
# Service Mesh for Mere Mortals

A guide to Istio and how to use service mesh platforms

Bruce Basil Mathews

# Service Mesh for Mere Mortals

By Bruce Basil Mathews

*I would like to dedicate this book to my wife, Lisa, and my son, Aaron, who continue to convince me that I have something important to say on the subject.*

# Preface

When the folks at Mirantis first approached me to begin setting down thoughts about the importance of Service Meshes, and the Ins-And-Outs of their features and benefits, I thought first about Laurel and Hardy.

In each of their films, they are placed in a situation requiring a great deal of planning and engineering to accomplish, such as hoisting a piano up a vast number of stairs or wallpapering an entire room.

Usually, they try to determine the best, most logical way of performing the task at hand. But, as they go about devising the hoist or creating a Rube Goldberg machine to apply the paste, things begin to fall apart rapidly, leaving both of them sitting in the shambles of what seemed to be a great idea.

This strikes me as being exactly like those attempting to transition between the prior generations' monolithic application architectures and more current virtualized architectures, and finally to a state of the art microservices-oriented architecture.

They see that containers alone won't solve the problem, so they mate it up with an orchestration engine like Kubernetes. Then, seeing that the orchestration engine falls short in terms of security, isolation, visibility and standardization, they begin to form all of these complex policies and Rube Goldberg pieces of code designed to try to fill the gaps in what they have deployed.

Unfortunately, that approach has about as much chance of succeeding as Oliver and Stanley getting the piano up the stairs or wallpapering the room successfully.

With this book, I am attempting to provide enough food for thought about how Service Mesh technology can help avoid such pitfalls as trying to recreate the wheel in terms of platform services, or allowing potential security issues to exist in the environment while filling the gaps in security, isolation, visibility, and standardization needed to make your microservices implementation work!

Hopefully, after reading this book, you won't be turning to me to say "Well, Bruce... This is another fine MESH you've gotten us into!"

## Acknowledgements

I want to acknowledge a debt of gratitude to the various developers of today's version of the Service Mesh, such as Google, Buoyant, and Kong, all of whom have taken some time over the past five years to explain to me their versions of the Service Mesh in great detail, and to provide me with a plethora of documentation. Much of what is written in this book stems from those learning sessions.

I would also like to thank the folks at Mirantis, specifically Nick Chase, Shaun O'Meara, and Rosheen Golden, for giving me the opportunity to present my thoughts in this area. I could never have acquired the level of knowledge I have on the subject had it not been for my time at Mirantis.

# Table of Contents

# Introduction

The reason I am writing this book is to help those who have already made the leap from monolithic application development, targeted for mainframes, to virtualized three-tier SOA oriented software development, and who have finally arrived at a point where they have begun investigation of microservices architecture for application development.

Many of you who have reached this plateau started out using technologies such as Docker for containerization and have already found the need for some type of automated orchestration using engines such as Mesos or Kubernetes for this purpose.

As you may or may not yet have discovered, though the orchestration engines do a very good job of coordinating and maintaining a specific number of instances of a microservice, scaling them up and down as needed, they are NOT very good at maintaining a Zero Trust security model, nor are they even able to participate fully in terms of Role-Based Access Control or strict Network Security, or even "standardization" of Platform Services such as Service Discovery, Load Balancing, or Application Level Encryption without the significant addition of external services and modifications to their Internal Resource definitions.

This is where the need for a Service Mesh add-on comes into play. Service Meshes provide Policy Engines and other features specifically targeted at filling these voids.

This book provides a deeper understanding of the available Service Meshes and their features and benefits, but most of all, it gives you the experience of actually using a Service Mesh so you can gain enough of an understanding of how they work to take on your own projects.

Since there are clear indications in the industry that specific types of Service Meshes are being used more commonly than others, I have made the deliberate choice to provide more detailed information about Service Meshes employing a Sidecar Architecture. The most popular one of the Sidecar oriented Service Meshes today is Istio, so we'll be using that for our examples. The concepts, however, are the same for all of the Sidecar Service Meshes.

## The Basics: How Did We Get Here?

Before discussing Service Meshes, let's lay a little foundation and provide a bit of context. We will be supplying the needed foundation in the first few chapters, but never fear! If you are a bit more advanced on the subject, we'll be getting into some really hairy specifics and some things that even most knowledgeable readers may not be aware of in the later chapters. (But in an understandable way, I promise.) So please bear with me as we get warmed up!

Beginning up front with something even many techies may not realize, the beginnings of the service mesh start with the development of the three-tiered architecture model for applications, which was used to develop and deploy a large majority of web-targeted applications.

At the time, the "mesh" was fairly simplistic; access to the "web tier" initiated a call to the "app tier," which in turn passed a call to the "database tier." The whole thing was tailored to provide sufficient performance back to the "web tier," where presumably an end user was awaiting a response. As the internet grew and applications became more and more complex, however, the three-tiered application model began to break down under heavier and heavier load.

This led to the advent of microservices that decomposed the monolithic web, app, and data tiers into multiple discrete components that operated independently.

At the same time, the dedicated networks between tiers became more and more generalized, and the concept of "east-west" traffic between all microservices (as opposed to the "north-south" traffic between the application and external consumers) was adopted.

Companies that relied on fast internet and network throughput to provide their services, such as Netflix, Google, Facebook and Twitter, initially followed the standard coding practices developed for "C" and "Java" and created vast libraries of functions and platform services, such as load balancing, telemetry, and circuit breaker patterning to "standardize" these operations for the developer across all of their microservices.

Voila! The birth of the modern Service Mesh (still in its infancy and needing its diapers changed) was upon us.

As these libraries of functions and operations became more widely used, some cracks in the armor started quickly showing up. Here are a few examples:

- One of these companies owning a library, (say, Twitter and Finagel) decides to add a feature, or to change the behavior of an existing feature. First, every instance where that library is in use by all developers needs to be recompiled. Upon update, the developer's code using the library may, and probably will, break. So, the whole thing has to be coordinated across a massive amount of the internet for each change to occur.
- Say a feature is becoming "deprecated" and it will cease to be included in the library as it is superseded by a newer feature or capability. The same type of difficult coordination is involved.
- Since there were and are no "standards" for the contents, protocols, or access points to these libraries, if you want to move an application from Twitter to Google, it must be rewritten.

The current cloud-native "work around" for the use of libraries is the use of proxies, which take in requests and spit out responses and provide a layer of abstraction in front of the actual library. A proxy-based architecture allows for updates and changes that do not require recompilation or extreme coordination to implement. Implementing the service mesh in proxies places the decisions for the platform functions and features used in applications in the hands of the platform operations and engineering teams, leaving the developer to focus on the application's logic.

So, at this point, we had decomposed monolithic applications into manageable individual services. Some were implemented as REST URLs, which didn't allow for individual scalability. To solve that, many were implemented as virtual machines,  but that was wasting the space and maintenance used by an operating system for very little benefit.

Containerization solved this problem, but didn't address the fundamental aspects of automated scaling up and down, network security, and platform service standardizations that would need to follow.

In addition, distributing these containers across multiple servers exponentially compounds the problems introduced, and requires some type of global, reusable solution.

Orchestration engines help with the scaling and operational issues of maintaining application uptime, but platform standardization, automated service discovery, load balancing, encryption, and so on were still left in the hands of the application developer to resolve.

## Why Didn't We Leave This to the Container Orchestrator?

In order to understand why developers of container orchestration engines such as Mesos, Openshift, and Kubernetes did not simply embed all of the needed features and functions to manage and maintain north-south and east-west network traffic in a cluster of containers, you first want to understand some of the background behind it, or what a container "orchestrator" is and what is designed and NOT designed to do.

Container orchestration automates the scheduling, deployment, networking, scaling, health monitoring, and management of containers. Deploying and scaling containers up and down across an enterprise can be challenging without some type of automation to perform load-balancing and resource allocation, all while maintaining strict security.

Container orchestration automates many of the more mundane tasks related to these operations in a predictably repeatable way. This makes the processes far more efficient, as they no longer involve human intervention to execute.

Although the automation parts of the orchestration engine make things easier, they also make security far more difficult to manage properly. For example, many organizations running container orchestration technologies assign full cluster administrator privileges to their users for day-to-day operational requirements that cannot be automated, but multiple applications are hosted across the same clusters of physical servers, so these administrators have access to applications other than their own.

This architecture breaks away from the entire concept of "least privilege." This problem can be mitigated by implementing a Role Based Access Control (RBAC)-based management scheme within the clusters, and Service Meshes provide policy engines that make implementation of an RBAC-based management scheme easier to achieve.

## The role of kube-proxy and the ingress controller

Orchestration engines typically route the network traffic between individual nodes over a virtual overlay network, which is usually managed by the orchestrator, and is often opaque to existing network security and management tools. The routing rules governing north-south and east-west traffic are maintained by a container hosted on each node of the cluster called the kube-proxy and containers hosted in the control plane called the kubernetes ingress controller.

The kube-proxy container proxies the network connections between nodes required by the application and maintains network rules on each of the nodes.

External connections to pods are handled by an edge proxy, known as an ingress controller because it is configured and maintained using ingress resources within the kubernetes framework. These network rules, which may come from external services such as load balancers and so on, allow network communication to your pods from network sessions inside or outside of your cluster.

All of this traffic is automated and "hidden" from external network operations.

Another complication of the kube-proxy/ingress approach implemented within the Kubernetes framework is that the various methods provided to accomplish the same end, such as exposing a service endpoint externally from within the cluster, are all implemented very differently.

For example:

1. The Kubernetes `NodePort` implementation exposes a `Service` endpoint against a single port on a single node.
2. The default load balancing mechanism within the Kubernetes framework consumes an IP address for each `Service`, and requires a separate `LoadBalancer` instance for each IP address.
3. The ingress methodology is implemented strictly using the Kubernetes API framework, so it can't apply to all cases needed to integrate with external access points as some external access points, such as F5 LoadBalancers, etc. are not recorded as a part of the Kubernetes Framework itself.

## Where the Service Mesh fits in

Service Meshes help standardize all of these needs within a central flexible methodology and service that relies more on declarative application programming than on imperative application programming. In other words,it enables the developer to focus on the "what" and the "why" versus the "how." Service Meshes also typically provide entry points for the introduction of outside probes to be used for network management and telemetry gathering purposes under strict security control.

Container orchestrators strive to maintain the scale and density of workloads. This means that by default, they can place workloads of differing sensitivity levels on the same host without some type of intervention in the process. Service Meshes can incorporate node labels within policies that enable workloads to be pinned to specific nodes, allowing for isolated deployments. (Kubernetes can achieve a similar effect by creating taints, tolerations and admission controllers, but because they are defined separately, and because of other limitations, they're not a complete solution.)

As you can see, Service Meshes fill several holes that the container orchestrator could not manage without significant rigidity and overhead being introduced into their operation, and they do it in a consistent and isolated way. This is why the use of a Service Mesh is becoming so popular among application developers.

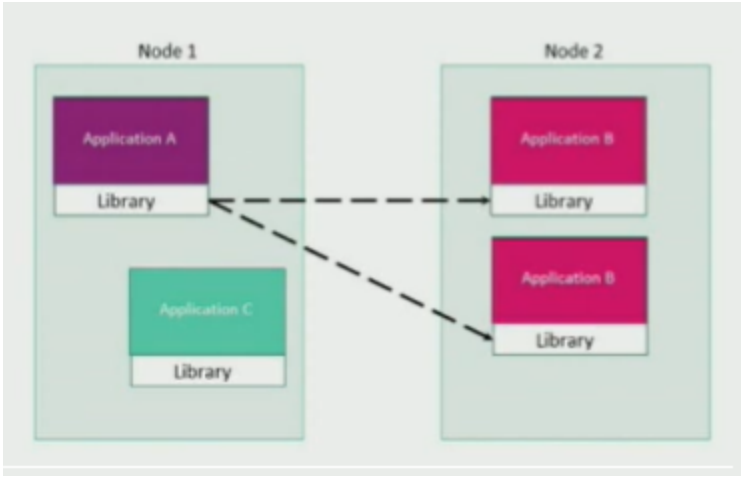
There are currently many proxied Service Mesh options to choose from that fill the need quite handily. We will be going over them in the next few chapters. Keep reading!

## How Did Sidecar Architecture Evolve?

So, this "proxy" thing didn't just happen, it evolved over time. The first iteration of applications that called themselves Service Meshes were actually "libraries" along the same lines as the BIG PLAYERS (Facebook, Google, Twitter,and so on). The only differences between them and the BIG PLAYERS was that the integrated calls with the BIG PLAYERS' platforms had been removed and replaced with targeted features and functions more generic in nature and more tailored to handling use cases beyond the needs of the BIG PLAYERS' applications.

This was, indeed, an advancement in that it freed the application developer from having to rewrite the application every time they moved it from one BIG PLAYER platform to another. However, the technique suffered from the same maladies as the BIG PLAYERS' versions. Whenever the library was changed, the code that took advantage of it had to be recompiled with the new version. Any deprecated calls had to be repaired, and new features would need to be taken into account individually. Developers spent more time incorporating the new versions than creating new functions and features in their own application, something you NEVER want to have happen!

## The Node Agent Architecture

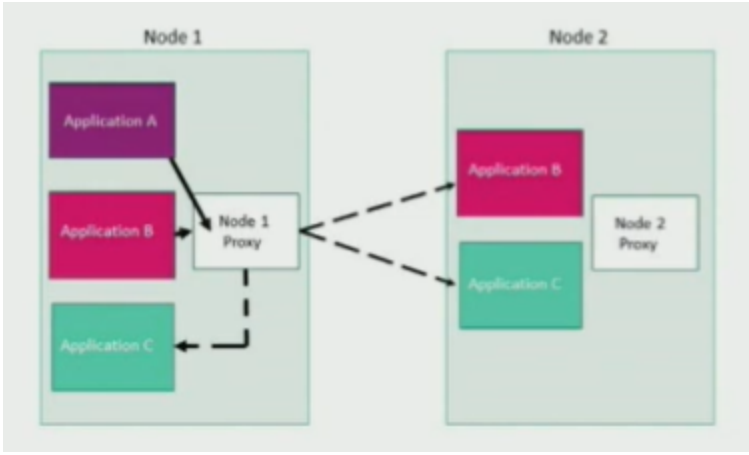



The next iteration attempted to resolve the update, upgrade deficiencies by creating a per node instance of the functions and features managed at the worker node level for each grouping of containers hosted on the platform. This became known as the Node Agent architecture.

The Node Agent architecture made it much easier to replace and repair, depending on the number of workers in your clusters. It also made it easier to introduce new features and capabilities as the ease of rollout reduced the time to implementation.

However, since ALL of the east-west traffic from the node was being routed through a single instance of the Node Agent, if, for whatever reason, the Node Agent ceased to function properly, or to transmit the east-west or north-south data that it was designed to handle, the "blast radius" would be ALL of the containers being hosted on that worker node at that time. A single failure could take out an entire application. That would NOT be good.

## The Sidecar architecture

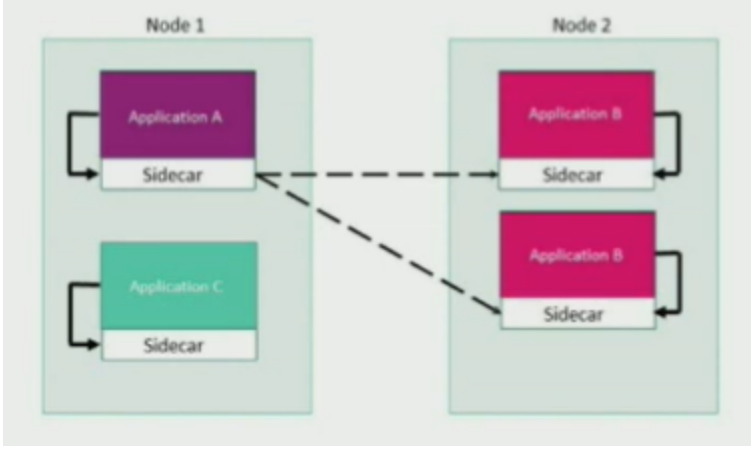



Finally, a completely new and different architecture was developed. It involved the automated spawning of a separate container for each container deployed in a namespace.

The purpose of the second container was to perform the redirection of all of the north-south and east-west network traffic to and from the original container -- in other words, to act as a "proxy."

This new architecture would allow for the traffic to be acted upon based on rules from a single control point, such as those concerning encryption, load balancing, telemetry capture, and so on, and they could be applied by an external operations and platform source without the container having to be made aware of the redirection rules.

This second container became known as a "sidecar." New features and capabilities would be introduced at the controller layer making them much more easily managed. If a sidecar failed to perform the redirection properly, the "blast radius" would be limited to a single container.




For these, and other reasons that we shall dive into later in the book, the Sidecar Architecture has now become the prominent player in the Service Mesh space.

**Service Meshes Using a Sidecar Architecture**

The three primary players in the Service Mesh space that take advantage of a sidecar architecture approach are:

- Istio
- LinkerD
- Kong

Although there are others out there claiming to be in the Service Mesh space commercially, the fact is that many of them, such as Aspen Mesh and AWS Mesh, are founded and developed on top of the Istio Service Mesh framework. Almost all of the frameworks, including Istio, rely on Envoy as their Layer 7 proxy at the sidecar level, though both LinkerD and Kong have written their own L7 proxy.

In this book, we'll concentrate on Istio.

## Istio As a Service Mesh In Depth

Istio (ISS-tee-oh) was named by the Founder of Tetrate, Varun Talwar, and Google Principal Engineer Louis Ryan, in 2017, when they were both at Google. The word, "istio" (ιστίο) is Greek for "sail", and extended the greco-nautical theme that was established by the Kubernetes team. (The word kubernetes, which was also developed at Google, is Greek for pilot, or helmsman.)

Istio (sail) and its meshy cousin Istos (ιστός,) which means mast, net, or web, both come from the ancient Greek root Istimi (ἵστημι), which means, "to make stand." The two Istio developers had an idea that kubernetes alone would need a little "help" in order to stand up in an enterprise. Thus the name.

The Istio application is actually an open source service mesh, originally developed by Google in 2017 and turned over to the Open Source community in 2020 (with some Googlish caveats.)

Istio helps organizations run distributed, microservices-based apps anywhere. Istio claims to enable organizations to secure, connect, and monitor microservices in order to modernize their enterprise apps more swiftly and securely. It is intended to be used in a transparent way with minimal code changes required to your existing images, enabling them to use all of the features and functions available within the Istio framework.

Also, use of Istio is language independent. So if your development team uses multiple languages to code your images, each can still use the Istio Service Mesh without changing code or being locked into one language.

Istio is also said to be platform independent, which means it will run in a variety of environments, such as in the cloud and on bare metal on premises and against multiple orchestrators such as Kubernetes and Mesos.

The Istio framework consists of two parts:

- Control Plane
- Data Plane

The Istio Control Plane is where everything is configured to dynamically program the operations of the server proxies as policies or the environment are updated. This limits the need for changes to the sidecars that would require any recompilations. That would get very messy with thousands of sidecars to deal with! In March of 2020, Istio introduced a new version of their Control Plane components that reduced the multiple components into a single binary called `istiod`. This move seemed counter intuitive, as we are constantly preaching the gospel of microservices and service decomposition. But when we dug a little deeper, the move to a single binary for all Control Plane services makes a ton of sense.

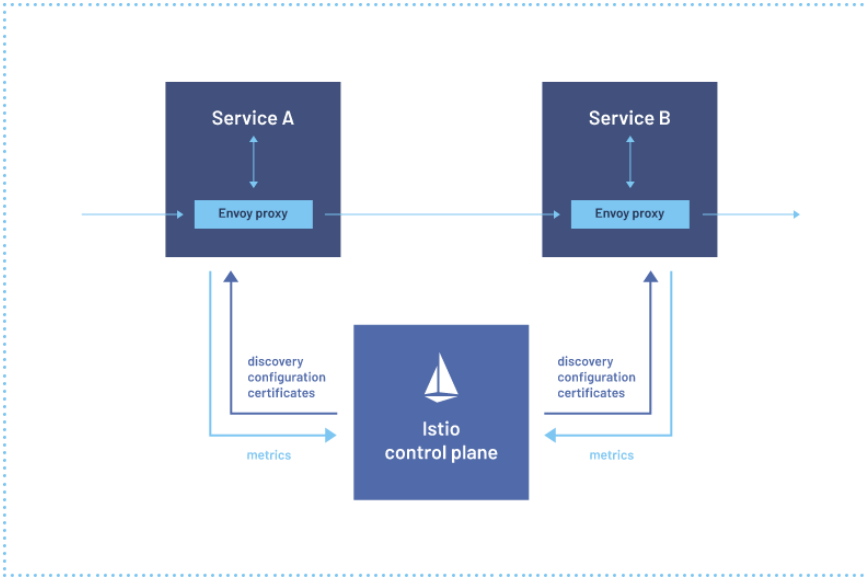
Normally, the Istio developers would have continued to provide separate microservices such as Pilot for discovery services, Galley to maintain the configuration information, Citadel to maintain generated certificates for authentication, and the Mixer service to provide extensibility and centralized traffic control. However, the vast majority of Istio implementations are performed and operated by a single team of people within an organization, making the separation of components unnecessary, if not counterproductive.

The versions of the microservice components had always been released as a unit and were tested together at a particular revision, so creating a single binary made the release process easier and far more rapid with a more guaranteed result.

Also, because all of the microservice components for Istio are installed in the same Kubernetes namespace. The deployment as a single binary ensures the proper placement during installation, limiting any security concerns to that single namespace.

Finally, this move to a single binary allowed for the Envoy proxy to be written in C++ and everything else to be written in Go, increasing performance and rapidity of upgrade/update for all components. As a result of this configuration change, installation, configuration, maintenance and scalability all became easier. Startup time has been reduced as it only takes a single pod starting for all services to be available.

Finally, resource utilization was also significantly reduced as the single pod, once started, provides all of the services needed to operate.




Now that we know what our tools are, let's look at using them.

## What Are the Common Features and Capabilities of These Service Meshes

Now that we have our context, we're ready to talk specifically about Service Meshes.

Service Meshes generally provide some form of the following capabilities:

- Service Discovery
- Load Balancing
- Encryption
- Observability
- Traceability
- Authentication
- Authorization
- Circuit Breaker Pattern support

Let's take a deeper look at each one of these capabilities by seeing them function within Istio.

## Create a Sidecar Oriented Service Mesh with Istio

Let's take a simple concrete example of using a Service Mesh implemented in a Kubernetes cluster to provide a really good idea of how this whole thing works.

You may be setting up your Kubernetes on a public cloud provider, in which case, you can follow the instructions for installing the Kubernetes part of the equation using the vendor's built in platforms, such as GKE for Google, AKS for Azure, or EKS for the AWS platform.

In this case, I will be providing documentation for using the kOs Kubernetes distribution to install a simple cluster. You can do this on Bare Metal if you have the resources, or using a tool such as Virtualbox (http://virtualbox.org). The process goes like this:

1. Set up the base Kubernetes portion of the cluster.
2. Use this cluster as a foundation for our Istio deployment.
3. Install one of the demonstration applications provided in the Istio distribution that will contain examples of multiple Service Mesh features, including Service Discovery, Load Balancing, Encryption, Observability, Traceability, and Circuit Breaker Patterning.
4. Walk through each step used to enable each feature.

Let's start by installing the Kubernetes cluster.

# Install the Kubernetes cluster

If you already have a Kubernetes cluster at your disposal, you should be able to follow along without having to install a new cluster. If not, you can easily get one using kos. kos is an all-inclusive Kubernetes distribution, configured with all of the features needed to build a Kubernetes cluster simply by copying and running an executable file on each target host.

## Install kos

While kos makes it pretty straightforward to install Kubernetes on multiple machines, for the purpose of this book, we actually don't need more than a single node, and because we don't need anything but localhost access, installation is pretty straightforward.

(For instructions on creating a three-node cluster, or if you need to access the cluster from outside the local machine, I've added more instructions at https://www.mirantis.com/blog/more-service-mesh-info/.)

kos does require Linux, but if you are running Windows or MacOS, you can create a virtual machine using VirtualBox (http://virtualbox.org).

Kubernetes defines two types of nodes: controllers and workers. kos enables you to create nodes of either type, but we're going to create one that does both:

1. First download and execute the install script:

```
curl -sSLf https://get.k0s.sh | sudo sh
```

2. Next install the service:

```
sudo k0s install controller --single
```

3. Finally, start the service:

```
sudo k0s start
```

From there, you just need to have configured access to the Kubernetes client, kubectl. For instructions on installing kubectl for your operating system, you can see the documentation at https://kubernetes.io/docs/tasks/tools/. As for pointing it at your cluster, on the local machine, you can execute:

```
export KUBECONFIG=/var/lib/k0s/pki/admin.conf
```

If you're working from another machine or with a tool such as Lens (http://k8slens.dev), simply copy over the file. (For access from another machine, remember to check https://www.mirantis.com/blog/more-service-mesh-info/!)

So now that you have kubectl pointing to a Kubernetes cluster, let's install Istio and deploy a demonstration application called BookInfo.

## Deploying Istio Service Mesh Into the k0s cluster

I am including instructions for installing Istio and its related components on the Ubuntu operating system running a Kubernetes cluster as a reference for you. You will want to find alternative instructions for variations in your platforms and configuration if required. (Again, if you have an alternate environment one place to start is https://www.mirantis.com/blog/more-service-mesh-info/.)

Install the Istio Service Mesh by issuing the following commands:

```
curl -L https://istio.io/downloadIstio | sh -
cd istio-1.11.2
sudo chmod +x bin/istioctl
sudo cp bin/istioctl /usr/local/bin/istioctl
istioctl install --set profile=demo -y
```

This last command will create the `istio-system` namespace and install all components to effectively run this example. The output from the command should look like this:

```
✔ Istio core installed
✔ Istiod installed
✔ Egress gateways installed
✔ Ingress gateways installed
✔ Installation complete
```

At this point, we are ready to install the demonstration application known as BookInfo under the Istio Service Mesh.

## Deploying the BookInfo Application for Testing

Now that we have Istio Service Mesh deployed, let's put an application into it so we can test its operation.

The BookInfo simple Reviewer application displays information about reviews of the Shakespeare play Comedy of Errors, one of Shakespeare's funniest plays in which no one dies. We are using it as the example application because it yields concrete examples of most of the seven capabilities built into the Istio Service Mesh: Service Discovery, Load Balancing, Encryption, Observability, Traceability, Authentication, Authorization, and Circuit Breaker Pattern support. Once the application is installed, we will be going over each one of these features.

To install the BookInfo application, we need to follow these steps:

1. Create the `bookinfo` namespace

   ```
   kubectl create namespace bookinfo
   ```

2. Label the namespace to automatically create a sidecar for every container created in the `bookinfo` namespace

   ```
   kubectl label namespace bookinfo \
     istio-injection=enabled
   ```

3. Change directory to the Istio directory for the following commands to work

   ```
   cd istio-1.11.2
   ```

4. Create the `bookinfo` application in the `bookinfo` namespace using the YAML file provided with the download.

   ```
   kubectl -n bookinfo apply -f \
   samples/bookinfo/platform/kube/bookinfo.yaml
   ```

This will install all of the components of the BookInfo application into the bookinfo namespace in the cluster. The output should look like this:

```
service/details created
serviceaccount/bookinfo-details created
deployment.apps/details-v1 created
service/ratings created
serviceaccount/bookinfo-ratings created
deployment.apps/ratings-v1 created
service/reviews created
serviceaccount/bookinfo-reviews created
deployment.apps/reviews-v1 created
deployment.apps/reviews-v2 created
deployment.apps/reviews-v3 created
service/productpage created
serviceaccount/bookinfo-productpage created
deployment.apps/productpage-v1 created
```

5. Now you can get a list of the running services in the `bookinfo` namespace:

   ```
   kubectl -n bookinfo get services
   ```

   The output should look something like this:

   ```
   NAME          TYPE        CLUSTER-IP
   EXTERNAL-IP   PORT(S)    AGE
   details       ClusterIP   10.0.0.212
   <none>        9080/TCP   29s
   kubernetes    ClusterIP   10.0.0.1
   <none>        443/TCP    25m
   productpage   ClusterIP   10.0.0.57
   <none>        9080/TCP   28s
   ratings       ClusterIP   10.0.0.33
   <none>        9080/TCP   29s
   ```

```
reviews       ClusterIP   10.0.0.28
<none>        9080/TCP   29s
```

6. Display all of the pods running in the `bookinfo` namespace

   ```
   kubectl -n bookinfo get pods
   ```

   This command shows all of the pods used to deliver the BookInfo application in a running state. The output should look something like this:

   ```
   NAME                                READY
   STATUS    RESTARTS   AGE
   details-v1-558b8b4b76-2llld         2/2
   Running   0          2m41s
   productpage-v1-6987489c74-lpkgl     2/2
   Running   0          2m40s
   ratings-v1-7dc98c7588-vzftc         2/2
   Running   0          2m41s
   reviews-v1-7f99cc4496-gdxfn         2/2
   Running   0          2m41s
   reviews-v2-7d79d5bd5d-8zzqd         2/2
   Running   0          2m41s
   reviews-v3-7dbcdcbc56-m8dph         2/2
   Running   0          2m41s
   ```

7. You can now test to see if the application is running properly by issuing the following command to pull the title of the output page:

```
kubectl -n bookinfo exec "$(kubectl -n
bookinfo get pod -l app=ratings -o
jsonpath='{.items[0].metadata.name}')" -c
ratings -- curl -sS
productpage:9080/productpage | grep -o
"<title>.*</title>"
```

The output from this command should look as follows:

```
<title>Simple Bookstore App</title>
```

8. Now we will associate the BookInfo application with an Istio Service Mesh Ingress Gateway to enable the mesh to control the application flow by issuing the following command:

```
kubectl -n bookinfo apply -f \
 samples/bookinfo/networking/bookinfo-
 gateway.yaml
```

The output of this command should look as follows:

```
gateway.networking.istio.io/bookinfo-gateway
created
virtualservice.networking.istio.io/bookinfo
created
```

You can look at the YAML file to get a better sense of what's happening here, but for now, let's concentrate on using the Service Mesh itself.

9. This `istioctl` command verifies that the mesh is working correctly

```
istioctl -n bookinfo analyze
```

The output of this command should verify that things have been installed properly into the Istio Service Mesh. The output of the command should look like this:

```
✔ No validation issues found when analyzing namespace: bookinfo.
```

10. Now we need to know where to access the application. This command displays the Virtual IP Address on which the Load Balanced BookInfo application has been placed:

    ```
    kubectl get svc istio-ingressgateway -n istio-system
    ```

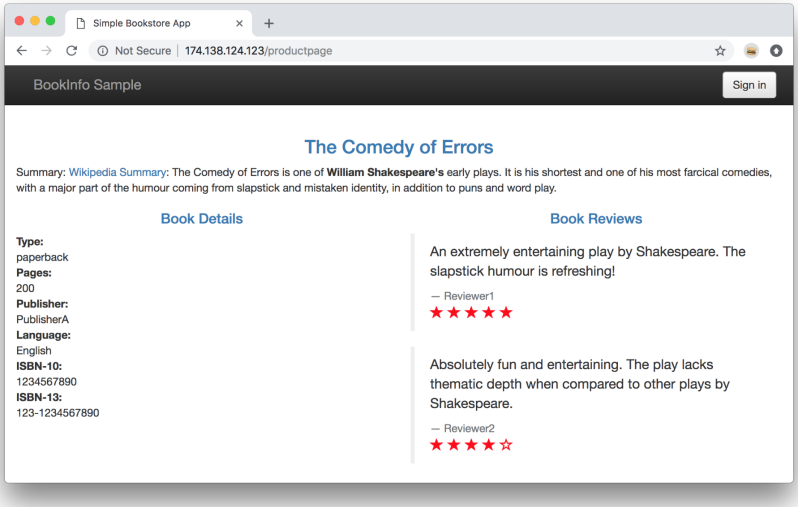
    If your Kubernetes cluster supports a `LoadBalancer`, the `TYPE` will be displayed as `LoadBalancer,` as in the following:

    ```
    NAME                   TYPE
    CLUSTER-IP       EXTERNAL-IP     PORT(S)
    AGE
    istio-ingressgateway   LoadBalancer
    172.21.109.129   130.211.10.121
    80:31380/TCP,443:31390/TCP,31400:31400/TCP
    17h
    ```

Based on the example above, you can now access the BookInfo application by pointing your favorite browser at the following URL:

```
http://172.21.109.129/productpage
```

Make sure to use the correct `CLUSTER-IP` or `EXTERNAL-IP,` as appropriate. The application will look like this:




When you refresh the URL, the stars displayed will randomly shift from RED to BLACK to NO STARS. This is intentional and is connected to the Load Balancing and Circuit Breaker Patterning Destination Rules defined within the BookInfo application. We'll look at these rules as we go along.

So far, based on what we have installed, we are using Service Discovery and Load Balancing as parts of this example. Now let's look a little deeper and add some more features to apply to the BookInfo use case before we start creating our own code.

## Adding Observability and Traceability to the Istio Service Mesh

Now that BookInfo is installed and operating through the Istio Ingress Gateway, let's add some observability add-ons to the implementation to enable us to see what is going on in the mesh and where there may be bottlenecks occurring or rules being applied. Run the following commands on the master node to install the observability and traceability add-ons:

```
kubectl apply -f samples/addons
kubectl rollout status deployment/kiali \
  -n istio-system
```

The response to the last command, which is designed to wait until the whole deployment is completed so as to not let any part of the deployment be used without all parts being installed completely, should look like this:

```
Waiting for deployment "kiali" rollout to finish:
0 of 1 updated replicas are available...
deployment "kiali" successfully rolled out
```

Kiali is a management console for the Istio-based service mesh you just installed. The Kiali graphic user interface displays various dashboards, observability, and metrics, and lets you operate your mesh with extensive configuration and validation capabilities. Kiali shows the structure of your service mesh graphically by capturing and displaying traffic topology and presenting telemetry data regarding the health of your mesh.

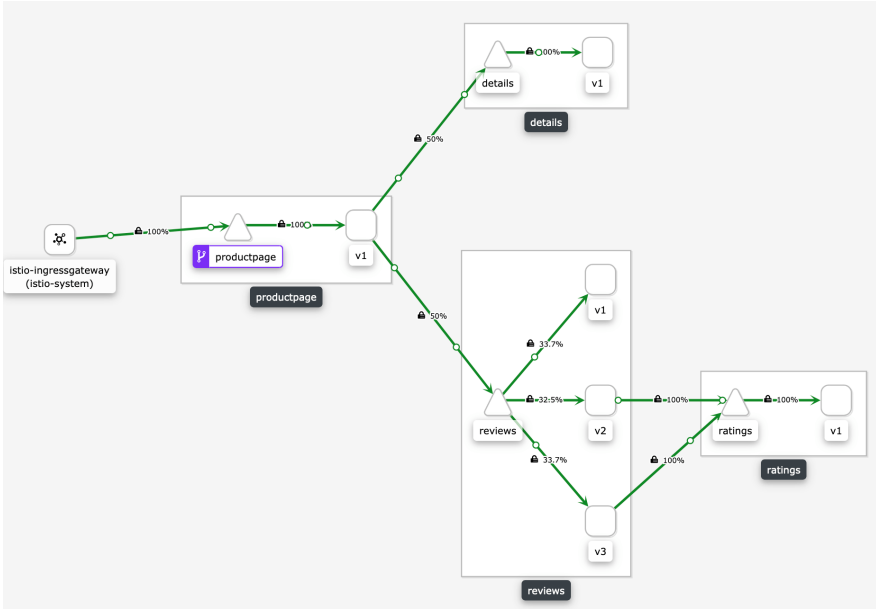
You can access the Kiali dashboard on the master node by issuing the following command in a command window. The command will then use the desktop to display Kiali from a browser:

```
istioctl dashboard kiali
```

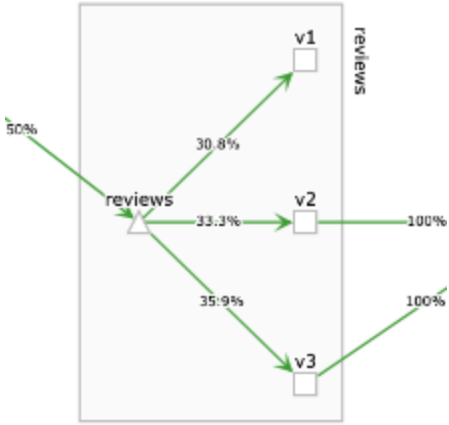
If the server doesn't have a desktop, you can access the Kiali application by using Kubernetes port forwarding on the server hosting the Kiali container from port number 20001. (See https://www.mirantis.com/blog/more-service-mesh-info/ for more information.)

Make sure you are reloading the BookInfo application in your browser to generate traffic to and from the application.

Now that you are seeing the Kiali interface, in the left navigation menu, select Graph, and in the Namespace drop down, select `bookinfo`. The display should look like this:



Now, I want to point out a few things about this graph. First, I want you to focus in on the reviews section depicted here:




Note that the review responses are being distributed (Load Balanced) across three different versions of the ratings portion of the application. The three service instances, identified as V1, V2, and V3, are all "known" to the application via the version tag, and are translated to the actual endpoint via Service Discovery. The three instances are then load balanced because there is a LoadBalancer Destination Rule that defines V1, V2, and V3 as the ratings access points. This gives you the RED, BLACK and NO STARS being presented as each "version" of the ratings application are configured differently.

Next, I want you to notice the V1 version of the ratings application. It's not displaying NO STARS because that's how it's designed, it's displaying NO STARS because the request doesn't progress to the ratings backend in the first place. This is because there is a SECOND destination Rule in place forming a Circuit Breaker Pattern indicating that the V1 version of the ratings application has been deprecated and should no longer be used.

In addition, ALL of the traffic from ALL of the entry and exit points from the ProductPage to the Ratings backend are being protected by STRICT mTLS as defined in the service for the `bookinfo` pod, as we'll see later. We'll also see other features, such as observability.

They're all defined as YAML documents, which you can see if you look at the complete file, stored in Github at https://github.com/istio/istio/blob/master/samples/bookinfo/platform/kube/bookinfo.yaml. To make things more manageable, we'll go through individual sections as we get to them.

Now let's shift gears a bit and provide more detail on each of the features we have put into action in the BookInfo application.

## Service Mesh Service Discovery

Generally speaking, Service Discovery is how your applications and services find each other over a network. Given the input of the name of a service, service discovery tells your application where that service lives (that is, on what IP/port pairs its instances are running.)

Service Discovery is really one of the lowest level expectations of a Service Mesh and is an essential component of multi-service applications because it enables services to refer to each other by name, independent of where they're deployed. Service Discovery also automates the process of determining what instances of available services fulfil a given request.

Service Discovery processes get called automatically in order to return a list of services that satisfy the application's request, so as monolithic applications became more decomposed and replaced with microservices, the need for such Service Discovery mechanisms became far more complex.

Take, for example, a simple task such as name resolution. It used to be that a single Domain Name Service, tying a name to the IP Address of a hosting application service, satisfied the result. Now, Service Discovery not only needs to take into account physical server locations, but also the dispersed geography, including services no longer held under a single enterprise's control.

Load Balancing also complicates matters, as the redirections from one-to-many may be returned from a source different from the origination point. Additionally, when new instances of a specific service become available as a result of application scaling, the Service Discovery mechanism needs to be made immediately aware of a new point of return for the same service. This means that the Service Discovery feature provided by a Service Mesh needs to be as sophisticated and service aware as the Service Mesh it supports.

Istio, for example, takes advantage of the Kubernetes list of services and endpoints to populate and maintain its Service Registry. It uses this Service Registry to instruct the Envoy proxies as to where to direct the application's traffic.

When there are multiple instances of the same microservice available, by default, Envoy uses the Istio Service Registry to send the traffic to each instance in a round-robin fashion. Istio provides a Traffic Management API that can be used to add, alter, or remove specific behaviors by adjusting the Kubernetes Custom Resource Defilitions (CRDs) that define them.

## A Simple Example of Using Service Discovery in Istio Service Mesh

There are two distinct services involved with delivery of Istio's service discovery mechanism and registry: Virtual Services and Destination Rules. These two features form the key building blocks of Istio's traffic routing capability.

A virtual service lets you configure how requests are routed to an actual service within an Istio service mesh, building on the basic connectivity and discovery provided by Istio and the Kubernetes endpoint API. The Virtual Service defined within the Istio and Kubernetes resources consist of routing rules that get evaluated sequentially.

The evaluated rules enabled Istio to match each request to the Virtual Service to a specific physical destination hosted within the mesh. Virtual Services enable Istio to decide on the appropriate traffic behavior to and from one or more hostnames. Routing Rules are used within the Virtual Service definition to tell the Envoy proxy how and where to send the virtual service's traffic to specific services or hostnames.

A simple Virtual Service use case is to send traffic to different versions of a service, specified as subsets. A good example of this is the reviews service displayed within the BookInfo application, where the v2 version of the review shows RED STARS, the v3 version of the load balanced group shows BLACK STARS, and the v1 version, which is deprecated, shows NO STARS in the page display. Clients send requests to the virtual service host as if it was a single endpoint, and the Envoy proxy routes the traffic to different versions of the service, depending on the virtual service rules contained in the Virtual Service definition.

For example, this definition of a Virtual Service sends requests to different versions of the service depending on whether the request comes from a specific user:

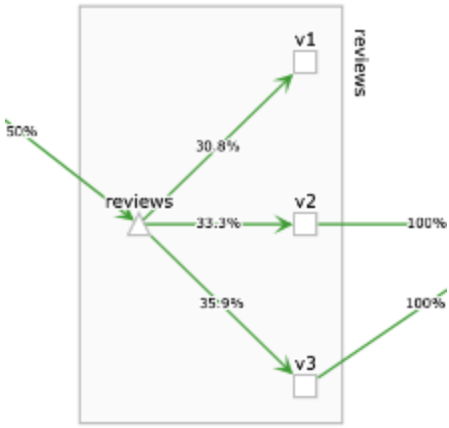
```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
```

```
hosts:
- reviews
http:
- match:
  - headers:
      end-user:
        exact: jason
  route:
  - destination:
      host: reviews
      subset: v2
- route:
  - destination:
      host: reviews
      subset: v3
```

So in this case, requests have two options: if they match the first condition (they have a header of `end-user: jason`) they are routed to host `reviews`, subset `v2`. Otherwise, they are routed to subset `v3`.

Note that a Virtual Service hostname can be a Kubernetes service short name such as `reviews`, as in this case, an IP address, or a DNS name that resolves, implicitly or explicitly, to a Fully Qualified Domain Name (FQDN).

Virtual Service hosts don't actually have to be part of the Istio service registry, enabling the developer to model traffic for Virtual Service hosts that don't have routable entries inside the mesh. Once again, by looking into the Kiali Management Console, you can see the review service subsets, identified by v1, v2, and v3, in operation:




The Services Subset tells the application what to include/exclude in the Service Group, defining the "what." If you want to change the behavior of the defined services, you need to establish a Destination Rule to tell the traffic targeted to the Service Group or Service Group Subset "where" to send the traffic. A Virtual Service or Service subset matches on a rule and evaluates a destination to route traffic to, whereas Destination Rules define available subsets of the Service to send the traffic after the routing has been established.

You can see this in action as follows. First add the VirtualService defined above to a text file and create it on the cluster:

```
kubectl create -f jasonisspecial.yml
```

Now send a curl request that specifies the header:

```
curl -header "end-user: jason" \
  http://172.21.109.129/productpage
```

If you check the Kiali dashboard, you can see that no matter how many times you send the request, it always goes to `v2`.

Try it without the header:

```
curl -header "end-user: jason" \
  http://172.21.109.129/productpage
```

to verify that it goes to v3 instead.

## Service Mesh Load Balancing

Load balancers increase application capacity by enabling the concurrent number of users to be increased by distributing the workload across multiple instances providing the same service. In addition, Load Balancers enhance reliability of applications by automatically redirecting requests to alternate microservices in the event of a failure.

They also improve the overall performance of applications by decreasing the workload utilization on individual servers, providing monitoring, management, maintenance and network usage, delivering service to the application, and by performing application-specific tasks defined within their configuration parameters.

Before we start coding, let's look at the theory behind Load Balancers and Istio.

There are basically two different types of load balancing to be discussed here. There is external load balancing for ingress and egress to a given cluster, and there is a whole lot of internal load balancing required for communication between services within the cluster.

Load balancing for cloud-native applications can be performed at different network layers.

Like the resilience strategies we just discussed, load balancing can be performed at layer 4 (the network or connection level). At this layer, the microservice and orchestration engines are responsible for resilient delivery of the service. For example, if a particular microservice instance stops responding to requests because of a failure at its location, such as a network outage, new requests would automatically be routed to other instances of that microservice that are still operating normally.

Load Balancing can also be introduced at layer 7 of the model (this is the layer that human users, as well as most other applications, directly connect to). Generally speaking, Network Administration is responsible for providing resilience strategies at this layer.

You can actually create a cloud-oriented or bare metal on-premises external Load Balancing service when you create any new service within the Kubernetes framework. The Load Balancing service is then passed along via Service Discovery to the Service Mesh.

There are a couple of different ways to do this if the deployed environment supports load balancing. The most well known is to use Kubernetes' native LoadBalancer object by adding the line `type: LoadBalancer` to the services configuration file, or via the kubectl command line when explicitly exposing a service by adding the `--type=LoadBalancer` switch to the command.

Kubernetes Ingress Controller capability offers the most popular method for true load balancing. The Ingress resource defines a set of rules that govern traffic flow and control access, and the daemon applies the rules inside a specialized Kubernetes pod. The Ingress controller has its own sophisticated capabilities and built-in features for load balancing, and can be customized to integrate with specific physical load balancer vendors and cloud platforms to provide load balanced access to Kubernetes hosted applications. The standard Kubernetes Ingress resource makes it easy to configure SSL/TLS termination, HTTP load balancing, and Layer 7 routing.

Using this native setup, when in-cluster services communicate, a load balancer called kube-proxy forwards requests to service pods at random. Each worker node in a Kubernetes cluster hosts a kube-proxy container process, which is placed in a pod in the kube-system namespace, per Kubernetes documentation.

The kube-proxy microservice manages the forwarding of traffic addressed to the virtual IP addresses (VIPs) used within the cluster's Kubernetes Service object definitions to the appropriate backend destination microservices. Though there are several modes of operation that the kube-proxy microservice can be configured to use, the two most popular are based on iptables or IP Virtual Server (IPVS).

The kube-proxy configuration for iptables, generally used as the default, takes advantage of Linux kernel-level Netfilter rules to configure all routing for Kubernetes services between instances. When load balancing for multiple backend pods, it uses the unweighted round-robin scheduling algorithm. The algorithm can be generally described as the dedication of a time slice (or quanta) being assigned to each process in equal portions and in circular order, handling all processes without priority.

The kube-proxy IPVS configuration is also built on the Netfilter framework, but it implements load balancing at Layer 4 in the Linux kernel. This configuration supports a greater variety of load balancing algorithms, including variations such as least connections, which directs network connections to the server with the least number of established connections, and shortest expected delay, which, as the name implies, assigns network connections to the server with the shortest expected delay time to respond.

## Istio Load Balancing

You can use Istio to add more complex load balancing methods, enabled by their use of the Envoy proxy. By default, Istio uses a round-robin load balancing policy. Each service instance in the instance pool gets a request sequentially as each instance of the same service was discovered. It returns to the top of the list when the last service instance in the list has had a turn. Istio also supports the following additional algorithms when specified in the destination rules for a given service or service instance:

- Random: requests are directed randomly to discovered members of the load balancing pool list

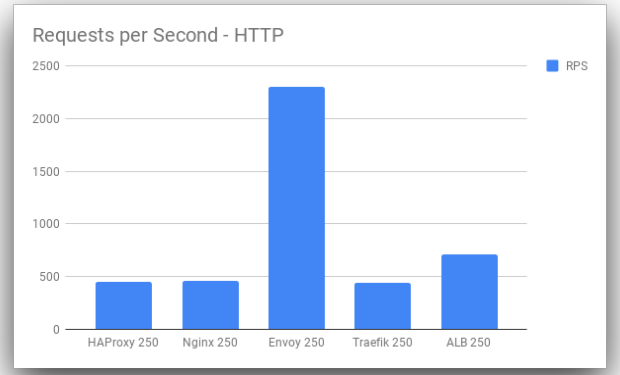
- Weighted: the developer to specify a certain percentage of all traffic requests to specific instances of the load balancing pool, which becomes quite handy when performing A/B or other types of testing to implement new versions of an application
- Least Request: sends traffic requests to the instance with the least queued requests, which assumes that each request has the same weight

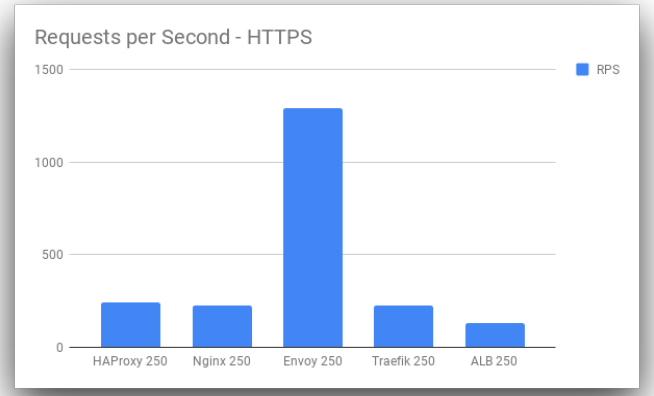
Although these three algorithms are the ones predominantly in use and most heavily tested, since Istio is founded on the Envoy proxy, these other algorithms are available at the developer's discretion:

- Ring Hash: based on mapping all hosts onto a circle such that the addition or removal of a host from the load balanced pool as the changing number of entries in the circle only affects one request out of every total member of the pool. It relies on the minimum ring size which is specified at runtime
- Original Destination: the upstream host is selected based on the downstream connection metadata. This allows the connection to remain "sticky" no matter how many instances are available in the load balancing pool
- Panic Threshold: establishes a minimum number of healthy services/hosts, usually set to 50%, to direct requests to, in the event of multiple failures. It is intended to mitigate situations where host failures could cascade throughout the cluster as application request load increases
- Zone Aware:  sends as much traffic to the local zone instance of an application as possible. (Note that there are multiple caveats to be met in order to attempt using this algorithm.)

- Load Balancer Subsets: configured to divide hosts within an upstream cluster into subsets based on metadata attached to the hosts, this algorithm ensures that hosts within the load balancer pool meeting the proper metadata criterion are selected. In the event of service failures, load balanced operations fall back to using any host in the load balancer pool.

Note that all of these algorithms involve the use of the Envoy proxy, which has a much higher throughput than other proxies. For example, here are a few Load Balancing statistics for the Envoy proxy employed by Istio that I thought might be of interest.






It appears that the envoy proxy service can handle a much larger number of Requests Per Second for both nonsecure and secure browser traffic than several competitors.

But that's enough theory. Let's see it in use!

## A Simple Load Balancing via Destination Rules Example Using Istio Service Mesh

Just as Virtual Services define how the application routes its traffic to a given destination, Destination Rules are used to configure what happens to traffic once it reaches that destination. Destination Rules define policies that apply to traffic intended for a service after routing has occurred. These rules specify configuration for load balancing, connection pool size from the sidecar, and outlier detection settings to detect and evict unhealthy hosts from the load balancing pool.

The example Destination Rule below, taken once again from the BookInfo example provided in the Istio distribution, configures three different subsets for the BookInfo load Balancing Destination Service, each with different load balancing policies:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule
spec:
  host: my-svc
  trafficPolicy:
    loadBalancer:
      simple: RANDOM
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
    trafficPolicy:
      loadBalancer:
        simple: ROUND_ROBIN
  - name: v3
    labels:
      version: v3
```

This Destination Rule definition file contains three destinations in the pool that are defined by a Version Tag within the Load Balancing pool. In this case, the default loadBalancer policy is RANDOM, but the v2 version uses the ROUND_ROBIN policy instead.

We can use the same method of applying Destination Rules to Load Balancing subsets to define the Service Ports. The example below shifts the input and output from the ingress Port 80 to Port 9080 for the BookInfo application. You can see that the DestinationRule contains a global `trafficPolicy` for a service with the label `bookinfo-ratings-port`. The following rule uses the least connection load balancing policy for all traffic to port 80, while it uses a round robin load balancing setting for traffic to the port 9080:

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: bookinfo-ratings-port
spec:
  host: ratings.prod.svc.cluster.local
  trafficPolicy: # Apply to all ports
    portLevelSettings:
    - port:
        number: 80
      loadBalancer:
        simple: LEAST_CONN
    - port:
        number: 9080
      loadBalancer:
        simple: ROUND_ROBIN
```

As you can see, Load Balancing can have many permutations and intricacies that require fine tuning. If the developer were left to their own devices, there might be thousands of variations for operations teams to manage. Istio, with their Service and Destination Rule definitions help to standardize the toolset. Amd ease the burden of managing and maintaining Load Balancing while providing a plethora of options to the developer.

## Service Mesh Encryption

Today, we are all striving for a Zero Trust security model. Zero Trust is a security framework requiring all users, both inside and outside the organization's network, to be authenticated, authorized, and continuously validated from a security perspective before being granted access to applications and data.

It has been touted that ninety percent of all internet traffic is encrypted to prevent outside people and processes from eavesdropping, and hackers from attempting man-in-the-middle attacks.

This principle has been carried through to include data movement between services and microservices in our rapidly containerizing world. In enterprise environments, we don't want just anybody calling our service, even if they are on the same network. In a Service Oriented Architecture (SoA) for example, it may only be valid for one particular service to call a specific backend service, and nobody else, especially not end users.

You want to encrypt all of your data from the moment it enters your network until it leaves (and beyond if you can pull it off!) But there is a price to be paid for this type of security. Time is the major one. It takes time to encrypt and decrypt data streams. It also takes resources, particularly CPU resources as the encryption/decryption processes are math intensive. There is also the added task of maintaining the seed data securely.

These and other factors can cause many headaches for encryption/decryption processing. Fortunately, the Service Mesh piece of the encryption equation has been well thought out and has reached a level of sophistication and maturity that makes the use of encryption within the Service Meshes realm a no-brainer.

Kubernetes has a built-in  rudimentary secret distribution capability and a control-plane certificate management component that service meshes can connect to, enabling the mesh to secure the microservices running in the Kubernetes cluster.

In order to secure the data driving an application as it flows from one service to another, the Service Meshes can be used to encrypt traffic between them automatically. The Mutual Transport Layer Security, or mTLS protocol, addresses this security need by providing client and server side security for service to service communications, enabling organizations to enhance their network security without incurring a significant amount of additional operational overhead.

Before moving on, it's important for us to understand how mTLS works. Since I am not a networking or security person by trade, the process has been somewhat dumbed down so that I can understand it.

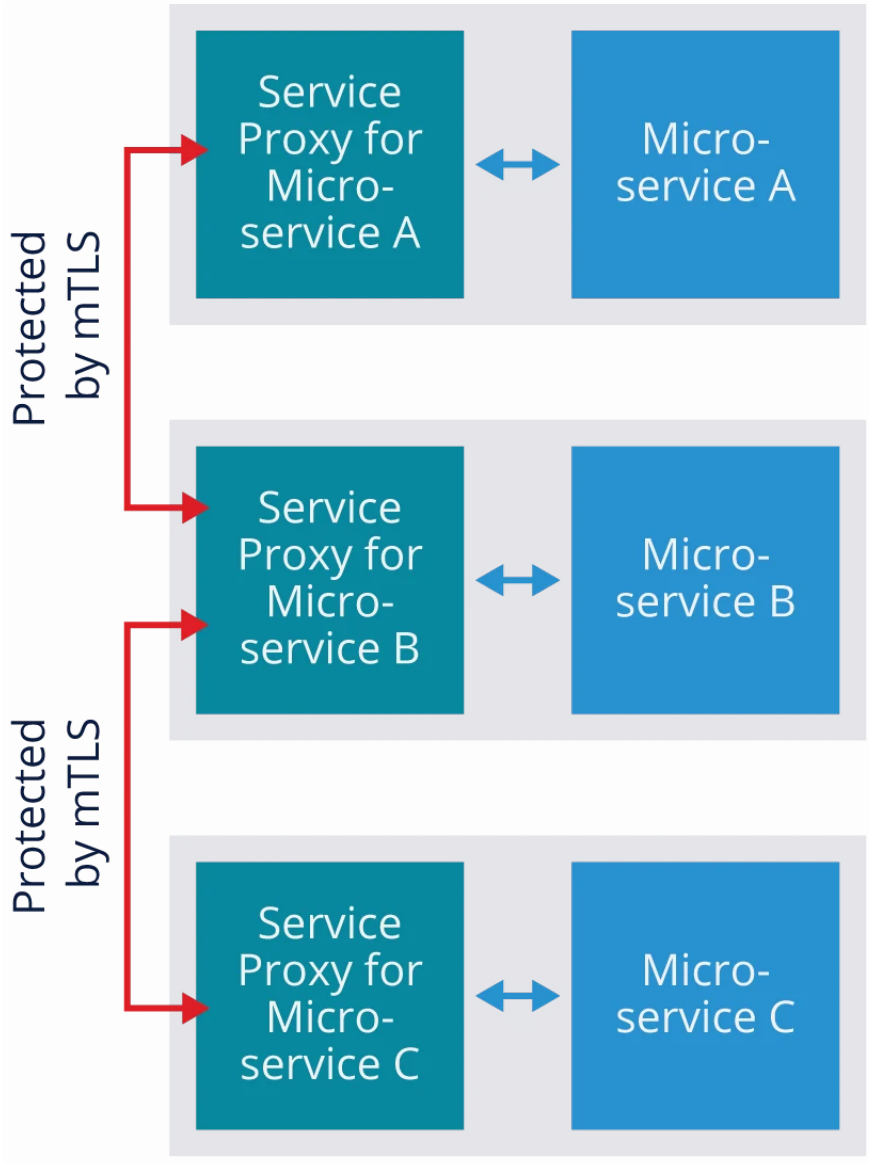
The "m" in the mTLS acronym stands for "mutual," meaning that both the client and the server sides verify each other's identities before proceeding on to the HTTP exchange of data.

From a certificate perspective, a root certificate authority (CA) is stored on the server side of the equation. Requests are only allowed from devices or services with a corresponding client certificate.

When a request reaches the service, the request responds with a new request for the client to present a certificate. If the device fails to present the certificate, the request is not allowed to proceed. If the client presents a certificate, the new request completes a key exchange to verify. Voila! Data is verified and securely passed. Best of all, the Developer didn't have to write a line of code for this to occur. No human intervention was required to complete the process. And yet, secured exchange of data took place.

Since the use of mTLS is well documented and understood, almost all service meshes work on the same principles to secure communications between microservices. Many service meshes offer a solid mTLS feature set, but they may differ in the use cases their particular implementations are designed to cover out of the box and the specific method in which the use of mTLS is deployed.

Once enabled in the Service Mesh, the feature looks like this.




**Istio Service Mesh Encryption**

Istio provides a feature-rich and mature methodology for use of mTLS throughout the service mesh that is implemented within the control plane, providing sophisticated and secure service-to-service communication. Istio's implementation of mTLS within the service mesh can be used at the application and service layer without the need for any code changes.

The Istio control plane (and all service mesh control planes for that matter) generally provides two specific capabilities: a certificate authority that handles certificate signing and management, and a configuration API server that distributes communication policies to the sidecar proxies, enabling them to become a Policy Enforcement Point (PEP). When two microservices need to communicate, the sidecars are responsible for establishing a secure proxy-to-proxy connection and encrypting the traffic going through it. The Istio mTLS implementation can be set to a very granular level, such as for a specific service, or across an entire namespace, or even over the entire service mesh.

Istio automatically configures workload sidecars to use mTLS when calling other workloads by default. Out of the box, Istio configures the destination workloads using `PERMISSIVE` mode. This setting allows a service to accept both plain text and mTLS oriented traffic.

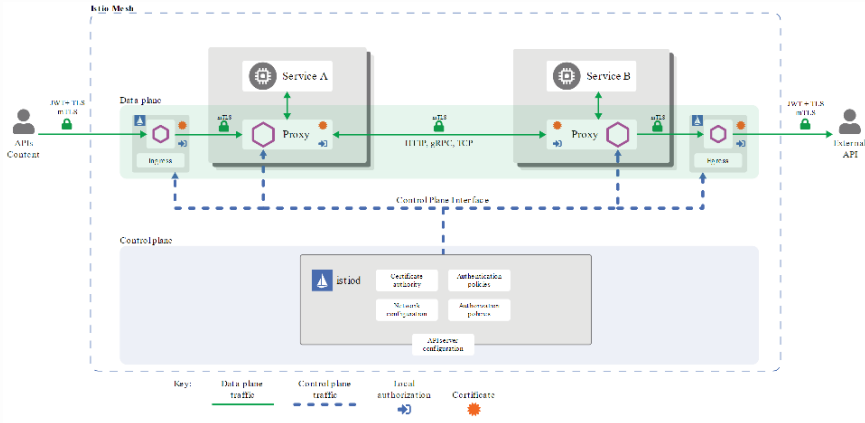
The ability to allow for both types of traffic is helpful when transitioning workloads from non-mTLS to mTLS secured traffic, as `PERMISSIVE` mode enables microservices that cannot use mTLS to be moved into the mesh and still communicate.

However, the `PERMISSIVE` setting weakens your security, in that it allows plain text to traverse between services. At some point, using the Istio Service Mesh to your best advantage, you will want to eliminate this potential, so you will first have to ensure that ALL services and microservices in your application are capable of authenticating and certifying, and that all service ports are "named" correctly within the service registry. Once these two factors are achieved, you can fairly safely change the mode setting from `PERMISSIVE` to `STRICT` to only allow mTLS traffic.

Microservices have particular security needs:

- Microservices need traffic encryption to defend against man-in-the-middle attacks.
- Microservices need mutual TLS and fine-grained access policies to provide flexible service access control.
- Microservices need auditing tools to determine who did what at what time.

Istio Security provides a comprehensive security solution to solve these issues. Istio security mitigates both insider and external threats against your data, endpoints, communication, and platform. Here is what the Istio mTLS Security model looks like.




Now let's see this in action.

## A Simple Example of Enabling Authentication and Authorization to an Istio Service Mesh

This example assumes that you have already installed a Kubernetes cluster. You will need to install the Istio Service Mesh in a default configuration to start, by issuing the following command:

```
istioctl install --set profile=default
```

You can then globally enable Mutual TLS (mTLS) in STRICT mode as a test by issuing the following commands:

```
kubectl create ns foo
kubectl apply -f <(istioctl kube-inject -f \
samples/httpbin/httpbin.yaml) -n foo
kubectl apply -f <(istioctl kube-inject -f \
samples/sleep/sleep.yaml) -n foo
kubectl create ns bar
kubectl apply -f <(istioctl kube-inject -f \
samples/httpbin/httpbin.yaml) -n bar
```

```
kubectl apply -f <(istioctl kube-inject -f \
samples/sleep/sleep.yaml) -n bar
kubectl create ns legacy
kubectl apply -f samples/httpbin/httpbin.yaml \
-n legacy
kubectl apply -f samples/sleep/sleep.yaml \
-n legacy
kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "default"
  namespace: "istio-system"
spec:
  mtls:
    mode: STRICT
EOF
```

Once you enable the use of mTLS as STRICT within Istio, you can prove that mTLS is being used throughout the mesh by issuing the commands to test it.

The command itself generates traffic between three namespaces: foo, bar and legacy in a loop. The command is just listing the pods in each of the three namespaces. Hopefully, you can just copy and paste the command from below as I did. Knowing me, I would have fat fingered something otherwise:

```
for from in "foo" "bar" "legacy"; do for to in
"foo" "bar" "legacy"; do kubectl exec "$(kubectl
get pod -l app=sleep -n ${from} -o
jsonpath={.items..metadata.name})" -c sleep -n
${from} -- curl "http://httpbin.${to}:8000/ip" -s
-o /dev/null -w "sleep.${from} to httpbin.${to}:
%{http_code}\n"; done; done
```

The output should look like this:

```
sleep.foo to httpbin.foo: 200
sleep.foo to httpbin.bar: 200
sleep.foo to httpbin.legacy: 200
sleep.bar to httpbin.foo: 200
sleep.bar to httpbin.bar: 200
sleep.bar to httpbin.legacy: 200
sleep.legacy to httpbin.foo: 000
command terminated with exit code 56
sleep.legacy to httpbin.bar: 000
command terminated with exit code 56
sleep.legacy to httpbin.legacy: 200
```

The response of `200` indicates that the command succeeded. The response of `56` indicates that the command failed to authenticate.

Now let's look at observability.

## Service Mesh Observability

Observability, at its core, provides the developer and operator with the ability to monitor an application's state in real time, and understand, at least at a high level, what the heck is happening when something goes wrong with the application. When people speak of observability, they usually focus on these two most common components: metrics and telemetry. In our situation, however, it's important to realize that measuring "performance" can be a little more complicated than you might expect.

Metrics is a generic term for any data captured about a subject. Telemetry refines the definition of metrics to include the concept of collecting that data in order to measure something about what is being observed.

Google, one of the BIG PLAYERS, looked at observability in terms of distributed computing systems and came up with four specific metrics that can be applied to all applications regardless of whether they are monolithic, virtualized, or containerized microservices:

- Latency: how long it takes to receive a reply for a request
- Traffic Flow: how much demand the system is under, such as requests per second
- Error Count: calculates what percentage of requests result in an error
- Saturation Point: presents how close the available resources, like processing or memory, are coming to being fully utilized

However, there is actually a completely different way of observing how well or badly an application is performing against the available resources that the application is hosted on, and it is one of the most accurate ways of gauging the performance of Process-oriented Operating Systems such as Linux and Unix against the physical hardware upon which it is being run.

This method determines areas of constraint and measures the effect of changes made to alleviate those constraints. The method relies on being able to capture and snapshot the state of all processes at a point of maximum utilization.

This data, once extracted, is constructed into a model of the overall system as a whole, which includes Active Resources, such as CPU and Disk, along with Passive Resources such as Memory. The time in a wait queue and the time in a work queue are calculated separately and totaled.

A specific mathematical algorithm known as Mean Value Approximation can then be applied using each of these numbers. The Mean Value Approximation (MVA) algorithm uses the mean value approximation formula to solve network queuing models.The Queuing algorithm is applied to each active and passive resource involved in providing service to each workload. A level of service can then be calculated and derived for each of the resources individually and for the system (or network) as a whole.

Queueing models enable a number of useful steady state performance measures to be determined, including:

- the average number in the queue, or the system,
- the average time spent in the queue, or the system,
- the statistical distribution of those numbers or times,
- the probability the queue is full, or empty, and

- the probability of finding the system in a particular state

These performance measures are important, as issues or problems caused by queueing situations are often related to customer dissatisfaction with a service. Computer systems, and for that matter all types of systems that receive requests and process them, have a response time "R" that includes some time waiting in queue if the server is busy when a request arrives. The wait time increases sharply as the server utilization (ρ) increases.

For queueing systems there is a simple equation that describes this exactly, but for more complicated systems, this equation is only approximate. This equation has become known as a Stretch Factor.

The Stretch Factor in a computing network is calculated as a ratio between the computing service time plus queue time versus the computing service time.

This method of calculation represents the best single figure of merit used to characterize system performance because it takes into account all factors related to the demand of each process running on the system and its precise resource consumption.

The lowest possible value for stretch factor is 1.0. When the stretch factor exceeds 2.0, there is a bottleneck somewhere in the system and, in all probability, some users will not be satisfied with the level of service.

Current Service Meshes usually support collecting all of these metrics and a great deal more out of the box, so you are in luck. Modern Service Meshes offer multiple ways to access the metrics they capture. Almost all of the popular Service Meshes, including Istio, LinkerD and Kong also provide visualization tools for viewing the metrics through graphical interfaces. Users can even export them through APIs for use in other tools.

## Istio Service Mesh Observability

Istio, for example, generates detailed telemetry for all service communications within a mesh automatically. The captured telemetry empowers operators to troubleshoot, maintain, and optimize their applications without placing any additional demands on the operations teams or individual microservice developers.

Istio generates the following types of telemetry in order to provide overall service mesh observability:

- Metrics: Istio generates a set of service metrics based on the Google definition of monitoring (latency, traffic flow, error count, and saturation point). Additionally, Istio provides detailed metrics for the Service Mesh Control Plane. All of these metrics can be displayed using the default Grafana Dashboards, which tap into a Prometheus database in which the metrics are captured. The Grafana templates and Prometheus database can be deployed during or after installation of the Service Mesh.
- Distributed Traces: Istio also generates distributed trace spans for each service. Tracing will be discussed in more detail in the next section.

- Access Logs: Istio can generate a full record of each request as traffic flows into and out of the services hosted in the mesh. The metrics captured in the Access Logs include source and destination metadata. We will also be discussing this feature more fully in the next section.

Istio metrics collection begins with the sidecar proxies (Envoy). Each proxy generates a large volume of metrics regarding all ingress and egress traffic being moved through the proxy. The proxies also provide detailed statistics regarding administrative functions being performed within the proxy. The metrics captured also include configuration and health information vital to determining the availability of the service through the proxy.

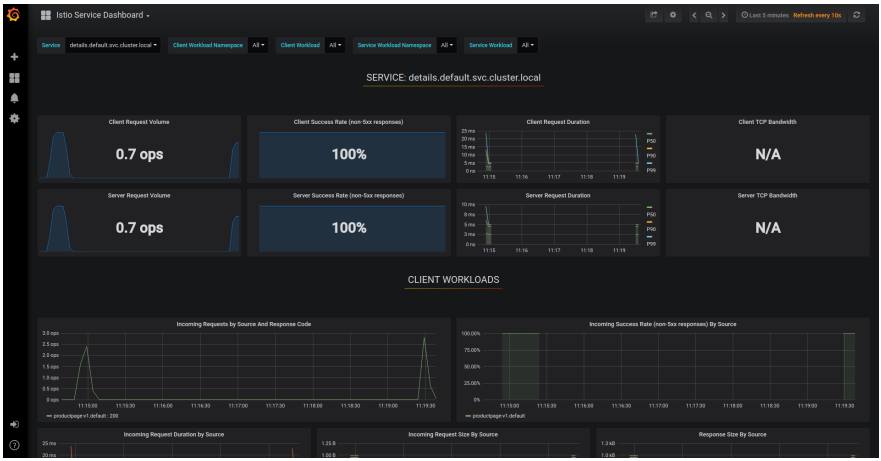
Istio enables operators to decide which of the Envoy metrics are generated and collected from each workload instance with a sidecar, but to reduce the CPU overhead associated with metrics collection and avoid producing too much data for the system to record, Istio only enables a small subset of the Envoy-generated statistics by default. Operators can easily expand the set of collected proxy metrics when required by changing the Prometheus configuration rules.

In addition to the proxy-level metrics, Istio provides a set of service level metrics to monitor the service-to-service communications generated across the mesh. The metrics cover the same four basic service monitoring needs as defined by Google, Latency, Traffic, Errors, and Saturation. The Istio installation comes with a default set of Grafana dashboards for monitoring service behaviors based on the four metrics. Use of the service-level metrics are entirely optional.

The standard Istio metrics are exported to Prometheus by default.

It is important to mention that the Istio control plane provides a collection of observability metrics as well. These metrics enable monitoring of Istio itself, separate from that of the services within the mesh.

Below is an example of the Istio Visualization tool, Grafana, in operation.




Let's take a look at the actual tools.

## Implementing Observability Tools for Telemetry and Visualization As a Stand-alone within an Istio Service Mesh

If you have not done this before, I recommend using Helm charts to deploy the tools, but for more information on installing manually, please check out https://www.mirantis.com/blog/more-service-mesh-info/. The tools most commonly used for this purpose are:

- Prometheus
- AlertManager
- Grafana

Prometheus, AlertManager and Grafana have become a standard for most Service Meshes. Many Service Meshes even include them in their distributions, but, if they aren't included in your deployment, the instructions below will help you to get them installed.

1. Download Helm itself:

```
curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/
master/scripts/get-helm-3
```

2. Make the script executable and run it:

```
chmod 700 get_helm.sh
./get_helm.sh
```

3. Execute the following commands to install Prometheus, AlertManager and Grafana. Start by adding the repository:

```
helm repo add prometheus-community \
  https://prometheus-community.github.i
o/helm-charts
helm repo add stable \
  https://charts.helm.sh/stable
helm repo update
```

4. Install the Prometheus stack:

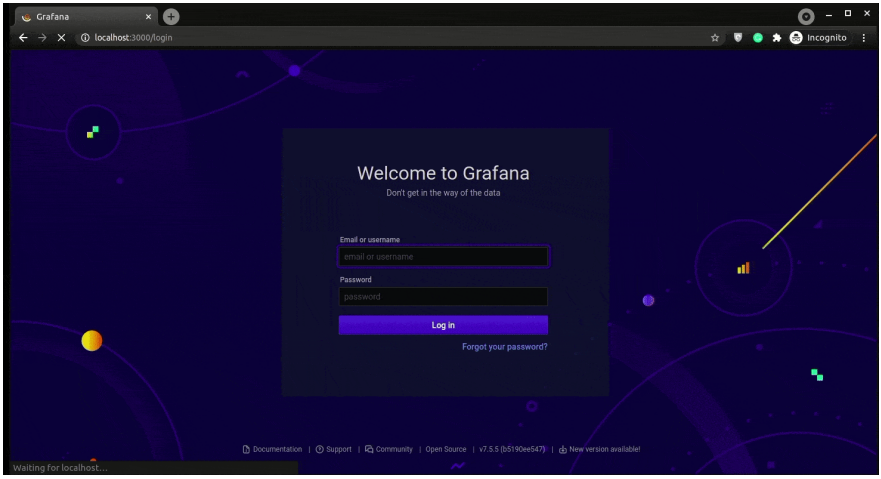
```
helm install prometheus \
  prometheus-community/kube-prometheus-stack
```

5. Forward requests to port 3000 so we can pull it up in the browser:

```
kubectl port-forward \
  deployment/prometheus-grafana 3000
```

6. Log in to Grafana using localhost:3000 in your favorite browser.

```
username: admin
password: prom-operator
```




The Grafana templates provided with Istio are focused on capturing and displaying the four Golden Metrics we discussed previously. This observability allows you to anticipate bottlenecks in performance and reliability.

Now let's look at traceability.

## Service Mesh Traceability

Distributed tracing, also called distributed request tracing, is a feature in most Service Meshes used to profile and monitor applications built using a microservices architecture. It helps pinpoint where failures occur and what causes poor performance.

Distributed tracing provides a way to monitor and understand behavior by monitoring individual requests as they flow through a mesh. Traces empower mesh operators to understand service dependencies and the sources of latency within their service mesh.

In distributed systems, tracing shows all the relationships across microservices hosted within the mesh. Visibility into those relationships gives the developer the observability to understand how their code is behaving in production.

Traceability, when combined with other observability components and Artificial Intelligence methodologies and types of automation, holds the promise of delivering the outline of steps needed to ensure cloud native applications work as expected, and deliver optimal performance and value to their consumers.

The Cloud Native Computing Foundation (CNCF) is working to develop standards for Distributed Tracing under the names OpenTracing and OpenCensus. OpenTracing is a vendor-agnostic API to help developers easily instrument tracing into their code base, while OpenCensus is an open source library that enables instrumentation of distributed microservice apps to collect traces and metrics across a wide variety of languages, platforms, and environments.

OpenCensus and OpenTracing have merged to form a new standard for both called OpenTelemetry. OpenTelemetry was released as the next major version of OpenCensus and OpenTracing, and is a collection of tools, APIs, and SDKs. You can use OpenTelemetry to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) for analysis in order to understand your microservice's performance characteristics and flow.

Distributed Tracing support at the service mesh level actually provides:

- Proxy timing and data in your traces
- Traces that extend to ingress points

- Distributed tracing functionality without needing to couple application code to a particular library

## Istio Service Mesh Traceability

Istio leverages Envoy's distributed tracing feature to provide tracing integration out of the box. Istio's distributed tracing through Envoy enables developers to obtain visualizations of call flows in large service oriented architectures, which can be invaluable in understanding and eliminating sources of latency. Envoy supports three features related to system wide tracing:

- Request ID generation
- Client trace ID joining
- External trace service integration

When using Envoy as a proxy, end-to-end traces consist of one or more spans. A span is a collection defining a logical unit of work that has a start time and duration. The span can contain metadata associated with the collection. Each span generated by Envoy contains the following data:

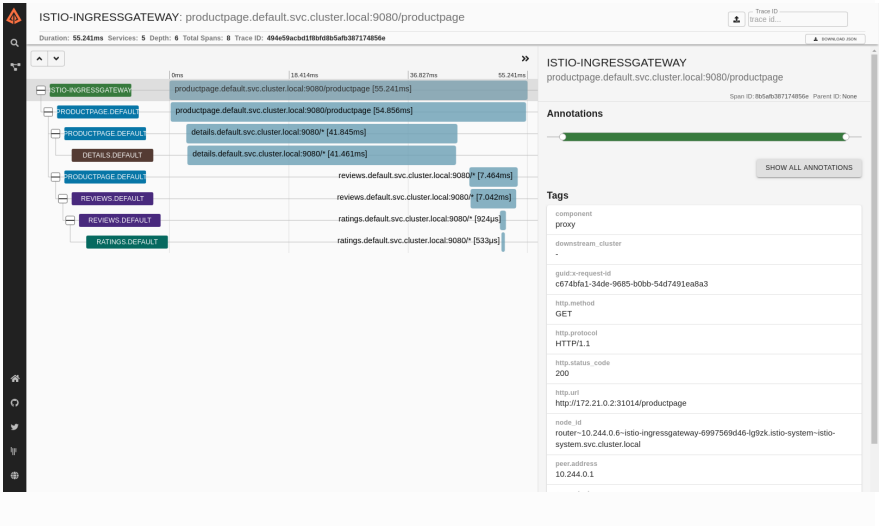
- Originating service cluster set
- Start time and duration of the request
- Originating host set
- Downstream cluster set
- HTTP request URL, method, protocol and user-agent
- Additional custom tags set
- Upstream cluster name, observability name, and address
- HTTP response status code
- GRPC response status and message (if available)

- An error tag when HTTP status is 5xx or GRPC status is not "OK"
- Tracing system-specific metadata.

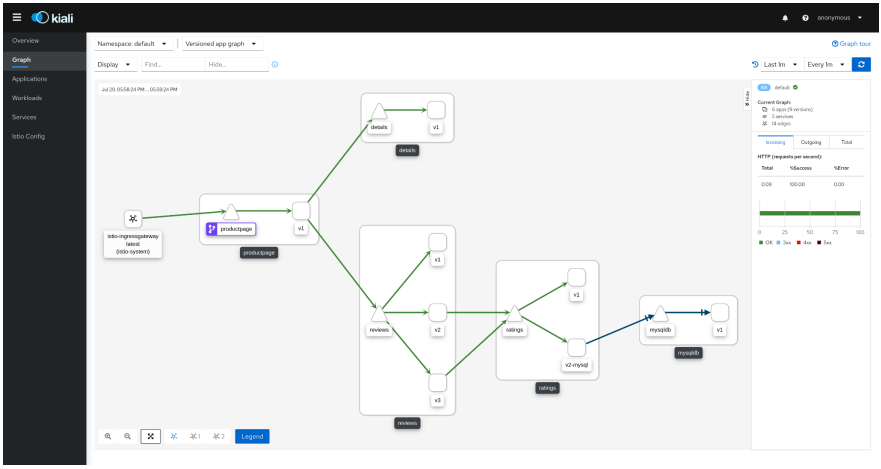
Envoy automatically sends spans to tracing collectors. Istio provides options to install various tracing backends and to configure proxies to send trace spans to them automatically. Four of the most commonly integrated backend tracing capture applications are Zipkin, Kiali, Jaeger and Lightstep task docs about how Istio works with those tracing systems.

Below are images of the four most popular backend visualization tools used with the Istio Service Mesh.

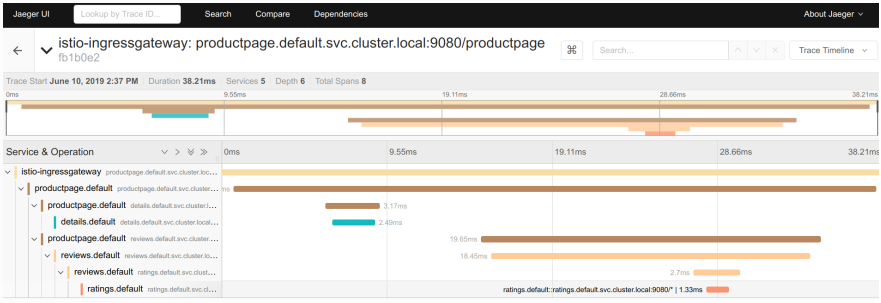
## Zipkin



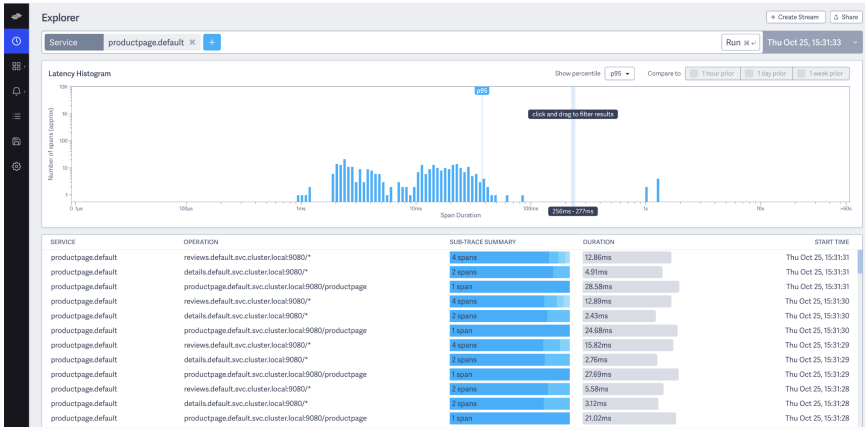
## Kiali




## Jaeger



## Lightstep




## Installing Service Mesh Traceability Components Separately

If you are doing this for the first time, I would not recommend you installing these components individually, but rather, using the built-in tools provided by the Service Mesh vendor to get to where you want to be more rapidly. However, some of the vendors, such as Istio, have removed some of the Traceability components most widely used from their distributions, which may leave you no choice but to pursue the manual deployment tasks I am about to describe.

There are several components to be deployed within your Kubernetes cluster to enable tracing. Generally, the most common components are listed below:

- Kiali

- Jaeger

We installed Kiali earlier, in the chapter "**Create a Sidecar Oriented Service Mesh with Istio**", but let's look at installing Jaeger.

## Hardware Requirements

For a production environment,, you should take the following configuration into consideration before deployment of these new components:

- For master nodes: 16GB RAM, 4vCPUs, 40GB of hard disk space.
- For worker nodes: 8GB RAM, 1vCPU, 16GB of hard disk space.

For development,, the requirements are lower, depending on how demanding your applications are, and how many services you're planning on running at the same time on your cluster.

## Prerequisites

After installing a Kubernetes cluster and a Service Mesh like Istio, Install Helm 3 using the following commands.

```
curl -fsSL -o get_helm.sh \
  https://raw.githubusercontent.com/helm/helm/
  master/scripts/get-helm-3
chmod 700 get_helm.sh
./get_helm.sh
```

Install an Ingress-Controller to support the Jaeger implementation. Using Helm 3 to provide an nginx ingress-controller:

```
helm repo add ingress-nginx \
  https://kubernetes.github.io/ingress-nginx
helm repo update
helm install ingress-nginx \
  ingress-nginx/ingress-nginx
```

You can verify the creation of the ingress-controller using by issuing the following command:

```
kubectl get pods -n ingress-nginx -l \
   app.kubernetes.io/name=ingress-nginx --watch
```

Now let's install Kiali.

## Kiali Installation

Now you can install Kiali using the following helm command:

```
$ helm install \
    --namespace istio-system \
    --set auth.strategy="anonymous" \
    --repo https://kiali.org/helm-charts \
    kiali-server \
    kiali-server
```

If you deployed an Istio Service Mesh, you can now use the following command to access the Kiali interface:

```
$ istioctl dashboard kiali
```

You can use the Kiali interface to observe the BookInfo application to show the interaction between Service Subsets, Destination Rules and Circuit Breaker Patterns within the Service Mesh. The v1, v2 and v3 versions of the "reviews" portion of the application depicted in the graphic presentation of the Kiali interface are defined in the application as a Service Subset. The Destination Rule defined against the Service Subset sends traffic to only the v2 and v3 version as the v1 version has a Circuit Breaker Pattern telling the traffic not to take that path as the v1 version is deprecated.

## Jaeger Installation

You can install the Jaeger Operators using the following commands:

```
$ kubectl create namespace observability
$ kubectl create -n observability -f \
https://raw.githubusercontent.com/jaegertracing/j
aeger-operator/master/deploy/crds/jaegertracing.i
o_jaegers_crd.yaml
$ kubectl create -n observability -f \
https://raw.githubusercontent.com/jaegertracing/j
aeger-operator/master/deploy/service_account.yaml
$ kubectl create -n observability -f \
https://raw.githubusercontent.com/jaegertracing/j
aeger-operator/master/deploy/role.yaml
$ kubectl create -n observability -f \
https://raw.githubusercontent.com/jaegertracing/j
aeger-operator/master/deploy/role_binding.yaml
$ kubectl create -n observability -f \
https://raw.githubusercontent.com/jaegertracing/j
aeger-operator/master/deploy/operator.yaml
```

```
$ kubectl create -f \
https://raw.githubusercontent.com/jaegertracing/j
aeger-operator/master/deploy/cluster_role.yaml
$ kubectl create -f \
https://raw.githubusercontent.com/jaegertracing/j
aeger-operator/master/deploy/cluster_role_binding
.yaml
```

Once the jaeger-operator deployment in the namespace observability is ready, create a Jaeger instance, like this:

```
$ kubectl apply -n observability -f - <<EOF
apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: simplest
EOF
```

The Jaeger data is accessible through the Kiali interface display.

## Service Mesh Authentication

A service mesh provides the ability to do authentication between the application developer's services to ensure traffic flowing in the cluster in which the application is hosted is secure. There are 4 different authentication options available with a sidecar service mesh:

- JWT Validation in Applications
- JWT Validation at Ingress
- Istio Ingress TLS passthrough + JWT Validation at Sidecars
- Istio mTLS + JWT Validation

A JSON web token (JWT), or "jot" for short, is a standardized container format that can be validated and/or encrypted when used to securely transfer information between two parties. JWT, is an open, industry standard RFC 7519 method for representing claims, and defines the structure of the information in the container format that gets sent across the network. The container format comes in two forms, serialized and deserialized.

The serialized form is used to transfer data through the network with each request and response. The deserialized form of the token consists of a header and a payload. They are both plain JSON objects. Thus the name JSON web token.

The JWT header is used to describe the cryptographic operations applied to the token. The information contained in the header defines whether the JWT is signed and additionally encrypted. If the header indicates encryption is to be used, it will also include what cryptographic algorithms are used to perform the encryption.

Workflows that involve an exchange of data initiated by one or more users through a network can use JWT as a vehicle to provide a compact and secured container that delivers each consumer's authentication data with each transfer.

This methodology was originally only applied to server-side sessions where a user's data is stored on a server using various storage methods. Since server-side sessions were very difficult to scale and forced complex scenarios to replicate data across all of the web servers supporting an application, smart developers came up with a better method where the data needed was stored on the client side of the equation.

This model, quite naturally, became known as client-side sessions. A client-side's session data is stored on the web client and sent to a server with each request.

Though the client-side sessions method solved the scalability and replication issues of server-side sessions, it left the potential for tricky users to try to alter the data in the cookies holding identity and other personal data on the client-side in order to try to cheat.

The JWT package was created to solve this problem by placing the data in an unhackable form. Since the JWT package can be signed, the server can verify the authenticity of the session data with every transfer and trust that it hasn't been hacked.

The JWT package can be encrypted too. This obscures the contents of the JWT and stops users from reading or modifying it.

When using JWT with signing and encryption together, the developer can ensure secure transmission of information. The two standards that are included as part of the JWT standard that describe these security features are JSON Web Signature (JWS) and JSON Web Encryption (JWE). (Yes, I know... more acronyms for us to learn.)

JWS actually allows for the use of two different standards:

- HMAC
- RSASSA

HMAC is otherwise known as the shared-secret scheme for verifying the signature of the package and the data that follows. RSASSA uses a public-private key scheme to verify the signature and data. This method allows for things like Single-Sign On to work.

JWT encryption, known as JSON Web Encryption (JWE), goes over and above the JWS method though it uses the same two signing verification methods. However, in the case of public-private key verification, JWE works differently than JWS because it can't guarantee the veracity of the package as it doesn't own the public-private key to do so. So, to have a truly secured transport, you need to employ both JWS and JWE together.

As you can see, this is a truly complex process that takes a great deal of coding to deal with, unless, of course, you are using a Service Mesh to connect all of your microservices in a cluster. In that case, you get these features and capabilities as part of the mesh.

Makes things easier, doesn't it?

## Istio Service Mesh Authentication

As you'll remember from the previous sections of this eBook, Istio Service Mesh authentication policies apply to requests that a service receives. To specify client-side authentication rules in mutual TLS, you need to specify the use of TLS Settings in the Destination Rules configuration. Like other Istio configurations, you can specify authentication policies in yaml files. You deploy policies using kubectl. Istio stores mesh-scope policies in the root namespace. Policies that have a namespace scope are stored in the corresponding namespace. If the developer configured a selector field in the yaml file, the authentication policy only applies to workloads matching the conditions the developer configured.

The selector fields in the authentication.yaml file help you specify the scope of the policies to be enforced:

- Mesh-wide policy
- Namespace-wide policy
- Workload-specific policy

Mesh-wide policies are specified for the root namespace without or with an empty selector field. Namespace-wide policies are specified for a non-root namespace without or with an empty selector field. Workload-specific policies are defined in the regular namespace, with a non-empty selector field.

Below is an example of two sections of an authentication.yaml file from the BookInfo application distributed with the Istio Service Mesh for reference:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
```

```
  name: "example-peer-policy"
  namespace: "foo"
spec:
  selector:
    matchLabels:
      app: reviews
  mtls:
    mode: STRICT
```

Notice the STRICT setting that requires that ALL traffic to and from the service be encrypted using mTLS. Notice, too, that the selector is app specific, meaning that it applies the specific workload identified by "reviews".

This section specifies that a JSON Web Token (JWT) is required to access the service through the Gateway.

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: "jwt-example"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  jwtRules:
  - issuer: "testing@secure.istio.io"
    jwksUri:
"https://raw.githubusercontent.com/istio/istio/re
lease-1.11/security/tools/jwt/samples/jwks.json"
```

Request authentication policies specify the values needed to validate a JWT. These values include the following:

- The location of the token in the request

- The issuer or the request
- The public JSON Web Key Set (JWKS)

Out-of-the-Box, Istio checks the presented token, if presented against the rules in the request authentication policy, and rejects requests with invalid tokens. But, when requests carry no token, they are accepted by default unless otherwise specified in the authentication policy.

## Implementing Authentication Without Authorization

First of all, let's start with the clarification that Authentication and Authorization are really two different things. Authentication is the process of verifying the identity of an individual. Authorization is a process with which we can allow or restrict resources from being accessed by an individual.

It is possible to implement Authentication without implementing Authorization, but this may not provide sufficient granular security to satisfy the needs of your microservice-based applications.

The example found at the URL at the end of this paragraph, is provided to show how the developer could create a Node.js oriented application and take advantage of OpenID to implement an Authentication-only scheme. The developer could use Passport.js, a popular OpenID library, for such a purpose. The tutorial providing all of the steps to implement OpenID in an Angular/Node.js application called "angular_dashboard" can be found at the following URL: https://blog.jscrambler.com/setting-up-authentication-using-angular-node-and-passport/

Though this implementation protects the application from unverified access, it doesn't protect the pages of the dashboard or specific attributes displayed within the dashboard from being accessed and displayed by a user who shouldn't be allowed to see them for some reason. You would also need to implement the OAuth standard parts of the library to meet this security need.

## Service Mesh Authorization

A service mesh provides the ability to enforce service-to-service and end-user-to-service authorization for all of the microservices and application resources hosted in the mesh. Authorization implemented within the mesh can provide the ability to secure your services and enforce the principle of least privilege. This is the only avenue to meeting the needs of a Zero Trust environment.

There are two authorization types that can be implemented with a service mesh:

- Role Based Access Control (RBAC)
- Attribute Based Access Control (ABAC)

When you design an Authorization scheme for a Service Mesh, you should take into account the following four factors:

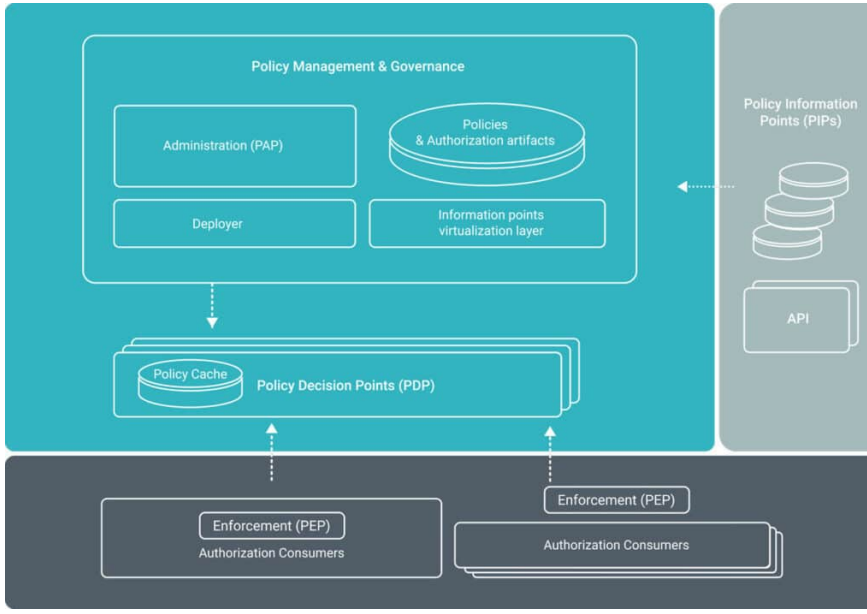
- Enforcement: should be embedded within the application/service based on the assets and resources the application/services are using
- Decisions: should be focused on who can access what, when, ideally with access rights that can be set and reset dynamically based on multiple factors so that they can be as accurate as possible at all times
- Management: Management of access policies should be maintained in a consistent way across all applications and services. Policies should be up to date at all times and should be expressed in a language that meets the needs of business. The access policies should also address who can manage the policies, how version control is maintained, and how deployment cycles are maintained as part of the policies

- Governance: establishes accountability for the creation, maintenance and update of the access policies for the organization

Service Meshes can assist in defining each of these components in reference to their organization in a very specific way. The Service Meshes Authorization mechanisms can be broken apart and managed by the accountable organizations without having to alter existing methods to achieve. The categories can be addressed as follows:

- PAP - policy administration point: The interface that is used to define and manage the access control policies that are then stored in the Policy Store
- PDP - policy decision point: The decision point is responsible for checking whether access is granted or blocked, based on the requested information
- PIPs - policy information point: The information points are the source of attributes of both identities and assets
- PEP - policy enforcement point: The enforcement point typically sits between the service you want to protect and the client application/service.

Put them all together, and they look like this.




This architecture keeps everything nice and separated to allow the responsible organization to administer as needed without impacting the other policy points in the diagram.

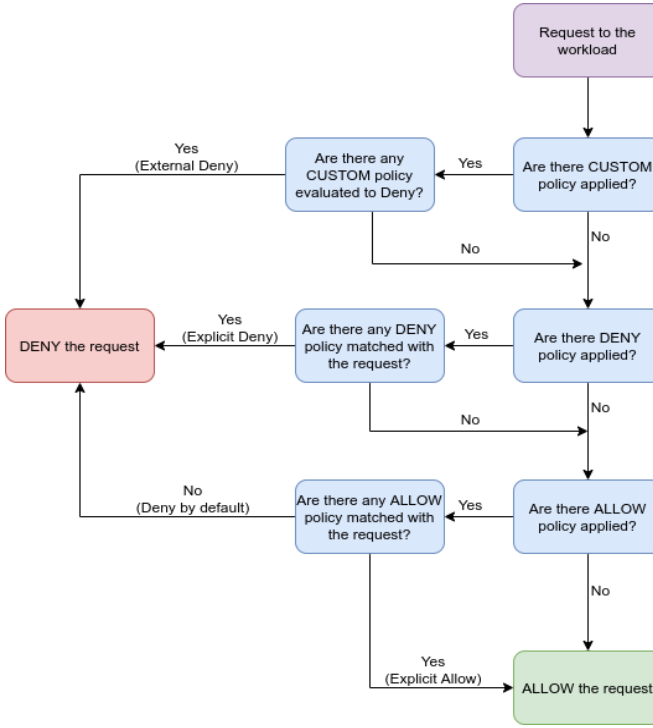
## Istio Service Mesh Authorization

Istio's authorization features provide mesh-wide, namespace-wide, and workload-wide access control for workloads hosted in the mesh. The authorization policies, created as yaml files and deployed via the Kubernetes `kubectl` command, enforce access control to the inbound traffic in the server side Envoy proxy.

At a granular level, each Envoy proxy runs an authorization engine that authorizes requests as they are received. The Envoy authorization engine evaluates each request context against the current authorization policies and returns the result of the evaluation as either ALLOW or DENY. If a workload has no authorization policies applied, by default, Istio will allow all requests. The following authorization example YAML file shows an authorization policy that denies requests if the source is not the `foo` namespace:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: httpbin-deny
 namespace: foo
spec:
 selector:
   matchLabels:
     app: httpbin
     version: v1
 action: DENY
 rules:
 - from:
   - source:
       notNamespaces: ["foo"]
```

The diagram below shows the sequence in which the authorization policies are evaluated:


Request to the workload
Are there CUSTOM policy applied?
Yes
Are there any CUSTOM policy evaluated to Deny?
Yes
(External Deny)
No
Are there DENY policy applied?
Yes
Are there any DENY policy matched with the request?
Yes
(Explicit Deny)
DENY the request
No
Are there ALLOW policy applied?
Yes
Are there any ALLOW policy matched with the request?
No
(Deny by default)
No
Yes
(Explicit Allow)
ALLOW the request

## Service Mesh Circuit Breaker Patterning

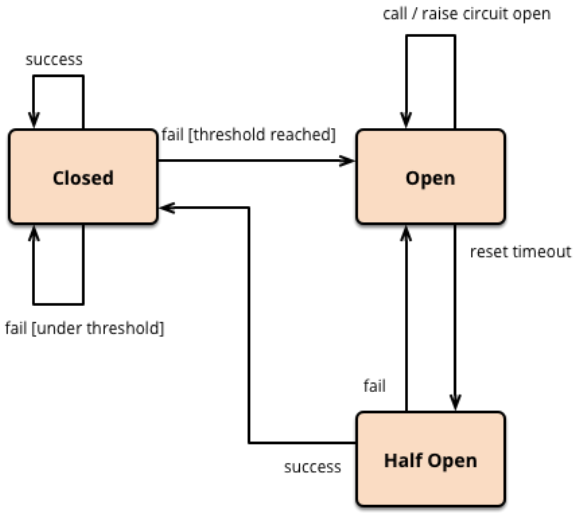
In a Microservice-driven architecture, when one or more instances of the same or different services are unavailable or are still stumbling along but with high latency or intermittent failures, it can result in a cascading failure across the entire enterprise. Standard application retry logic can only make things worse for the service.

This is where Circuit Breaker Patterning comes in. The circuit breaker concept is relatively straightforward. The Circuit Breaker wraps a function with a monitor that tracks failures and suboptimal performance. The circuit breaker has 3 distinct states: Closed, Open, and Half-Open.

In the Closed state, everything is normal. The circuit breaker remains in the closed state and all calls pass through to the services. The Circuit Breaker switches to an Open state when the number of failures exceeds a predefined threshold. Instead of retrying the function, the circuit breaker returns an error for calls without attempting to execute. After a predefined timeout period, the circuit switches to a half-open state to test if the underlying problem still exists.

If the function succeeds after the timeout period, the circuit breaker resets to the normal, closed state. If a single call fails in this half-open state, the breaker is once again tripped. In this way, the Circuit Breaker proxy can prevent a single failure from cascading through an entire application.

The Circuit Breaker Pattern model looks like this:




## An Example of Using Circuit Breaker Patterning in an Istio Service Mesh

Istio adds fault tolerance to Microservice applications hosted in the service mesh without any changes to the code through use of the Envoy proxy. The features include:

- Retries and Timeouts.
- Circuit breakers.
- Health checks.
- Outlier Detection.
- Fault injection.

Envoy provides a set of features out-of-the-box to handle failure scenarios using the following parameters as guides:

- Maximum Connections
- Maximum Pending Requests
- Maximum Requests

We can see how this plays out in the creation of DestinationRules to take advantage of this pattern, such as:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: serviceC
spec:
  host: serviceC
  subsets:
    - name: serviceC-v1
      labels:
        version: v1
    - name: serviceC-v2
      labels:
        version: v2
      trafficPolicy:
        connectionPool:
          http:
            http1MaxPendingRequests: 10
            maxRequestsPerConnection: 1
          tcp:
            maxConnections: 1
        outlierDetection:
            baseEjectionTime: 20s
            consecutiveErrors: 1
            interval: 10s
            maxEjectionPercent: 100
```

Istio Circuit Breaker Patterns are specified in the Destination Rules definition under the Traffic Policy section. The example provided above for reference shows how the maximum connections, requests per connection and pending requests counts can be used to control the flow of traffic from within the Destination Rule.

It is important to understand the behavior of Circuit Breaker Patterning within Istio before attempting to take full advantage of the feature in a Production setting. Vikas Kumar, in his blog post, "Demystifying Istio Circuit Breaking" observed the following: The developer can set the limit for circuit breaker using tcp.maxConnections or http2MaxRequests along with http1MaxPendingRequests to establish Closed and Open behavior.

On the client-side, each client proxy applies the limit independently. If the limit is 100, then each client proxy can have 100 outstanding requests before they apply local throttling. The limit on the client proxy is for the destination service overall, not for individual replicas of the destination service.

On the destination side, each destination service proxy applies the limit separately. If there are 50 active pods of the service, each pod can have a maximum of 100 outstanding requests from client proxies before switching to an Open state and returning a 503 error.

## Istio Service Mesh Circuit Breaker Patterning

Istio Service Mesh Circuit Breaker Patterning allows you to write applications that limit the impact of failures, latency spikes, and other undesirable effects of network peculiarities. In this example, we will configure circuit breaking rules and then test the configuration by intentionally "tripping" the circuit breaker. We will use examples provided in the Istio distribution to do this.

First, you will need to install Istio on your Kubernetes cluster in its default configuration. Next, we will install the httpbin sample application using the following command:

```
kubectl apply -f samples/httpbin/httpbin.yaml
```

Next, we will create the Destination Rule in which the Circuit Breaker Pattern has been defined:

```
kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: httpbin
spec:
  host: httpbin
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
    outlierDetection:
      consecutive5xxErrors: 1
      interval: 1s
      baseEjectionTime: 3m
      maxEjectionPercent: 100
EOF
```

Now we need a client to drive the httpbin application in order to "trip" the Circuit Breaker. We will use the fortio example client application supplied with the Istio Service Mesh distribution to fill this role by issuing the following commands:

```
kubectl apply -f \
 samples/httpbin/sample-client/fortio-deploy.yaml
export FORTIO_POD=$(kubectl get pods \
 -lapp=fortio -o
```

```
 'jsonpath={.items[0].metadata.name}')
kubectl exec "$FORTIO_POD" -c fortio \
 -- /usr/bin/fortio curl -quiet
 http://httpbin:8000/get
 HTTP/1.1 200 OK
…
```

The output of the last command will verify that the request succeeded. Now, we will "trip" the Circuit Breaker by issuing the following command:

```
kubectl exec "$FORTIO_POD" -c fortio -- \
 /usr/bin/fortio load -c 3 -qps 0 -n 30 -loglevel
Warning http://httpbin:8000/get
20:32:30 I logger.go:97> Log level is now 3
Warning (was 2 Info)
Fortio 1.3.1 running at 0 queries per second,
6->6 procs, for 30 calls: http://httpbin:8000/get
Starting at max qps with 3 thread(s) [gomax 6]
for exactly 30 calls (10 per thread + 0)
20:32:30 W http_client.go:679> Parsed non ok code
503 (HTTP/1.1 503)
20:32:30 W http_client.go:679> Parsed non ok code
503 (HTTP/1.1 503
…
```

The output from this command shows that there are 100% failures of the application being cut off from failed execution by the Circuit Breaker. In this example, the Circuit Breaker will remain in the Open position until the number of requests is reduced from 3 to 2.

# Conclusion

I hope this book has succeeded in providing a deeper understanding of the available Service Meshes, their architectures, and their features and benefits. I have personally been convinced that there are clear trends in the industry toward specific types of Service Meshes.

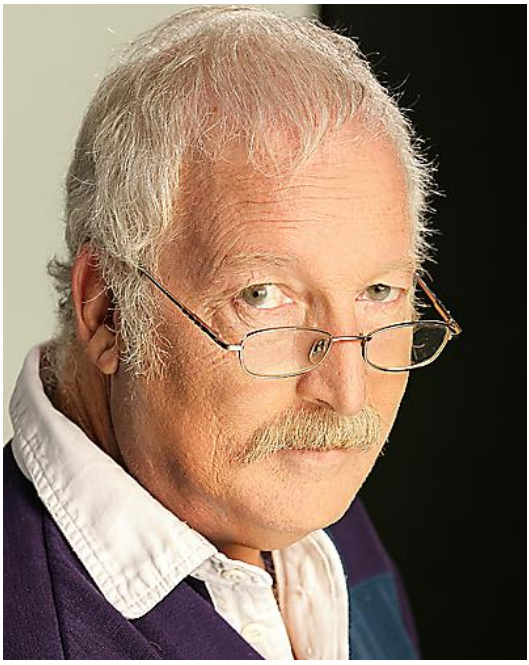
I hope that you, as the reader, will agree with the deliberate choice to focus more on Service Meshes employing a Sidecar Architecture, Istio being the most popular one of the Sidecar oriented Service Mesh today. If so, you were able to follow along with my thinking throughout the book. I would greatly appreciate any feedback you may have with this regard as this book represents my first effort of this magnitude.

To recap the major points, orchestrators such as Kubernetes and Mesos need a Service Mesh to fill in some of the major network security, observability, traceability, and platform oriented services that are not addressed within their frameworks. Service Meshes seem to fill these needs well. The prominent architecture for Service Meshes today are based on Sidecar technology, and Istio is the most prominent player in that Service Mesh architecture today. But keep watching this space as things change on a dime!

If you are already using an orchestration engine like Kubernetes, it is pretty easy to give Istio a shot to see if it will truly provide the benefits I have outlined!

Let me know! My email address is: bard_in_love@yahoo.com.

## About the Author



Bruce Basil Mathews is an open source expert specializing in OpenStack, Kubernetes and service mesh. Having worked in the computer industry for more than 40 years, including at Mirantis and HP, he officially retired from his IT career in 2021. Currently, he is pursuing his true passion as a professional stage, screen and voiceover actor. He has been a member of SAG-AFTRA/AEA since 2000.

Visit his website at: http://brucebasilmathews.com

Also, don't forget to look for Bruce's next book listed soon on Amazon!