LEARNING MADE EASY

VMware Special Edition

Service Mesh

for dummies®

A Wiley Brand

Accelerate modern app development

Explore service mesh use cases

Transform multi-cloud networks

Brought to you by

vmware®

Niran Even-Chen

Oren Penso

Susan Wu

About VMware

VMware software powers the world's complex digital infrastructure. The company's cloud, networking and security, and digital workspace offerings provide a dynamic and efficient digital foundation to customers globally, aided by an extensive ecosystem of partners. Headquartered in Palo Alto, California, VMware is committed to being a force for good, from its breakthrough innovations to its global impact. For more information, please visit **www.vmware.com/company.html.**

Service Mesh

VMware Special Edition

by Niran Even-Chen, Oren Penso, and Susan Wu

for dummies®
A Wiley Brand

Service Mesh For Dummies®, VMware Special Edition

Published by
John Wiley & Sons, Inc.
111 River St.
Hoboken, NJ 07030-5774
www.wiley.com

ISBN 978-1-119-66034-7 (pbk); ISBN 978-1-119-66036-1 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Associate Publisher: Katie Mohr

Editorial Manager: Rev Mengle

Business Development Representative: Karen Hattan

Production Editor: Tamilmani Varadharaj

Special Help: Sergio Pozo, Pere Monclus, Mark Schweighardt

Table of Contents

INTRODUCTION 1
About This Book 1
Foolish Assumptions 1
Icons Used in This Book 2
Beyond the Book 2

CHAPTER 1: **The Rise of Microservices and Cloud-Native Architecture** 3
Recognizing the Need for Agility 3
Understanding That Application Architectures Are Changing 4
Service-oriented architecture 4
Microservices 5
Seeing That Distributed Applications Require a Reliable Network 9
Looking at How Kubernetes and Microservices Work Together 9

CHAPTER 2: **Service Mesh: A New Paradigm** 11
Identifying Challenges in Microservices Architectures 11
Introducing Service Mesh 13

CHAPTER 3: **Service Mesh Use Cases** 17
Traffic Management 17
Network reliability 18
Circuit breaking 19
Rate limiting 20
A/B testing 21
Canary releases 21
Traffic steering 23
Egress control 25
Observability 26
Metric collection 26
Distributed traces 27
Security 28
mTLS authentication 28
Istio authorization 29

CHAPTER 4: **Recognizing Complexity Challenges in Service Mesh** 31
Installing Istio 31
Cluster models 32
Network models 33
Control plane models 35
A Traffic-Shifting Configuration Example 38

CHAPTER 5: **Transforming the Multi-Cloud Network with NSX Service Mesh** 43
Introducing NSX Service Mesh 43
Increasing the Scope of the Mesh 44
VMware NSX Service Mesh Architecture 45
Introducing Global Namespaces 46
Federation and Intra-Service Mesh Interoperability 48
NSX Service Mesh Use Cases 50
Multi-cloud and hybrid cloud patterns 50
Business continuity 51
End-to-end mutual transport layer security (mTLS) 51
Rolling upgrades at scale 52
Predicting end-to-end response time 52

CHAPTER 6: **Ten (Or So) Resources to Help You Get Started with Service Mesh** 55
Online Resources 55
Discussion Groups 56
Books 56
User Stories 57
Interactive Labs 57
How-To Guides 58
Online Courses 58
Conferences and Meetups 58
Podcasts 59

Introduction

There's a common myth that a service mesh is only for a microservices architecture, but the truth is that a service mesh can benefit any enterprise that uses service-to-service communications in its application infrastructure — including traditional, monolithic applications and modern, cloud-native apps built on a microservices architecture.

About This Book

Service Mesh For Dummies consists of six chapters that explore

- The evolution of microservices and cloud-native architecture (Chapter 1)
- The basics of service mesh (Chapter 2)
- Service mesh use cases (Chapter 3)
- The complexity of service mesh (Chapter 4)
- Building a multi-cloud network with NSX Service Mesh (Chapter 5)
- Additional service mesh resources (Chapter 6)

Each chapter is written to stand on its own, so if you see a topic that piques your interest, feel free to jump ahead to that chapter. You can read this book in any order that suits you (though we don't recommend upside down or backward).

Foolish Assumptions

It's been said that most assumptions have outlived their uselessness, but we assume a few things nonetheless!

Mainly, we assume that you work for an organization that is interested in learning how service mesh can create an abstraction layer for both your traditional and modern cloud-native applications, regardless of where the application resides — whether on-premises or in a public cloud on virtual machines, containers, or bare-metal servers.

We also assume that you're either an application developer or a platform operator who supports application development. We assume you have an understanding of technology concepts such as cloud computing, networking, virtualization, and containers. As such, this book is written primarily for technical readers.

If any of these assumptions describes you, then this is the book for you! If none of these assumptions describes you, keep reading anyway! It's a great book and when you finish reading it, you'll know quite a lot about modern cloud-native application architectures and the service mesh!

Icons Used in This Book

Throughout this book, we occasionally use special icons to call attention to important information. Here's what to expect:

REMEMBER

This icon points out important information you should commit to your nonvolatile memory, your gray matter, or your noggin!

TECHNICAL STUFF

If you seek to attain the seventh level of NERD-vana, perk up! This icon explains the jargon beneath the jargon!

TIP

Tips are appreciated, never expected — and we sure hope you appreciate these useful nuggets of information.

WARNING

These alerts point out the stuff your mother warned you about. Well, probably not, but they do offer practical advice to help you avoid potentially costly or frustrating mistakes.

Beyond the Book

There's only so much we can cover in 64 short pages, so if you find yourself at the end of this book, thinking, "Gosh, this was an amazing book — where can I learn more?," just go to `www.vmware.com`.

IN THIS CHAPTER

- Recognizing the need for business agility
- Evolving from monolithic to service-oriented architecture to microservices
- Understanding the critical role of the network
- Looking at how containers and Kubernetes have changed app development

Chapter 1
The Rise of Microservices and Cloud-Native Architecture

In this chapter, you explore the need for business agility, how application architectures are evolving, the increasing importance of the network in modern cloud-native architectures, and the rise of containers and Kubernetes.

Recognizing the Need for Agility

Digital transformation is driving the need for speed and no market vertical is exempt. Companies are under constant pressure internally and externally to innovate faster and provide value to the business. In the relentless quest to serve customers and users better than their competition, many businesses are increasingly building their own custom software, rather than buying the same commercial off-the-shelf applications used by their competitors, to differentiate their products and services and achieve competitive advantage.

Much of this transformation is achieved in software, and enterprises are hiring a growing number of developers to turn innovative ideas into reality. Today's digital creators and revenue generators — application developers — are evolving to achieve not only faster development cycles, but also faster delivery times and more frequent deployments.

Understanding That Application Architectures Are Changing

Application architectures are constantly changing. Over the past several years, the application space has evolved from monolithic to service-oriented architecture (SOA) to microservices.

From a software development point of view, in a monolithic application, all components that compile the application are packaged and tested as a single unit. If, for example, the user interface (UI) team needs to make a small change in the code, that small change can have a ripple effect throughout the entire application stack, requiring it to be recompiled and redeployed.

In a monolithic architecture, the architectural choices are fixed and the teams cannot choose their own programming languages and tools. The entire software development team is stuck using the same integrated development environment (IDE), working on the same release, and using waterfall testing.

REMEMBER

Disadvantages of monoliths include

- Fixed architectural choices
- Fixed development environment
- Same release cadence
- Waterfall testing

Service-oriented architecture

SOA is a software architecture in which distinct components of the application provide services to other components via a communications protocol over a network. SOA integrates distributed, separately maintained software components that communicate

with each other using an enterprise services bus (ESB) messaging protocol over an Internet Protocol (IP) network.

SOA represents a middle phase between monolithic architectures and microservices, in which the organization breaks out parts of its applications and represents them in the network as web services. There are two main roles in SOA: a *service provider* and a *service consumer.* The consumer layer is where the users (humans, other components of the apps, or third parties) interact with the SOA, and the provider layer consists of all the services within the SOA.

Using SOA, organizations can encourage component reuse to avoid having to develop commonly used services like ecommerce shopping carts and short message service (SMS). Instead, organizations can just publish these shared services in a service catalog, and applications can then consume them over the network.

REMEMBER

SOA is still the most commonly used architecture, but the adoption of microservices is growing fast.

Some advantages of SOA over monolithic architectures include

- Component reusability
- Improved scalability and availability
- Easy maintenance

The biggest limitation of SOA is the ESB, which is a single point of failure that potentially impacts the entire system. Every service communicates over the ESB, so if one of the services slows down, the ESB can be bogged down by requests for that service.

Microservices

Microservices can be thought of as the next evolution in application architecture. Instead of integrating reusable components like in SOA, services are created for specific business functions in a microservices architecture.

Web or mobile applications are composed of a suite of independent services such as user management, user roles, ecommerce cart, inventory, shipping, search engine, social media logins, and more. The services are independent of each other, which means that each service can be written in a different programming language, use a different framework, and use different databases.

They can also be scaled and revised independently of another service. Unlike SOA, which uses open standards and communicates using an ESB messaging protocol to communicate between themselves, microservices use lightweight HyperText Transfer Protocol (HTTP), representational state transfer (REST), or application programming interfaces (APIs) for communicating between themselves (see Figure 1-1).

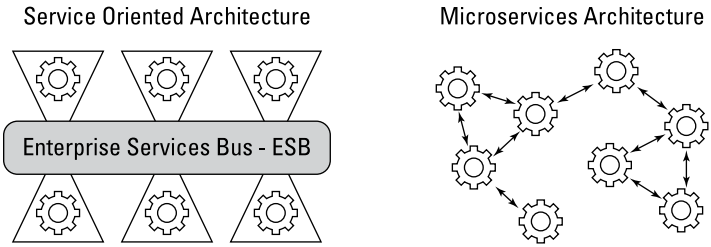

FIGURE 1-1: Comparing communications in SOA and microservices.

Table 1-1 summarizes the key differences between SOA and microservices.

TABLE 1-1 Differences between SOA and Microservices

SOA	Microservices
Maximizes component reusability	Decouples the monolithic app into services
DevOps and continuous delivery (CD) are used, but not mainstream	DevOps, continuous integration, and continuous delivery (CI/CD) are used
Focused on business functionality reuse	Focused on creating new services
Supports multiple messaging protocols	Uses lightweight protocols such as HTTP, REST or Thrift APIs
Use of containers is less common	Prevalent use of containers
SOA services share data storage	Each microservice can have independent data storage
Common platform for all services deployed on it	Application servers are not typically used; instead, cloud platforms are commonly used

To achieve even greater feature velocity and faster production than SOA, a microservices architecture breaks up entire monoliths into smaller units with smaller team sizes, each with independent workflows, freedom to choose the appropriate architecture components, and different governance models. Whereas an SOA architecture breaks some of the application into smaller parts that are published and consumed via an API over the network, microservices take the same concept a step further. In a microservices architecture, it's not just parts of the application that are broken up; the entire application is broken up into loosely coupled services that can be developed, maintained, and run independently from the other parts. After the app is rearchitected, all the parts communicate over the network via APIs, in exactly the same manner as SOA.

There are many benefits of a microservices architecture over SOA, including the following:

- **Freedom to choose the right technologies for the right job:** In both SOA and microservices architectures, services can be developed in different programming languages and tools. Teams using either architecture can choose the most appropriate technology for the problem they're trying to solve. However, in SOA, each team needs to know about the common communication mechanism. With microservices, the services can operate and be deployed independently of other services. It's far easier to deploy new services and scale independently. In the case of a monolith, the architectural choices are fixed and the teams cannot choose programming languages and tools. They're stuck using the same IDE and the same framework.
- **Independent workflow and full autonomy:** SOA encourages sharing of components, whereas microservices focus on independent services with minimal dependencies. Microservices give your team control over the full stack they require to deliver a feature. The benefit of this separation is a reduction in the amount of coordination required with other teams. The workflow is independent from other teams, and the risk of negatively affecting other teams is minimized. As SOA relies on multiple services to fulfill a business request, systems built on SOA are likely to be slower than microservices and revised less frequently than microservices.

- **Independent scalability:** With microservices, you can scale each service according to its workload demands and performance needs. However, in the case of a monolithic application, scaling horizontally across more servers can lead to overprovisioning and underutilization when the workload demand drops.
- **Easier rollback:** If each feature only requires a change to a single microservice, then that feature can be rolled back without affecting the workflows of other teams. Microservices can also improve security by reducing the attack surface of the application and increase reliability by reducing the possibility of an outage due to a single fault.
- **Ability to release independently and more frequently:** Microservices limit the scope of changes and reduce the amount of coordination required between teams. Teams can release according to their own schedules instead of being bound to the single cadence of a monolith. A showstopper bug found in a monolith holds back the whole release, whereas in microservices the individual services can be released independently.
- **Independent communication:** In microservices, services communicate independently. If one of the services has a memory fault, then only that microservice is affected. All the other microservices will continue to handle requests without interruptions.
- **Easier upgrade path:** Upgrading the framework used by a large application is nontrivial and can be risky, even under the best of conditions. Upgrades are much harder when you need to coordinate sweeping, interlinked changes across multiple teams. Smaller, independent services give you the option of only upgrading the services that require the update or allowing you to perform a rolling upgrade for one service at a time and/or one team at a time.
- **Protection from change:** Monoliths have lines of code that may be unchanged for months, or even years. However, some parts of the code require more maintenance than other parts. The ability to separate the parts of the code that frequently churn from the parts of the code that don't change can reduce the risk of accidental regressions.

- **Defined scope:** An independent service is much easier to define and understand, especially if the service is maintained by the same team. Even if the service needs to be refactored down the road, the same team can keep the design consistent. A monolith may become inconsistent as the architecture evolves, due to decisions made by the different teams that maintain the application over time.

Seeing That Distributed Applications Require a Reliable Network

The network is the glue that brings microservices together to deliver an app. As you may imagine, microservices communicate significantly over the network — it's the connection between your app's microservices. Enterprise networks have traditionally been designed and built to provide redundancy, but you add a network dependency to your application logic, the potential for network — and thus, application — failures grows proportionally with the number of connections that your applications depend upon.

Some web companies have had to develop special frameworks and libraries to alleviate some of the challenges of an unreliable network. For example, Netflix created projects like Ribbon, Hystrix, and Eureka to solve these types of problems. Twitter, Facebook, and Google have all undertaken similar projects. However, adding networking stacks into an app introduces additional challenges. For example, when the framework is updated, the applications also need to be updated.

Looking at How Kubernetes and Microservices Work Together

The advent of Linux containers and container orchestration from Kubernetes have fundamentally transformed the way applications are developed and vastly improved deployment velocity by focusing on orchestrating containers through each stage of a well-automated pipeline. Individual services can be packaged as containers running in Kubernetes pods, complete with their

respective language runtime, such as a Java virtual machine (JVM) or Python, and all the necessary dependencies, such as language-specific frameworks (such as Spring or Express) and libraries (such as Java ARchives [JARs] or NPMs [originally Node Package Managers]). Development teams can now manage their pipelines independent of the language or framework that runs inside the container. Kubernetes provides application availability, elasticity, and overall management of complex distributed, polyglot applications.


REMEMBER

Microservices and Kubernetes work together hand-in-hand. Kubernetes provides round-robin load balancing but does not get involved with how each of the application components, running in its own pod, interacts with the others.

The tooling and infrastructure required to quickly deploy distributed applications is rapidly maturing, but it's still missing a set of capabilities to describe how services interact. A service mesh is the missing link for infrastructure architects and developers.

IN THIS CHAPTER

- Understanding microservices architecture challenges
- Addressing microservices challenges with a service mesh

Chapter 2
Service Mesh: A New Paradigm

In this chapter, you learn about the challenges introduced by a microservices architecture and how a service mesh solves these challenges.

Identifying Challenges in Microservices Architectures

The benefits of moving to a microservices architecture (discussed in Chapter 1) are well understood. But breaking an application into smaller components also introduces new challenges and complexities, including the following (see Figure 2-1):

- Many endpoints to monitor, scale, and secure
- Inconsistent operational visibility and remediation
- Disjointed security, auditing, and compliance
- Polyglot fragmented libraries and code bloat

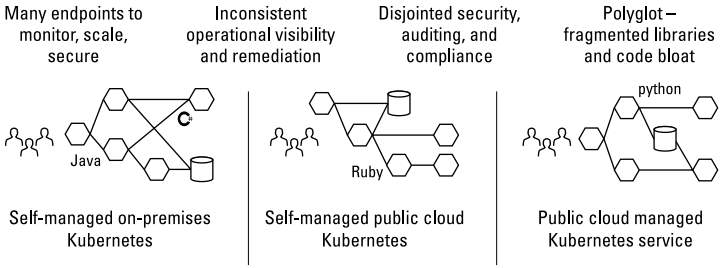

FIGURE 2-1: Microservices introduce lots of new opportunities — and challenges.

In a microservices architecture, latency is introduced between services in the service chain, which will affect the entire application and the user experience.

Another challenge is troubleshooting and identifying the root cause of issues when they occur. This is usually done using tracing, which allows you to understand the communication and transaction flow of the applications. But in a distributed microservices-based application, where do you look for the root cause when issues arise? How can you know which log to search for the root cause of a problem? Applications built in microservices archtectures are composed of many different services, often written in different coding languages. This is known as a *polyglot application.* In such an environment, how do you troubleshoot issues in different development languages?

Before service mesh, there were various ways to address these challenges — and there still are. For example, language-specific libraries address issues such as encryption, service discovery, and traffic management; likewise, an application programming interface (API) gateway — which sits in the application path — provides similar capabilities. But these projects have some challenges of their own. Libraries, for example, are not language and platform agnostic, but are instead mostly focused on a specific development language, like Java. A centralized solution such an API gateway or northside proxies does not provide a full solution.

Introducing Service Mesh

A service mesh is an abstraction layer that takes care of the following:

- Service-to-service communication (service discovery and encryption)
- Observability (monitoring and tracing)
- Resiliency (circuit breakers and retries)

Microservices should be all about business logic. Developers should be able to focus their development work on coding the "special sauce" in their apps for the business. For example, if a microservice is a web server, it needs to fetch data and present it. But a lot of maintenance or "housecleaning" work is needed (see Figure 2-2) — things like service discovery and connection details. A microservice that needs to communicate with other services needs to know how to find them, how to define the connection details, and whether the connection is encrypted. You also need to tell the service what to do in case of connection errors or failures: Should it retry? If so, how many times? Also, you need a way to detect latency and define where to send latency information and what to do if latency is too high.

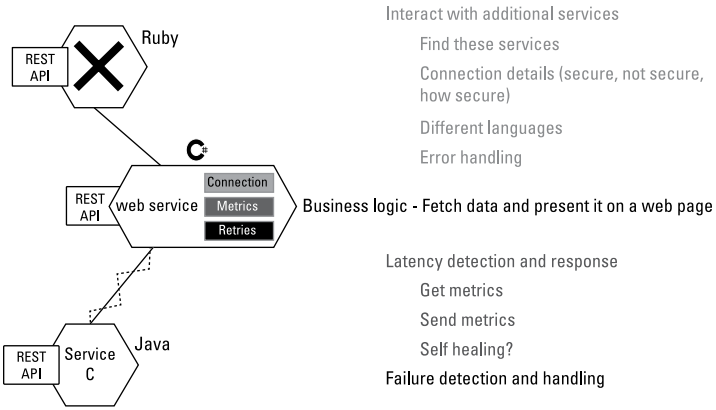

FIGURE 2-2: Necessary maintenance functions in a microservices architecture.

TIP

Business logic is the purpose of the service, such as a web server that fetches data and presents it as a web page.

However, none of this "housecleaning" work differentiates the business, and it can't be considered business logic. With a service mesh, you can abstract these functions to a separate entity called a *proxy.* The proxy sits in front of each microservice and all communications are passed through it. The proxy is responsible for connection details, traffic management, error and failure handling, and collecting metrics for observability purposes. When proxies talk with other proxies, you get a service mesh.

In Kubernetes, the proxy is implemented as a "sidecar." The containers run in pods, where the sidecars act as "helper containers" to the main container, which runs the business logic. In a service mesh, the proxy runs as a sidecar (see Figure 2-3).

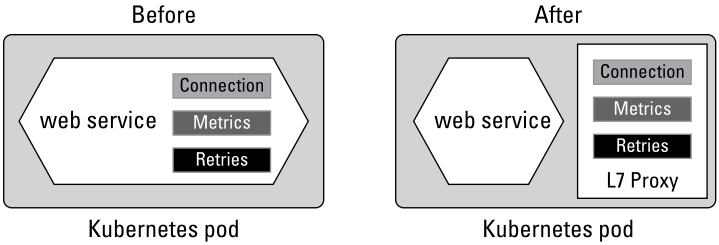

FIGURE 2-3: A Layer 7 proxy, or "sidecar," in a Kubernetes pod.

TIP

Service mesh projects like Istio, Linkerd, Consul, Kong, and Cilium have gained momentum over the past several years. Istio, which was initiated by Google, currently has the most momentum in the open-source, cloud-native space, with more than 19,500 stars, 6,400 commits, and 380 contributors in the Istio GitHub repository and numerous companies building service mesh solutions that involve Istio, including VMware, Avi Networks, Cisco, OpenShift, NGINX, Rancher, Tufin Orca, Tigera, Twistlock, and Aspen Mesh.

Projects like Istio, LinkerD, and others add a control plane to manage the sidecar proxies. In Istio, for example, you can apply a configuration to the mesh with YAML ("YAML Ain't Markup Language") files, using a declarative API. This means you can provide an end-state definition, rather than a series of steps.

This makes the configuration of the communication nonproprietary and easy to handle. The "housekeeping" code required to handle communications, security, and retries — which is nondifferentiating to the business — is abstracted using the sidecar proxy, and developers can instead focus on writing business logic that creates business value.

REMEMBER

Developer time is very expensive — if not the most expensive time in the business — so moving to a configurable abstraction layer can save a lot of time — and money — for the business.

Istio's architecture is divided into two levels: the data plane, which is based on Envoy (most service mesh projects that use a sidecar architecture utilize Envoy), and a control plane to manage the proxies. With Istio, you inject the proxies into all the Kubernetes pods in the mesh.

Istio also uses Envoy proxies to provide access in and out of the mesh, thereby providing a very clear demarcation line for the entry and exit points of the mesh. Traffic coming into the mesh or leaving it via an Envoy proxy that acts as an ingress or egress gateway (or both), where traffic originates outside the service mesh and goes via the egress gateway will return via the ingress gateway.

For example, a service running inside the service mesh (for example, Service B) can originate traffic to external services (for example, YouTube) internally (see Figure 2-4). You can easily configure the service mesh to handle the way this traffic leaves the service mesh via the egress gateway using a declarative definition (that is, an intended state).

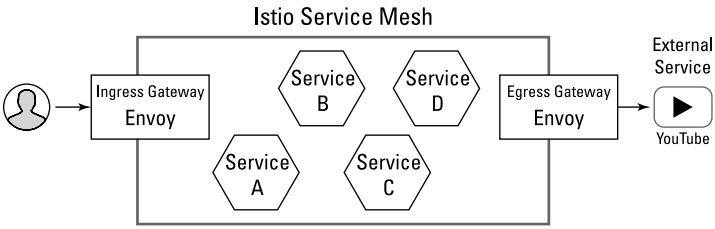

FIGURE 2-4: Envoy proxies provide entry and exit points in the service mesh.

The Istio control plane is composed of three main components (see Figure 2-5):

- **Pilot:** The pilot is responsible for "piloting" the mesh and programming the envoy proxies with traffic management, security, and more.
- **Mixer:** The mixer acts as the aggregation point for all telemetry in the mesh. The Envoy traffic is sent to the mixer, where external tools can interact with it for observability and monitoring purposes.
- **Citadel:** The citadel is the certificate authority (CA) for the mesh. It's responsible for providing the certificate (identity) for all the microservices and is a key element in delivering mutual transport layer security (mTLS) authentication and encryption for traffic in flight in the mesh.

FIGURE 2-5: The Istio control plane consists of the pilot, mixer, and citadel.

IN THIS CHAPTER

- Managing traffic
- Enabling observability
- Ensuring security

Chapter 3
Service Mesh Use Cases

Istio use cases tend to fall into three main areas: traffic management, observability, and security. In this chapter, you explore Istio service mesh use cases.

Traffic Management

The ability of Istio (discussed in Chapter 2) to enforce policy at any point in the network enables a number of very useful traffic control features, including traffic splitting, rate limiting, circuit breaking, and programmable rollouts such as canary deployments.

The power of Istio lies in its ability to express configurations at Layer 7, the Application Layer (see Figure 3-1). Decoupling traffic flow from infrastructure in this way makes it an application concern under the purview of the app owners. App owners, DevOps engineers, and SREs understand the business context and the needs of the app users, and they're ultimately responsible for the releases, application reliability, and customer experience, as well as ensuring the platform can handle the demands of the apps.

Yet another advantage of Istio is that the developers don't necessarily have to understand the internals of Kubernetes to be able to configure traffic rules at an application level.

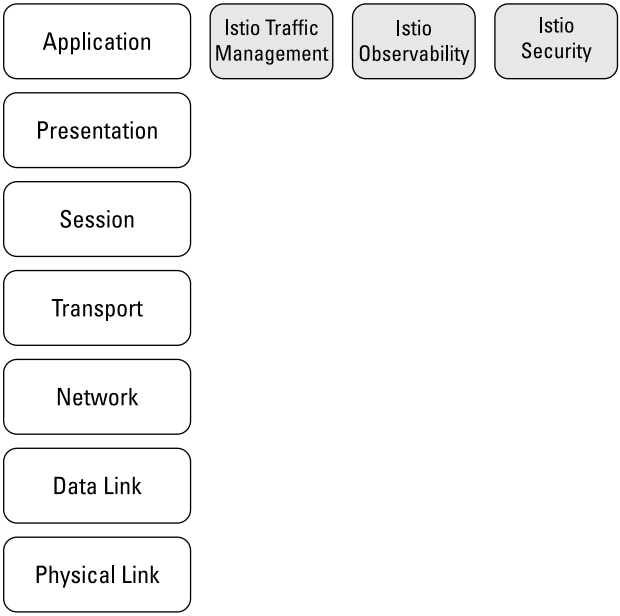

FIGURE 3-1: Istio functionality is implemented at Layer 7 (the Application Layer) of the Open Systems Interconnection (OSI) stack.

Network reliability

Istio is able to handle network failures in the service mesh automatically and transparently, by retrying failed requests within the parameters set up by the app owners (see Figure 3-2). They can define the timeout budgets for retries and jitter thresholds to restrict the impact of the increased traffic caused by retries on upstream services.

WARNING

If there's a 503 Service Unavailable error because the server is completely unavailable due to scheduled maintenance, the app operator would need to make sure that the application has resiliency built in to handle the 503 errors returned from Envoy when it stops retrying.

TIP

One of the main benefits of offloading the retries to the proxy is that you can define the application resiliency settings between services independently of the programming language, because the configuration layer is language agnostic.

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: recommendation-v2-retry
spec:
  destination:
    namespace: tutorial
    name: recommendation
  precedence: 3
  route:
  - labels:
      version: v2
  httpReqRetries:
    simpleRetry:
      perTryTimeout: 2s
      attempts: 3
```

FIGURE 3-2: App owners can define parameters such as failed request retries.

Circuit breaking

A sudden and intense spike in traffic, such as a distributed denial of service (DDoS) attack can quickly overload a network. Microservices-based distributed applications would struggle to process the backlog of requests coming all at once due to the sudden spike in traffic.

In Istio, an app owner can set a threshold when a service is generating 503 errors to remove it from the load balancer pool. New requests coming in will not be routed to the unhealthy service (see Figure 3-3). This is known as *circuit breaking*, and it's configured by defining a connection pool of concurrent Transmission Control Protocol (TCP) connections and pending HyperText Transfer Protocol (HTTP) requests.

Figure 3-4 shows an example of a YAML ("YAML Ain't Markup Language") configuration file for Istio's circuit-breaking features.


TIP

The circuit-breaking functionality in Istio is similar to Kubernetes liveness and readiness checks where you can define thresholds for load balancer ejection and readmission.

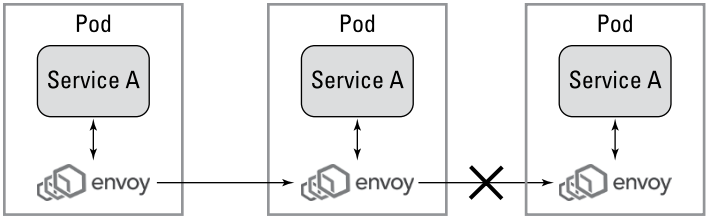

FIGURE 3-3: Traffic is not routed to the unhealthy service.

```
apiVersion: config.istio.io/v1alpha2
kind: DestinationPolicy
metadata:
  name: bookinfo-circuit-breaker
spec:
  destination:
    name: details
    labels:
      version: v1
  circuitBreaker:
    simpleCb:
      # Maximum number of connections on destination backend
      maxConnections: 1
      # Maximum number of pending requests to destination backend
      httpMaxPendingRequests: 1
      # Minimum time circuit will be opened
      sleepWindow: 3m
      # Time between ejection sweep analysis
      httpDetectionInterval: 1s
      # Maximum percentage of hosts to eject if circuit is triggered
      httpMaxEjectionPercent: 100
      # Number of 5XX codes before circuit should be opened
      httpConsecutiveErrors: 1
      # Max number of requests per connection to a backend
      httpMaxRequestsPerConnection: 1
```

FIGURE 3-4: An example of a circuit breaker configuration YAML file for Istio.

Rate limiting

Related to circuit breaking is rate limiting, a feature in Istio that allows you to enforce limits on the rate of requests that match certain criteria. This feature can be used to ensure that certain requests are not overused. This is analogous to the public

application programming interface (API) service, which will throttle you back when you exceed a predefined rate of requests.

Defining rate limits involves specifying which parameters to count, their maximum values, and the window of time in which to enforce the limit. These counts need to be tracked centrally in the cluster to ensure they aren't exceeded. Rate-limiting checks, therefore, happen in the Mixer on the data path, rather than in Envoy. The Mixer can store these counts in memory (not recommended in production) or in Redis.


TIP

Careful configuration of rate limits will ensure that all users have fair access to Kubernetes resources.

A/B testing

Istio's traffic routing can be useful for A/B testing. When testing web applications, the app owners can test new features by sending a subset of customer traffic to the instances hosting the new features. They can then observe the telemetry and gain insights into the user interactions — whether the users prefer one feature over another, or prefer an implementation of one service over another.

Istio can be configured to direct traffic based on parameters such as percentage weight, cookie value, query parameter, HTTP headers, and so on.

A common use case for microservices A/B testing may include trying out new features for a subset of users or geographical regions, or testing an update on a reduced scale before a complete rollout.


TIP

It's always possible to roll back to a stable release from the new version if too many errors are detected. However, if testing new features via A/B testing can negatively impact a running system, the recommended approach would be to perform canary releases instead.

Canary releases

Canary releases could be considered a special case of A/B testing (discussed in the preceding section), in which the rollout happens much more gradually.

A canary release begins with a "dark" deployment of the new service version, in which the new service receives no traffic. If the service is observed to start healthy, a small percentage of traffic (for example, 1 percent) is directed to it (see Figure 3-5). Errors are continually monitored as continued health is rewarded with increased traffic, until the new service is receiving 100 percent of the traffic. The old instances can then be gracefully shut down.

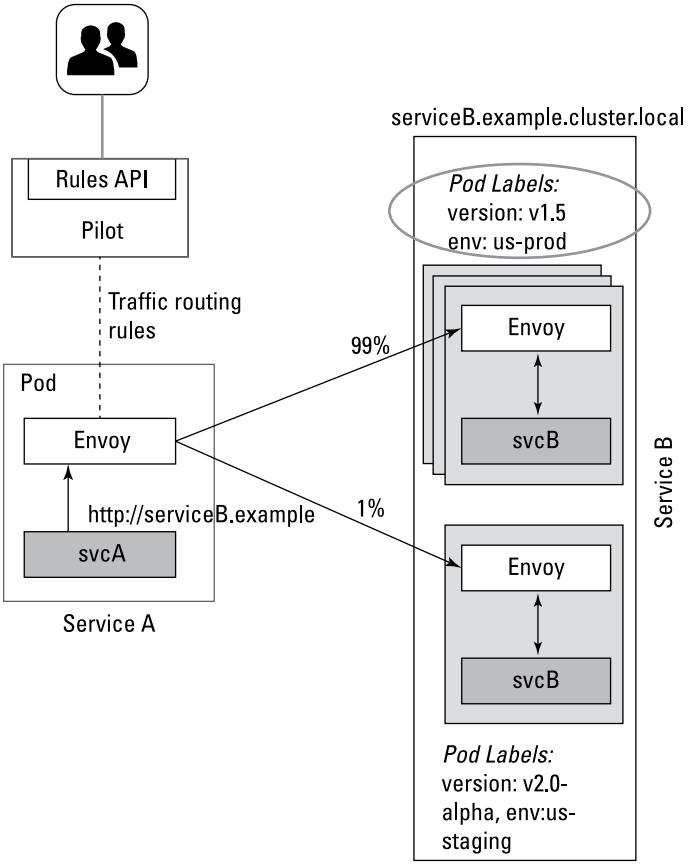

FIGURE 3-5: A canary release example in which 99 percent of the traffic is routed to version 1.5 of the software and 1 percent of the traffic is routed to version 2.0 of the software.

Istio traffic routing configuration can be used to perform canary releases by programmatically adjusting the relative weighting of traffic between service versions. Users would have to write a control loop to observe service health and adjust the amount of traffic given to the new service. Successful canary rollouts depend on being able to monitor application health.

Figure 3-6 shows an example of a YAML configuration file for Istio's traffic-splitting feature.

```
// A simple traffic splitting rule

destination: serviceB.example.cluster.local
match:
  source: serviceA.example.cluster.local
route:
- tags:
    version: v1.5
    env: us-prod
  weight: 99
- tags:
    version: v2.0-alpha
    env: us-staging
  weight: 1
```

FIGURE 3-6: An example of a traffic-splitting rule in YAML.

Traffic steering

Traffic-steering rules enable app owners to control where incoming traffic will be sent based on application or web attributes such as authentication (Jason to service A and Linda to service B), location (United Kingdom to service A and United States to service B), device (watch to service A and mobile to service B), or anything else that is passed in the application header. In Figure 3-7, Android users are authenticating with service B located on Pod 3, whereas iPhone users are directed to authenticate with service B located on Pod 4.

Traffic steering has been done in previous architectures by hard-coding the rules into the application and using software libraries. It's much easier to do traffic steering in Istio because of the abstraction Istio offers and because it's easy to change the parameters in a YAML file.

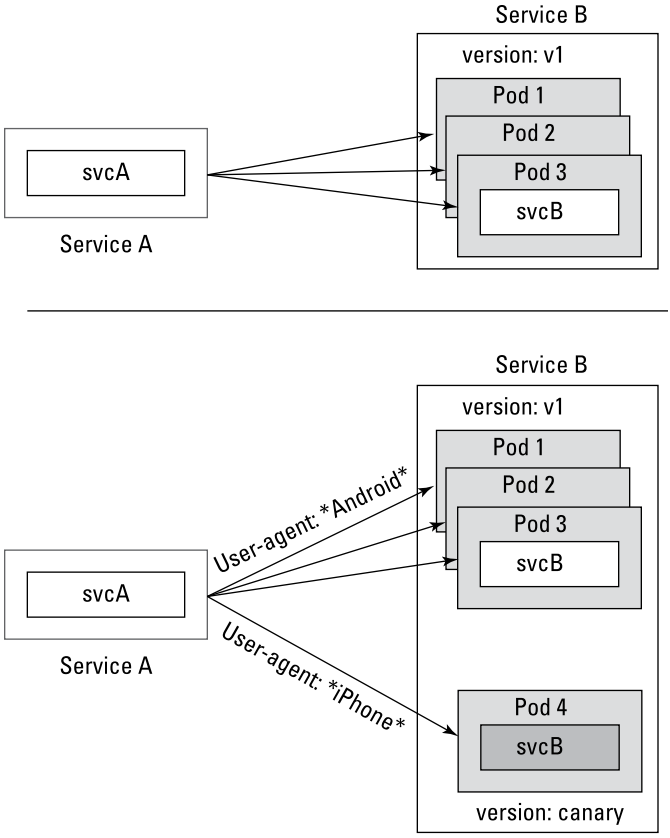

FIGURE 3-7: A traffic-steering example.

For example, if an app owner wants to direct traffic from one location to the same location where the application is hosted, the app owner can simply implement the rules in the YAML file. For example, U.S. traffic is steered to the app hosted in North America and European Union (EU) traffic is steered to the app hosted in the EU.

Figure 3-8 shows an example of a YAML configuration file for Istio's traffic-steering feature.

```
// Content-based traffic steering rule

destination: serviceB.example.cluster.local
match:
  httpHeaders:
    user-agent:
      regex: ^(.*?;)?(iPhone)(;.*)?$
precedence: 2
route:
- tags:
    version: canary
```

FIGURE 3-8: A YAML file showing how to configure the traffic-steering rule for two types of mobile users, Android and iPhone.

TIP

Developers need to make sure that the information on which the traffic decision is made is inserted in the HTTP headers.

Egress control

By default, Istio does not permit connections to services outside the mesh. However, Istio provides two in-mesh ways to define outbound connections to a permitted Uniform Resource Locator (URL). Controlling egress using URLs provides an advantage over legacy firewalling techniques using static Internet Protocol (IP) ranges, because cloud-native apps are increasingly hosted using IPs that dynamically change.

One example use case is a microservice app that performs read-write actions on a database. The database is hosted on the public cloud outside the service mesh without a static IP. The egress needs to be configured so that the microservice app can connect with the database outside the mesh. One way would be to whitelist the entire IP range of the cloud provider, but this approach creates potential security risks.

A more secure approach would be to create a dedicated Istio proxy (also known as an egress gateway) through which all egress traffic must pass in order to exit from the mesh. Using the gateway provides additional controls, and you can use the Kubernetes network policy to restrict all egress from the cluster except for the traffic from the egress gateway.

Observability

Although service discovery is part of the load-balancing functionality of Istio as the data is coming from the proxies, observability is a category on its own. Istio's Pilot component consumes information from the underlying platform service registry (that is, Kubernetes) and provides a platform-independent service discovery interface (see Figure 3-9). With this interface, tools like Kiali and VMware NSX Service Mesh can provide service observability.

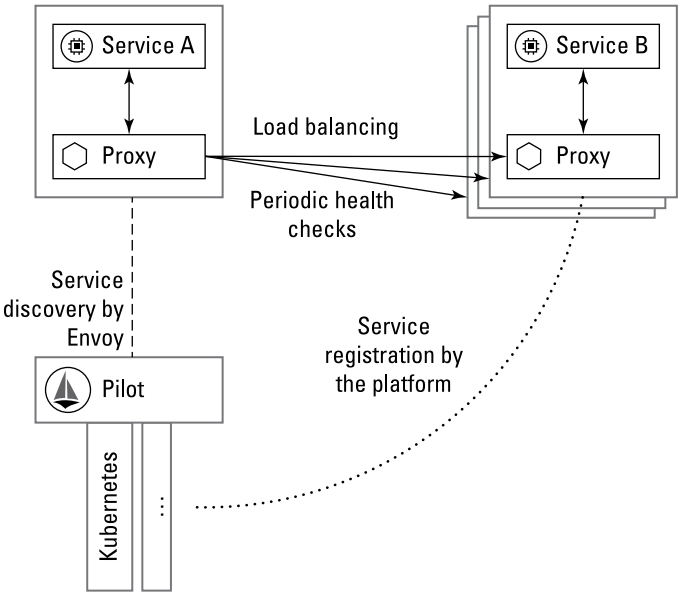

FIGURE 3-9: Istio's Pilot can discover new services and registers the new service (B) into the platform.

Metric collection

The Envoy proxies send metrics to the mixer and provide a way of monitoring and understanding the behavior of the services (see Figure 3-10).

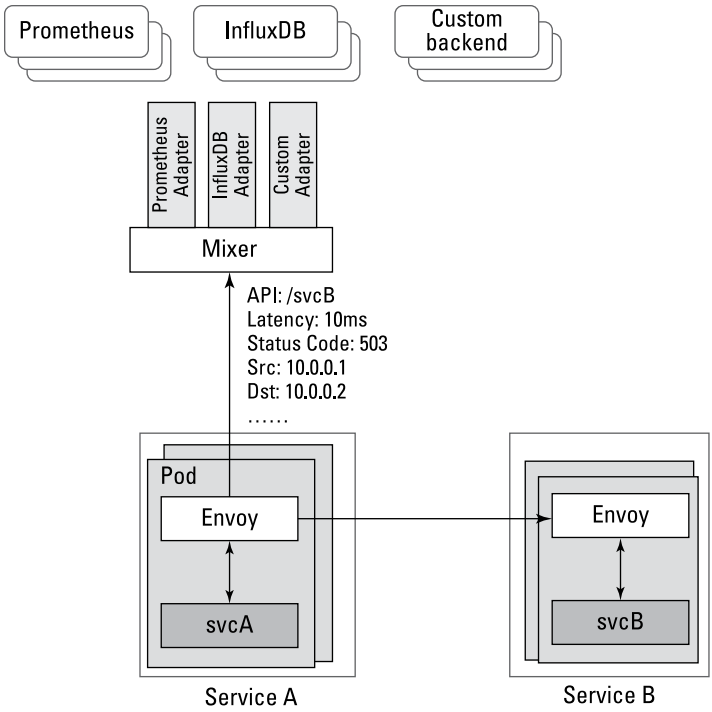

FIGURE 3-10: Envoy proxies send metrics to the mixer in Istio.

The metrics generated by the Envoy proxies allow app owners to understand app behavior such as the volume of traffic, the error rates within the traffic, and the response times for requests. The metrics also enable an understanding of how the service mesh itself is behaving and its overall health. The metrics in Istio provide this level of observability.

Like all other configuration files in the service mesh, the collection of metrics is driven by a YAML configuration file. The metrics in Istio are divided into proxy-level metrics and service-level metrics.

Distributed traces

Distributed tracing is a concept that has existed in application programming for many years, but it's particularly difficult to achieve in a distributed microservices application. Istio supports

tracing by monitoring transactions as they flow through a mesh and allows application owners to understand service dependencies and the sources of latency within the service mesh.

Distributed tracing is implemented through the Envoy proxies, which automatically generate trace spans on behalf of the application. The developer needs to configure the application to forward trace information. When tracing is required for troubleshooting, the application operator can configure a tracing tool — such as Zipkin, Jaeger, or others — to present and analyze the tracing information and identify the root cause of latency or errors in the application.

Security

Istio security use cases include mutual transport layer security (mTLS) authentication and Istio authorization.

mTLS authentication

Istio's service-to-service communication flows through the Envoy proxies. The Envoy proxies can generate mTLS tunnels between the services, and each service will have its own unique certificate providing it an identity. The certificates are managed by the Citadel component (the root certificate authority of the mesh), which is also responsible for certificate rotation — a nontrivial task for any organization, especially when done at scale (see Figure 3-11). The encryption is done at Layer 7, which means the payload is encrypted, not the traffic.

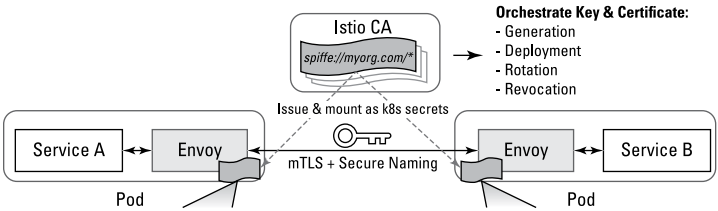

FIGURE 3-11: mTLS authentication in Istio.

There are two types of encryption policies: *restrictive,* in which the service is configured to not accept nonencrypted traffic, and *permissive,* in which the service will fall back and accept nonencrypted traffic as needed.

Istio authorization

Istio authorization is analogous to micro-segmentation at Layer 7. Micro-segmentation is sometimes considered to be synonymous with the security principle of "zero trust." Micro-segmentation means that no traffic will be allowed that is not explicitly permitted ("zero trust") by inspecting every request for access.

In software-defined networking (SDN) solutions like VMware NSX Data Center, the distributed firewall inspects the packets based on Layer 4 (Transport Layer) ports and also application signatures (Layer 7), while the enforcement is done in Layer 4. In Istio, both the control and enforcement are purely Layer 7.

Istio authorization provides namespace-level, service-level, and method-level access control for services in the Istio mesh. Istio can segment the services based on Layer 7 constructs, such as remote procedure call (RPC) level authorization and role-based access control (RBAC) with conditions.

The Istio documentation describes RPC-level authorization as follows:

> Authorization is performed at the level of individual RPCs. Specifically, it controls "who can access my bookstore service" or "who can access method `getBook` in my bookstore service." It is not designed to control access to application-specific resource instances, like access to "storage bucket X" or access to "3rd book on 2nd shelf." Today this kind of application-specific access control logic needs to be handled by the application itself.

The RBAC with conditions method applies access control based on user identity and a group of additional attributes. It's the combination of RBAC with the flexibility of attribute-based access (ABAC).

IN THIS CHAPTER

- Installing Istio and choosing a deployment model
- Enabling Istio service mesh capabilities in a Kubernetes cluster
- Looking at a configuration example

Chapter 4
Recognizing Complexity Challenges in Service Mesh

The process of installing and configuring objects, understanding the architecture, and getting true value from a service mesh takes time, and for now all the service meshes still have some sharp edges that need to be smoothed out. The technology is still on its way to being fully ready for huge production environments. In the meantime, some vendors are trying to solve this complexity and add value to the service mesh.

In this chapter, you learn how to install Istio and configure traffic shifting for a simple A/B testing scenario and the complexities in a single use case, let alone at scale.

Installing Istio

To get started with Istio, you need to install it into your Kubernetes cluster and get an external Internet Protocol (IP) address assigned to the ingress gateway.

TIP

Installing Istio has a few steps to it, which can be found at `https://istio.io/docs/setup`. There are also different deployment models described at `https://istio.io/docs/concepts/deployment-models`. These documents basically cover the way Istio can currently be deployed.

The following are the supported Kubernetes cluster deployment models:

- Cluster models
- Network models
- Control plane models

Cluster models

The single Kubernetes cluster model is the most common deployment model (see Figure 4-1). Many organizations choose this model because it provides the simplest way to use Istio and Kubernetes, and organizations usually prefer to keep their Kubernetes clusters independent from one another.

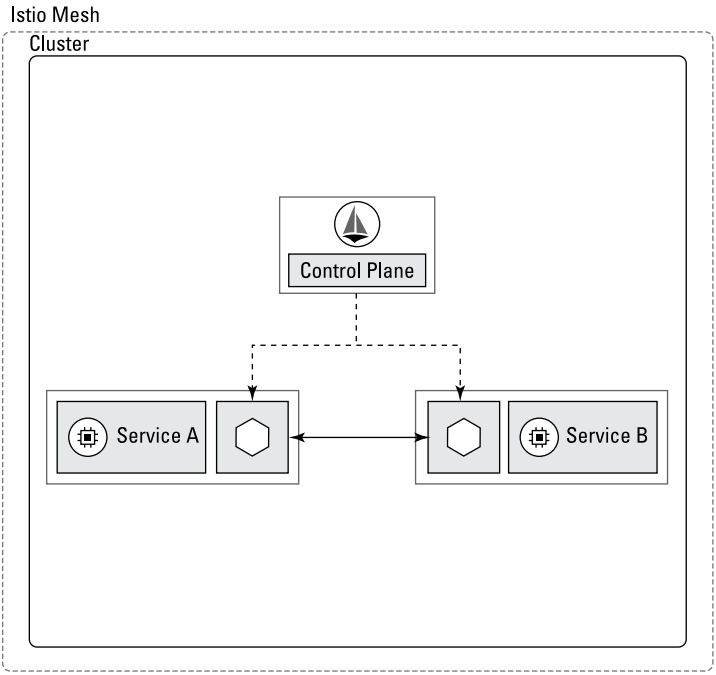

FIGURE 4-1: A service mesh with a single cluster.

The multi-cluster deployment model is less common than the single-cluster deployment model (see Figure 4-2). Stretching the Istio service mesh across Kubernetes clusters allows for multi-cluster use cases such as isolation between tenants, business continuity (high availability and disaster recovery), and location awareness, but having a single Istio control plane that is stretched across multiple Kubernetes clusters eliminates the independence of each cluster.

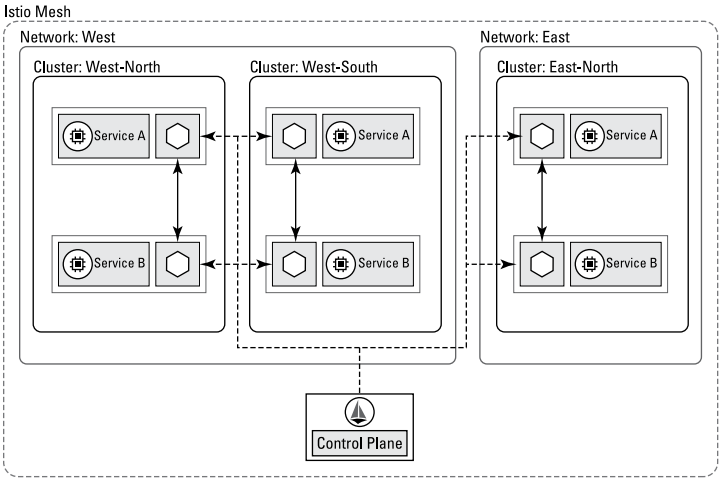

FIGURE 4-2: A service mesh with multiple clusters.

Network models

The single network model provides a true mesh where everything is connected to each other (see Figure 4-3). Although this model makes life simple for operating Istio, when deploying multiple clusters this will usually not be the case.

A multi-network model is used in a larger-scale network than a single flat network and provides support for use cases such as using overlapping IP schemes of clusters, crossing of administrative boundaries, fault tolerance, and compliance (see Figure 4-4).

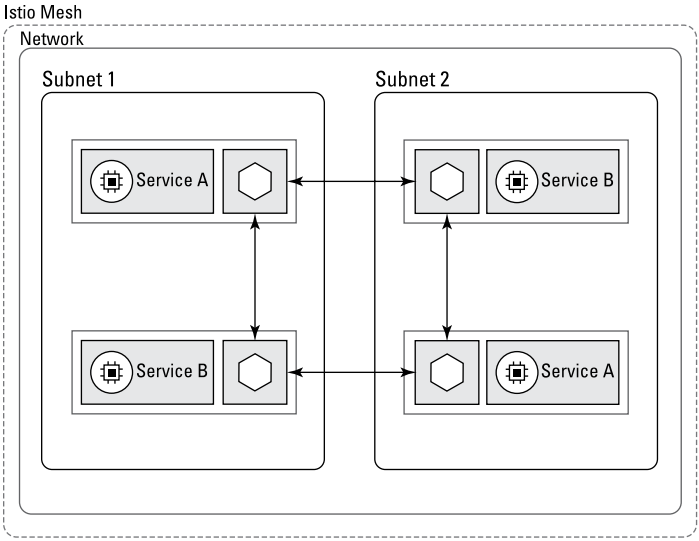

FIGURE 4-3: A service mesh with a single network.

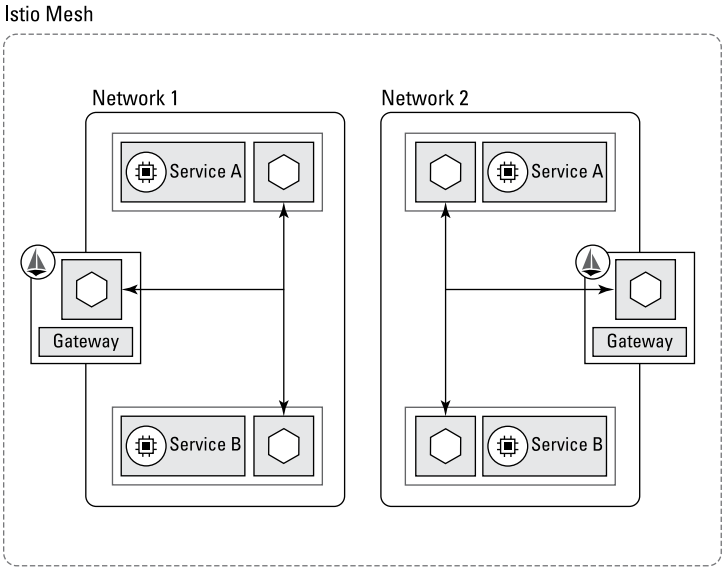

FIGURE 4-4: A service mesh with multiple networks.

Control plane models

From a control plane perspective, Istio supports multiple models as well. The single control plane is the simplest deployment model and the easiest to stand up and maintain (see Figure 4-5).

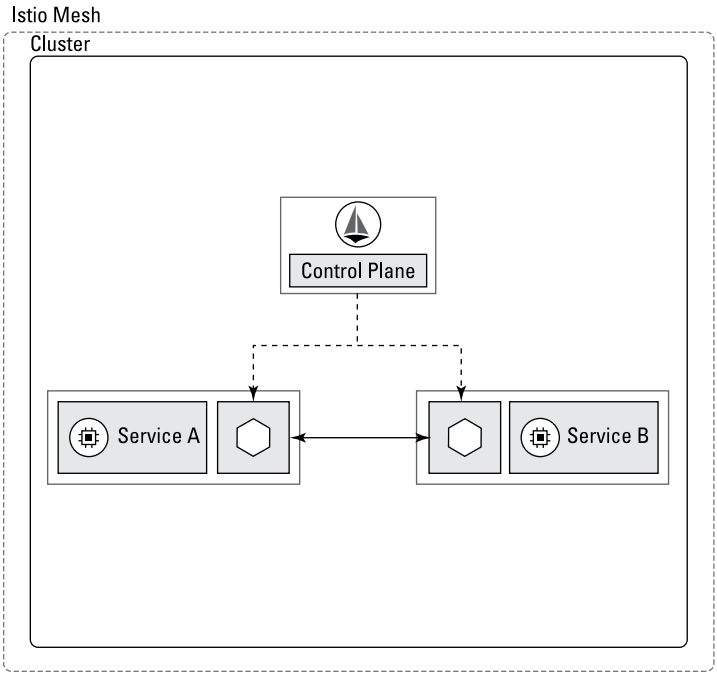

FIGURE 4-5: A service mesh with a single control plane.

A multi-cluster, shared control plane allows for multiple clusters, using a single Istio control plane (see Figure 4-6). This provides uniformity of service mesh across clusters with the simplicity of a single control plane, but at the cost of having a single point of failure.

Lastly, in a multi-cluster, multi-control plane (replicated) model (see Figure 4-7), you need to set up replication between multiple control planes and set the gateway in each mesh to allow control plane traffic between them. Setup and maintenance are manual processes, and this model also does not cover non-Istio service meshes.

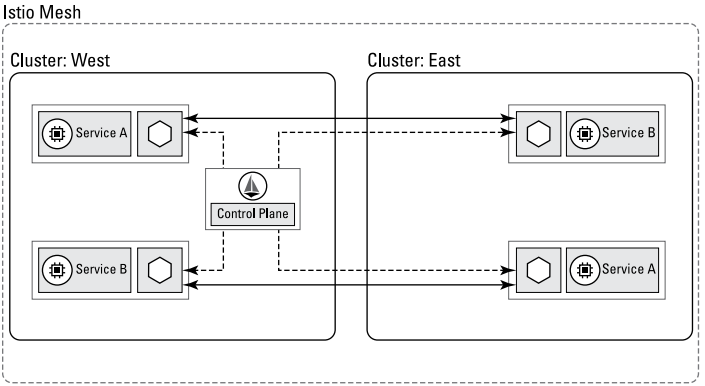

FIGURE 4-6: Two clusters sharing a single control plane.

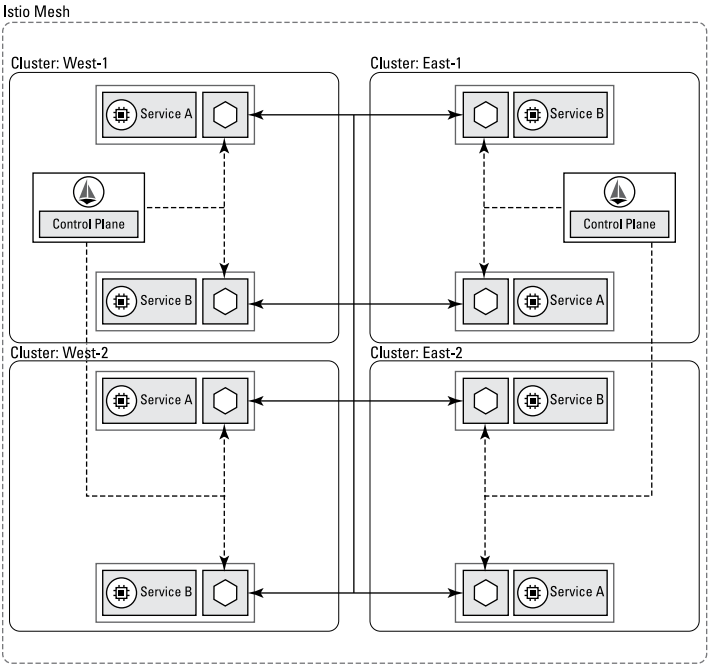

FIGURE 4-7: A service mesh with control plane instances for each region.

TIP

Detailed instructions for deploying a multi-cluster, multi-control plane (replicated) model can be found at `https://istio.io/docs/setup/install/multicluster/gateways`.

Whatever model is chosen, after Istio is deployed, you need to enable the service mesh in the Kubernetes cluster. As described in the architecture components of Istio, to enable the service mesh capabilities you have to inject the Envoy proxy into every pod that you want to be part of the mesh. You can decide if you want to inject the sidecar to all namespaces or a specific namespace by labeling the namespace.

After injecting the proxy, you need to create the different objects that structure the mesh to manage the networking within and outside the mesh.

The architecture has a clear entry point, and that's the first object created in the installation process. To create the environment to run an app inside a mesh you need to create several objects: an Istio Ingress Gateway, one or more VirtualServices, and zero or more DestinationRules (see Figure 4-8).

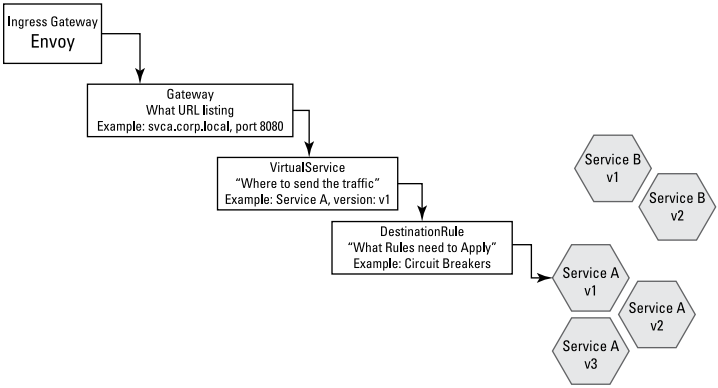

FIGURE 4-8: Objects in an Istio service mesh.

If you want to change parameters to different services inside the mesh, you have to create multiple roles inside a VirtualService, create one VirtualService per application/developer team, or create as many roles as services that need different parameters.

The complexity starts here, with the number of objects and breaking point to the base structure of the mesh. Even though the creation and management will probably be automated and orchestrated, the number of failing points and the way that you manage those points, currently by using declaration files, makes operation of Istio difficult in large environments.

If there are multiple Kubernetes clusters, operators would have to manage many Istio deployments independently and there would be potentially hundreds of different declaration file configuration files across the isolated service meshes.

A Traffic-Shifting Configuration Example

Let's use the example of traffic shifting for a single service version in a single app to demonstrate the complexity and failing points that exist.

To progressively upgrade a service and avoid outages, a DevOps engineer may want to route a portion of the traffic — say, 10 percent — to a specific new version of a service and the remaining 90 percent of traffic to an older version.

In the example in Figure 4-9, we use the common Bookinfo app from the Istio repository located at `https://istio.io/docs/examples/bookinfo`.

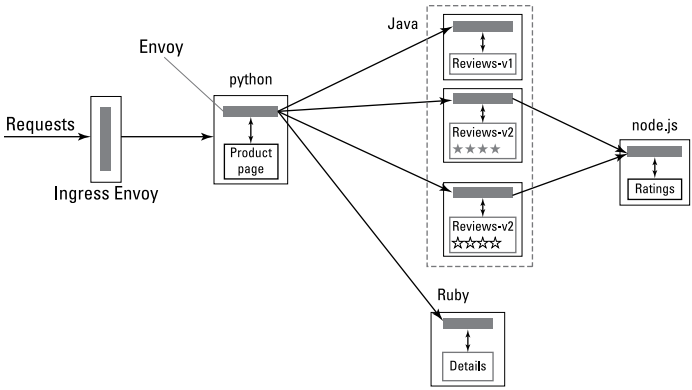

FIGURE 4-9: The Bookinfo application.

After the installation, the first Ingress controller of the mesh is created and an external IP is assigned to the service as the entry point to everything that is being managed by Istio.

Now, the DevOps engineer wants to route the traffic to a specific version of the backend Review service. First, you need to configure an internal Istio gateway (see Figure 4-10). This is a second gateway that the first Ingress controller will route traffic to.

```
1. apiVersion: networking.istio.io/v1alpha3
2. kind: Gateway
3. metadata:
4.   name: bookinfo-gateway
5. spec:
6.   selector:
7.     istio: ingressgateway # use istio default cont
   roller
8.   servers:
9.   - port:
10.      number: 80
11.      name: http
12.      protocol: HTTP
13.    hosts:
14.    - “*”
```

FIGURE 4-10: Configuring an internal gateway for the Bookinfo app.

The gateway will route the traffic into a VirtualService. You need to create a general VirtualService that will route traffic to the front end of the app so that you can browse the app (see Figure 4-11).

After creating the basic structure of the networking for your app, you can leverage the Layer 7 management capabilities and create an additional VirtualService to control the internal network inside the mesh and route some of the traffic to a specific version of a specific service. In the example, 10 percent of the traffic will be routed to v2 while all other traffic will still go to v1 (see Figure 4-12).

Figure 4-13 shows an example of the configuration file for the VirtualService you need to route part of the traffic. If you multiply this configuration file by the amount of changes you need to make, you see that you end up with a lot of failing points.

```
1. apiVersion: networking.istio.io/v1alpha3
2. kind: VirtualService
3. metadata:
4.   name: bookinfo
5. spec:
6.   hosts:
7.   - “*”
8.   gateways:
9.   - bookinfo-gateway
10.  http:
11.  - match:
12.    - uri:
13.        exact: /productpage
14.    - uri:
15.        prefix: /static
16.    - uri:
17.        exact: /login
18.    - uri:
19.        exact: /logout
20.    - uri:
21.        prefix: /api/v1/products
22.    route:
23.    - destination:
24.        host: productpage
25.        port:
26.          number: 9080
```

FIGURE 4-11: Configuring a VirtualService to route traffic to the front end of the app.

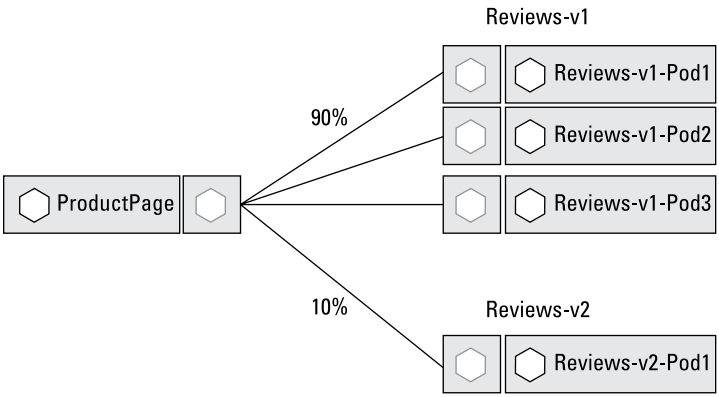

FIGURE 4-12: A traffic-steering example.

```
1. apiVersion: networking.istio.io/v1alpha3
2. kind: VirtualService
3. metadata:
4.   name: reviews
5.   ...
6. spec:
7.   hosts:
8.   - reviews
9.   http:
10.  - route:
11.    - destination:
12.        host: reviews
13.        subset: v1
14.      weight: 90
15.    - destination:
16.        host: reviews
17.        subset: v2
18.      weight: 10
```

FIGURE 4-13: A configuration file example.

To create this example of a progressive upgrade, four objects needed to be created:

- External ingress gateway (Istio entry point)
- Internal gateway (Bookinfo gateway)
- Front-end VirtualService (Bookinfo VirtualService)
- DestinationRules (Bookinfo DestinationRules)

These objects need to be maintained throughout the upgrade process while changing the weight of traffic routed — and that's for just one service of one application.

There are many parameters (such as weight, application users, destination rules, application service versions, and gateways) that you can leverage, but you must also manage them in the declaration files. Also, the one-to-one mapping between the service mesh to a Kubernetes cluster will require you to manage multiple meshes, multiplying the challenge.

In a scaled environment, creating multiple upgrades in the version cadence of a modern app is pretty much impossible. For example, Amazon is doing more than 1,000 upgrades an hour — every 11.6 seconds something is being changed in the mesh. Think about the operation behind that. And that doesn't even address scaling issues and other use cases that are complex to control and manage with service mesh.

Although service mesh solves many of the challenges in connecting and securing service-to-service communication, it also introduces an order of complexity that enterprises are left to address on their own. We explain how to overcome these challenges with VMware NSX Service Mesh in Chapter 5.

IN THIS CHAPTER

- Learning the basics of NSX Service Mesh
- Going beyond service-to-service communication
- Looking at the VMware NSX Service Mesh architecture
- Getting different service meshes to work together
- Exploring NSX Service Mesh use cases

Chapter 5
Transforming the Multi-Cloud Network with NSX Service Mesh

In this chapter, you learn about VMware's enterprise service mesh offering, NSX Service Mesh. In the same way that VMware abstracts physical compute in vSphere and physical networks in NSX Datacenter, VMware is now abstracting the cloud with NSX Service Mesh, part of the NSX suite.

Introducing NSX Service Mesh

A service mesh addresses challenges (discussed in Chapter 2) associated with a microservices architecture. However, the service mesh itself introduces new challenges (also discussed in Chapter 2) related to multiple Kubernetes cluster/multi-cloud operations and the limited scope of most service meshes today, which focus on services alone.

NSX Service Mesh solves these challenges and more. By abstracting the service mesh from the physical boundaries of a single Kubernetes cluster and a single cloud, and by extending the scope from service-to-service communication to users-to-service-to-data communication, NSX Service Mesh is able to control, secure, and operate applications, no matter where their components are deployed.

Increasing the Scope of the Mesh

All service mesh implementations are focused on bringing visibility, traffic management, and security to service-to-service communications at Layer 7. But application flows are not limited to interservice communications; users also access data (via those services). In traditional applications running in on-premises infrastructure, you manage access between application components using IP addresses on the network and permissions that are based on internal identity sources that are under your control.

In a multi-cloud, multi-platform environment, where you may not have access to the underlying infrastructure, you need to move up the stack to manage communication and access between users, services, and data, by abstracting out the underlying physical infrastructure (see Figure 5-1).

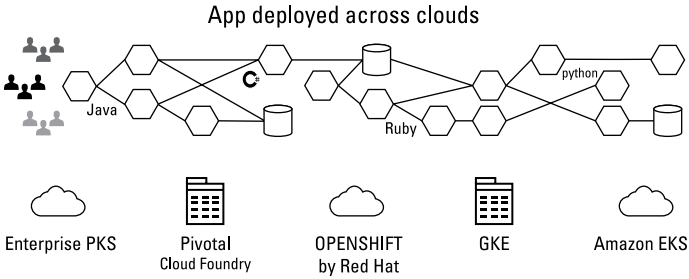

FIGURE 5-1: Modern apps are deployed across multi-cloud and hybrid environments.

VMware is leading an effort in the open-source community to extend the visibility and control of communication from just service-to-service communications to include users, services, and data. This effort includes work on the data services proxy filters, which will allow the Envoy proxy to decode the data wire protocols and SQL queries. This allows NSX Service Mesh to have visibility into the entire transaction all the way to the datastore. With this visibility, access can be logged for auditing and security purposes. When this level of visibility is achieved, policies can be used to control access to the data (see Figure 5-2).

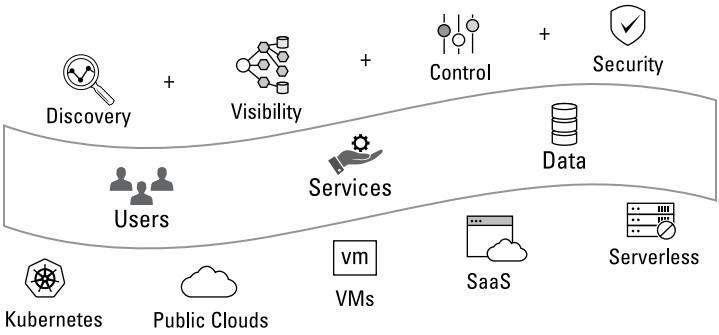

FIGURE 5-2: VMware is working with the open-source community to extend visibility and control in the service mesh to include users, services, and data.

VMware NSX Service Mesh Architecture

VMware NSX Service Mesh uses Istio as a data plane abstraction for Kubernetes workloads. When deploying Istio, it's typically tied to a single Kubernetes cluster. Istio users don't stretch it across more than one Kubernetes cluster, as most prefer each cluster to be able to operate independently from other Kubernetes clusters. For this reason, NSX Service Mesh acts as a control plane for many data plane Istio deployments managing the life cycle of Istio from onboarding to Day 2 and Day 3 operations. NSX Service Mesh only handles the life cycle of the service mesh (Istio, in this case); it does not handle the life cycle of Kubernetes. When onboarding a new cluster on NSX Service Mesh, the service will perform the deployment of a curated version of Istio, which is signed and supported by VMware. This Istio deployment is the same as

the upstream Istio in every way, but it also includes an agent that communicates with the NSX Service Mesh control plane. Istio installation is not the most intuitive, but the onboarding process of NSX Service Mesh simplifies the process significantly.

NSX Service Mesh acts as an abstraction layer on many data plane service meshes. The solution applies the same concepts of service mesh such as traffic control, security, and observability — not to a single Kubernetes cluster or cloud, but across Kubernetes clusters, clouds, and third-party service meshes.

The architecture is constructed of a local Istio data plane with its own local control plane and a central control plane, which is the NSX Service Mesh service (see Figure 5-3).

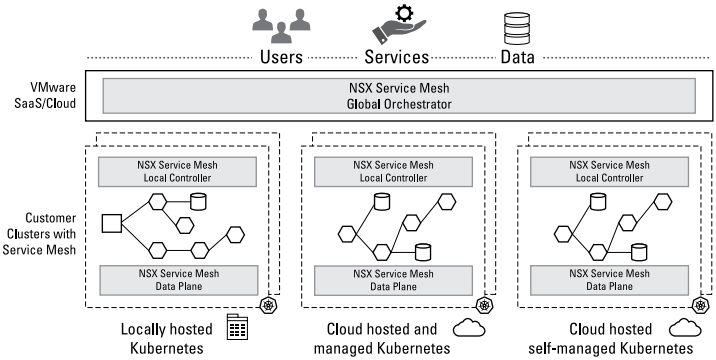

FIGURE 5-3: The NSX Service Mesh architecture.

Before going into the underlying technology of NSX Service Mesh, it's important to first understand how this architecture is logically managed.

Introducing Global Namespaces

Global namespaces are the primary management construct within NSX Service Mesh and one of its main differentiation points. Kubernetes provides service discovery and scheduling, and service mesh does this in a fully distributed way, but what about scenarios in which there is more than one Kubernetes cluster where services are running?

There are many reasons and use cases for deploying multiple Kubernetes clusters rather than a single huge cluster, including the following:

- **The difficulty of updating large Kubernetes clusters:** You need to coordinate more activities between the applications and the tenant — this complexity is the reason many organizations are moving from a monolith to microservices in the first place.
- **Isolation between multiple tenants:** In Kubernetes, the tenancy construct is the namespace, which is a folder (a binder, if you will) of logical constructs that are part of the same app or tenant. However, namespaces are not a good tenancy model because they don't provide any isolation between tenants. If true multi-tenancy is required, you may need to utilize multiple clusters.
- **High availability:** You may also want to distribute your application components on multiple clusters for high-availability purposes. In this case, you could run your application on multiple Kubernetes clusters in the same region and have a local load balancer between them to provide high availability, or you could deploy multiple Kubernetes clusters across regions with a global load balancer for disaster recovery and disaster avoidance.
- **Separating stateful and stateless services:** Data in stateful services is handled differently from data in stateless services and may, therefore, need to be separated in multiple Kubernetes clusters.

Istio supports multi-cluster Kubernetes, but most deployments have a one-to-one relationship with the Kubernetes cluster, because organizations want to keep their clusters independent from one another. With NSX Service Mesh, you can create resource groups, such as users in user groups, data in data groups, and services in service groups. The system can do this for you automatically based on user-defined criteria. By arranging these objects in groups, you can then create a "virtual sandbox" for an app, which includes all its components without regard to where these components reside — whether in multiple Kubernetes clusters, sites, or clouds. This is the global namespace (GNS), which is like a namespace in Kubernetes (see Figure 5-4), but instead

of being tied to a Kubernetes cluster, VMware NSX Service Mesh elevates the GNS from the physical world. Each NSX Service Mesh GNS manages its own service discovery, observability, encryption, policies, and service-level agreements (SLAs) (see Figure 5-5).

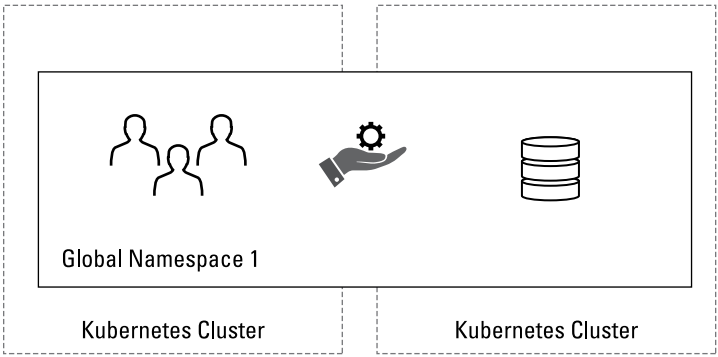

FIGURE 5-4: Global namespaces in Kubernetes clusters.

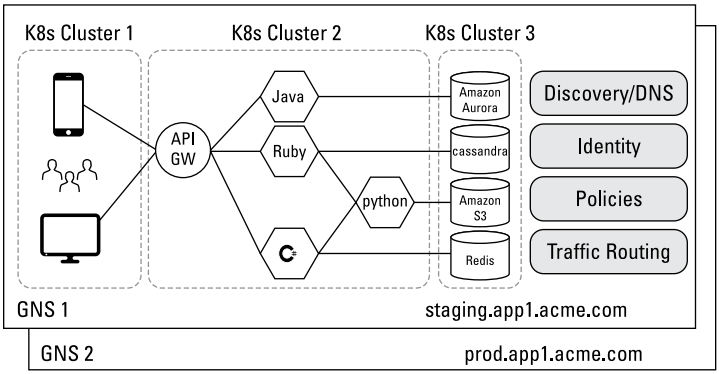

FIGURE 5-5: Global namespaces in NSX Service Mesh.

Federation and Intra-Service Mesh Interoperability

NSX Service Mesh has a layered architecture with a federated approach. The layers of the solution include the following:

- **The service itself:** This is the top control plane for the entire service mesh.

» **The control plane:** The NSX Service Mesh is built as a Software as a Service (SaaS) offering (an on-premises version will be available in the future). This service communicates and manages the underlying data plane service mesh components.

This layered architecture is fine for a service mesh under NSX control, but that isn't always the case. There may be application components in another service mesh, such as Google Anthos, HashiCorp Consul Connect, or some other service mesh (see Figure 5-6). Also, Istio is a great data plane service mesh and has a lot of momentum in the Kubernetes space, but it isn't the only one and it doesn't cover all types of workloads. There are also service meshes such as Cilium, Consul by HashiCorp, Linkerd, and so on. Organizations need to be able to manage their application components no matter where those components reside.

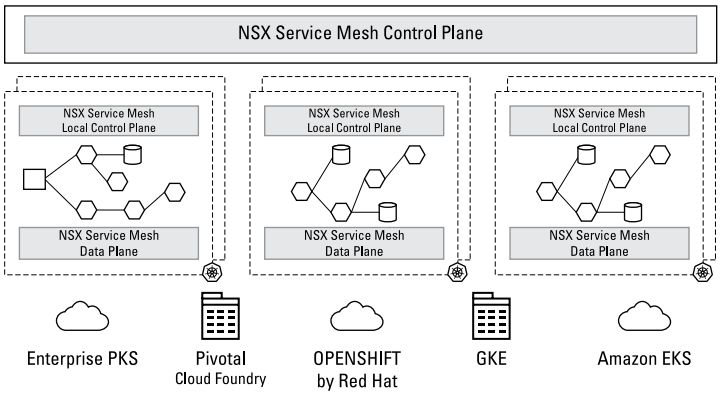

FIGURE 5-6: Application components may reside in multiple service mesh locations.

VMware is leading a new open-source service mesh interoperation project that is called Hamlet (`https://github.com/vmware/hamlet`).

This project is a collaboration between VMware, Google Cloud's Anthos, HashiCorp, and Pivotal and is a recognition that service mesh has become a vital part of microservices infrastructure.

Hamlet facilitates federation of service discovery between different service meshes of potentially different vendors. Through an API, service meshes can be interconnected to deliver the associated benefits of observability, control, and security across

different organizational unit boundaries, and potentially across different products and vendors (see Figure 5-7).

With service mesh interoperation, where each service mesh exists within a different and untrusted organizational unit boundary (and hence workloads are loosely coupled), each mesh can be of the same or different vendors, can have the same or different control and data plane implementations, be single or multi-cluster, and can provide the same or different functionality as a product while still providing interoperation across the meshes. This is the problem that Hamlet solves.

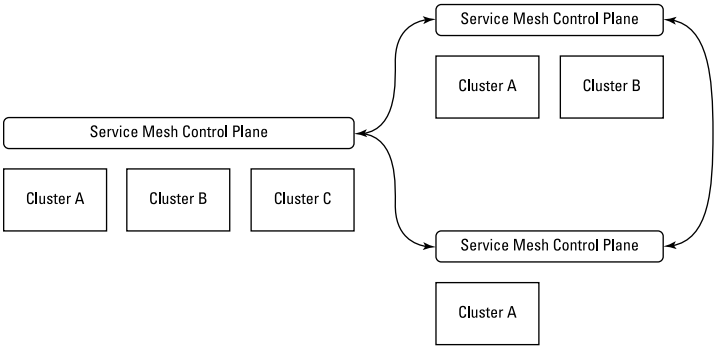

FIGURE 5-7: Service mesh interoperation across several different organizational unit boundaries.

NSX Service Mesh Use Cases

There are many use cases today for NSX Service Mesh and future use cases that may not yet have been imagined.

Multi-cloud and hybrid cloud patterns

There are many reasons why organizations deploy an application in multiple Kubernetes clusters and multiple clouds, whether on-premises, public, or hybrid, including the following:

- Separation of duties (services are developed by different business units)
- Separation of stateless and stateful services to different Kubernetes clusters
- Services consumption (using a data service in one cloud and an app service in a different cloud)

- Business continuity
- Reduced blast radius and more

With NSX Service Mesh, you can apply policies on the GNS level, such as security and traffic management policies. You also get the observability you need to operate and troubleshoot your applications and act upon any service-level objective (SLO) policy violations.

Business continuity

The general practice for business continuity with Kubernetes-deployed apps and cloud-native apps is to deploy the app in more than one cluster, either in the same region for high availability (HA) or a remote region for disaster recovery (DR), with a load balancer between them to direct traffic to both clusters, usually to both in an active-active configuration. In NSX Service Mesh, you can group these clusters into a GNS, integrate them with load balancing (local or global load balancing), and program it automatically to redirect traffic in case of failure. NSX Service Mesh can visualize the GNS configuration and traffic flows between the clusters for faster detection of health issues in the cluster.

End-to-end mutual transport layer security (mTLS)

Achieving end-to-end encryption for in-flight traffic is not easy but, in many cases, it's required for regulatory purposes. This requires a top-level CA that will provide a trusted identity to each node on the network. In the case of micro-services architecture, those nodes are the pods that run the services — and there are a lot of them. Istio can set up end-to-end mTLS encryption utilizing the CA function called Citadel. Citadel will manage the certificates for the services and will automatically rotate them every 90 days.

NSX Service Mesh builds upon this capability in Istio and expands it across multiple clusters and clouds — and even service meshes. The implementation of end-to-end encryption of in-flight traffic is applied at the GNS level, where you can apply an encryption policy that also supports different settings, such as faster certificate rotation than the default setting in Istio, or setting up permissive and restrictive policies.

Rolling upgrades at scale

When upgrading microservices through a canary upgrade or blue-green upgrade, you introduce a new version of a service side-by-side with the old version, and then move a small percentage of traffic over to the new version and monitor it for errors and latency. If the new service is working fine, you then move some more traffic over until 100 percent of your users are using the new version. At that point, you can decommission the old version of the service.

As we describe in previous chapters, this type of upgrade can be performed relatively easily with Istio using a simple YAML ("YAML Ain't Markup Language") file that defines the routing rules to send a percentage of traffic to the new service (split traffic). However, when you need to perform dozens, or even hundreds, of such upgrades a day it's no longer such an easy task. How do you manage these upgrades, monitor them, and make changes at scale? NSX Service Mesh provides an easy way to manage multiple rolling upgrades from a single console and automatically manage it across Istio deployments. By defining a single or multiple service upgrade for a GNS, you can define the rules that determine how much traffic to shift and the steps to take, as well as what to do in case of errors or failures. You can then monitor all your upgrades from a single dashboard.

TIP

You can also test a version by simulating an upgrade without going "all in" and switching out versions.

Predicting end-to-end response time

Achieving application SLAs in a distributed architecture can be very complex and challenging. If latency goes up in your application because of a service load that goes up in the chain, just autoscaling it out (deploying more instances of it) may cause an adverse effect on the entire application due to the ripple effect (one service scale creates pressure on downstream services). It's even more complicated when you're doing this across multiple clouds because you need to consider cross-cloud latencies and be able to look at the health of all the services in the application service chain (see Figure 5-8).

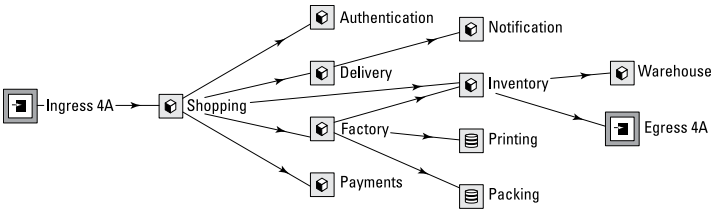

FIGURE 5-8: Service chain map example.

NSX service mesh allows you to assign an end-to-end latency SLA policy to an application and to automatically optimize and self-heal distributed microservices applications to achieve the SLA.

REMEMBER

VMware NSX products and solutions are not bound to any specific type of workload or cloud. NSX Service Mesh will continue to encompass additional types of workloads beyond Kubernetes and Istio. It's aimed to be extended to support virtual machines (VMs), as well as SaaS and Function as a Service (FaaS) in the future. The NSX Service Mesh scope is broader than just service-to-service communications on Kubernetes and extends to include users, data, and services across Kubernetes clusters and clouds, as well as service meshes.

IN THIS CHAPTER

- Exploring Istio online resources and other books
- Learning from other users
- Getting hands-on experience with labs and how-to guides
- Taking online courses and attending conferences
- Checking out podcasts
- Working with VMware resources

Chapter 6
Ten (Or So) Resources to Help You Get Started with Service Mesh

Ready to get started? We've put together the following list of materials and tutorials to help you enhance your understanding of Istio and VMware NSX Service Mesh.

Online Resources

The Istio project publishes a wide range of resources to help you get grounded on the project and keep up with the latest information. Check out the following resources:

- `https://istio.io/docs`
- `https://github.com/istio/istio`

Enterprises are deploying Kubernetes on-premises and in public clouds, and there's a real need to federate between services meshes. Check out the following online resources to learn about service mesh interoperability between Kubernetes clusters:

- **Open Source Service Mesh Interoperation:** `https://blogs.vmware.com/networkvirtualization/2019/08/new-open-source-service-mesh-interoperation-collaboration.html`
- **Simplify Hybrid Deployments with VMware NSX Service Mesh and Google Cloud Services:** `https://youtu.be/4iYaF4nBM_o`
- **How SPIFFE Helps Istio in Service Mesh Federation:** `https://youtu.be/SCBksDnA2rU`

Discussion Groups

Join a discussion group to post questions and actively contribute to the Istio project and follow @Istio on Twitter:

- `https://discuss.istio.io`
- `https://twitter.com/istiomesh`

Books

When you're ready to take a deeper dive into service mesh, why not get a blueprint from technical experts to help you understand what's going on "under the hood"?

- *Istio in Action,* by Christian E. Posta
- *Mastering Service Mesh Architecture,* by Anjali Khatri and Vikram Khatri
- *Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe,* by Lee Calcote and Zach Butcher

User Stories

Many enterprise users have taken the journey from Kubernetes to service mesh. Start your own journey by learning from others who have deployed service mesh in production:

- **The Life of a Packet Through Istio:** `www.infoq.com/presentations/life-packet-istio`
- **Trulia blog: Microservice Observability with Istio:** `www.trulia.com/blog/tech/microservice-observability-with-istio`
- **Autotrader UK: Using Istio to increase agility for the United Kingdom's largest automotive marketplace:** `https://cloud.google.com/customers/auto-trader-uk`
- **Lyft's Envoy: From Monolith to Service Mesh:** `www.microservices.com/talks/lyfts-envoy-monolith-service-mesh-matt-klein`
- **Namely: A Crash Course For Running Istio:** `https://medium.com/namely-labs/a-crash-course-for-running-istio-1c6125930715`
- **Pinterest: Running Envoy at the Edge:** `https://youtu.be/4x5WjxAMvKY`
- **eBay: Kubernetes and Istio on Google Kubernetes Engine:** `https://video.cube365.net/c/908977`
- **Ygrene Energy Fund: Using Istio's Mixer for Network Request Caching:** `www.youtube.com/watch?v=x1SomOy431I`

Interactive Labs

Ready to go get some hands-on experience? Try these interactive labs to get a virtual service mesh experience:

- **Katacoda:** `www.katacoda.com/courses/istio`
- **VMware:** `https://nsx.techzone.vmware.com/kubernetes-nsx`

How-To Guides

The best way to learn a new technology is to get hands-on experience. These step-by-step resources will help you get started:

- **Install Istio on any Kubernetes Cluster:** `https://istio.io/docs/setup/install/kubernetes`
- **Install Istio on Minikube:** `https://dzone.com/articles/istio-service-mesh-the-step-by-step-guide-part-2-t`

Online Courses

Why not take a class to enrich your understanding of service mesh? There are many free and low-cost options, including the following:

- **Linux Academy:** `https://linuxacademy.com/course/service-mesh-with-istio-part-1`
- **Udemy:** `www.udemy.com/istio-service-mesh-for-cloud-native-apps-on-kubernetes`
- **Lynda:** `www.lynda.com/Kubernetes-tutorials/Kubernetes-Service-Mesh-Istio/751332-2.html`

Conferences and Meetups

Join a local Meetup group on service mesh at `www.meetup.com/topics/service-mesh`.

Engage in dialog with the Istio user community at `www.meetup.com/topics/istio`.

The Linux Foundation hosts a vendor-neutral conference on service mesh. Find out more at `https://events.linuxfoundation.org/events/servicemeshcon-2019`.

Podcasts

Listen to podcasts on service mesh:

- www.infoq.com/ServiceMesh/podcasts
- www.se-radio.net/2019/03/se-radio-episode-361-daniel-berg-on-istio-service-mesh

Transform your multi-cloud network with service mesh

A service mesh is an abstraction layer that takes care of service-to-service communication, observability, and resiliency in modern, cloud-native apps built on a microservices architecture. In *Service Mesh For Dummies*, you'll discover how a service mesh can help you address challenges with microservices architecture and how to accelerate modern app development and operations. You will also learn how to address the challenges that service mesh itself brings using VMware NSX Service Mesh.

Inside...

- Understand the business need for agility
- Recognize microservices challenges
- Discover service mesh capabilities
- Explore service mesh use cases
- Enable service mesh in Kubernetes
- Get started with VMware NSX Service Mesh

vmware

Niran Even-Chen (@niranec) is a Principal SE at VMware, Office of the CTO, and a 3xVCDX. **Oren Penso** (@openso) is a Kubernetes Solution Engineer in VMware's Cloud Native Business Unit. **Susan Wu** (@susanwu88) is a Senior Product Marketing Manager in VMware's Networking and Security Business Unit.

ISBN: 978-1-119-66034-7
Not For Resale

for dummies
A Wiley Brand

Also available as an e-book

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.