The InfoQ eMag / Issue #86 / July 2020

Service Mesh Ultimate Guide

InfoQ

Service Mesh Features

Service Mesh Implementations and Products

Exploring the (Possible) Future of Service Meshes

FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

Service Mesh Ultimate Guide

IN THIS ISSUE

5 – The Service Mesh Pattern
6 – Service Mesh Features
7 – Service Mesh Architecture: Looking Under the Hood
8 – Use Cases
9 – Antipatterns
10 – Service Mesh Implementations and Products
10 – Service Mesh Comparisons: Which Service Mesh?
11 – History of the Service Mesh
12 – Exploring the (Possible) Future of Service Meshes
13 – FAQ
16 – Additional Resources
16 – Glossary

PRODUCTION EDITOR Ana Ciobotaru / COPY EDITORS Lawrence Nyveen & Susan Conant / DESIGN Dragos Balasoiu
GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

Key Takeaways

- A service mesh manages all service-to-service communication within a distributed (potentially microservice-based) software system. It accomplishes this typically via the use of "sidecar" proxies that are deployed alongside each service through which all traffic is transparently routed.
- Proxies used within a service mesh are typically "application layer" aware (operating at Layer 7 in the OSI networking stack). This means that traffic routing decisions and the labeling of metrics can draw upon data in HTTP headers or other application layer protocol metadata.
- A service mesh provides dynamic service discovery and traffic management, including traffic shadowing (duplicating) for testing, and traffic splitting for canary releasing, incremental rollout, and A/B type experimentation.
- A service mesh also supports the implementation and enforcement of cross cutting requirements, such as security (providing service identity and TLS) and reliability (rate limiting, circuit-breaking).
- As a service mesh is on the critical path for every request being handled within the system, it can also provide additional "observability," such as distributed tracing of a request, frequency of HTTP error codes, global and service-to-service latency.
- There are clear benefits provided by the use of a service mesh, but the tradeoffs of added complexity and the requirement of additional runtime resources should be analyzed.
- Service mesh technology is rapidly becoming part of the (cloud native) application platform "plumbing." The interesting innovation within this space is happening in relation to the higher-level abstractions and the human-focused control planes.
- Popular service meshes include: Linkerd, Istio, Consul, Kuma, and Maesh. Supporting technologies within this space include: Layer 7-aware proxies, such as Envoy, HAProxy, NGINX, and MOSN; and service mesh orchestration, visualization, and understandability tooling, such as SuperGloo, Kiali, and Dive.

Daniel Bryant

is leading change within organisations and technology. His current technical expertise focuses on 'DevOps' tooling, cloud/container platforms and microservice implementations. Daniel ia a leader within the London Java Community (LJC), contributes to several open source projects, writes for well-known technical websites such as InfoQ, DZone and Voxxed, and regularly presents at international conferences such as QCon, JavaOne and Devoxx.

Around 2016, the term "service mesh" appeared to spring from nowhere in the arenas of microservices, cloud computing, and DevOps in. However, as with many concepts within computing, there is actually a long history of the associated pattern and technology.

The arrival of the service mesh has largely been due to a perfect storm within the IT landscape. Developers began building distributed systems using a multi-language (polyglot) approach, and needed dynamic service discovery. Operations began using ephemeral infrastructure, and wanted to gracefully handle the inevitable communication failures and enforce network policies. Platform teams began embracing container orchestration systems like Kubernetes, and wanted to dynamically route traffic in and around the system using modern API-driven network proxies, such as Envoy.

This eMag aims to answer pertinent questions for software architects and technical leaders, such as: what is a service mesh?, do I need a service mesh?, and how do I evaluate the different service mesh offerings?

The Service Mesh Pattern

The service mesh pattern is focusing on managing all service-to-service communication within a distributed software system.

Context

The context for the pattern is twofold: First, that engineers have adopted the microservice architecture pattern, and are building their applications by composing multiple (ideally single-purpose and independently deployable) services together. Second, that the organization has embraced cloud native platform technologies such as containers (e.g., Docker), orchestrators (e.g., Kubernetes), and proxies/gateways (e.g., Envoy).

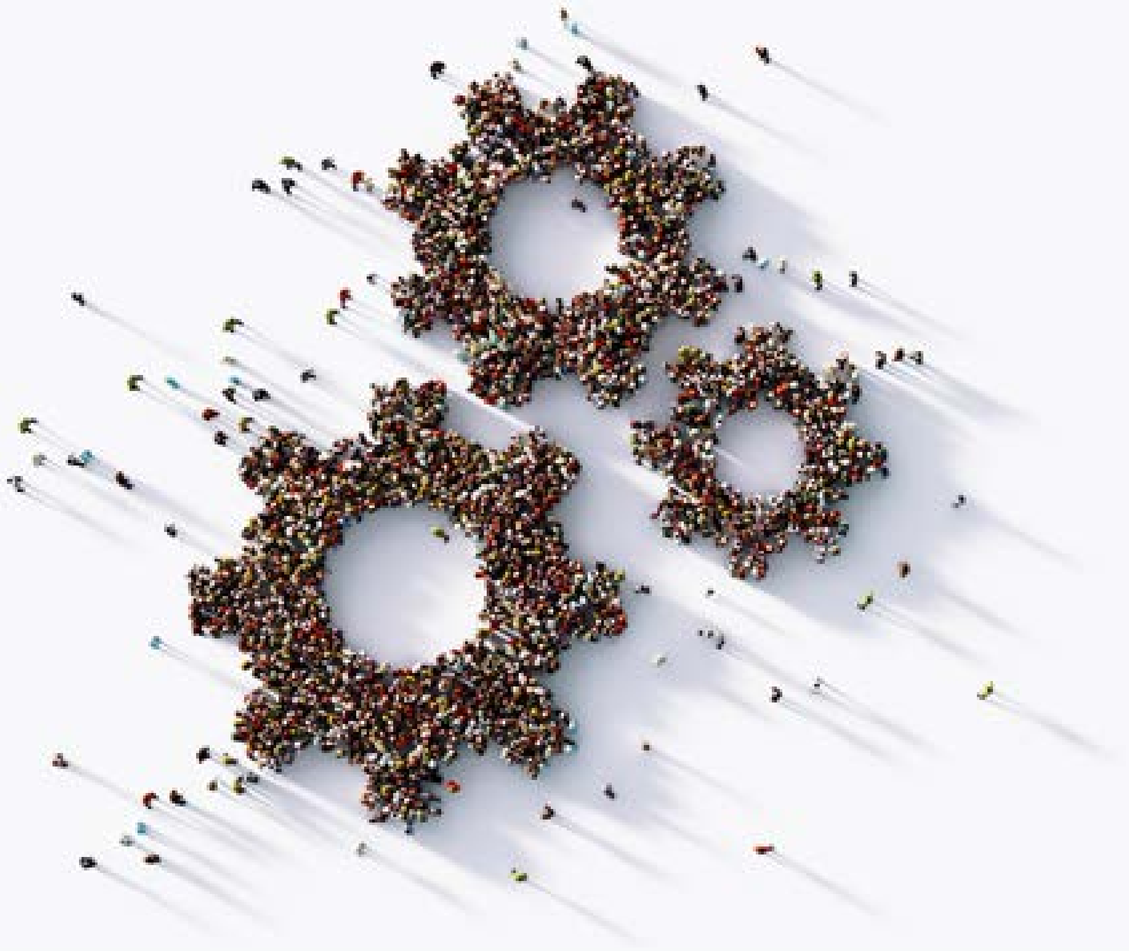
Intent

The problems that the service mesh pattern attempts to solve include:

- Eliminating the need to compile into individual services a language-specific communication library to handle service discovery, routings, and application-level (Layer 7) non-functional communication requirements.
- Externalizing service communication configuration, including network locations of external services, security credentials, and quality of service targets.
- Providing passive and active monitoring of other services.
- Decentralizing the enforcement of policy throughout a distributed system.
- Providing observability defaults and standardizing the collection of associated data.
  - Enabling request logging
  - Configuring distributed tracing
  - Collecting metrics

Structure

The service mesh pattern primarily focuses on handling traditionally what has been referred to as "east-west" remote procedure call (RPC)-based traffic: request/response type communication that originates internally within a datacenter and travels service-to-service. This is in contrast to an API gateway or edge proxy, which is designed to handle "north-south" traffic: Communication that originates externally and ingresses to an endpoint or service within the datacenter.



Service Mesh Features

A service mesh implementation will typically offer one or more of the following features:

- Normalizes naming and adds logical routing, (e.g., maps the code-level name “user-service” to the platform specific location “AWS-us-east-1a/prod/users/v4”)
- Provides traffic shaping and traffic shifting
- Maintains load balancing, typically with configurable algorithms
- Provides service release control (e.g., canary releasing and traffic splitting)
- Offers per-request routing (e.g., traffic shadowing, fault injection, and debug re-routing)
- Adds baseline reliability, such as health checks, timeouts/deadlines, circuit breaking, and retry (budgets)
- Increases security, via transparent mutual Transport Level Security (TLS) and policies such as Access Control Lists (ACLs)
- Provides additional observability and monitoring, such as top-line metrics (request volume, success rates, and latencies), support for distributed tracing, and the ability to “tap” and inspect real-time service-to-service communication
- Enables platform teams to configure “sane defaults” to protect the system from bad communication

Service Mesh Architecture: Looking Under the Hood

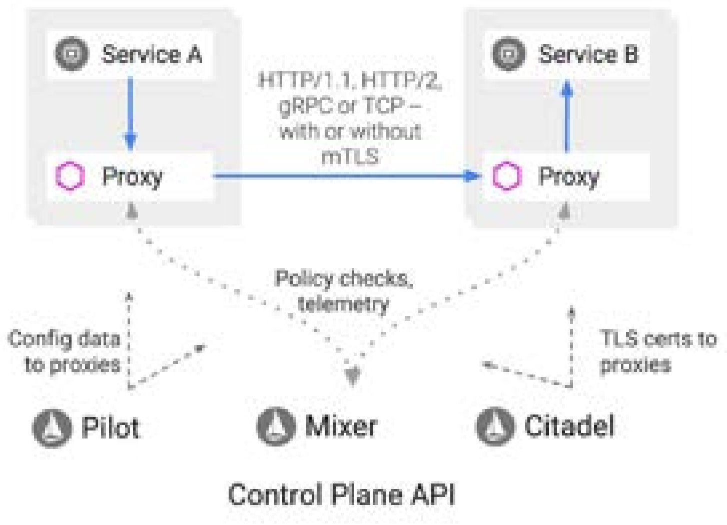
A service mesh consists of two high-level components: a data plane and a control plane. Matt Klein, creator of the Envoy Proxy, has written an excellent deep-dive into the topic of "service mesh data plane versus control plane."

Broadly speaking, the data plane "does the work" and is responsible for "conditionally translating, forwarding, and observing every network packet that flows to and from a [network endpoint]." In modern systems, the data plane is typically implemented as a proxy, (such as Envoy, HAProxy or MOSN), that is run out-of-process alongside each service as a "sidecar."

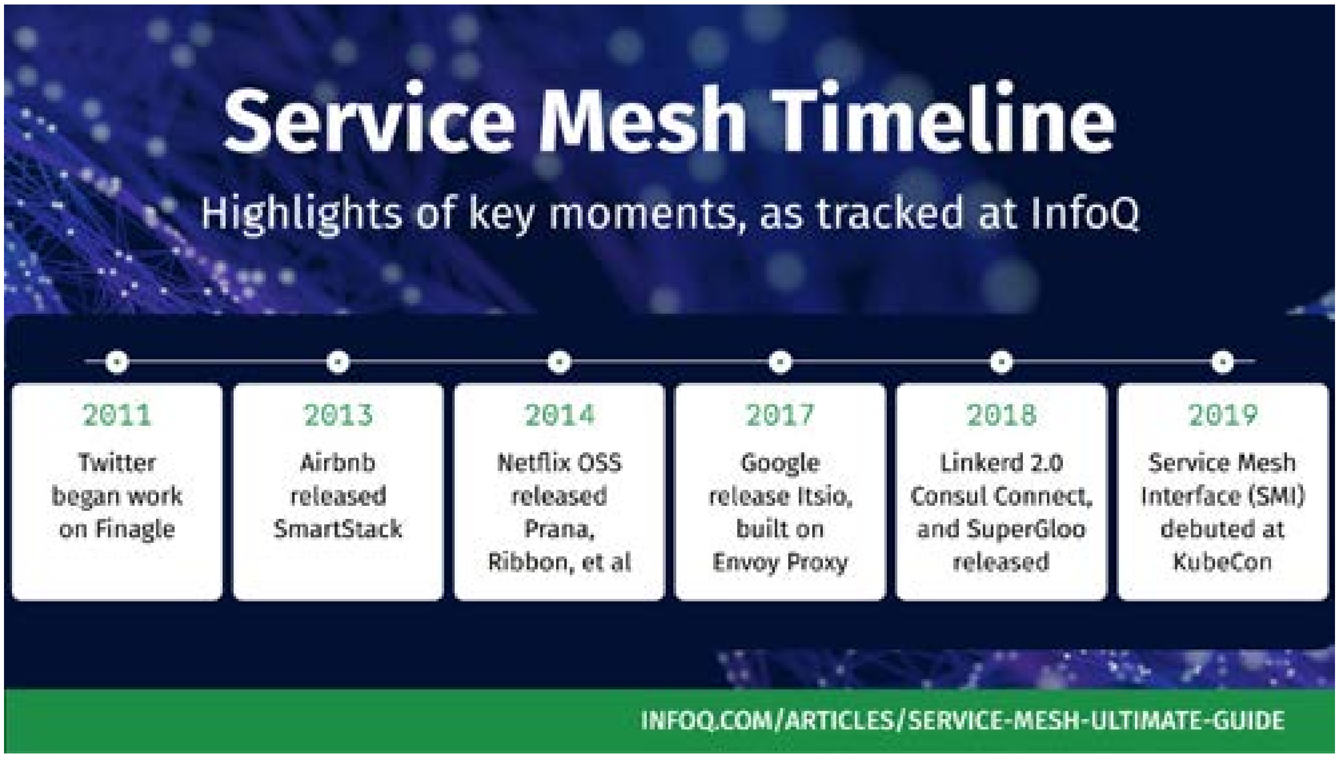
Klein states that within a service mesh, the data plane "touches every packet/request in the system, and is responsible for service discovery, health checking, routing, load balancing, authentication/authorization, and observability." There is work underway within the CNCF to create a Universal Data Plane API, based on concepts from Klein's earlier blog post The Universal Data Plane API. This proposal extends the xDS API that has been defined and implemented by Envoy and is supported in other proxies such as MOSN.

A control plane "supervises the work," and takes all the individual instances of the data plane — a set of isolated stateless sidecar proxies—and turns them into a distributed system. The control plane doesn't touch any packets/requests in the system, but instead, it allows a human operator to provide policy and configuration for all of the running data planes in the mesh. The control plane also enables the data plane telemetry to be collected and centralized, ready for consumption by an operator; Red Hat has been working on Kiali for just this use case.

The diagram below is taken from the Istio architecture documentation, and although the technologies labeled are specific to Istio, the components are general to all service mesh implementation.



Istio architecture, demonstrating the how the control plane and proxy data plane interact (courtesy of the Istio documentation)


Service Mesh Timeline
Highlights of key moments, as tracked at InfoQ
2011
Twitter began work on Finagle
2013
Airbnb released SmartStack
2014
Netflix OSS released Prana, Ribbon, et al
2017
Google release Itsio, built on Envoy Proxy
2018
Linkerd 2.0 Consul Connect, and SuperGloo released
2019
Service Mesh Interface (SMI) debuted at KubeCon
INFOQ.COM/ARTICLES/SERVICE-MESH-ULTIMATE-GUIDE

Use Cases

There are a variety of use cases that a service mesh can enable or support.

Dynamic Service Discovery and Routing

A service mesh provides dynamic service discovery and traffic management, including traffic shadowing (duplicating) for testing, and traffic splitting for canary releasing and A/B type experimentation.

Proxies used within a service mesh are typically “application layer” aware (operating at Layer 7 in the OSI networking stack). This means that traffic routing decisions and the labeling of metrics can draw upon data in HTTP headers or other application layer protocol metadata.

Service-to-Service Communication Reliability

A service mesh supports the implementation and enforcement of cross cutting reliability requirements, such as request retries, timeouts, rate limiting, and circuit-breaking. A service mesh is often used to compensate (or encapsulate) dealing with the eight fallacies of distributed computing. It should be noted that a service mesh can only offer wire-level reliability support (such as retrying an HTTP request), and ultimately the service should be responsible for any related business impact such as avoiding multiple (non idempotent) HTTP POST requests.

Observability of Traffic

As a service mesh is on the critical path for every request being handled within the system, it can also provide additional “observability,” such as distributed tracing of a request, frequency of HTTP error codes, global and service-to-service latency. Although a much overused phrase in the enterprise space, service meshes are often proposed as a method to capture all of the data necessary to implement a “single pane of glass” view of traffic flows within the entire system.

Communication Security

A service mesh also supports the implementation and enforcement of cross cutting security requirements, such as providing service identity (via x509 certificates), enabling application-level service/network segmentation (e.g. «service A» can communicate with «service B», but not service C») ensuring all communication is encrypted (via TLS), and ensuring the presence of valid user-level identity tokens or "passports."

Antipatterns

It is often a sign of a maturing technology when antipatterns of usage emerge. Service meshes are no exception.

Too Many Traffic Management Layers (Turtles All the Way Down)

This antipattern occurs when developers do not coordinate with the platform or operations team, and duplicate existing communication handling logic in code that is now being implemented via a service mesh. For example, an application implementing a retry policy within the code in addition to a wire-level retries policy provided by the service mesh configuration. This antipattern can lead to issues such as duplicated transactions.

Service Mesh Silver Bullet

There is no such thing as a "silver bullet" within IT, but vendors are sometimes tempted to anoint new technologies with this label. A service mesh will not solve all communication problems with microservices, container orchestrators like Kubernetes, or cloud networking. A service mesh aims to facilitate service-to-service (east-west) communication only, and there is a clear operational cost to deploying and running a service mesh.

Enterprise Service Bus (ESB) 2.0

During the pre-microservice service-oriented architecture (SOA) era the Enterprise Service Buses (ESB) implemented a communication system between software components. Some fear that many of the mistakes from the ESB era will be repeated with the use of a service mesh.

The centralized control of communication offered via ESBs clearly had value. However, the development of the technologies was driven by vendors, which led to multiple problems, such as: a lack of interoperability between ESBs, bespoke extension of industry standards (e.g., adding vendor-specific configuration to WS-* compliant schema), and high cost. ESB vendors also did nothing to discourage the integration and tight-coupling of business logic into the communication bus.

Big Bang Deployment

There is a temptation within IT at large to believe that a big bang approach to deployment is the easiest approach to manage, but as research from Accelerate and the State of DevOps Report, this is not the case. As a complete rollout of a service mesh means that this technology is on the critical path for handling all end user requests, a big bang deployment is highly risky.

Service Mesh Implementations and Products

The following is a non-exhaustive list of current service mesh implementations:

- Linkerd
- Istio
- Consul
- Kuma
- Maesh
- AWS App Mesh

Service Mesh Comparisons: Which Service Mesh?

The service mesh space is extremely fast moving, and so any attempt to create a comparison is likely to quickly become out of date. However, several comparisons do exist. Care should be taken to understand the source's bias (if any) and the date that the comparison was made.

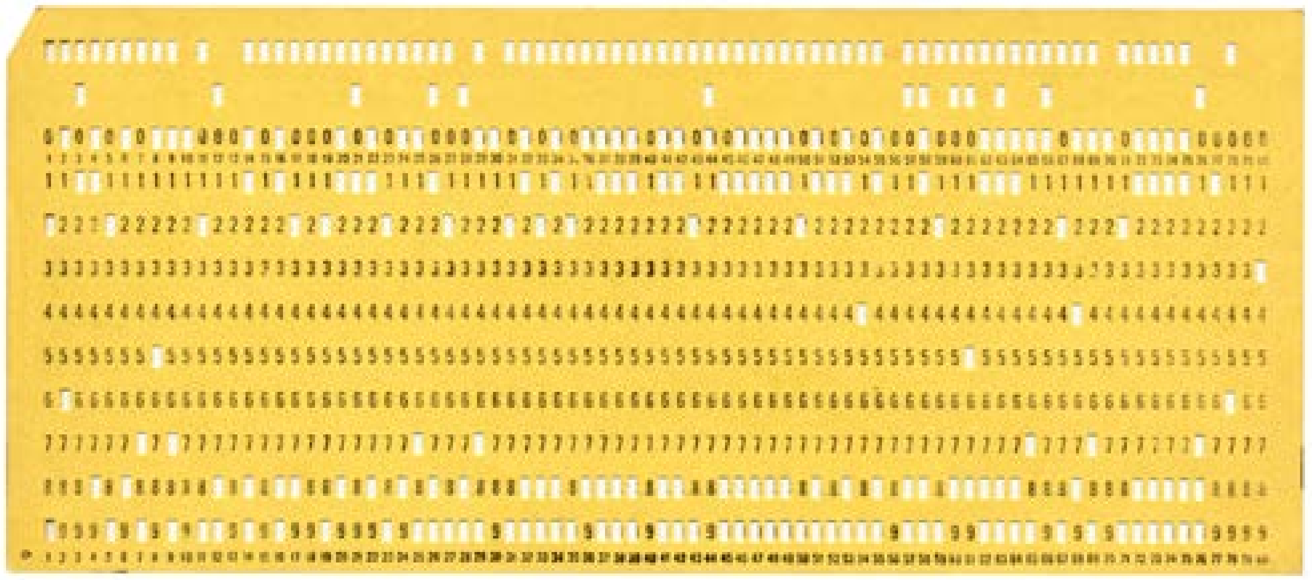
- https://layer5.io/landscape
- https://kubedex.com/istio-vs-linkerd-vs-linkerd2-vs-consul/ (correct as of May 2019)
- https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/ (up to date as of October 2019)
- https://servicemesh.es/ (last published February 2020)

InfoQ always recommends that service mesh adopters perform their own due diligence and experimentation on each offering.

Service Mesh Tutorials

For engineers or architects looking to experiment with multiple service meshes the following tutorials, playgrounds, and tools are available:

- Layer 5 Meshery — a multi service mesh management plane.
- Solo's SuperGloo — a service mesh orchestration platform
- KataCoda Istio tutorial
- Consul service mesh tutorial
- Linkerd tutorial



History of the Service Mesh

InfoQ has been tracking the topic that we now call service mesh since late 2013, when Airbnb released SmartStack, which offered an out-of-process service discovery mechanism (using HAProxy) for the emerging “microservices” style architecture. Many of the previously labeled “unicorn” organizations were working on similar technologies before this date. From the early 2000s Google was developing its Stubby RPC framework that evolved into gRPC, and the Google Frontend (GFE) and Global Software Load Balancer (GSLB), traits of which can be seen in Istio. In the earlier 2010s, Twitter began work on the Scala-powered Finagle from which the Linkerd service mesh emerged.

In late 2014, Netflix released an entire suite of JVM-based utilities including Prana, a “sidecar” process that allowed application services written in any language to communicate via HTTP to standalone instances of the libraries. In 2016, the NGINX team began talking about “The Fabric Model,” which was very similar to a service mesh, but required the use of their commercial NGINX Plus product for implementation.

Other highlights from the history of the service mesh include the releases of Istio in May 2017, Linkerd 2.0 in July 2018, Consul Connect and Super-Gloo in November 2018, service mesh interface (SMI) in May 2019, and Maesh and Kuma in September 2019.

Even service meshes that emerged outside of the unicorns, such as HashiCorp's Consul, took inspiration from the aforementioned technology, often aiming to implement the CoreOS coined concept of "GIFEE"; Google infrastructure for everyone else.

For a deep-dive into the history of how the modern service mesh concept evolved, Phil Calçado has written a comprehensive article "Pattern: Service Mesh."

Exploring the (Possible) Future of Service Meshes

As service mesh technology is still within the early adoption phase, there is a lot of scope for future work. Broadly speaking, there are four areas of particular interest: adding support for use cases beyond RPC, standardizing the interface and operations, pushing the service mesh further into the platform fabric, and building effective human control planes for service mesh technology.

Kasun Indrasiri has explored "The Potential for Using a Service Mesh for Event-Driven Messaging," in which he discussed two main emerging architectural patterns for implementing messaging support within a service mesh: the protocol proxy sidecar, and the HTTP bridge sidecar. This is an active area of development within the service mesh community, with the work towards supporting Apache Kafka within Envoy attracting a fair amount of attention.

Christian Posta has previously written about attempts to standardize the usage of service meshes in "Towards a Unified, Standard API for Consolidating Service Meshes." This article also discusses the Service Mesh Interface (SMI) that was recently announced by Microsoft and partners at KubeCon EU. The SMI defines a set of common and portable APIs that aims to provide developers with interoperability across different service mesh technologies including Istio, Linkerd, and Consul Connect.

The topic of integrating service meshes with the platform fabric can be further divided into two sub-topics.

First, there is work being conducted to reduce the networking overhead introduced by a service mesh data plane. This includes the data plane development kit (DPDK), which is a userspace application that "bypasses the heavy layers of the Linux kernel networking stack and talks directly to the network hardware," and work by the Cilium team that utilizes the extended Berkley Packet Filter (eBPF) functionality in the Linux kernel for "very efficient networking, policy enforcement, and load balancing functionality." Another team is mapping the concept of a service mesh to L2/L3 payloads with Network Service Mesh, as an attempt to "re-imagine network function virtualization (NFV) in a cloud-native way."

Second, there are multiple initiatives to integrate service meshes more tightly with public cloud platforms, as seen in the introduction of AWS App Mesh, GCP Traffic Director, and Azure Service Fabric Mesh.

The Buoyant team is leading the charge with developing effective human-centric control planes for service mesh technology. They have recently released Dive, a SaaS-based "team control plane" for platform teams operating Kubernetes. Dive adds higher-level, human-focused, functionality on top of the Linkerd service mesh, and provides a service catalog, an audit log of application releases, a global service topology, and more.

FAQ

What is a service mesh?

A service mesh is a technology that manages all service-to-service, "east-west," traffic within a distributed (potentially microservice-based) software system. It provides both business-focused functional operations, such as routing, and nonfunctional support, for example, enforcing security policies, quality of service, and rate limiting. It is typically (although not exclusively) implemented using sidecar proxies through which all services communicate through.

How does a service mesh differ from an API gateway?

A service mesh manages all service-to-service, "east-west," traffic within a distributed (potentially microservice-based) software system. It provides both business-focused functional operations, such as routing, and nonfunctional support, for example, enforcing security policies, quality of service, and rate limiting.

An API gateway manages all ingress, "north-south," traffic into a cluster, and provides additional support for cross-functional communication requirements. It acts as the single entry point into a system and enables multiple APIs or services to act cohesively and provide a uniform experience to the user.

If I am deploying microservices, do I need a service mesh?

Not necessarily. A service mesh adds operational complexity to the technology stack, and therefore is typically only deployed if the organization is having trouble scaling service-to-service communication, or has a specific use case to resolve.

Do I need a service mesh to implement service discovery with microservices?

No. A service mesh provides one way of implementing service discovery. Other solutions include language-specific libraries (such as Ribbon and Eureka, or Finagle)

Does a service mesh add overhead/latency to my service-to-service communication?

Yes, a service mesh adds at least two extra network hops when a service is communicating with another service (the first is from the proxy handling the source's outbound connection, and the second is from the proxy handling the destination's inbound connection). However, this additional network hop typically occurs over the localhost or loopback network interface, and adds only a small amount of latency (on the order of milliseconds). Experimenting with and understanding whether this is an issue for the target use case should be part of the analysis and evaluation of a service mesh.

Shouldn't a service mesh be part of Kubernetes or the "cloud native platform" that applications are being deployed onto?

Potentially. There is an argument for maintaining separation of concerns within cloud native platform components (e.g., Kubernetes is responsible for providing container orchestration and a service mesh is responsible for service-to-service communication). However, work is underway to push service mesh-like functionality into modern Platform-as-a-Service (PaaS) offerings.

How do I implement, deploy, or rollout a service mesh?

The best approach would be to analyse the various service mesh products (see above), and follow the implementation guidelines specific to the chosen mesh. In general, it is best to work with all stakeholders and incrementally deploy any new technology into production.

Can I build my own service mesh?

Yes, but the more pertinent question is should you? Is building a service mesh a core competency of your organization? Could you be providing value to your customers in a more effective way? Are you also committed to maintaining your own mesh, patching it for security issues, and constantly updating it to take advantage of new technologies? With the range of open source and commercial service mesh offerings that are now available, it is most likely more effective to use an existing solution.

Which team owns the service mesh within a software delivery organization?

Typically the platform or operations team own the service mesh, along with Kubernetes and the continuous delivery pipeline infrastructure. However, developers will be configuring the service mesh properties, and so both teams should work closely together. Many organizations are following the lead from the cloud vanguard such as Netflix, Spotify, and Google, and are creating internal platform teams that provide tooling and services to full cycle product-focused development teams.

Is Envoy a service mesh?

No. Envoy is a cloud native proxy that was originally designed and built by the Lyft team. Envoy is often used as the data plane with a service mesh. However, in order to be considered a service mesh, Envoy must be used in conjunction with a control plane in order for this collection of technologies to become a service mesh. The control plane can be as simple as a centralized config file repository and metric collector, or a comprehensive/complex as Istio.

Can the words "Istio" and "service mesh" be used interchangeably?

No. Istio is a type of service mesh. Due to the popularity of Istio when the service mesh category was emerging, some sources were conflating Istio and service mesh. This issue of conflation is not unique to service mesh—the same challenge occurred with Docker and container technology.

Which service mesh should I use?

There is no single answer to this question. Engineers must understand their current requirements, and the skills, resources, and time available for their implementation team. The service mesh comparison links above will provide a good starting point for exploration, but we strongly recommend that organizations experiment with at least two meshes in order to understand which products, technologies, and workflows work best for them.

Can I use a service mesh outside of Kubernetes?

Yes. Many service meshes allow the installation and management of data plane proxies and the associated control plane on a variety of infrastructure. HashiCorp's Consul is the most well known example of this, and Istio is also being used experimentally with Cloud Foundry.

Additional Resources

- InfoQ Service Mesh homepage
- The InfoQ eMag - Service Mesh: Past, Present, and Future
- The Service Mesh: What Every Software Engineer Needs to Know about the World's Most Over-Hyped Technology
- Service Mesh Comparison
- Service Meshes

Glossary

API gateway: Manages all ingress (north-south) traffic into a cluster, and provides additional. It acts as the single entry point into a system and enables multiple APIs or services to act cohesively and provide a uniform experience to the user.

Consul: A Go-based service mesh from HashiCorp.

Control plane: Takes all the individual instances of the data plane (proxies) and turns them into a distributed system that can be visualized and controlled by an operator.

Data plane: A proxy that conditionally translates, forwards, and observes every network packet that flows to and from a service network endpoint.

East-West traffic: Network traffic within a data center, network, or Kubernetes cluster. Traditional network diagrams were drawn with the service-to-service (inter-data center) traffic flowing from left to right (east to west) in the diagrams.

Envoy Proxy: An open-source edge and service proxy, designed for cloud-native applications. Envoy is often used as the data plane within a service mesh implementation.

Ingress traffic: Network traffic that originates from outside the data center, network, or Kubernetes cluster.

Istio: C++ (data plane) and Go (control plane)-based service mesh that was originally created by Google and IBM in partnership with the Envoy team from Lyft.

Kubernetes: A CNCF-hosted container orchestration and scheduling framework that originated from Google.

Kuma: A Go-based service mesh from Kong.

Linkerd: A Rust (data plane) and Go (control plane) powered service mesh that was derived from an early JVM-based communication framework at Twitter.

Maesh: A Go-based service mesh from Containous, the maintainers of the Traefik API gateway.

MOSN: A Go-based proxy from the Ant Financial team that implements the (Envoy) xDS APIs.

North-South traffic: Network traffic entering (or ingressing) into a data center, network, or Kubernetes cluster. Traditional network diagrams were drawn with the ingress traffic entering the data center at the top of the page and flowing down (north to south) into the network.

Proxy: A software system that acts as an intermediary between endpoint components.

Segmentation: Dividing a network or cluster into multiple sub-networks.

Service mesh: Manages all service-to-service (east-west) traffic within a distributed (potentially micros-ervice-based) software system. It provides both functional operations, such as routing, and nonfunctional support, for example, enforcing security policies, quality of service, and rate limiting.

Service Mesh Interface (SMI): A work-in-progress standard interface for service meshes deployed onto Kubernetes.

Service mesh policy: A specification of how a collection of services/endpoints are allowed to communi-cate with each other and other network endpoints.

Sidecar: A deployment pattern, in which an additional process, service, or container is deployed alongside an existing service (think motorcycle sidecar).

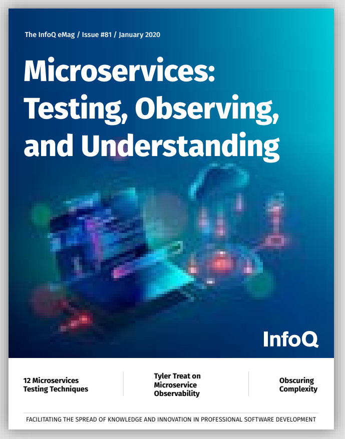
Single pane of glass: A UI or management console that presents data from multiple sources in a unified display.

Traffic shaping: Modifying the flow of traffic across a network, for example, rate limiting or load shedding.

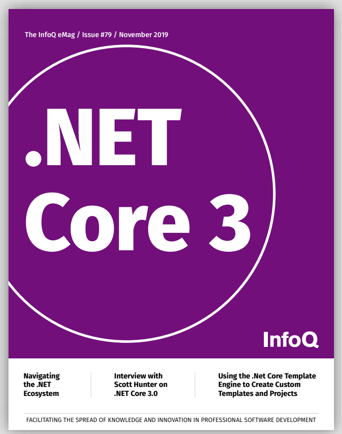
Traffic shifting: Migrating traffic from one location to another.

InfoQ   @ InfoQ   InfoQ   InfoQ

Curious about previous issues?


The InfoQ eMag / Issue #81 / January 2020
Microservices: Testing, Observing, and Understanding
InfoQ
12 Microservices Testing Techniques
Tyler Treat on Microservice Observability
Obscuring Complexity
FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

This eMag takes a deep dive into the techniques and culture changes required to successfully test, observe, and understand microservices.


The InfoQ eMag / Issue #79 / November 2019
.NET Core 3
InfoQ
Navigating the .NET Ecosystem
Interview with Scott Hunter on .NET Core 3.0
Using the .Net Core Template Engine to Create Custom Templates and Projects
FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

In this eMag we explore some more of the benefits of .NET Core and how it can benefit not only traditional .NET developers, but all technologists who need to bring robust, performant and economical solutions to market.

The InfoQ eMag / Issue #77 / October 2019
Taming Complex Systems in Production
InfoQ
An Engineer's Guide to a Good Night's Sleep
Sustainable Operations in Complex Systems with Production Excellence
Testing in Production—Quality Software, Faster
FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

To tame complexity and its effects, organizations need a structured, multi-pronged, human-focused approach, that: makes operations work sustainable, centers decisions around customer experience, uses continuous testing, and includes chaos engineering and system observability. In this eMag, we cover all of these topics.

InfoQ