

LEARNING MADE EASY

VMware 2nd 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
Sergio Pozo
Susan Wu

About VMware

VMware is a leading provider of multi-cloud services for all apps, enabling digital innovation with enterprise control. As a trusted foundation to accelerate innovation, VMware software gives businesses the flexibility and choice they need to build the future. Headquartered in Palo Alto, California, VMware is committed to building a better future through the company's 2030 Agenda. For more information, please visit www.vmware.com/company.

Service Mesh

**for
dummies®**
A Wiley Brand



Service Mesh

VMware 2nd Special Edition

**by Niran Even-Chen, Oren Penso,
Sergio Pozo, and Susan Wu**

**for
dummies®**
A Wiley Brand

Service Mesh For Dummies®, VMware 2nd Special Edition

Published by

John Wiley & Sons, Inc.

111 River St.

Hoboken, NJ 07030-5774

www.wiley.com

Copyright © 2022 by John Wiley & Sons, Inc., Hoboken, New Jersey

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Trademarks: Wiley, For Dummies, the Dummies Man logo, The Dummies Way, Dummies.com, Making Everything Easier, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: WHILE THE PUBLISHER AND AUTHORS HAVE USED THEIR BEST EFFORTS IN PREPARING THIS WORK, THEY MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES REPRESENTATIVES, WRITTEN SALES MATERIALS OR PROMOTIONAL STATEMENTS FOR THIS WORK. THE FACT THAT AN ORGANIZATION, WEBSITE, OR PRODUCT IS REFERRED TO IN THIS WORK AS A CITATION AND/OR POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE PUBLISHER AND AUTHORS ENDORSE THE INFORMATION OR SERVICES THE ORGANIZATION, WEBSITE, OR PRODUCT MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING PROFESSIONAL SERVICES. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR YOUR SITUATION. YOU SHOULD CONSULT WITH A SPECIALIST WHERE APPROPRIATE. FURTHER, READERS SHOULD BE AWARE THAT WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ. NEITHER THE PUBLISHER NOR AUTHORS SHALL BE LIABLE FOR ANY LOSS OF PROFIT OR ANY OTHER COMMERCIAL DAMAGES, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR OTHER DAMAGES.

ISBN 978-1-119-86667-1 (pbk); ISBN 978-1-119-86668-8 (ebk)

For general information on our other products and services, or how to create a custom *For Dummies* book for your business or organization, please contact our Business Development Department in the U.S. at 877-409-4177, contact info@dummies.biz, or visit www.wiley.com/go/custompub. For information about licensing the *For Dummies* brand for products or services, contact BrandedRights&Licenses@Wiley.com.

Publisher's Acknowledgments

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

Project Editor: Elizabeth Kuball

Acquisitions Editor: Ashley Coffey

Editorial Manager: Rev Mengle

Client Account Manager:
Cynthia Tweed

Production Editor:

Saikarthick Kumarasamy

Special Help: Larry Miller,
Pere Monclus

Table of Contents

INTRODUCTION	1
About This Book	1
Foolish Assumptions	2
Icons Used in This Book	2
Beyond the Book	3
CHAPTER 1: The Rise of Microservices and Cloud-Native Architecture	5
Recognizing the Need for Agility	5
Understanding That Application Architectures Are Changing	6
Service-oriented architecture	6
Microservices	7
Seeing That Distributed Applications Require a Reliable Network	11
Looking at How Kubernetes and Microservices Work Together	12
CHAPTER 2: Service Mesh: A New Paradigm	13
Identifying Challenges in Microservices Architectures	13
Introducing Service Mesh	15
Introducing Istio	18
CHAPTER 3: Service Mesh Use Cases	21
Service Discovery and Routing	21
Observability	24
Metrics	25
Distributed tracing	26
Availability and Resiliency	26
SLO definition and auto-remediation	27
Retries	27
Circuit breakers	28
Rate limiting	28
Traffic steering	29
Traffic-routing resiliency	30
Security	32
Service authentication and in-flight data encryption	33
Service authorization (auth-z)	34

CHAPTER 4: Introducing VMware Tanzu Service Mesh..... 35

- Exploring VMware Tanzu Service Mesh..... 35
- Introducing Global Namespaces 39
 - Service discovery..... 39
 - End-to-end mTLS..... 42
 - Application publishing and high availability 42
 - Actionable SLOs (intelligent autoscaling)..... 43
 - Traffic management (progressive upgrades) 45
 - Access control policies 46
 - API operations and security..... 46
 - PII data leakage protection 47
 - East-west threat detection 48

CHAPTER 5: Ten Resources to Help you Get Started with Service Mesh 51

- Blogs 51
- Books 52
- Conferences and Meetups 52
- Documentation..... 53
- Discussion Groups 53
- Getting Started Guides 54
- Hands-On Labs and Online Courses..... 54
- Podcasts and User Stories..... 55
- Videos and Tutorials 56
- VMware Products and Reference Designs 56

Introduction

Modern, cloud-native applications are composed of a collection of microservices, each performing a specific function. A typical modern application might consist of hundreds or thousands of microservices, all of which need to communicate with each other to give users what they want.

For services-to-services communications in modern applications to work, developers would have to program the business logic for their apps and manage the communication, security, and observability logic on top of that.

With hundreds or thousands of microservices, you can imagine how much complexity and overhead this burden adds to application development.

A service mesh is an abstraction that can provide a form of developer automation. By grouping and automating the repetitive logic tasks into a Layer 7 Proxy and delegating the tasks to the service mesh, the developer only needs to focus on writing the business logic.

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 — from traditional, monolithic applications to modern, cloud-native apps built on a microservices architecture.

About This Book

Service Mesh For Dummies consists of five chapters that explore

- » The evolution of microservices and cloud-native architecture (Chapter 1)
- » The service mesh paradigm (Chapter 2)
- » Service mesh use cases (Chapter 3)
- » VMware Tanzu Service Mesh (Chapter 4)
- » Additional service mesh resources (Chapter 5)

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 usefulness, 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 understand technology concepts such as cloud computing, networking, virtualization, and containers. As such, this book was 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 icons in the margin 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'll 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 <https://tanzu.vmware.com/service-mesh>.

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

This chapter explores 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. To differentiate their products and services and achieve a competitive advantage, many businesses are building their own custom software rather than buying the same commercial, off-the-shelf applications used by their competitors.

Much of this transformation is achieved in software. 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.



REMEMBER

In a monolithic architecture, the architectural choices are fixed, and the teams can't choose their own programming languages and tools. Plus, the entire software development team is stuck using the same integrated development environment (IDE), working on the same release, and using *waterfall testing*, in which testing is separate from software development. In waterfall testing, the development and testing teams may work separately, and acceptance and regression testing are performed after software development is completed.

Service-oriented architecture

SOA is a software architecture in which distinct components of the application provide services to other components via a communication protocol over a network. SOA integrates distributed, separately maintained software components that communicate with each other using an enterprise service 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:

- » **Service provider:** The provider layer consists of all the services within the SOA.
- » **Service consumer:** The consumer layer is where the users (humans, other components of the apps, or third parties) interface with the SOA.

Using SOA, organizations can encourage component reuse to avoid having to develop commonly used services like e-commerce shopping carts and Short Message Service (SMS). Instead, organizations can just publish these shared services in a service catalogue, and applications can then consume them over the network.



REMEMBER

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

Some advantages of SOA over monolithic architectures include:

- » Component reusability
- » Improved scalability and availability
- » Easy maintenance

The biggest limitation of the traditional implementation 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

A microservice architecture implements a modern SOA architecture that solves the challenges discussed in the preceding section.

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, e-commerce 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).

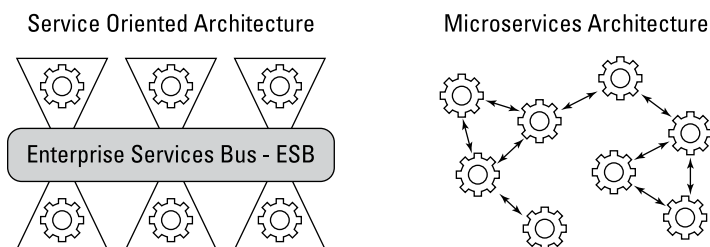


FIGURE 1-1: Comparing communications in SOA and microservices.

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

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.

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 and continuous integration/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 Apache 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.

A microservices architecture has many advantages 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 can't choose programming languages and tools. They're stuck to 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 when 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 Eureka, Hystrix, and Ribbon to solve these types of problems. Facebook, Google, and Twitter have all undertaken similar projects. However, adding networking stacks to 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 has 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 along with their dependencies (such as language-specific frameworks and libraries) and deployed into Kubernetes, thereby simplifying the path to production. 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 go hand-in-hand. As organizations modernize their applications and build container-based Kubernetes applications, they're confronted by a lab-to-production gap that challenges their ability to operationalize the applications. There are many point solutions in the Kubernetes ecosystem (see the CNCF Cloud Native Interactive Landscape at <https://landscape.cncf.io>) that they would have to stitch together. Application teams can quickly develop and validate Kubernetes applications in development environments, but a very different set of connectivity, security, and operational considerations await networking and operations teams that deploy the applications to production environments. Implementing a service mesh can address the networking and security requirements to rapidly bring applications to production.

- » Understanding microservices architecture challenges
- » Addressing microservices challenges with a service mesh
- » Finding out about Istio

Chapter 2

Service Mesh: A New Paradigm

This chapter covers 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):

- » Distributed applications are running on multiple run times and on multiple cloud providers creating application silos
- » Many endpoints to monitor, troubleshoot, scale, and secure, which increases operational costs and hinders operational agility

- » Many network connections to inspect and secure, which increases network latency and operational costs and hinders operational agility
- » Disjointed security, auditing, and compliance, which makes achieving compliance a difficult and time-consuming task

Transformation Challenges

How to consistently connect, secure, control and monitor modern apps?

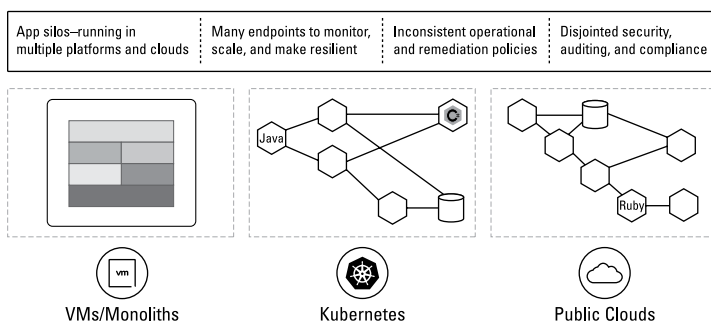


FIGURE 2-1: Microservices can deliver the promise of agility, but it can also introduce operational challenges.

Before service mesh architectures appeared, there were various ways to address these challenges — and there still are. For example, language-specific libraries address issues such as service discovery, encryption, and resiliency; likewise, an application programming interface (API) gateway — which sits in the path from a client to the application — provides similar capabilities.

However, both approaches have their own challenges. Libraries, for example, are language (for example, Java) and platform (for example, Spring) specific, preventing developers from realizing the promise of freedom, bloating their applications with unnecessary networking and security code, and tying the life cycle of their applications to the life cycles of the libraries and frameworks they're using. A centralized solution such as an API gateway only provides a solution for traffic directed from a client to the front end of an application, but not between the components of the application.

Introducing Service Mesh

A service mesh is a modern connectivity and security run-time platform that takes care of:

- » Service-to-service communication (service discovery and routing)
- » Service-to-service security (authentication, authorization, encryption, and traffic inspection)
- » Observability (monitoring and distributed tracing)
- » Resiliency (service-level objectives [SLOs], circuit breakers, and retries)

The premise of the service mesh architecture is that microservices should be all about business logic, and development teams should focus solely on building business logic rather than dealing with the connectivity and security requirements of the applications. For example, if a microservice is a web server, it needs to fetch data and present it. But a lot of maintenance or “housekeeping” work is needed (see Figure 2-2) — things like service discovery and connectivity details of the data endpoints. A microservice that needs to communicate with other services needs to know how to find them, how to define the connection details, whether the connection is encrypted, how to authenticate and authorize the connection, and so on. 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.

However, none of this “housekeeping” work adds value to the business, and it isn’t considered business logic. With a service mesh, you can abstract these functions to a platform, which enforces them through an entity called a *proxy*.



TECHNICAL
STUFF

A proxy sits in front of each microservice, and all communications traverse it, so the proxy virtually impersonates its companion microservice in an architectural pattern known as a *sidecar proxy*. Each proxy is responsible for managing the connectivity, providing observability, and enforcing reliability and security properties required for its companion microservice. This proxy-to-proxy communication creates the service mesh architectural pattern.

Proxies behave both as forward and reverse proxies, depending on the direction of the communication: a forward proxy impersonates the caller service or client, while a reverse proxy impersonates the receiver service or server.

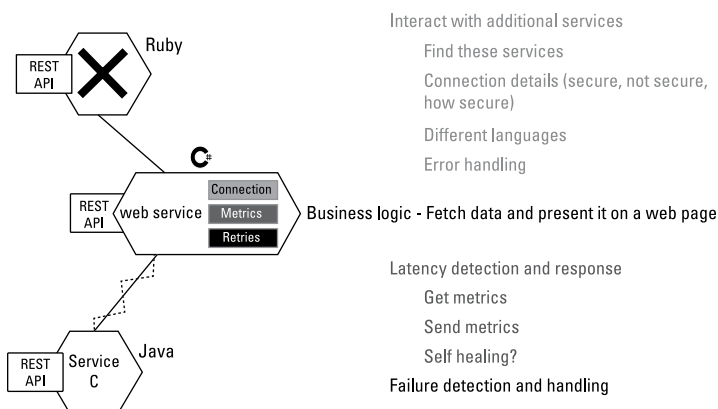


FIGURE 2-2: Necessary maintenance functions in a microservices architecture.

In a Kubernetes application platform, containers run in pods. The sidecar proxies are deployed as containers in the same pod, creating a true sidecar service that captures all traffic going in and out of the microservice (see Figure 2-3).

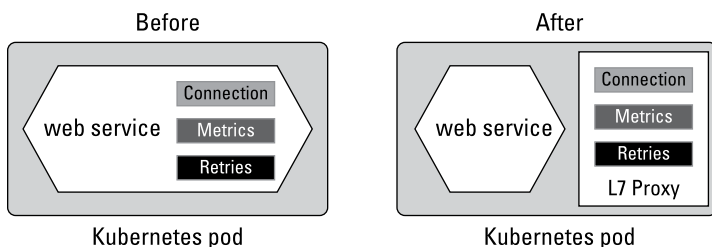


FIGURE 2-3: A Layer 7 proxy, or *sidecar*, in a Kubernetes pod.



TIP

Service mesh projects like Consul, Istio, Kuma, and others 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 29,800 stars, 17,428 commits, and 780 contributors in the Istio GitHub repository and numerous companies building service mesh products that are based on Istio, including Aspen Mesh, Cisco, F5, NGINX,

RedHat OpenShift, SUSE Rancher, Turfin Orca, Twistlock, and VMware.

Some of the most prominent multi-cloud service mesh commercial products on the market that differentiate from open-source service mesh projects beyond simplifying its usage include the following:

- » **Cillium Service Mesh:** Open-source and commercial offerings with an Extended Berkeley Packet Filter (eBPF) implementation instead of a sidecar proxy one.
- » **Consul Service Mesh (HashiCorp):** Open-source and commercial offerings with a “bring your own” (BYO) proxy data plane.
- » **Kuma Service Mesh (Kong):** Open-source and commercial offerings with an Envoy-based unified data plane for north-south API gateway and east-west sidecar traffic, and a single control plane.
- » **Linkerd Service Mesh:** Open-source and commercial offerings with a custom proxy implementation for the data plane.
- » **Tanzu Service Mesh (VMware):** Commercial offering with Envoy-based data plane, offering API-security capabilities and reliability capabilities through SLO monitoring and auto-remediation.

Projects like Consul, Istio, Kuma, 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 separated from the business logic perspective, making it easier to manage the life cycle. The “housekeeping” code required to handle communication, security, observability, and resiliency — which is non-differentiating to the business — is declared by application operators and enforced using the sidecar proxy via the service mesh control plane. Developers can instead focus on writing business logic that creates business value.



Developer time is very expensive, so moving to a modern connectivity and security run-time platform separated from the application run time itself can save a lot of time and cost for the business.

Introducing Istio

Istio's architecture is divided into two levels: the data plane, which is based on the Envoy proxy, and a control plane to manage the life cycle and configure the proxies. Istio injects the sidecar proxies into all the Kubernetes pods forming the desired service mesh.

Istio also uses Envoy proxies to provide access in and out of the mesh (with the function of ingressing and egressing traffic to and from the mesh), thereby providing a very clear demarcation line for the entry and exit points of the service mesh. Traffic coming into the mesh or leaving it via an Envoy proxy — which 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.



Istio uses an extended version of the Envoy proxy. Envoy is a high-performance proxy developed in C++ to mediate all inbound and outbound traffic for all services in the service mesh. Envoy proxies are the only Istio components that interact with data plane traffic.

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).

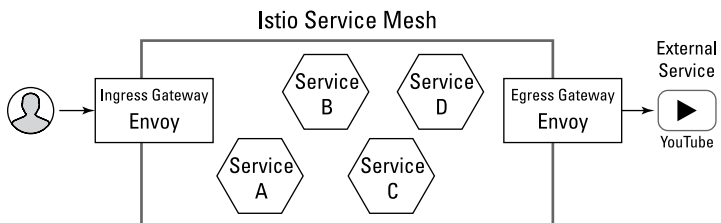


FIGURE 2-4: Envoy proxies provide entry and exit points in the service mesh.

The Istio control plane (istiod) provides service discovery, configuration, and certificate management (see Figure 2-5). Istiod converts high-level routing rules that control traffic behavior into Envoy-specific configurations, and propagates them to the sidecars at run time.

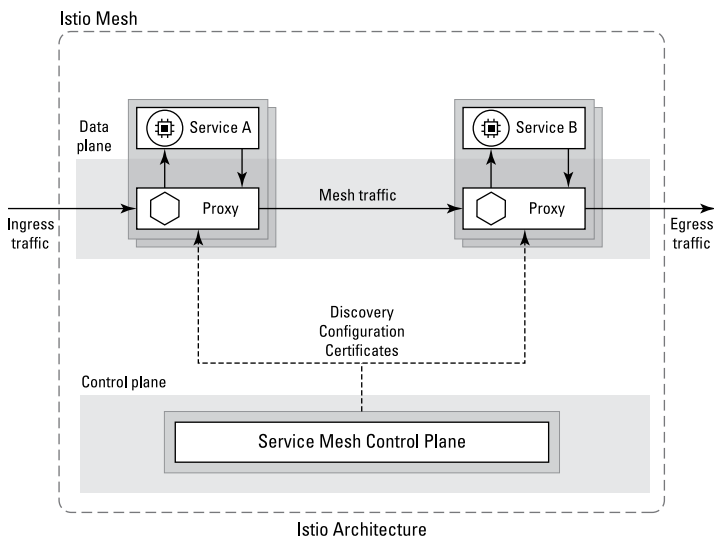


FIGURE 2-5: The Istio architecture consists of the data plane and the service mesh control plane.

- » Managing communication
- » Enabling observability
- » Ensuring availability and resiliency
- » Providing security

Chapter 3

Service Mesh Use Cases

Service mesh use cases tend to fall into four main areas: service-to-service communication, observability, availability and resiliency, and security. This chapter explores these use cases.

Service Discovery and Routing

The ability to provide a service directory and ways for microservices to register in it and pull the endpoints of other microservices they need to communicate with (for example, the Domain Name Server [DNS]) is provided by default by all service meshes — both open-source projects and commercial products). The ability to route traffic between microservices when they're contained in the same platform and administrative boundary (for example, a single Kubernetes cluster) is also provided by default.

When an organization is starting with Kubernetes as its de facto application platform, or when an organization is mature but its development teams are still small or very fragmented, it will tend to run development and test builds in the same Kubernetes cluster. Small clusters may be assigned to each team or one big cluster may be assigned to multiple teams. The tendency today

in more organizations is to have many small clusters rather than a few big ones. At a minimum, most organizations separate their nonproduction services from their production services by placing them in different clusters. In more complex scenarios, organizations might choose multiple clusters to separate services across tiers, locales, teams, or infrastructure providers. The most common reasons for separating clusters include the following:

- » **Isolation and multi-tenancy:** Separating the control plane and data plane of services, primarily to improve reliability or address security needs. Isolation usually comes up when considering the following:
 - *Environment:* More often than not, organizations run their development, staging/test, and production services across separate clusters, often running on different networks and cloud projects.
 - *Workload tiering:* Often, organizations that have many complex applications tier their services, choosing to run their more critical services on separate clusters from their less critical ones.
 - *Reduced impact from failure:* When organizations want to limit the impacts of an operator mistake, cluster failure, or related infrastructure failure, they can split their services across multiple clusters.
 - *Upgrades:* When organizations are concerned about potential issues with upgrading in place (that is, upgrading automation failure, application flakiness, or the ability to roll back), they can choose to deploy a copy of their services in a new cluster.
 - *Security/regulatory compliance requirements:* Organizations can choose to isolate services for many reasons, including keeping workloads that are subject to regulatory requirements separate from less-sensitive ones, or running third-party (less-trusted) services on separate infrastructure from first-party (trusted) services (clusters).
 - *Jurisdiction/data regulatory compliance requirements:* Data residency and other jurisdictional processing requirements can require compute and storage to live within a specific region, requiring infrastructure to be deployed in multiple data centers or cloud providers.

» **Scale and availability:** Placing services in specific locations to address availability, latency, and locality needs. Location usually comes up when considering the following:

- *Latency:* Certain services have latency requirements that must be met by physically locating that workload in a specific location (or geography). This need can occur if upstream services or end users are sensitive to latency, but it can also easily occur if the workload itself is sensitive to downstream service latency.
- *Availability:* Running the same service across multiple availability zones in a single cloud provider (or across multiple providers) can provide higher overall availability.
- *Data gravity:* A large amount of data, or even certain database instances, can be difficult, impossible, or even inadvisable to consolidate in a single cloud provider or region. Depending on the processing and service requirements, an application might need to be deployed close to its data.
- *Legacy infrastructure/services:* Just as data can be difficult to move to the cloud, some legacy infrastructure is difficult to move. Although these legacy services are immobile, being able to modernize around them allows organizations to increase development velocity.
- *Local/edge compute needs:* As organizations want to adopt application modernization practices in more traditional work environments, like warehouses, factory floors, retail stores, and so on, this necessitates managing many more workloads on many more pieces of infrastructure.

» **Eliminate vendor lock-in (see Figure 3-1):** In addition to compliance, there are other reasons organizations may need to use different vendors. For example, their applications may use application services or specialized features (beyond the platform), like databases or artificial intelligence (AI), from a specific cloud vendor. Or different cloud vendors in an organization's regions may offer different availability, cost, security, and compliance features.

» **Developer choice:** Organizations often benefit from being able to provide developers choice in the cloud-managed services that they consume. Generally, choice lets teams move more quickly with tools that are best suited to their needs at the expense of needing to manage additional resources allocated in each provider.

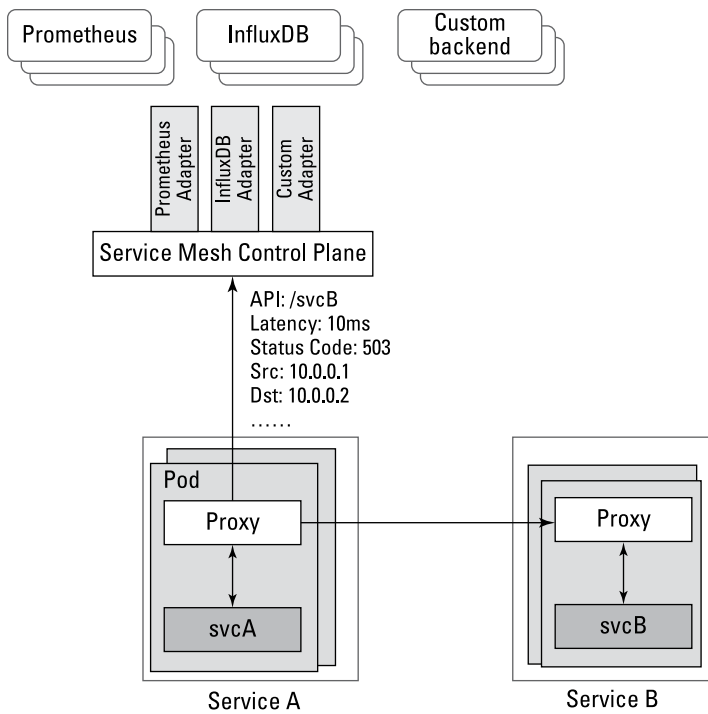


FIGURE 3-1: Proxies send metrics to the Service Mesh Control Plane in a service mesh.



As applications are becoming more distributed in nature in multi-clouds, or as application dependencies are distributed across multi-clouds, service meshes that can provide service discovery, traffic routing, and some kind of traffic resiliency when workloads are distributed across multiple clouds are becoming more common.

Observability

Application observability is a diverse topic, and different application teams may have different requirements, which are generally related to troubleshooting (because something doesn't work, because it is not performant, and so on). At a glance, developers and application operators need to use traffic metrics and application programming interface (API) call traces between the different microservices that compose their applications.

To do this, application operators have to instrument microservices and set up metric aggregators like Prometheus and dashboards like Grafana. There are different ways to instrument microservices. With service meshes that follow the sidecar-proxy architecture (see Chapter 2), the proxy automatically captures all the traffic in and out of the microservices and exports the metrics to the service mesh control plane, where developers and application operators can check and use these metrics. Relying on Envoy sidecars means that the observability instrumentation is transparent to applications and automatically provided, not requiring any code changes or application redeployments, regardless of the programming language or frameworks used to build the applications.

Metrics

The four golden signals of metrics monitoring are latency, traffic, errors, and saturation. If you can only measure four metrics of your user-facing system, focus on these four to understand the behavior of your application:

- » **Latency:** The time it takes to service a request, for both successful and failed requests.
- » **Traffic:** How much demand is being placed on a microservice (depending on the nature of the microservice, this could be requests per second, input/output [I/O] rate, and so on).
- » **Errors:** The rate of requests that fail.
- » **Saturation.** How close your microservices are to the maximum capacity (for example, memory they can use, throughput they can provide, and so on).

In most commercial products, service graphs with service-to-service relationships and golden metrics help developers and application operators understand who connects to each service and the service dependencies. The best service meshes provide service graphs of different granularity, like applications distributed across clusters, concentrated within one single cluster, within a namespace, for groups of services, and so on, allowing developers and application operators to analyze relationships and metrics for their applications, wherever they're deployed.

Preconfigured metric dashboards and charts for individual microservices are commonly used to analyze them with filtering

and breakdowns, including by destination service, source service, metrics, and more. Operators should have dashboards for full clusters, namespaces, groups of services, and so on. Ideally, these dashboards should include golden metrics for microservices and also metrics around the infrastructure that support those microservices. This functionality helps platform operators to get information on how the microservices are laid over the workload nodes they run on and the different infrastructure performance and health metrics like central processing unit (CPU), memory, disk, and more. Through this relationship between infrastructure and applications, platform operators can correlate microservices problems with the infrastructure.



REMEMBER

The four golden signals of metric monitoring are latency, traffic, errors, and saturation.

Distributed tracing

Distributed tracing is a method used to profile and monitor applications to understand service dependencies and the sources of latency within the composite application. The concept has existed for many years, but it has been particularly difficult to achieve in a distributed microservices application, because transactions flow through a mesh of many services.

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

Availability and Resiliency

Applications commonly need to remediate different conditions related to the four golden signals of monitoring (discussed earlier in this chapter). For example, in Kubernetes, new pods are scheduled by the platform when they're unhealthy, and operators can manually schedule new pods and kill existing ones. Application operators can also scale up and scale down pods. Remediations are generally manual tasks done after the fact — the operators must watch the metrics and then remediate.

The breadth of remediation actions or preventive actions that a service mesh provides to application operators depends entirely on the service mesh implementation. However, there is a common set (discussed in the following sections) that is found across the most common open-source projects and commercial products.

SLO definition and auto-remediation

With service-level objectives (SLOs), application operators can monitor the health of microservices and applications. Using the different golden metrics that service meshes capture, application operators can define error budgets to check how application health is evolving over time. Operators can also define SLOs for individual services or for groups of services. Finally, SLOs help application operators make informed decisions around which parts of an application need work and where feature development can be accelerated, as well as helping them configure automated remediations.



TIP

Examples of automated remediations (discussed in the following sections) include retries, circuit breakers, rate limiting, traffic steering, and traffic routing resiliency. Although these are resiliency functions that can be used independently, their value increases when they can be automatically triggered from SLOs.

Retries

Service meshes are able to handle network failures on behalf of the microservices of your application. The service mesh can be configured to automatically and transparently retry failed requests within the parameters set up by the application operators. 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

As an example, if there's a Hypertext Transfer Protocol (HTTP) "503 Service Unavailable" error because the server is completely unavailable due to scheduled maintenance, the application operator would need to make sure that the application has retries configured in the service mesh to handle the 503 errors returned from the proxy.



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 and outside of the application.

Circuit breakers

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 service meshes, an application operator can set a threshold when a service is generating HTTP 503 errors to remove it from the load balancer pool that handles traffic to the multiple instances of the same microservice. New requests coming in will not be routed to the unhealthy instances (see Figure 3-2). This is known as circuit breaking and it's configured by defining a connection pool of concurrent Transmission Control Protocol (TCP) connections and pending HTTP requests.

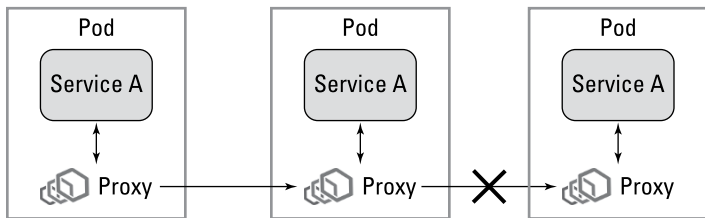


FIGURE 3-2: Traffic is not routed to the unhealthy service.



TIP

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

Rate limiting

Related to circuit breaking is rate limiting, a feature in service meshes that allows application operators to enforce limits on the rate of requests that match certain criteria. This feature can be used to throttle traffic when it exceeds a rate of requests pre-defined by the application operator.



TIP

Defining rate limits involves specifying which parameters to count, their maximum values, and the window of time in which to enforce the limit.

Traffic steering

Traffic-steering rules enable application owners to control where incoming traffic to a pool of instances of a microservice will be sent. Rules are based on attributes such as authentication (send Jason to Service A and send Linda to Service B), location (send United Kingdom to Service A and send United States to Service B), device (send watch to Service A and send mobile to Service B), or anything else that is passed in the HTTP header. In Figure 3-3, 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.

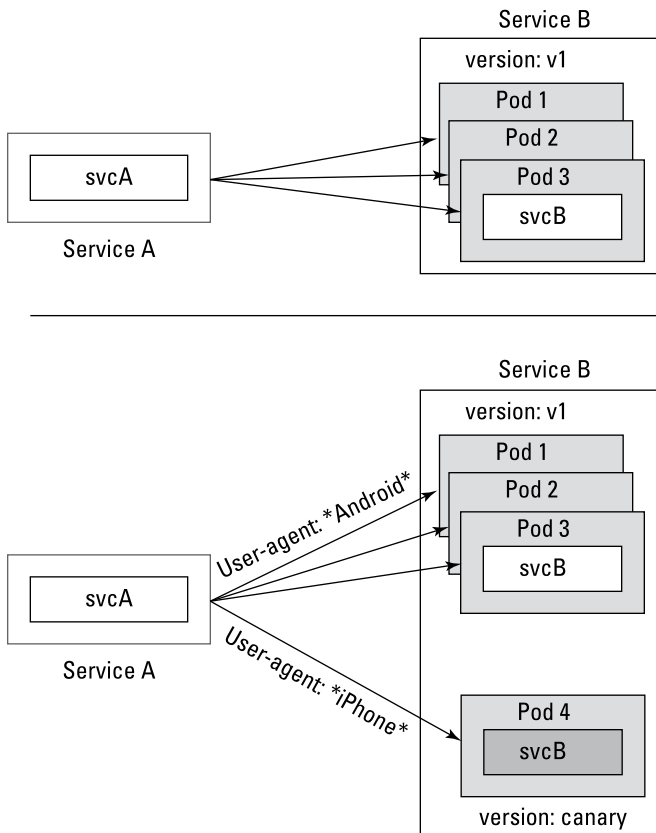


FIGURE 3-3: A traffic-steering example.

Traffic steering has been accomplished in previous architectures by hardcoding the rules into the application and using software libraries. With service meshes, these rules are created by application operators, with the life cycle managed independently of the application, and enforced by the proxies.

Specific traffic-steering use cases include the following:

- » **A/B testing of microservices:** When testing web applications, application operators 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 an implementation of one service over another.
- » **Canary releases:** Canary releases are generally used to deploy new versions of microservices that only contain nonfunctional improvements. 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 3-4). 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.

Traffic-routing resiliency

There are normally two scenarios in which application operators need to provide resiliency for routing traffic to services:

- » When application operators publish applications for external consumers (like users or other applications) where there is one “fronting” microservice
- » For the rest of microservices that compose an application when they aren't colocated in the same administrative domain (for example, the same cluster)

Although these use cases may sound similar, there are fundamental differences from an architecture perspective.

```

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 canary release.

In the first use case, traffic is coming from outside the service mesh (typically referred to as *north-south traffic*) and must traverse the ingress gateway of the service mesh, as well as an upstream load balancer. When two or more instances of the same published microservice are available in different administrative domains (for example, two or more clusters), the load balancer can be programmed to route traffic to all instances, only to one or to a group of instances, fail over to healthy instances (eject the unhealthy ones), and so on, with different load-balancing policies that can be configured by an application operator. Synchronizing the configuration between the service mesh and the external load balancer can be very painful because it's basically a manual process, or it can be automated using do-it-yourself (DIY) tools.



TIP

Some service meshes can automate this process for you.

In the second use case, traffic is going across the different administrative domains within the service mesh (typically referred to as *east–west traffic*) and traversing ingress and egress gateways of the service meshes of each (for example, a cluster). Like the previous use case, the service mesh can be configured with different load-balancing policies in the ingress gateways to route traffic to different services or to all of them and so on. The main difference from the previous use case is that, in this one, there is no external load balancer and the routing function executes at the cluster service mesh ingress gateways.

Security

Application operators must follow security and compliance guidelines to mitigate insider threats and reduce the risk of a data breach. This is generally achieved by ensuring that all communications between applications are encrypted and mutually authenticated. There are more advanced use cases, but these two are hard requirements necessary to comply with regulatory and security governance measures.

To achieve service-to-service authentication and encryption, application operators have to instrument security into their microservices. Developers can satisfy these requirements in middleware by setting up a mutual or one-way Transport Layer Security (TLS) connection each time a microservice connects to another (for east–west traffic) or a client connects to a microservice (for north–south traffic). However, this approach poses new challenges as security and compliance teams are increasingly making development teams accountable for security and compliance, which requires additional training and resources.

However, in service meshes that follow the sidecar-proxy architecture, the proxy automatically and transparently instruments the microservices and can enforce all the required security functions for both east–west and north–south traffic patterns, without requiring any code changes or application redeployments, regardless of the programming language or frameworks used to build the applications.

Service authentication and in-flight data encryption

You may be familiar with TLS, which is used in the HTTP Secure (HTTPS) protocol to allow browsers to trust web servers and encrypt data that's exchanged. When simple TLS is used, the client determines that the server can be trusted by validating its certificate. Mutual TLS (mTLS) is an implementation of TLS in which both the client and the server present certificates to each other and verify each other's identities.

With mTLS, application operators can do the following:

- » **Mitigate risk of replay or impersonation attacks that use stolen credentials.** Proxy-based service meshes rely on TLS certificates to authenticate microservices rather than bearer tokens such as JavaScript Object Notation (JSON) Web Tokens (JWT) issued directly to users. Because TLS certificates are bound to the TLS channel, it is not possible for an entity within an environment to impersonate another by simply replaying the authentication token without access to the private keys.
- » **Ensure encryption of data in transit.** Using mTLS also ensures that all HTTP, Google Remote Procedure Call (gRPC), and TCP communications are encrypted in transit, thereby providing data confidentiality.

Service meshes provide a top-level (root) certificate authority (CA) in the control plane, which provides a certificate to the CA present in each workload cluster. This local-to-the-cluster CA provides a certificate to each sidecar, which acts as the identity of the workload. The CA automatically manages the life cycle of the certificates. Often, this certificate follows the Secure Production Identity Framework for Everyone (SPIFFE) standard, as in the case of Istio, for example.



TIP

Some service meshes are not limited to single-cluster topologies and can seamlessly work across clusters, environments (on premises and cloud), and different platform vendors (for example, open-source Kubernetes, Tanzu Kubernetes Grid [TKG], Amazon Elastic Kubernetes Service [EKS], Azure Kubernetes Service [AKS], Google Kubernetes Engine [GKE], OpenShift Container Platform [OCP], and so on). Requests between services in the mesh are

allowed by default. Authorization policies are applied in the sidecar proxy, which will determine if a request should be allowed or denied based on the policies.

Service authorization (auth-z)

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

Service authorization policies provides access control for the services in the mesh. Service authorization provides namespace-level, service-level, and method-level access control for services in the mesh. The service mesh can segment the services based on Layer 7 constructs.

Requests between services in the mesh are allowed by default. Authorization policies are applied in the sidecar proxy, which will determine if a request should be allowed or denied based on the policies.

- » Learning the basics of VMware Tanzu Service Mesh
- » Discovering global namespace policies and capabilities

Chapter 4

Introducing VMware Tanzu Service Mesh

A service mesh addresses challenges associated with a microservices architecture but also introduces new challenges (see Chapter 2). This chapter explains how VMware Tanzu Service Mesh unleashes the real power of a service mesh and addresses service mesh challenges.

Exploring VMware Tanzu Service Mesh

The real power of a service mesh extends beyond service-to-service communication abstraction. Just like other abstractions, you can build more value into it. Tanzu Service Mesh delivers this value.

A service mesh, no matter which project or product you're going with, is quite challenging to operate at scale. You need to take care of many moving parts to make it work in a large environment. In addition, most service meshes don't address multi-cluster service mesh deployments, not to mention extending the service mesh across different sites and clouds. The value in Tanzu Service Mesh begins with simplifying operations.

Tanzu Service Mesh takes the concept of a service mesh and elevates it to more of a distributed application framework that extends far beyond service-to-service communications and provides advanced security capabilities, resiliency, and automated operations for the application — regardless of which clouds its services are running on.

Tanzu Service Mesh differentiation starts with its architecture. Instead of clustering and synchronizing control planes, Tanzu uses a federated approach for multi-Kubernetes cluster and multi-cloud deployments. This is a significant difference from other service mesh offerings because it removes a dependency on complicated architectures that can reduce availability.

Tanzu Service Mesh provides a three-tier architecture. The top tier, referred to as the *Global Controller*, is delivered as a software-as-a-service (SaaS) offering. The Global Controller sends control operations in a federated manner to the local controllers, which comprise the second tier. These local controllers are basically a managed, curated, vanilla Istio control plane running on the Kubernetes cluster. The third tier consists of the data plane proxies. Tanzu Service Mesh utilizes the Envoy project as the data plane proxies on Kubernetes (see Figure 4-1).

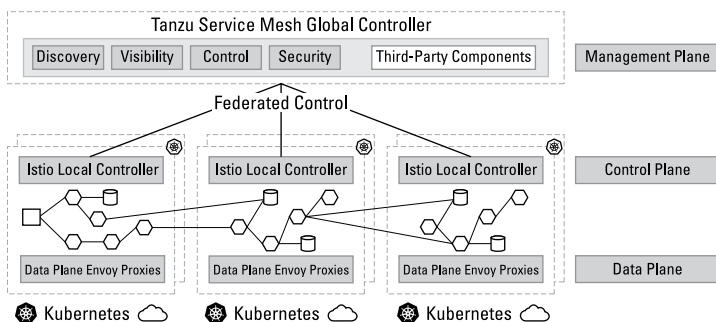


FIGURE 4-1: The Tanzu Service Mesh three-tier architecture.



TIP

Tanzu Service Mesh supports many popular Kubernetes platforms, including Amazon Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE), Microsoft Azure Kubernetes Service (AKS), Red Hat OpenShift, Upstream Kubernetes, VMware Tanzu Kubernetes Grid, and more.

There are several reasons and use cases for an organization to distribute an application's services across multiple Kubernetes clusters instead of using a single huge Kubernetes cluster, including the following:

- » **Difficulty 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. To avoid such a large “blast radius,” organizations can limit the scope and size of their Kubernetes clusters. However, this approach often requires placing an application's services in different clusters.
- » **Isolation between multiple tenants:** In Kubernetes, the tenancy construct is the namespace, which is similar to a folder (a binder, if you will) of logical constructs that are part of the same application or tenant. However, namespaces are not a good tenancy model because they provide limited isolation between tenants. If true multi-tenancy is required, you may need to utilize multiple Kubernetes 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:** Stateful services (also referred to as *data services*) are usually handled differently from stateless services and, therefore, may need to be separated in multiple Kubernetes clusters with different backup, disaster recovery, and upgrade plans.



REMEMBER

A major component of Tanzu Service Mesh in the client clusters is Istio. Upon onboarding a client Kubernetes cluster into the Tanzu Service Mesh SaaS solution, a managed and curated version of Istio will be deployed onto your client cluster (a client cluster is your Kubernetes cluster where the application workloads are running). The Tanzu Service Mesh Global Controller uses Istio for certain local control capabilities while also managing the life cycle of that Istio deployment. Customers can choose to utilize this

Istio deployment directly or utilize Tanzu Service Mesh’s application programming interface (API) and create a global namespace (discussed later in this chapter), which provides automated Istio operations and adds additional layers of policy.

When it comes to multi-cluster/multi-cloud support, although Istio itself supports multi-Kubernetes cluster deployments, it requires control plane synchronization. This requirement adds a dependency on complicated clustering topologies between the Kubernetes cluster control planes.

With Tanzu Service Mesh, the top layer, the Global Controller, manages and controls the local Istio control planes, programming them and managing their life cycle (see Figure 4-2). The Istio control planes on the client clusters are federated by the Global Controller and only communicate back to the Global Controller — never with one another.

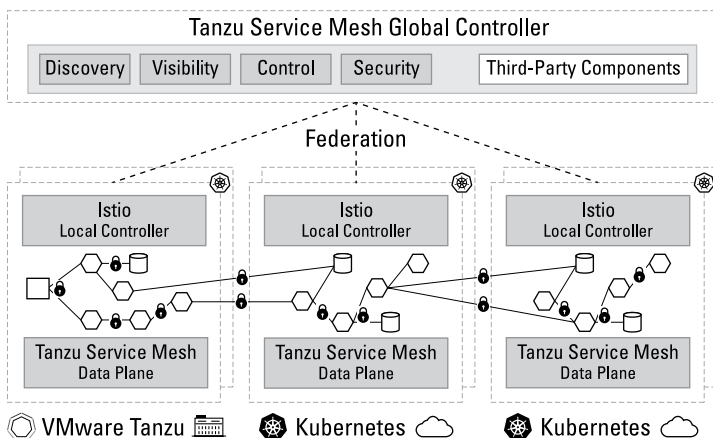


FIGURE 4-2: The Tanzu Service Mesh federated three-tier architecture.

This federated design removes a major dependency (reliance on replication of control planes) for multi-Kubernetes cluster service mesh deployments and greatly simplifies operations while increasing system availability. Data plane communication between the application service proxies on the clusters don’t flow through the SaaS Global Controller. Whether they run on one Kubernetes cluster, multiple Kubernetes clusters, across sites, or

even across clouds, they communicate directly with one another from proxy to proxy.

Introducing Global Namespaces

Global namespaces, as the name implies, are a logical construct that exists in the Global Controller and logically ties together services that need to communicate with one another. These services can be part of a single application, part of a clustered data service, or part of any other construct in which its workloads need to communicate with one another and be secured as a unit. When mapping services in different clusters and clouds into a global namespace, you can apply policies to them by applying the policy on that global namespace. The policies will then apply to the services within it.



TECHNICAL
STUFF

A global namespace is a unique concept in Tanzu Service Mesh. A global namespace defines an application boundary. It connects the resources and workloads that make up the application into one virtual unit to provide consistent traffic routing, connectivity, resiliency, and security for applications across multiple clusters and clouds. Each global namespace is an isolated domain that provides automatic service discovery and manages service identities within that global namespace.

Tanzu Service Mesh supports an ever-growing number of configurations and policies in a global namespace, which we discuss in the following sections (see Figure 4-3).

Service discovery

Microservices brought the promise of agility to software development. Applications are reduced to individual pieces called *microservices*, and these microservices communicate using APIs. Microservices enable continuous deployment and upgrade of individual functionality separately and independently from the other parts of the application.

But, as always, a new promise also brings new challenges. One such challenge is how services now communicate with one another. In monolithic apps, different functions are written in the same code base and communicate through remote procedure calls

(RPCs) in memory. Microservices communicate over the network, which introduces overhead and complexity when developing cloud-native applications in a distributed architecture, because you must take care of things such as:

- » Ensuring services can find each other
- » Securing communications
- » Making applications resilient and highly available

Global Namespaces: Cross-Cloud and Strong Isolation Decoupling applications from infrastructure

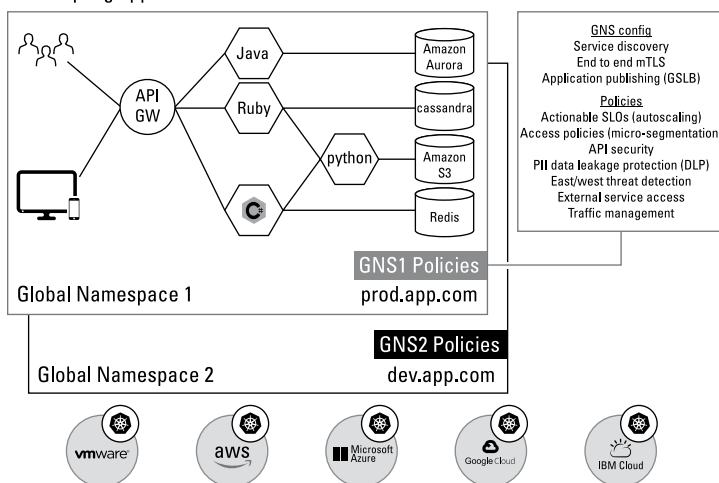


FIGURE 4-3: Global namespace configurations and policies in Tanzu Service Mesh.

The services in a microservices architecture talk with one another using APIs, but reachability is based on Internet Protocol (IP) addresses where Service A reaches out to the IP address of Service B. At the same time, you don't want your applications, and ultimately your developers, to rely on IP addresses because IP addresses are very infrastructure-specific and may not be directly visible to the services (think, Network Address Translation [NAT] boundaries). Also, in the cloud-native world, IP addresses are ephemeral and constantly changing. Imagine having to update your application every time a service changed its IP address — it would be an operational nightmare and would defeat the whole purpose of moving to distributed architectures in the first place.

So, how do services find each other if not based on IP addresses? If Service A needs to communicate with Service B, how would Service A “know” the IP address of the ingress point to Service B? You may be thinking this is easy — the Domain Name System (DNS). Since the dawn of IP addressing, the DNS has been the “phone book” for IP networks. But with microservices, there’s a big challenge — someone or something needs to update the DNS records because IP addresses and service locations change constantly.

If you’re building your application in a single Kubernetes cluster, the problem is easily solved with Kubernetes DNS. The cluster “knows” about all the IP addresses of all the services that are running on it automatically. The developers just have to refer to the services by their “Kubernetes name.”

However, in a multi-Kubernetes cluster or multi-cloud environment, relying on the Kubernetes name and mechanisms is pretty limiting, and it becomes more complicated because the different Kubernetes clusters don’t exchange DNS information. With Istio and other service mesh technologies, this is addressed by synchronizing control plane information. With Tanzu Service Mesh global namespaces, VMware Tanzu Service Mesh abstracts all that complexity and apply a simple federated “service discovery” mechanism. A *service discovery domain name* (for example, `acme.lab`) is applied to the global namespace. After you map services to the global namespace, it will automate the DNS entries in all the clusters based on that domain name, so now Service A calls Service B not by using its Kubernetes local name, but instead by using the global namespace domain name (for example, `B.acme.lab`). In this way, the service mesh will always know the ingress IP address for Service B wherever it may be based in the global namespace name. This eliminates any need for manual configurations of DNS or any replications of data between clusters. Developers can simply specify the business logic name of each service in the application and “inject” the service discovery domain of the global namespace. The services communication will always work without anyone needing to know or configure a single IP address. This will work in a single Kubernetes cluster, multiple clusters, and across clouds and vendors.



REMEMBER

Service discovery is the first interaction with the mesh, and it’s the only way developers should attach their applications to it. Now you can start layering more sophisticated services on top.

End-to-end mTLS

All service mesh technologies can attach a unique x509 certificate to each microservice. These certificates are then used by the services to authenticate and establish Mutual Transport Layer Security (mTLS) encryption and authorization access policies. But for this to work, the certificates need to be part of the same trust chain. It's simpler to achieve trust when two services are running on the same Kubernetes cluster using the same single certificate authority (CA) in the cluster to issue certificates to the services. But in a multi-cluster/multi-cloud deployment, it's challenging to create a trust between independent control planes.

Like service discovery, Tanzu Service Mesh takes a federated approach to this challenge. When onboarding clusters, the Global Controller will deploy a CA to each cluster and create a root CA in the cloud, or connect them to the customer's corporate CA as root. In this way, the CAs on the Kubernetes clusters have the same trust chain. Now you can authenticate and establish mTLS between every service that is part of a global namespace, even if they're running on multiple clouds, thereby allowing you to secure the communications and establish mTLS encryption without virtual private networks (VPNs) or special gateways. The mTLS encryption across a global namespace is from proxy to proxy with no terminations, thereby preventing man-in-the-middle attacks.

Application publishing and high availability

Most service mesh implementations are focused on service-to-service communications, sometimes referred to as *east-west communications*. Tanzu Service Mesh treats east-west and north-south communications as attributes of the application that need to be managed as part of the same mesh. For this purpose, Tanzu Service Mesh is integrated with VMware NSX Advanced Load Balancer and Amazon Route 53 DNS services to control the north-south policies as part of a global namespace.

After a service (or multiple services) in a global namespace is published, Tanzu Service Mesh configures the integrated global server load balancing (GSLB) service based on its policy, including all health checks and load balancing algorithms. At this point, the GSLB service automatically sends traffic anywhere the published service exists and manages the external facing certificate

imported by the application operator on the ingress controller of the clusters where that service exists. This provides users of the application with access to the published service running in the global namespace in a highly available manner. Also, when deploying the published service in a new Kubernetes cluster or cloud that's part of the global namespace, the application automatically expands by configuring the GSLB to point to the new location without any human intervention or code change, thereby saving time and resources for the business.

At this point, “localization abstraction” has been established. The Tanzu Service Mesh Global Controller is automating the instantiation of service discovery, identity (certificates), mTLS, and load balancing of outbound access into the application. You can now *burst* (expand) to any new cloud or scale to more clusters, or contract automatically as needed, and all communications will be securely established without needing to change a single line of code in your applications.

Actionable SLOs (intelligent autoscaling)

One of the challenges of ensuring a high quality of service is being able to measure the factors that reflect the quality of user experience, such as latencies and error rates.

Tanzu Service Mesh provides these metrics out of the box, without needing additional plug-ins or code changes. Metric levels are displayed in real-time graphs in the Tanzu Service Mesh Console user interface. Tanzu Service Mesh provides an interface where you can configure service-level objective (SLO) targets and select the service-level indicators (SLIs) that determine service health.

Tanzu Service Mesh SLO helps application operators make informed decisions around which parts of the application may need work and where feature development can be accelerated. It also helps configure the Tanzu Service Mesh Service Autoscaler, which further helps ensure that the SLO is met.

With Tanzu Service Mesh SLO, you can observe and monitor the health of your services inside a global namespace or, for services directly in their cluster namespaces, in the user interface, as well as through the API.

SLOs are offered in two ways in Tanzu Service Mesh:

- » **Monitored SLOs:** Monitored SLOs provide an indicator of the performance of the services and whether these services meet the target SLO condition based on the SLIs specified for the service. The monitored SLO policies can be configured either for services inside a global namespace (GNS-scoped SLOs) or for services directly in the cluster (org-scoped SLOs). When configured for a global namespace service, the SLO budget is depleted when any of the service instances inside the global namespace, violates SLIs.
- » **Actionable SLOs:** Like monitored SLOs, actionable SLOs provide an indicator of the health of the services and track how well the services meet the defined SLO target. Unlike monitored SLOs, each actionable SLO can only target a single GNS-scoped service, and they can help influence the service resiliency actions like autoscaling. The actionable SLO policies can be configured for services inside a global namespace. When configured for a global namespace service, the SLO budget is depleted when any of the service instances inside the global namespace violates SLIs.

Tanzu Service Mesh provides application resiliency and scaling through global server load balancing (GLSB) for high availability and failover, cloud bursting, and elastic scalability (see Figure 4-4).

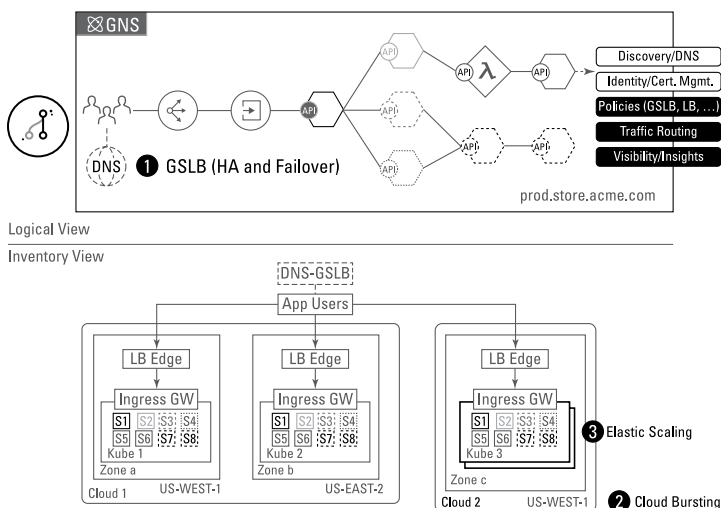


FIGURE 4-4: Tanzu Service Mesh provides application resiliency, elastic autoscaling, and cloud bursting.



You can configure monitored SLOs to monitor the behavior of one or more services and inspect associated performance graphs. With actionable SLOs, you can monitor and also influence autoscaler decisions when SLIs are violated.

Traffic management (progressive upgrades)

Traffic management has many use cases, including traffic shifting based on user headers, resiliency features (such as circuit breakers), and different upgrade strategies. With Tanzu Service Mesh, you can use the native capabilities in Istio that make use of YAML (YAML Ain't Markup Language) declarative manifests on a cluster-by-cluster basis or, with Tanzu Service Mesh being a multi-Kubernetes cluster multi-cloud solution, you can use these capabilities in a scalable distributed cross-cloud manner. For the purpose of operating and managing upgrades, Tanzu Service Mesh has a capability known as *progressive upgrades*, which manages canary and blue-green across clusters at scale.

When upgrading microservices through a canary 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 Hypertext Transfer Protocol (HTTP) errors and latency. If the new service is working fine, you 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 described in Chapter 3, this type of upgrade can be performed relatively easily with Istio using a simple YAML file that defines the routing rules to send a percentage of traffic to the new service (split traffic). However, when you need to perform hundreds of such upgrades a day, it's no longer such an easy task to monitor them and make changes at scale.

With progressive upgrades, Tanzu Service Mesh allows you to define a single upgrade or multiple service upgrades in a global namespace. 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. And you can monitor all your upgrades from a single dashboard.



TIP

You can also test a version with Tanzu Service Mesh progressive upgrades by simulating an upgrade without going “all in” and switching out versions.

Access control policies

Access control policies (ACPs) provide granular access control between microservices. With Tanzu Service Mesh, you can create *authorization policies* (known as *AuthZ*) that are based on the fact that all services, no matter where they reside (as long as they’re a part of the same global namespace), can authenticate with one another so you can apply policies that control access to them. This means that a global namespace becomes an *application security zone*, which the ACP is defined upon. In Tanzu Service Mesh, access can be controlled based on Transmission Control Protocol (TCP) ports or HTTP requests — even as granular as to the specific HTTP path. Tanzu Service Mesh’s dashboarding clearly shows the security posture of each application and which services can access which services (see Figure 4-5).

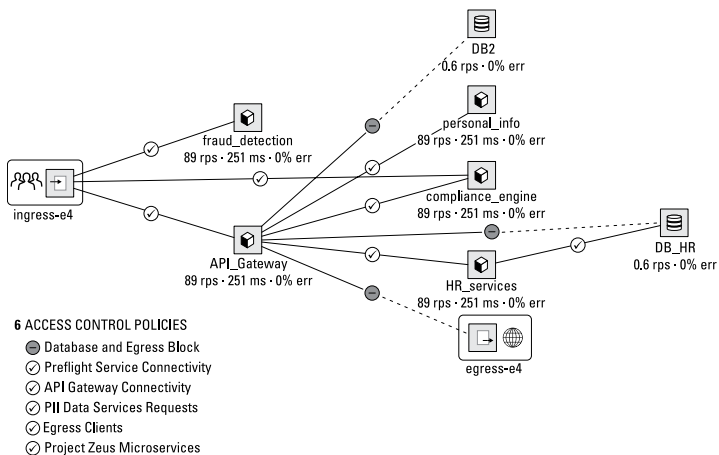


FIGURE 4-5: Access control policies in Tanzu Service Mesh.

API operations and security

Communication between microservices is done using API calls. Every API call passes through the Tanzu Service Mesh data plane. Tanzu Service Mesh can then provide certain capabilities

around operationalizing and securing the API calls, including the following:

- » **Document:** You can extract the API schema of your application allowing teams to document their APIs and validate them during application development. This is a significant benefit because there are usually gaps in the API documentation versus the actual APIs that are developed.
- » **Analyze:** You can perform analytics on your API calls and extract important information about their behavior, including information such as the performance of API calls, location, usage, by whom, how much, and more. This allows application operators to diagnose whether problems are occurring due to the API calls, their use, or another layer, and helps them generally pinpoint issues faster and better from the API layer down.
- » **Security:** You can control exactly which API calls should be allowed between microservices. Whereas access policies and mTLS are foundational Zero Trust architecture capabilities, API security would be considered an advanced Zero Trust capability. API security allows you to secure the application beyond access control whether a service allows a PUT call or a GET call to another service and which parameters work. This can be thought of as an advanced “API firewall.” An organization that has microservices deployed in production can also take the API schema captured during the development process and use it as a baseline for an API security policy in production. This means that if a service is performing an unauthorized API call, Tanzu Service Mesh will either alert about the incident or block it depending on what it’s programmed to do. This is the incarnation of Zero Trust.

PII data leakage protection

Tanzu Service Mesh not only takes care of service-to-service communications, but also provides data security for personally identifiable information (PII) including Social Security numbers (SSNs), bank routing numbers, credit card information, and so on. Tanzu Service Mesh can detect when such data is passing through its data plane proxies based on defined policies and provide alerts in the application views within a global namespace. Customers can create PII data leakage policies to alert or block when data is accessed inappropriately.

For example, if Service A is compromised and credit card numbers are being pulled in a way that violates the PII data policy, Tanzu Service Mesh will either alert the security operators or block the transaction, depending on how the policy is configured. Even if this capability is used just to model where PII data is passing, it's very useful for organizations with regulatory requirements. In Tanzu Service Mesh there are predefined rules to identify specific PII data patterns, and you can also create custom rules to detect PII.

Figure 4-6 shows the security posture of the application in Tanzu Service Mesh.

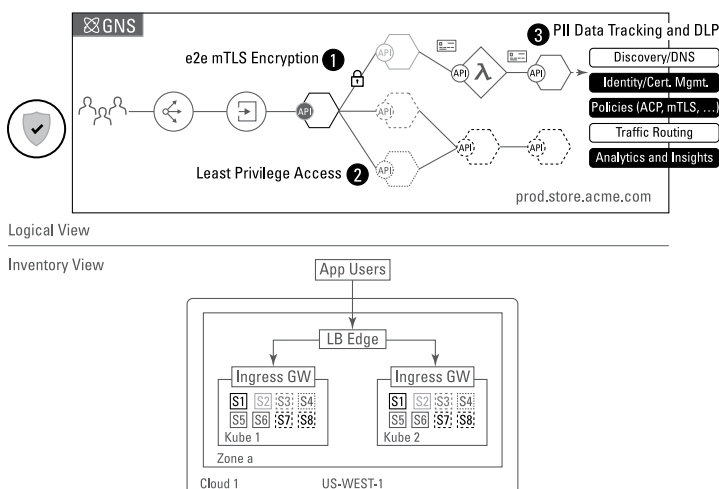


FIGURE 4-6: Tanzu Service Mesh provides zero trust security and PII data leak protection.

East-west threat detection

Another Zero Trust capability in Tanzu Service Mesh is east-west threat detection. Threat detection is usually done at the access point to the network using web application firewalls or other firewalls that can provide deep inspection capabilities. Threat actors typically use numerous tactics, techniques, and procedures (TTPs) to execute a successful attack. Given that many security solutions that are designed to stop an attack are deployed at the network perimeter, east-west traffic usually flows freely within the network, enabling a threat actor to move laterally with ease within your network.

Tanzu Service Mesh can apply threat detection for the Open Web Application Security Project (OWASP) Top Ten web application security risks to east–west traffic. This means that every proxy in the service mesh can detect and stop an OWASP 10 attack. You can also program custom attacks by importing attack signatures.

IN THIS CHAPTER

- » Exploring helpful blogs and books
- » Participating in conferences and meetups
- » Digging into documentation
- » Joining discussion groups
- » Getting started with step-by-step guides and hands-on labs
- » Checking out podcasts and user stories
- » Viewing videos and tutorials
- » Working with VMware resources

Chapter 5

Ten 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 Tanzu Service Mesh.

Blogs

The open-source projects and VMware publish numerous blogs to help you familiarize yourself with service mesh use cases and technology. Check out the following:

- » **Istio:** <https://istio.io/latest/blog>

- » **Kuma:** <https://kuma.io/blog>
- » **Linkerd:** <https://linkerd.io/blog>
- » **Open Service Mesh:** <https://openservicemesh.io/blog>
- » **VMware Network and Security Virtualization:** <https://blogs.vmware.com/networkvirtualization>
- » **VMware Tanzu:** <https://tanzu.vmware.com/blog>

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"? Here are some good places to start:

- » *Getting Started with Istio Service Mesh: Manage Microservices in Kubernetes* by Rahul Sharma and Avinash Singh (Apress)
- » *Istio in Action* by Christian E. Posta and Rinor Maloku (Manning)
- » *Istio: Up & Running: Using a Service Mesh to Connect, Secure, Control, and Observe* by Lee Calcote and Zack Butcher (O'Reilly Media)
- » *Mastering Service Mesh* by Anjali Khatri and Vikram Khatri (Packt Publishing)
- » *Istio Succinctly* by Rahual Rai and Tarun Pabbi (Syncfusion)
- » *Consul: Up & Running* by Luke Kysow (O'Reilly)
- » *Service Mesh Standard Requirements: Practical Tools for Self-Assessment* by Gerardus Blokdyk (5STARCook's)

Conferences and Meetups

Attend a conference or join a local Meetup group to learn from your peers operating server meshes:

- » **GOTO Conferences:** Watch the *Service Meshes: Istio, Linkerd — or No Mesh at All?* presentation at https://youtu.be/kwUgrSG_ZKI.

- » **IstioCon 2022:** The community conference for Istio service mesh. Learn more at <https://events.istio.io/istiocon-2022>.
- » **Meetup:** Join a local meetup at www.meetup.com/topics/service-mesh and engage in dialogue with the Istio user community at www.meetup.com/topics/istio.
- » **ServiceMeshCon:** A vendor-neutral conference on service mesh. Find out more at <https://events.linuxfoundation.org/servicemeshcon-europe>.

Documentation

The various open-source service mesh projects publish a wide range of resources to help you get grounded on the projects and info on how you can contribute. Check out the following resources:

- » **Istio:** <https://istio.io>
- » **Kuma:** <https://kuma.io>
- » **Linkerd:** <https://linkerd.io>
- » **Open Service Mesh:** <https://release-v1-0.docs.openservicemesh.io>

Although not a service mesh, documentation for related technologies include the following:

- » **Envoy service proxy:** www.envoyproxy.io
- » **Multi-Vendor Service Mesh Interoperation:** <https://github.com/vmware/hamlet>

Discussion Groups

Join a discussion group to post questions and actively contribute to the various open-source service mesh projects and follow them on Twitter:

- » **Istio:** <https://discuss.istio.io>
- » **Istio on Twitter:** <https://twitter.com/istiomesh>

- » **Kuma on Twitter:** <https://twitter.com/kumamesh>
- » **Linkerd on Twitter:** <https://twitter.com/linkerd>
- » **Open Service Mesh on Twitter:** <https://twitter.com/openservicemesh>

Getting Started Guides

These step-by-step guides will help you get started:

- » **Istio: Getting Started:** <https://istio.io/latest/docs/setup/getting-started>
- » **Istio: Platform Setup:** <https://istio.io/latest/docs/setup/platform-setup/>
- » **Kuma: Getting Started with Kuma Service Mesh:** <https://konghq.com/blog/getting-started-kuma-service-mesh>
- » **Linkerd: Getting Started:** <https://linkerd.io/2.11/getting-started>
- » **Open Service Mesh: Getting Started:** https://release-v1-0.docs.openservicemesh.io/docs/getting_started

Hands-On Labs and Online Courses

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
- » **Modern Apps Ninja: Tanzu for Kubernetes Operation:** https://modernapps.ninja/course/intrototko_tk5598/
- » **VMware Tanzu for Kubernetes Operations:** <https://labs.hol.vmware.com/HOL/catalogs/lab/10414>
- » **VMware: Tanzu Service Mesh Hands-on-Lab:** <https://labs.hol.vmware.com/HOL/catalogs/lab/8509>

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

- » **Coursera:** www.coursera.org/search?query=service%20mesh
- » **DevOps School:** <https://devopsschool.com/courses/istio>
- » **edX:** www.edx.org/course/introduction-to-service-mesh-with-linkerd
- » **KodeKloud:** <https://kodekloud.com/courses/istio-service-mesh>
- » **LinkedIn:** www.linkedin.com/learning/kubernetes-service-mesh-with-istio
- » **NobleProg:** www.nobleprog.com/cc/istio and www.nobleprog.com/cc/linkerd
- » **Pluralsight:** www.pluralsight.com/courses/istio-managing-apps-kubernetes
- » **Udemy:** www.udemy.com/topic/istio
- » **Virtual Pair Programmers:** www.virtualpairprogrammers.com/training-courses/Istio-training.html

Podcasts and User Stories

Listen to podcasts on service mesh and Linkerd:

- » **The InfoQ Podcast:** www.infoq.com/ServiceMesh/podcasts
- » **Buoyant:** <https://buoyant.io/media/podcasts>

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:

- » **Istio case studies:** <https://istio.io/latest/about/case-studies>
- » **Linkerd case studies:** <https://linkerd.io/community/adopters>

Videos and Tutorials

Watch the following videos and tutorials from the Just Me and Opensource YouTube channel (www.youtube.com/c/wenkatn-justmeandopensource) to go in-depth with Istio:

- » **Installing Istio in Kubernetes Cluster:** <https://youtu.be/WFu8OLXUETY>
- » **Deploying Istio Service Mesh in Kubernetes Using Istioctl:** <https://youtu.be/wdusXMYeddg>
- » **Istio Deploying Sample Bookinfo Application:** <https://youtu.be/7D6wvqdJ9oU>
- » **Monitoring Istio Mesh Using Grafana:** https://youtu.be/J_04pTPjR9o
- » **Kiali – Visualize Your Istio Service Mesh:** https://youtu.be/dpb_q42uzy4
- » **Istio Traffic Management – Request Routing Part 1:** <https://youtu.be/J4FM1-CICTE>
- » **Istio Traffic Management – Request Routing Part 2:** <https://youtu.be/OI8PcxHx3to>
- » **Istio Demo with Kiali and Traffic Management:** <https://youtu.be/0w23AQ-hrF4>

VMware Products and Reference Designs

VMware offers a wide range of resources to help you understand the enterprise requirements for service mesh. Visit the following pages to learn more:

- » **VMware Tanzu Service Mesh Documentation:** <https://docs.vmware.com/en/VMware-Tanzu-Service-Mesh/index.html>
- » **VMware NSX Advanced Load Balancer Documentation:** <https://docs.vmware.com/en/VMware-NSX-Advanced-Load-Balancer/index.html>

- » **VMware Tanzu for Kubernetes Operations on VMware Cloud on AWS Reference Design:** <https://docs.vmware.com/en/VMware-Tanzu/services/tanzu-reference-architecture/GUID-reference-designs-tko-on-vmc-aws.html>
- » **VMware Tanzu for Kubernetes Operations on vSphere Reference Design:** <https://docs.vmware.com/en/VMware-Tanzu/services/tanzu-reference-architecture/GUID-reference-designs-tko-on-vsphere.html>
- » **VMware Tanzu for Kubernetes Operations on vSphere with NSX-T Reference Design:** <https://docs.vmware.com/en/VMware-Tanzu/services/tanzu-reference-architecture/GUID-reference-designs-tko-on-vsphere-nsx.html>

Transform your multi-cloud network with service mesh

A service mesh is an abstraction layer that takes care of service-to-service communication, observability, availability and resiliency, and security 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'll also find out how to address the challenges that service mesh itself brings using VMware Tanzu 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 Tanzu Service Mesh

vmware®

Niran Even-Chen (@niranec) is a Senior Product Line Manager in the VMware's Networking and Security Advanced Business Group (NASBG) and a 3x VCDX.

Oren Penso (@openso) is the EMEA Field CISO for VMware Tanzu. **Sergio Pozo** is also a Senior Product Line Manager in VMware's NASBG. **Susan Wu** (@susanwu88) is a Senior Product Marketing Manager at VMware.

Go to **Dummies.com™**
for videos, step-by-step photos,
how-to articles, or to shop!

ISBN: 978-1-119-86667-1

Not For Resale

for
dummies®
A Wiley Brand



WILEY END USER LICENSE AGREEMENT

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