



Hanna Prinz  
Eberhard Wolff

# Service Mesh

The New Infrastructure for  
Microservices

INOQ

# **Service Mesh**

**The New Infrastructure for Microservices**

**Hanna Prinz  
Eberhard Wolff**

---

ISBN 978-3-9821126-1-9

innoQ Deutschland GmbH

Krischerstraße 100 · 40789 Monheim am Rhein · Germany

Phone +49 2173 33660 · WWW.INNOQ.COM

Layout: Tammo van Lessen with X<sub>3</sub>LaTeX

Design: Sonja Scheungrab

Print: Pinsker Druck und Medien GmbH, Mainburg, Germany

**Service Mesh – The New Infrastructure for Microservices**

Published by innoQ Deutschland GmbH

1st Edition · August 2019

Copyright © 2019 Hanna Prinz, Eberhard Wolff

This work is licensed under a Creative Commons Attribution 4.0

International License.

This book is available at <http://leanpub.com/service-mesh-primer>.

# Contents

<b>1</b>	<b>Intro</b>	<b>1</b>
<b>2</b>	<b>What is a Service Mesh?</b>	<b>3</b>
2.1	Architecture.....	3
2.2	Service Mesh Interface.....	3
<b>3</b>	<b>Why Service Meshes?</b>	<b>5</b>
3.1	Monitoring.....	5
3.2	Logging.....	7
3.3	Resilience.....	7
3.3.1	Circuit Breaker.....	8
3.3.2	Retry.....	8
3.3.3	Timeout.....	8
3.4	Routing.....	8
3.5	Security.....	9
3.6	Performance Impact.....	10
3.7	Legacy.....	11
3.8	Alternatives to Service Meshes.....	11
3.9	Conclusion.....	12
<b>4</b>	<b>Example with Istio</b>	<b>13</b>
4.1	Istio.....	13
4.2	Overview.....	14
4.3	How the Example Uses Istio.....	16
4.4	Monitoring with Prometheus and Grafana.....	17
4.5	Tracing.....	20
4.6	Visualization with Kiali.....	22
4.7	Logging.....	23
4.8	Resilience.....	25
4.9	How to Dig Deeper.....	29
4.10	Conclusion.....	30
<b>5</b>	<b>Other Service Meshes</b>	<b>33</b>
5.1	Linkerd 2.....	33
5.2	Consul.....	34
5.3	AWS App Mesh.....	35
5.4	When to Choose Which?.....	35
<b>6</b>	<b>Conclusion</b>	<b>37</b>



# 1 Intro

Microservices are still the most hyped approach to software architecture. That has led to a huge number of new technologies to support microservices. Docker<sup>1</sup> for instance is a great tool to package microservices in self-contained images and to run many microservices on a single machine, but with separate file systems and network interfaces. Kubernetes<sup>2</sup> runs Docker containers in a cluster and enables load balancing as well as fail over. It also adds features like service discovery and routing. Nowadays, Kubernetes has become a very important infrastructure technology and has use cases far beyond microservices.

Thus, microservices lead to a lot of innovation for infrastructures. Service meshes are the latest infrastructure development to support microservices even better. This primer gives an overview of service meshes – it describes not just the concepts, but also explains why service meshes are useful and which implementations exist. It even includes a hands-on example for trying out a service mesh.

If you are interested in the concepts of microservices, there is also the free *Microservices Primer*<sup>3</sup> and the *Microservices Book*<sup>4</sup>. The free *Microservices Recipes*<sup>5</sup> booklet talks about some other technologies for implementing microservices as does the *Practical Microservices*<sup>6</sup> book.

And now enjoy this booklet!

---

<sup>1</sup><https://www.docker.com>

<sup>2</sup><https://kubernetes.io/>

<sup>3</sup><https://microservices-book.com/primer.html>

<sup>4</sup><https://microservices-book.com/primer.html>

<sup>5</sup><https://practical-microservices.com/recipes.html>

<sup>6</sup><https://practical-microservices.com/>



# 2 What is a Service Mesh?

A service mesh is a dedicated infrastructure component that facilitates observing, controlling, and securing communication between services. Unlike earlier approaches such as Enterprise Service Buses (ESBs) or API Gateways, a service mesh embraces the distributed nature of modern microservice applications and only focuses on networking rather than business concerns.

## 2.1 Architecture

A service mesh is composed of two layers, the *data plane* and the *control plane*. The *data plane* consists of a number of service proxies, each deployed alongside every microservice instance. This is called the *sidecar pattern*: Capabilities that every service needs are extracted into an additional container (the sidecar) that is placed next to every service instance. Up to now, typical use cases for a sidecar have been basic monitoring and encryption of network connections. Therefore, a service proxy was deployed as a sidecar which intersects all network traffic. The configuration of these service proxy sidecar containers and any update to it had to be performed manually. In a service mesh, the service proxies, that make up the data plane, are configured automatically by the second layer of a service mesh: the *control plane*. Any change to the behavior of the service mesh configured by the developer is applied to the control plane and automatically distributed to the service proxies. As shown in figure 2.1, the *control plane* also processes telemetry data that is collected by the service proxies.

This architecture adds powerful features like monitoring, circuit breaking, canary releasing, and automatic mTLS (mutual TLS authentication) to a microservice application without the need to change a single line of application code.

## 2.2 Service Mesh Interface

The attention - or rather hype - has resulted in a multitude of service mesh implementations, each introducing different concepts and APIs. Meanwhile, a service mesh also turned out to be a suitable base for more advanced tools like

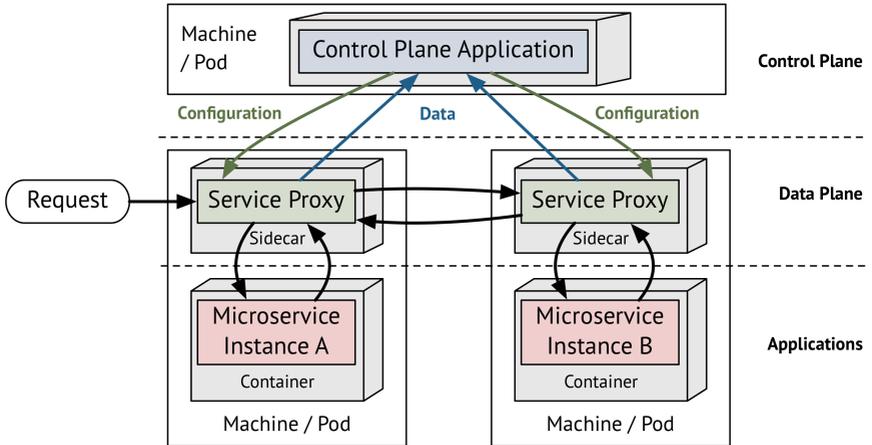


Figure 2.1 - Service Mesh Architecture

- Flagger<sup>1</sup>, for automating canary releasing,
- Squash<sup>2</sup>, a debugging tool for running microservices, and
- Kiali<sup>3</sup>, a service mesh dashboard.

The diverse landscape of service mesh implementations forced tool developers and users to bind their software to a specific service mesh. This led Microsoft, HashiCorp, Buoyant, and Solo.io to create Service Mesh Interface<sup>4</sup> (SMI), an API specification for service mesh features. Users and tools binding to SMI are able to use service mesh features independently of the implementation. Although SMI is still young, adapters implementing a part of SMI already exist for the three major service meshes Linkerd 2, Istio, and Consul.

<sup>1</sup><https://flagger.app>

<sup>2</sup><https://squash.solo.io>

<sup>3</sup><https://www.kiali.io>

<sup>4</sup><https://smi-spec.io>

# 3 Why Service Meshes?

By adding proxies to the data plane, service meshes make it possible to solve some basic problems for microservices infrastructures. This chapter explains the features services meshes provide in more details.

## 3.1 Monitoring

A microservices system should include a monitoring infrastructure that collects information from all microservices and makes them accessible. This is required to keep track of the metrics for the huge number of distributed microservices. You can implement alarms and further analysis based on these metrics.

The proxies in the data plane can measure basic information about the network traffic such as latencies or throughput. However, it is also possible to get more information from the network traffic. The service mesh needs to understand the protocol and interpret it. For example, HTTP has some defined status codes that enable the service mesh to determine whether a request was successful or not. That way the service mesh can also measure error rates and the like.

Of course, the service mesh cannot look into the microservices. So internal metrics about thread pools, database connection pools, and the like are beyond what the service mesh can provide. However, it is possible for the service mesh to use metrics that a platform like Kubernetes provides and measure these, too.

Relying on a service mesh for monitoring has some benefits:

- It has no impact on the code of the microservice at all. Metrics are measured only by the proxy, so any microservice will report the same metrics – no matter in what programming language it is written or which framework it uses.
- The metrics give a good impression about the state of the microservice. The metrics show the performance and reliability that a user or client would see. This is enough to ensure that service level agreements and quality of service are met.

So metrics provided by a service mesh might be enough to manage a microservices system. However, for a root cause analysis additional metrics from inside the microservices might be useful. In that case, the service mesh is still helpful because it provides a standardized monitoring environment that the service mesh already uses for its own metrics.

## Tracing

Tracing solves a common problem in microservices systems. A request to a microservice might result in other requests. Tracing helps to understand these dependencies, thus facilitating root cause analysis.

The proxies are able to intercept each request. But to do tracing, it is important to figure out which incoming request caused which outgoing requests. This can only be done inside the microservices. Usually, each request has some meta data such as a unique ID e.g. as part of HTTP headers, and that information is then transferred to each outgoing request. The data for each request is then stored in a centralized infrastructure.

A unique ID is not just valuable for tracing, but it is also useful to mark log entries that belong to one original request across all microservices.

The code of the microservices has to transfer the IDs of each request. So the service mesh cannot add tracing transparently to microservices. This compromises an important benefit of service meshes.

In addition, tracing is only useful for very specific cases. Most errors can be analyzed by looking into just one microservice and treating the other ones as black boxes. Tracing only adds value in very complex scenarios.

The tracing data can also be used to create dependency graphs. However, such a graph can be generated by observing the traffic between microservices. There is no need for an ID in each request and therefore no need to change the code of the microservices.

## 3.2 Logging

Logging is another important technology to gain more insight into a system. A service mesh collects information about the network communication. It could write that information also to a log file. Access logs that contain entries for each HTTP access are an important source of information to evaluate the success of web sites. So a log that contains only information about network traffic might be useful.

However, often log files are used to analyze errors in microservices. To do that, the log files have to include detailed information. The microservices will need to provide that information and log it. So just like with tracing, the microservice needs to be modified. In fact, for logging the service mesh adds little value because usually the information about the network traffic is not that useful to understand problems in the system.

The logging support of a service mesh has the advantage that developers do not have to care about these logs at all. Besides, the logs are uniform no matter what kind of technology is used in the microservices and how they log. Enforcing a common logging approach and logging format takes some effort. This is particularly true for microservices that use different technologies. So although the logs of the service mesh might not include information from inside the microservices, they are easy to get. Such a log might be better than no log at all or no uniform log.

## 3.3 Resilience

Resilience means that individual microservices still work even if other microservices fail. If a microservice calls another microservice and the called microservice fails, this will have an impact. Otherwise, the microservice would not need to be called at all. So the calling microservice will behave differently and might not be able to respond successfully to each request. However, the microservice must still respond. It must not block a request because then other microservices might be blocked and an error cascade might occur. Also delays in the network communication might lead to such problems.

A typical microservices architecture contains a large number of microservices. So it is quite likely that at least one microservice fails. If this leads to a cascade of error, the system will become quite unstable.

### 3.3.1 Circuit Breaker

*Circuit breakers* are one way to add some resilience to a microservices system. A conventional circuit breaker would cut a circuit if the circuit has been short-circuited. That protects the circuit from overheating. Circuit breakers in software systems do something similar: They protect a part of the system by cutting off some of the traffic. Because service meshes add proxies to the communication between the microservices, circuit breakers can be added without changing the code.

### 3.3.2 Retry

*Retries* repeat the failed requests. They are an obvious way to increase resilience: If the failure is transient, the retry can make the request succeed. However, retries also increase the number of requests. So the called microservice will have a higher load and might become unstable. A circuit breaker might be used to solve this problem.

### 3.3.3 Timeout

*Timeouts* make sure that the calling microservice is not blocked for too long. Otherwise, if all threads are blocked, the calling microservice might not be able to accept any more requests. So a timeout will not make it more likely that a request succeeds, rather it makes the request fail faster so that less resources are blocked.

## 3.4 Routing

Any microservices system needs some way to route requests between microservices and to route a request from the outside to the correct microservice. Implementing these features can be very simple. A reverse proxy might be enough to route requests from the outside to the correct microservices.

However, more advanced routing capabilities are useful, too:

- *Canary releasing* is a deployment process. First one instance of the new version of a microservice is deployed while there are still some instances of the old version around. Step by step, more and more traffic is routed to the new version. At the

same time, the new version is monitored. So if there is a problem with the new version, it will become obvious while some instances of the old version are still around and it is therefore easy to roll back. Advanced routing supports this process by splitting the traffic between the two versions. This might be done randomly or according to specific segments like devices or geo regions.

- Another way to mitigate the risk of a deployment is *mirroring*. In that case the new and the old version of a microservice run in parallel. Both receive the traffic and respond to it. It is possible to look at different behaviors in detail and to determine whether the new version works correctly. Istio automatically discards the responses of the new service. In that case, the risk for deploying a new version is essentially non-existent.
- *A/B testing* gives different users access to different versions of a microservice. The test helps to determine which version generates more revenue, for example. Again, randomly routing requests or routing requests for specific customer segments is a prerequisite for this approach.

It is quite easy to implement these features once the proxies of the service mesh are in place. The only difference might be that not just traffic between microservices must be handled, but also traffic from the outside.

## 3.5 Security

Obviously, it is possible to *encrypt* the network traffic between the proxies. However, there is still the network traffic between the proxy and the microservice. Usually the proxy and the microservice run on the same machine. So while the traffic might look like network traffic, it really goes through a loop-back device and not through a real network. Therefore, service meshes can implement encryption and confidentiality transparently.

To enable the encryption, a public key infrastructure is useful. If each microservice receives a certificate, it is easy to set up encrypted communication. It is even possible to use mTLS (mutual TLS authentication). That way, each microservice has a unique

certificate. It is therefore possible to implement *authentication* and ensure that a call actually originates from a specific microservice.

Of course, it is then possible to limit what a microservice can do. For example, the proxies might not allow specific requests to other microservices. That means that the service mesh can implement *authorization* for microservices.

Finally, it is possible to add a token to the HTTP communication. Standards like JWT<sup>1</sup> (JSON Web Token) define how information about a user and its roles can be transferred with each HTTP request. The proxy of the service mesh can examine the token and implement authorization based on the data provided.

## 3.6 Performance Impact

While service meshes have many benefits, they also add some challenges. For example, they add proxies to the communication. This increases the latency for network communication as the communication goes through more hops. In some cases, latency is very important. For example, for e-commerce applications a slightly higher latency can already impact revenue.

For that reason, it is important to keep the additional latency of the proxies as small as possible. For example, it is possible to collect data about requests in the proxy and send it to the service mesh infrastructure later. That way sending the information does not increase the latency even further. Also updates – for example to security policies, are distributed asynchronously to the proxies. That means updates to such policies might take some time until they are actually enforced in every proxy. Istio is optimized for low latency.

Of course, the service mesh itself needs to run. This will consume memory and CPU. However, hardware is becoming cheaper constantly and adding some hardware resources to improve the reliability of a system might be acceptable.

---

<sup>1</sup><https://jwt.io/>

## 3.7 Legacy

Service meshes are very useful for microservices architectures because they solve many challenges of distributed systems. The microservice architecture has been around for years and its popularity is still growing. But many teams have experienced that slicing their monolithic application takes a long time. A service mesh can add monitoring, routing, resilience, and security features to legacy parts of the application and facilitate integrating legacy and hybrid applications into modern architectures. Of all service mesh implementations, Istio offers the best support for this scenario because it is not limited to Kubernetes. It can integrate legacy systems that run on different infrastructures using service mesh expansion<sup>2</sup>.

## 3.8 Alternatives to Service Meshes

While service meshes provide a lot of features, there are also alternatives. Monitoring, logging, and resilience can be implemented as part of a microservice. There are numerous libraries that support the implementation of these features. However, an infrastructure to collect the monitoring and logging data still has to be setup. In addition, libraries are only available for specific programming languages or platforms. So if a microservice has to be implemented in a different programming language or on a different platform, another library must be used. That is an additional effort and makes it harder to use different technologies in the microservices. However, because there are no proxies involved, the performance might be better. Moreover, libraries don't rely on a modification of the infrastructure. So if just a few microservices should include the features mentioned above, libraries might be the better solution because the behavior of the infrastructure and therefore of all other microservices is not modified at all. Besides, in some cases developers cannot modify the infrastructure because a different part of the organization or even a different organization manages it.

For routing technologies reverse proxies or API gateways might be alternatives. They are specialized in providing just these features. Limiting the feature set might make

---

<sup>2</sup><https://istio.io/docs/setup/kubernetes/additional-setup/mesh-expansion/>

them easier to handle than a service mesh. However, it also means that they do not provide all the other features a service mesh has to offer.

For security, an infrastructure that provides certificates must be set up, certificates must be distributed, and the communication must be encrypted. While there are alternatives, they also require a modification to the infrastructure and have a limited set of features. In addition, non-authorized microservices must not be able to get certificates from the infrastructure. Service meshes control the proxies that need to receive the certificates. So it is quite easy for a service mesh to ensure that certificates are distributed securely because it controls all of the parts of the communication infrastructure.

## 3.9 Conclusion

Service meshes are based on a rather simple idea: Add proxies to the communication between microservices and add an infrastructure to configure the proxy and evaluate the data the proxies send. However, it turns out that this idea is quite powerful. It provides basic monitoring, security, analysis of the dependencies, and resilience. It is not necessary to change any code to enjoy these benefits.

However, for tracing the microservices have to forward the unique IDs for each call. So in that case some changes to the code are needed. Service meshes can provide only basic logging for the network traffic. That might be of little value. And of course the additional infrastructure comes with a cost: The latency increases and the additional infrastructure also consumes memory and CPU.

However, all in all, service meshes solve challenges that are quite important for microservices systems and are therefore a good addition to a microservices system.

## 4 Example with Istio

Typically, an example that shows a technology in action is a great way to understand how the technology actually works. This chapter shows a system consisting of multiple microservices and demonstrates how a service mesh can add value to a microservices architecture. The example runs on Kubernetes and uses Istio as a service mesh.

### 4.1 Istio

Istio is the most popular service mesh and was developed by Google and IBM. Just like Kubernetes, it is an Open Source reimplementation of a part of Google's internal infrastructure. Istio implements all service mesh features described in the previous chapter such as metrics, logging, tracing, traffic routing, circuit breaking, mTLS, and authorization. Although Istio is designed to be platform-independent, it started with first class support for Kubernetes.

Figure 4.1 reflects how the service mesh is located between the orchestrator (top) and the application (bottom). The four core components of Istio make up the control plane: Galley, Pilot, Mixer, and Citadel. They communicate with the service proxies to distribute configurations, receive recorded network traffic and telemetry data, and manage certificates. Istio uses *Envoy*<sup>1</sup> as service proxy, a widely adopted open source proxy that is used by other service meshes, too.

In addition to the typical service mesh control and data plane, Istio also adds infrastructure services. They support the monitoring of microservice applications. Instead of developing its own tools, Istio integrates established applications such as *Prometheus*, *Grafana*, and *Jaeger* and the service mesh dashboard *Kiali*. The image shows that the Istio control plane interacts with the orchestrator, which today is in most cases Kubernetes.

In a Kubernetes environment, Istio adds over 20 Custom Resource Definitions (CRDs), which demonstrates the complexity of the Istio API and the richness of configuration options. On the one hand, this allows full customization, but on the other hand it

---

<sup>1</sup><https://www.envoyproxy.io>

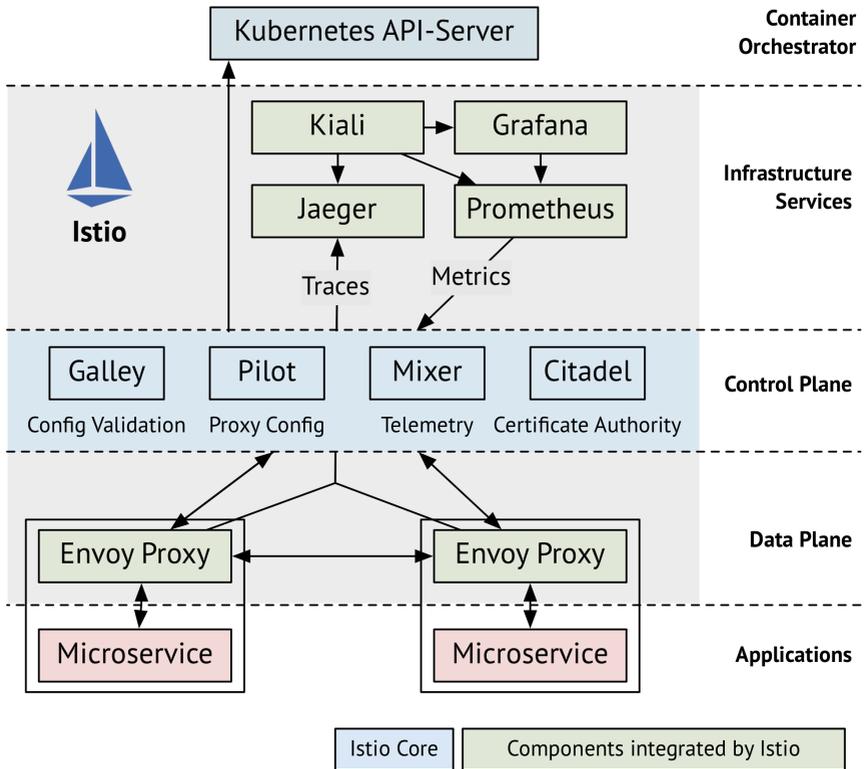


Figure 4.1: Istio Architecture

clearly affects the usability. Istio also adds a number of components (marked as Istio Core and integrated components in the figure) to an application which increases technical complexity.

## 4.2 Overview

The example in this chapter contains three microservices: Users can enter new orders via a web interface of the *order* microservice. The data about the order is transferred to the *invoicing* microservice that will create an invoice and present it to the user.

The *shipping* microservice will use the data about the order to create a shipment. The invoicing and the shipping microservice present a web interface to the user, too.

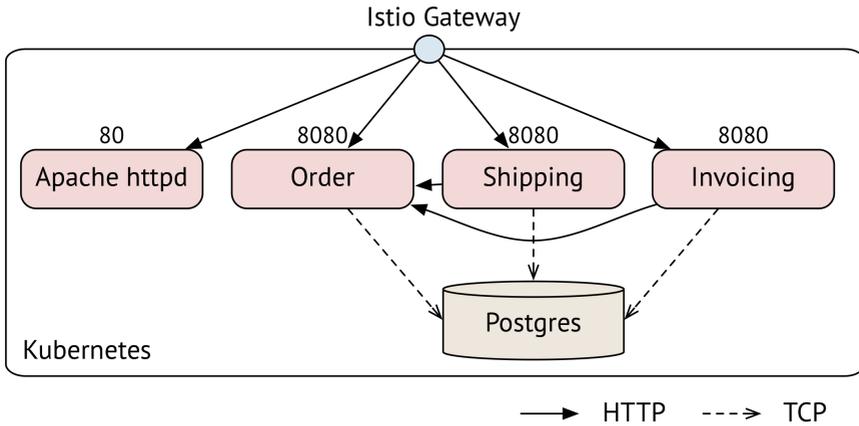


Figure 4.2: Overview of the Example

Figure 4.2 shows the structure of the example:

- Istio provides the *Ingress Gateway*. It forwards HTTP requests to the microservices. In addition to the features of the Kubernetes Ingress, the Istio Gateway supports Istio's features mentioned previously, such as monitoring or advanced routing.
- *Apache httpd* provides a static HTML page that serves as the home page for the example. The page has links to each microservice.
- *Order*, *shipping*, and *invoicing* are microservices. Shipping and invoicing poll data about the orders from the order microservice using REST. Istio understands HTTP and REST, so it can support this interface very well. The data format is based on JSON. The feed contains a simple JSON document with a list of links to the individual orders.

- All three microservices use the same *Postgres database*.

## Run the Example

You can access the code for the example at <https://github.com/ewolff/microservice-istio>. The documentation<sup>2</sup> explains in detail how to install the required software and how to run the example.

## 4.3 How the Example Uses Istio

A closer look at the `microservices.yaml` file shows that the deployment for the microservices contains no Istio-specific information. This is an advantage that makes it easier to use Istio. It also means that Istio works with any type of microservice no matter what programming language or frameworks are used.

However, Istio supports features such as resilience and monitoring as mentioned. Somehow Istio needs to collect information about the microservices.

### Sidecar

As mentioned in chapter 2, the idea behind a *sidecar* is to add another Docker container to each Kubernetes pod. Actually, if you list the Kubernetes pods with `kubectl get pods`, you will notice that for each pod it says 2/2:

```
[~/microservice-istio]kubectl get pods
NAME                                READY STATUS  RESTARTS  AGE
apache-7f7f7f79c6                   2/2   Running   0          8m51s
invoicing-77f69ff854                2/2   Running   0          8m43s
order-cc7f8866                      2/2   Running   0          8m43s
postgres-5ddddbbf8f                 2/2   Running   0          8m51s
shipping-5d58798cdd                 2/2   Running   0          8m43s
```

---

<sup>2</sup><https://github.com/ewolff/microservice-kubernetes/blob/master/HOW-TO-RUN.md>

So while there is just one Docker container configured for each Kubernetes pod, two Docker containers are in fact running. One container contains the microservice, and the other contains the sidecar that enables the integration into the Istio infrastructure.

Istio automatically injects these containers into each pod. During the installation described previously, `kubectl label namespace default istio-injection=enabled` marked the default namespace so that Istio injects sidecars for each pod in that namespace. Namespaces are a concept in Kubernetes to separate Kubernetes resources. In this example, the default namespace contains all Kubernetes resources that the user provides. The namespace `istio-system` contains all Kubernetes resources that belong to Istio itself.

The sidecars contain the proxies. Istio routes all traffic between the microservices through these proxies as described in chapter 2.

## 4.4 Monitoring with Prometheus and Grafana

As mentioned in section 2.1, a service mesh provides basic monitoring information without any additional effort. Istio includes *Prometheus*<sup>3</sup> to store metrics for analysis. Istio also includes *Grafana*<sup>4</sup>, a tool for the analysis of the metrics.

Of course, this approach supports only metrics that the proxy can measure. This includes all the information about the request, such as its duration or the status code. Also, information about the Kubernetes infrastructure – for example, CPU utilization – can be measured. However, data about the internal state of the microservice is not measured. To get that data, the microservice would need to report it to the monitoring infrastructure.

---

<sup>3</sup><https://prometheus.io/>

<sup>4</sup><https://grafana.com/>

## Prometheus

The documentation of the example<sup>5</sup> contains information how to run Prometheus in the example.

Prometheus stores all metrics and also provides a simple UI to analyze the metrics. Figure 4.3 shows the byte count for requests and the different destinations: the order microservice and also the Istio component that measures the telemetry data.

Prometheus is multi-dimensional. The destination is one dimension of the data. These metrics could be summed up by dimensions such as the destination to understand which destination receives how much traffic.

## Grafana

For more advanced analysis of the data, Istio provides an installation of Grafana. The documentation<sup>6</sup> explains how to use Grafana with the example.

The Grafana installation in Istio provides predefined dashboards. Figure 4.4 shows an example of the Istio service dashboard. It uses the shipping microservice. The dashboard shows metrics such as the request volume, the success rate and the duration. This gives a great overview about the state of the service.

Istio supplies also other dashboards. The Istio performance dashboard provides a general overview about the state of the Kubernetes cluster with metrics such as memory consumption or CPU utilization. The Istio mesh dashboard shows a global metric about the number of requests the service mesh processes and their success rates.

So just by installing Istio, basic monitoring for all microservices is already in place.

---

<sup>5</sup><https://github.com/ewolff/microservice-istio/blob/master/HOW-TO-RUN.md#prometheus>

<sup>6</sup><https://github.com/ewolff/microservice-istio/blob/master/HOW-TO-RUN.md#grafana>

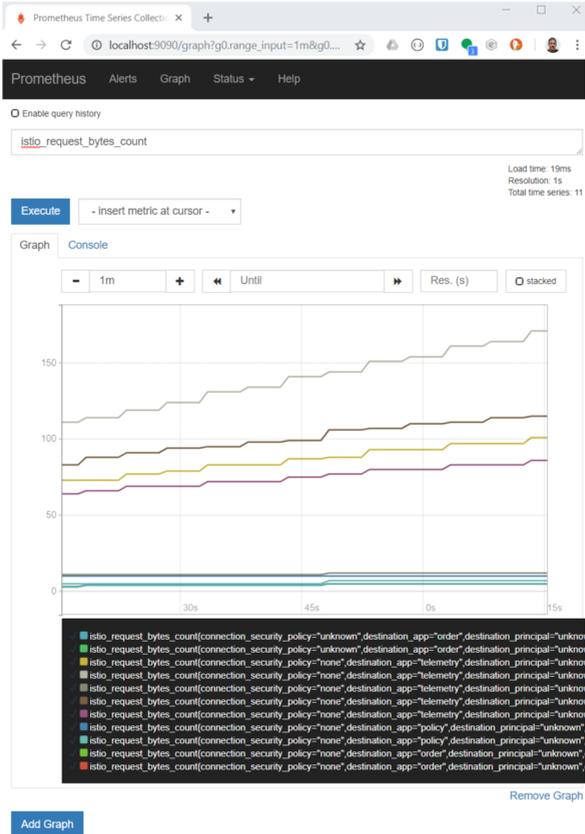


Figure 4.3: Prometheus with Istio Metrics

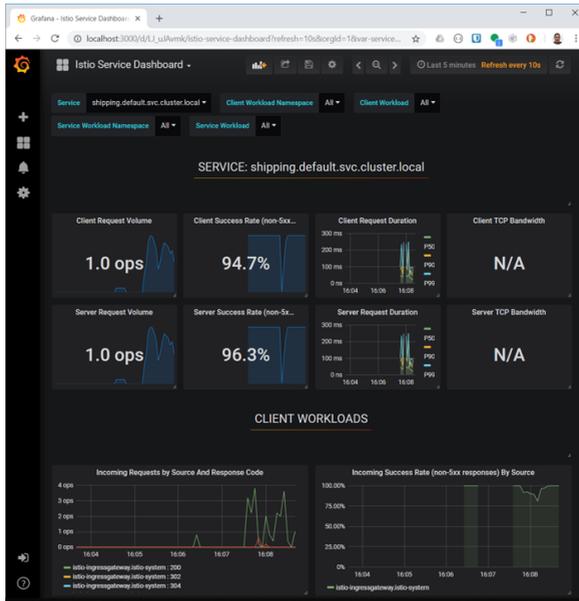


Figure 4.4: Grafana with Istio Dashboard

## 4.5 Tracing

As explained in section 2.1, tracing might be important to trace calls across microservices and to do a root cause analysis based on that information. For tracing, Istio uses Jaeger<sup>7</sup>.

The documentation of the example contains a section<sup>8</sup> about tracing.

Figure 4.5 shows an example of a trace for a request to the shipping microservice. The user started a poll for new data on the order microservice. Then the service contacted the Istio Mixer to make sure the policies are enforced.

Figure 4.6 shows a different type of information Jaeger provides: the dependencies between the microservices. The shipping and invoicing microservices use the order microservice to receive information about the latest orders. The order microservice

<sup>7</sup><https://www.jaegertracing.io/>

<sup>8</sup><https://github.com/ewolff/microservice-istio/blob/master/HOW-TO-RUN.md#tracing>

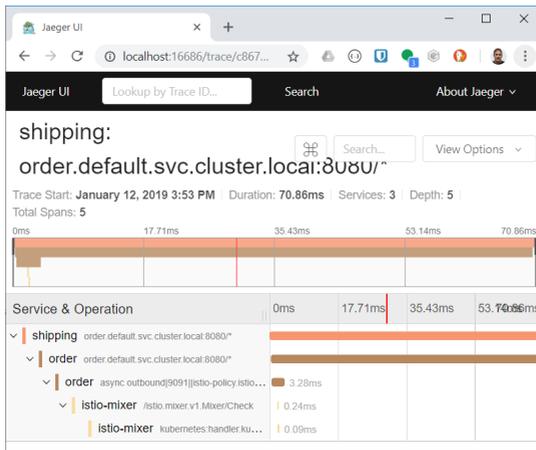


Figure 4.5: Jaeger Trace

reports metrics to Mixer. And finally, the order microservice is accessed by the Istio gateway when external requests are forwarded to it. This information about dependencies might be useful to get an overview about the architecture of the system.

To understand which incoming request caused which outgoing requests, Jaeger relies on specific HTTP header. The values in the headers of the incoming requests have to be added to any outgoing request. This means that tracing cannot be transparent to the microservices. They have to include some code to forward the tracing headers from the incoming request to each outgoing request.

## Code for Tracing

The example uses Spring Cloud Sleuth<sup>9</sup>. This is a powerful library that supports many features for tracing. Spring Cloud Sleuth just needs to forward the HTTP headers. So in application.properties the parameter spring.sleuth.propagation-keys contains the HTTP headers that must be forwarded. The x-b3-\* headers are automatically forwarded by Spring Cloud Sleuth so just the x-request-id and x-ot-span-context header have to be configured. Other languages require different means to forward the HTTP headers.

<sup>9</sup><https://spring.io/projects/spring-cloud-sleuth>

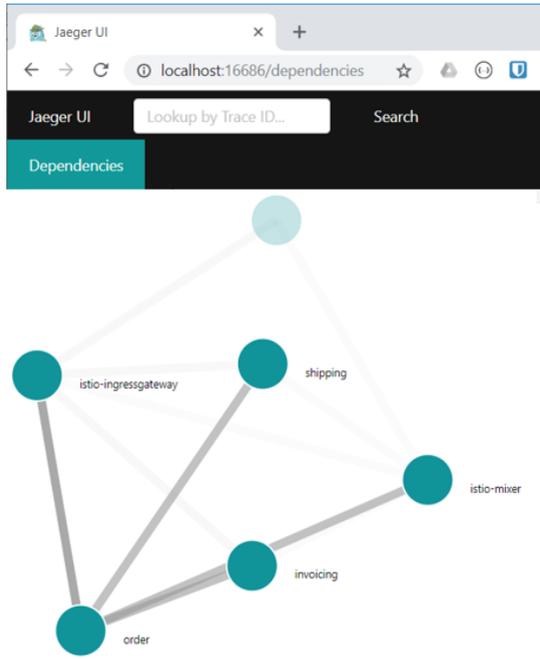


Figure 4.6: Jaeger Dependencies

## 4.6 Visualization with Kiali

With the information from the tracing, Jaeger can provide a graph depicting the dependencies between the microservices. However, Kiali<sup>10</sup> is a tool that is specialized in generating dependency graphs of microservices. It also shows traffic rates, latencies, and health of the services. That way, it provides not just a great overview of the microservices and their relationships, but also information about how they are communicating with each other. That makes it clearer what the status of the dependencies is and what is going on where in the microservices system. Figure 4.7 shows an example of a dependency graph.

The documentation<sup>11</sup> explains how to use Kiali for the example.

<sup>10</sup><https://www.kiali.io/>

<sup>11</sup><https://github.com/ewolff/microservice-istio/blob/master/HOW-TO-RUN.md#kiali>

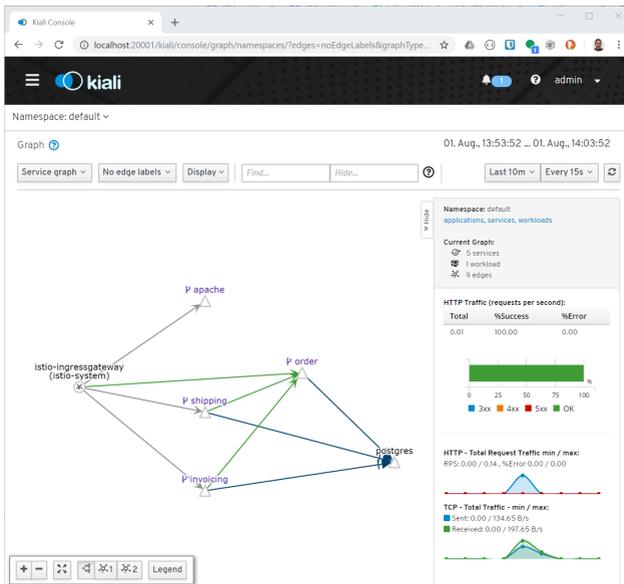


Figure 4.7 Dependencies in Kiali

## 4.7 Logging

As explained in section 3.2, service meshes are of little help for logging. However, Istio can forward information about each HTTP request to a logging infrastructure. That information can be used to analyze, for example, the number of requests to certain URLs and status codes. A lot of statistics for web sites rely on this kind of information.

The format in the logs can be configured. A log entry may contain any information Mixer received from the request.

### Logs in the Example

However, Istio does not provide an infrastructure to handle logs. For the example, a custom log infrastructure was set up. This infrastructure uses Elasticsearch to store logs and Kibana to analyze them.

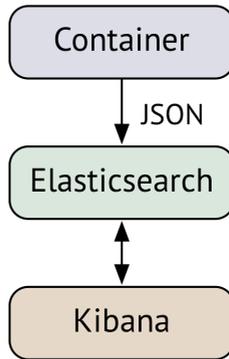


Figure 4.8 Logging in the Example

Figure 4.8 shows how logging is implemented in the example. Each microservice must directly write JSON data to the Elasticsearch server. So there is no need to write any log files which makes the system easier to handle. The need to parse the log information has also been eliminated; Elasticsearch can directly process it.

## Code for Logging

The example uses the Logback<sup>12</sup> Java library. The Logback Elasticsearch Appender<sup>13</sup> forwards the logs to Elasticsearch. The configuration in the file `logback-spring.xml` defines what information the microservices log.

The Istio infrastructure could log to the same Elasticsearch instance, too. However, for the example, it was decided that this is not necessary. With the current system, it is easy to find problems in the implementation by searching for log entries with severity error. Also logging each HTTP request adds little value. Information about the HTTP requests is probably already included in the logs of the microservices.

---

<sup>12</sup><https://logback.qos.ch/>

<sup>13</sup><https://github.com/internetitem/logback-elasticsearch-appender>

## 4.8 Resilience

Resilience means that a microservice should not fail if other microservices fail. It is important to avoid failure cascades that could bring down the complete microservices system.

### Measuring Resilience with Istio

As explained in section 3.3, failure cascades can happen if a called microservice returns an error. It could be even worse if the called microservice does return successfully, but takes a long time. In that case, resources such as threads might be blocked while waiting for a reply. In the worst case, all threads end up blocked and the calling microservice fails.

Such scenarios are hard to simulate. Usually the network is reasonably reliable. It would be possible to implement a stub microservice that returns errors, but that would require some effort.

However, Istio controls the network communication through the proxies. It is therefore possible to add delays and errors to specific microservices. Then the other microservices can be checked to see if they are resilient against the delays and failures.

### Fault Injection

The configuration is in the file `fault-injection.yaml` from the example. It makes 100 percent of the calls to the order microservice fail with HTTP status 500. See the documentation<sup>14</sup> about how to apply this configuration to your system.

Actually, the microservice will still work after applying the fault injection. If you add a new order to the system, though, it will not be propagated to the microservices shipping and invoicing. You can make those microservices poll the order microservice by pressing the pull button in the web UI of the shipping and invoicing microservices. In that case, an error will be shown. So the system is already quite resilient because it uses polling. If the shipping microservice would call the order microservice – for

---

<sup>14</sup><https://github.com/ewolff/microservice-istio/blob/master/HOW-TO-RUN.md#fault-injection>

example, to fulfill a request –, the shipping service would fail after the fault injection if no additional logic is implemented to handle such a failure.

## Delay Injection

Another possibility is to inject a delay, see the documentation<sup>15</sup>. If you make the shipping microservice poll the order microservice, it will take longer but it will work fine. Otherwise the system just works normally. So again, due to polling the system is already quite resilient.

## Implementing Resilience with Istio

Fault and delay injection are only useful to test the resilience of a system. However, as explained in section 3.3 circuit breakers are one way to actually make a system more resilient.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: order-circuit-breaker
spec:
  host: order.default.svc.cluster.local
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        http2MaxRequests: 1
    outlierDetection:
      consecutiveErrors: 1
      interval: 1m
```

---

<sup>15</sup><https://github.com/ewolff/microservice-istio/blob/master/HOW-TO-RUN.md#delay-injection>

```
baseEjectionTime: 10m
maxEjectionPercent: 100
```

The previous listing shows a configuration of a circuit breaker for Istio. See the documentation<sup>16</sup> for information about how to apply it. The configuration has the following settings:

- A maximum of one TCP connection is allowed for the service (`maxConnections`).
- In total, just one HTTP 1.1 (`http1MaxPendingRequests`) and one HTTP 2 (`http2MaxRequests`) request might be pending.
- Each minute, each microservice instance is checked (`interval`). If it has returned one error (`consecutiveErrors`) – an HTTP status 5xx or a timeout – it is excluded from traffic for ten minutes (`baseEjectionTime`). All instances of the microservice might be excluded from traffic in this way (`maxEjectionPercent`).
- Exceeding the connection limits (`maxConnections`, `http1MaxPendingRequests`, `http2MaxRequests`) will also result in an error state and trigger the circuit breaker.

These limits protect the microservice from too much load. And, if an instance has already failed, it is excluded from the work. That gives the instance a chance to recover.

The limits in the example are very low to make it easy to trigger the circuit breaker. If you use the `load.sh` script to access the order microservice's web UI, you will need to run a few instances of the script in parallel to receive 5xx error codes returned by the circuit breaker.

If the circuit breaker does not accept a request because of the defined limits, the calling microservice will receive an error. So the calling microservice is not protected from a failing microservice.

---

<sup>16</sup><https://github.com/ewolff/microservice-istio/blob/master/HOW-TO-RUN.md#circuit-breaker>

**Retry and Timeout** Retries and timeouts as explained in section 3.3 can be defined in the same part of the Istio configuration. That makes it easy to define a timeout per retry and a maximum time span that all retries together might take.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: order-retry
spec:
  hosts:
  - order.default.svc.cluster.local
  http:
  - retries:
      attempts: 20
      perTryTimeout: 5s
      retryOn: connect-failure,5xx
    timeout: 10s
    route:
    - destination:
        host: order.default.svc.cluster.local
```

The previous listing shows a part of the file `retry.yaml`. It configures retries and timeouts for the `order` microservice. Calls to the `order` microservice are retried up to 20 times. For each retry, a timeout of 5 seconds is configured. However, there is also a total timeout of 10 seconds. So if the retries don't succeed within 10 seconds, the call will fail. The Istio's default timeout is 15 seconds.

`retryOn` defines when a request is considered failed. In this case any HTTP status code `5xx` or connection failures such as timeouts are considered failed requests.

The rest of the file is not shown in the listing. It adds retries to the Istio Ingress gateway for the `order` microservice.

See the documentation<sup>17</sup> for information about how to make the order service fail and then fix the problem by applying this configuration to the system.

## Resilience: Impact on the Code

Istio's circuit breaker makes it more likely that a call to a microservice fails. The same is true for a timeout. Retries can reduce the number of failures. But even with Istio's resilience features, there will still be calls to microservices that fail. So while retry, timeout, and circuit breaker do not require any changes to the code, the code still needs to take care of failed requests. These result in a response with HTTP status code 5xx. How failures are handled is a part of the domain logic. For example, if the warehouse management is down, it might not be possible to determine the availability of certain items. Whether an order should be accepted – even though the availability of the ordered items is unknown –, is a business decision. Maybe it is fine to accept the order and deal with the unavailable items. Maybe this would disappoint customers or violate contracts and is therefore not an option. So there must be code that handles an HTTP status code 5xx and determines whether the order should still be processed.

## 4.9 How to Dig Deeper

Istio provides extensive documentation. You can use it to familiarize yourself with features of Istio that this chapter does not discuss:

- Istio provides a list of tasks<sup>18</sup>. They provide hands-on guidelines how to gain more experience with Istio and are therefore a great addition to this chapter.
- Istio provides support for security. See the security task<sup>19</sup> for some hands-on exercises for this feature. The exercises cover encrypted communication, authentication and authorization.

---

<sup>17</sup><https://github.com/ewolff/microservice-istio/blob/master/HOW-TO-RUN.md#retry>

<sup>18</sup><https://istio.io/docs/tasks/>

<sup>19</sup><https://istio.io/docs/tasks/security/>

- Istio also supports advanced routing. For example, the traffic shifting task<sup>20</sup> shows how to do A/B testing with Istio. The mirroring task<sup>21</sup> shows mirroring. With mirroring, two versions of the microservice receive the traffic. Mirroring can be used to make sure that the new and the old version behave in the same way.
- This example<sup>22</sup> discusses how Istio support logs, showing how log information can be composed from Mixer's data. This example outputs the logs to stdout and not to a log infrastructure.
- Also, this example<sup>23</sup> shows how Fluentd<sup>24</sup> collects the logs Istio provides from all microservices. The logs are stored in Elasticsearch and evaluated with Kibana.

## 4.10 Conclusion

The example shows how Istio provides a complete monitoring environment and basic monitoring information for the microservices system without any additional code. The microservices are treated as black boxes i.e. the metrics only cover the communication between the microservices and the infrastructure. However, this could in fact be enough for a production system as it shows the performance as experienced by the user.

Some code has to be added to the microservices for tracing so that they forward the tracing headers to the outgoing HTTP requests. Still the complete infrastructure is provided by Istio.

Istio does not provide an infrastructure for logging, but it can log information for each HTTP request to a logging infrastructure. However, logging is often used to look inside the microservices and understand what they actually do. This can only be implemented in the code of the microservices. Still Istio could at least provide some very basic information in the logs without any impact on the code or the microservices.

---

<sup>20</sup><https://istio.io/docs/tasks/traffic-management/traffic-shifting/>

<sup>21</sup><https://istio.io/docs/tasks/traffic-management/mirroring/>

<sup>22</sup><https://istio.io/docs/tasks/telemetry/metrics-logs/>

<sup>23</sup><https://istio.io/docs/tasks/telemetry/fluentd/>

<sup>24</sup><https://www.fluentd.org/>

Istio allows to simulate problems in the system by injecting delays and faults. That is useful to test the system's resilience. Istio's circuit breaker and retries even help to implement resilience. If a microservice fails, the system still needs to compensate that failure. Dealing with a failed service must be covered by the domain logic. The microservices in the example are self-contained i.e. all data for the logic is stored in the microservice. So a failed microservice has very limited impact. However, this is a feature of the architecture and not the service mesh.



# 5 Other Service Meshes

The last chapter has discussed Istio. While Istio is the most popular service mesh, the market is quite diverse and worth evaluating. The first service mesh implementation was Linkerd, developed in 2015. In 2017, Google and IBM joined forces to create the Istio service mesh after they found out that they had been working on similar ideas. The public attention Istio received through Google and IBM as main contributors was amplified by media campaigns and conference talks. By the end of 2017, Linkerd announced a new, more opinionated service mesh only for Kubernetes that was first named Conduit and later Linkerd 2. By 2018, the term service mesh was ubiquitous and more products and companies joined the party. Consul added service mesh features and AWS announced their own service mesh implementation AWS App Mesh.

## 5.1 Linkerd 2

Linkerd 2<sup>1</sup> is the successor of Linkerd. Although Linkerd had been adopted in large production systems, the software was too complex to configure and not performing well. These limitations induced Buoyant to develop a completely new service mesh. The result – formerly Conduit, today called Linkerd 2 – was presented in December 2017 as an open source project, written in Go and Rust. Linkerd 2 is an incubation project of the CNCF<sup>2</sup> (Cloud Native Computing Foundation) and currently the only service mesh in the CNCF portfolio.

Although Kubernetes is the most common platform to be used with Istio, it is designed to work with any environment. The complexity of Istio is to some extent caused by this platform neutral design. While the first version of Linkerd was designed alike, Linkerd 2 is committed to Kubernetes only. Linkerd 2 implements most service mesh features such as monitoring, routing, retries, timeouts, and mTLS. Circuit breaking, tracing, and authorization are missing. While most service mesh implementations use the Envoy service proxy, Linkerd includes its own service proxy implementation (linkerd-proxy).

---

<sup>1</sup><https://linkerd.io>

<sup>2</sup><https://www.cncf.io>

Rewriting Linkerd for usability and performance while integrating with Kubernetes seems to have worked out. The API of Linkerd 2 is much cleaner and more consistent than Istio's. It introduces only one Kubernetes CRD (Custom Resource Definition) and provides a carefully considered dashboard, shown in figure 5.1. Kubernetes users should seriously consider it since it includes most service mesh features in a production ready stage, has a small resource and performance footprint, and provides an excellent developer experience.

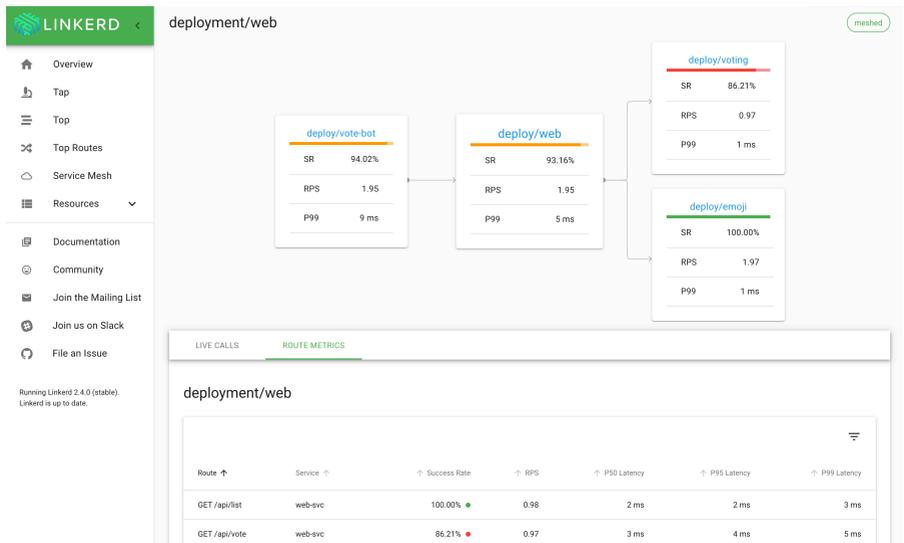


Figure 5.1: Linkerd Dashboard

## 5.2 Consul

Consul<sup>3</sup>, originally a solution for service discovery, recently added service mesh use cases. Consul supports service discovery, authorization, mTLS, monitoring through metrics, and routing capabilities. The latter were added by integrating with the Envoy

<sup>3</sup><https://www.consul.io>

proxy that Istio and other service meshes are also using. Resilience features like circuit breaking, retry, and timeout are yet to be developed. Like Istio, Consul does not depend on any specific orchestration platform, but is compatible with Kubernetes and Nomad.

## 5.3 AWS App Mesh

As the name suggests, AWS App Mesh<sup>4</sup> is a service mesh developed for AWS. It was first introduced in November 2018. Like Istio and Consul, AWS App Mesh uses a pre-configured but customizable Envoy as service proxy and adds its own control plane. It is integrated with other AWS tools like CloudWatch for metrics and X-Ray for traces. AWS App Mesh already supports essential service mesh features like monitoring, routing and mTLS. AWS users can activate App Mesh in Fargate, EC2, EKS, and EC2 and add configuration through JSON. In Kubernetes environments, configuration can be applied through CRDs and applied by the AWS App Mesh Controller<sup>5</sup>.

AWS is the most recent service mesh implementation, but since AWS has the biggest market share for cloud computing, AWS App Mesh is expected to stay and grow.

## 5.4 When to Choose Which?

The features a service mesh provides are useful for the vast majority of microservice applications. When it comes to choosing a specific service mesh implementation, it is hard to resist going with Istio as it is the most popular one. But experience shows that technical decisions are best based on requirements and problems rather than any hype, a project's public attention, or even its number of features.

As shown in figure 5.2, Istio has overtaken all other service mesh implementations in terms of feature completeness and configurability. But this rich feature set has turned Istio into a complex component that can be hard to manage in practice. In cases where not all features and their customizability are required, Linkerd 2, Consul, and AWS App Mesh might be better choices.

---

<sup>4</sup><http://aws.amazon.com/app-mesh/>

<sup>5</sup><https://github.com/aws/aws-app-mesh-controller-for-k8s>

		 Istio 1.2	 Linkerd 2.4	 Consul 1.6	 AWS App Mesh
General	<b>Platform</b>	Kubernetes, Nomad & Consul	Kubernetes	Kubernetes, Nomad	AWS Fargate, ECS, EKS, EC2
	<b>Service Proxy</b>	istio-proxy (extended Envoy)	linkerd-proxy	Envoy	Envoy
Monitoring	<b>Metrics</b>	✓	✓	✓	✓
	<b>Tracing</b>	Zipkin, Jaeger or LightStep	✗	Manually through Envoy	AWS X-Ray
	<b>Dashboard</b>	✓	✓	No service graph	No service graph
Routing	<b>Load Balancing</b>	✓	✓	✓	✓
	<b>Traffic Management</b>	✓	✓	✓	✓
Resilience	<b>Timeouts &amp; Retries</b>	✓	✓	✗	✓
	<b>Circuit Breaking</b>	✓	Announced for version 2.5	✗	✗
Security	<b>mTLS</b>	✓	✓	✓	✓
	<b>Authorization</b>	✓	✗	✓	✗

Figure 5.2: Features of service mesh implementations as of August 2019

Another criterion is the platform. If the whole application runs in Kubernetes anyway, users can benefit from the simplicity of Linkerd 2. If Consul or AWS are already used, the corresponding service mesh implementations might cause least friction. If multiple clusters or legacy applications outside of the cluster are involved, Istio provides appropriate concepts and configuration as mentioned in chapter 4.

Many microservice applications are dealing with a higher latency compared to monoliths. Communication in a monolith is always a method call in the same process. Communication between microservices goes through the network stack and has therefore a higher latency. A service mesh also adds to the latency, which is not completely balanced by improved load balancing strategies. In addition, the sidecars double the number of running containers which has an impact on resource consumption. Fair benchmarks in this space are hard to find because of the different feature sets and configuration. Most results show that although Istio's performance has improved over time, Linkerd 2 is performing better under high load and uses much less resources.

# 6 Conclusion

Microservices are here to stay. And service meshes are likely to be their long-term companions. Many monitoring, routing, resilience, and security features implemented inside each microservice can and will be taken over by a service mesh over time. These concepts have already proven themselves in the Google infrastructure. In some cases, the service mesh cannot – and should not – fully implement the desired functionalities. A service mesh has a black box view on the microservice. So internal information of the microservice is hidden from it. For example, the service mesh can only provide metrics about requests but not internal metrics like thread or database pool sizes. Therefore, in the example in chapter 4 logging had to be implemented in the microservices despite the service mesh.

## Asynchronous Communication

A service mesh is not limited to traditional microservices following the request-response paradigm. Microservices that communicate asynchronously also benefit from monitoring and security features as well as from tools like Prometheus, Grafana, Jaeger, or Kiali.

However, asynchronous microservices will probably not use HTTP at all so routing is of little use. Also the resilience features are not useful. Asynchronous microservices will work on a message queue eventually. So if the microservice is currently not available, processing the message will take more time - until the microservice is available again. The system should be able to deal with that, though.

## Maturity

Current implementations such as Istio, Linkerd, Consul, and AWS App Mesh are already mature and stable enough to be considered for production systems. However, it is essential to carefully test and compare the technologies with regard to the individual use case. As the example in chapter 3 shows, a service mesh is a piece of infrastructure and can be added and removed easily without changing application code. So it is relatively easy to test a service mesh for a concrete application without investing too much effort.

## Overhead

One of the few drawbacks of service meshes is the mental overhead they add for the developers. But the challenges service meshes solve must be dealt with in a microservices system anyway, which raises the question whether there is any simpler alternative to a service mesh. Compared to other available solutions, the mental overhead of a service mesh appears to be the smaller problem. The complexity of a service mesh seems rather necessary to deal with the complexity of the distributed microservice architecture itself.

However, not all service meshes have the same complexity. Istio provides many advanced features. That increases complexity but might only pay off in very few scenarios. Linkerd 2 on the other hand has less features, but is also less complex. For example, Linkerd 2 just defines one Kubernetes CRD (Custom Resource Definition) while Istio has many. These add to the complexity of the configuration of the service mesh. However, Istio can also be stripped down. It is possible to exclude any part of the system from the installation.

Recent efforts such as the SMI (Service Mesh Interface) intend to reduce mental overhead, standardize service mesh features, and will pave the way for even more tools on top of service mesh capabilities.

The second drawback is the technical overhead: A service mesh adds to the latency and the resource demands. However, this overhead will decrease since all current service mesh implementations are constantly improved.

## Service Mesh: Hype or Real?

So even though service meshes are a hype and there are some drawbacks, the advantages prevail in most cases. As the impact on the code is quite low, it is not too hard to give a service mesh a try and determine its value. Even if you decide not to use it in production, you will still learn something and understand modern infrastructure even better.