O'REILLY®

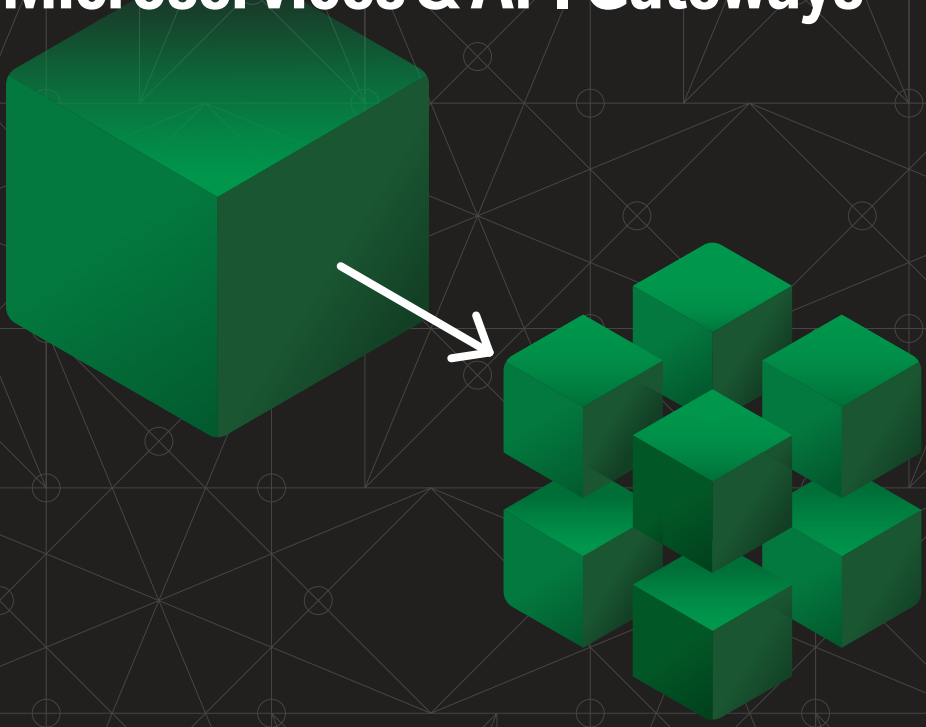
Compliments of NGINX

The Enterprise Path to Service Mesh Architectures

Decoupling at Layer 5

Lee Calcote

The NGINX Application Platform powers Load Balancers, Microservices & API Gateways



Load Balancing

Microservices

Cloud

Security

Web & Mobile Performance

API Gateway

FREE TRIAL

LEARN MORE

Learn more at nginx.com

NGINX

The Enterprise Path to Service Mesh Architectures

Decoupling at Layer 5

Lee Calcote

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

The Enterprise Path to Service Mesh Architectures
by Lee Calcote

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nikki McDonald
Editor: Virginia Wilson
Production Editor: Nan Barber
Copyeditor: Octal Publishing, Inc.

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2018: First Edition

Revision History for the First Edition
2018-08-08: First Release

This work is part of a collaboration between O'Reilly and NGINX. See our *statement of editorial independence*.

978-1-492-04176-4

[LSI]

Table of Contents

Preface

As someone interested in modern software design, you have heard of service mesh architectures primarily in the context of microservices. Service meshes introduce a new layer into modern infrastructures, offering the potential for creating and running robust and scalable applications while exercising granular control over them. Is a service mesh right for you? This report will help answer common questions on service mesh architectures through the lens of a large enterprise. It also addresses how to evaluate your organization's readiness, provides factors to consider when building new applications and converting existing applications to best take advantage of a service mesh, and offers insight on deployment architectures used to get you there.

What You Will Learn

- What is a service mesh and why do I need one?
  - What are the different service meshes, and how do they contrast?
- Where do services meshes layer in with other technologies?
- When and why should I adopt a service mesh?
  - What are popular deployment models and why?
  - What are practical steps to adopt a service mesh in my enterprise?
  - How do I fit a service mesh into my existing infrastructure?

Who This Report Is For

The intended readers are developers, operators, architects, and infrastructure (IT) leaders, who are faced with operational challenges of distributed systems. Technologists need to understand the various capabilities of and paths to service meshes so that they can better face the decision of selecting and investing in an architecture and deployment model to provide visibility, resiliency, traffic, and security control of their distributed application services.

Acknowledgements

Many thanks to Dr. Girish Ranganathan (Dr. G) and the occasional two "t"s Matt Baldwin for their many efforts to ensure the technical correctness of this report.

CHAPTER 1

Service Mesh Fundamentals

Why is operating microservices difficult? What is a service mesh, and why do I need one?

Many emergent technologies build on or reincarnate prior thinking and approaches to computing and networking paradigms. Why is this phenomenon necessary? In the case of service meshes, we'll blame the microservices and containers movement—the cloud-native approach to designing scalable, independently delivered services. Microservices have exploded what were once internal application communications into a mesh of service-to-service remote procedure calls (RPCs) transported over networks. Bearing many benefits, microservices provide democratization of language and technology choice across independent service teams—teams that create new features quickly as they iteratively and continuously deliver software (typically as a service).

Operating Many Services

And, sure, the first few microservices are relatively easy to deliver and operate—at least compared to what difficulties organizations face the day they arrive at many microservices. Whether that "many" is 10 or 100, the onset of a major headache is inevitable. Different medicines are dispensed to alleviate microservices headaches; use of client libraries is one notable example. Language and framework-specific client libraries, whether preexisting or created, are used to address distributed systems challenges in microservices environments. It's in these environments that many teams first con-

sider their path to a service mesh. The sheer volume of services that must be managed on an individual, distributed basis (versus centrally as with monoliths) and the challenges of ensuring reliability, observability, and security of these services cannot be overcome with outmoded paradigms; hence, the need to reincarnate prior thinking and approaches. New tools and techniques must be adopted.

Given the distributed (and often ephemeral) nature of microservices —and how central the network is to their functioning—it behooves us to reflect on the fallacy that networks are reliable, are without latency, have infinite bandwidth, and that communication is guaranteed. When you consider how critical the ability to control and secure service communication is to distributed systems that rely on network calls with each and every transaction, each and every time an application is invoked, you begin to understand that you are under tooled and why running more than a few microservices on a network topology that is in constant flux is so difficult. In the age of microservices, a new layer of tooling for the caretaking of services is needed—a service mesh is needed.

What Is a Service Mesh?

Service meshes provide policy-based networking for microservices describing desired behavior of the network in the face of constantly changing conditions and network topology. At their core, service meshes provide a developer-driven, services-first network; a network that is primarily concerned with alleviating application developers from building network concerns (e.g., resiliency) into their application code; a network that empowers operators with the ability to declaratively define network behavior, node identity, and traffic flow through policy.

Value derived from the layer of tooling that service meshes provide is most evident in the land of microservices. The more services, the more value derived from the mesh. In subsequent chapters, I show how service meshes provide value outside of the use of microservices and containers and help modernize existing services (running on virtual or bare metal servers) as well.

Architecture and Components

Although there are a few variants, service mesh architectures commonly comprise two planes: a *control plane* and *data plane*. The concept of these two planes immediately resonate with network engineers by the analogous way in which physical networks (and their equipment) are designed and managed. Network engineers have long been trained on *divisions of concern* by planes as shown in Figure 1-1.

The physical networking data plane (also known as the *forwarding plane*) contains application traffic generated by hosts, clients, servers, and applications that use the network as transport. Thus, data-plane traffic should never have source or destination IP addresses that belong to any network elements such as routers and switches; rather, they should be sourced from and destined to end devices such as PCs and servers. Routers and switches use hardware chips—application-specific integrated circuits (ASICs)—to optimally forward data-plane traffic as quickly as possible.

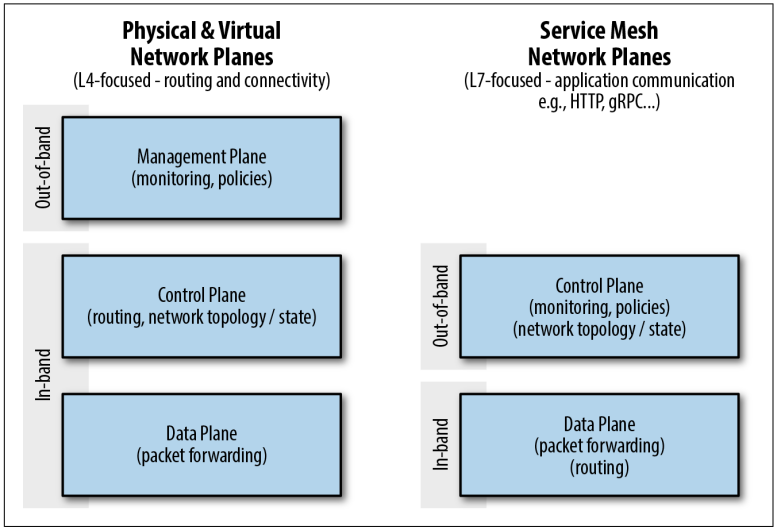


Figure 1-1. Physical networking versus software-defined networking planes

Let's contrast physical networking planes and network topologies with those of service meshes.

Physical network planes

The physical networking control plane operates as the logical entity associated with router processes and functions used to create and maintain necessary intelligence about the state of the network (topology) and a router's interfaces. The control plane includes network protocols, such as routing, signaling, and link-state protocols that are used to build and maintain the operational state of the network and provide IP connectivity between IP hosts.

The physical networking *management plane* is the logical entity that describes the traffic used to access, manage, and monitor all of the network elements. The management plane supports all required provisioning, maintenance, and monitoring functions for the network. Although network traffic in the control plane is handled in-band with all other data-plane traffic, management-plane traffic is capable of being carried via a separate out-of-band (OOB) management network to provide separate reachability in the event that the primary in-band IP path is not available (and create a security boundary).

Physical networking control and data planes are tightly coupled and generally vendor provided as a proprietary integration of hardware and firmware. Software-defined networking (SDN) has done much to insert standards and decouple. We'll see that control and data planes of service meshes are not necessarily tightly coupled.

Physical network topologies

Common physical networking topologies include *star*, *spoke-and-hub*, *tree* (also called *hierarchical*), and *mesh*. As depicted in Figure 1-2, nodes in mesh networks connect directly and nonhierarchically such that each node is connected to an arbitrary number (usually as many as possible or as needed dynamically) of neighbor nodes so that there is at least one path from a given node to any other node to efficiently route data.

When I designed mesh networks as an engineer at Cisco, I did so to create fully interconnected, wireless networks. Wireless is the canonical use case for mesh networks for which the networking medium is readily susceptible to line-of-sight, weather-induced, or other disruption, and, therefore, for which reliability is of paramount concern. Mesh networks generally self-configure, enabling dynamic distribution of workloads. This ability is particularly key to

both mitigate risk of failure (improve resiliency) and to react to continuously changing topologies. It's readily apparent why this network topology is the design of choice for service mesh architectures.

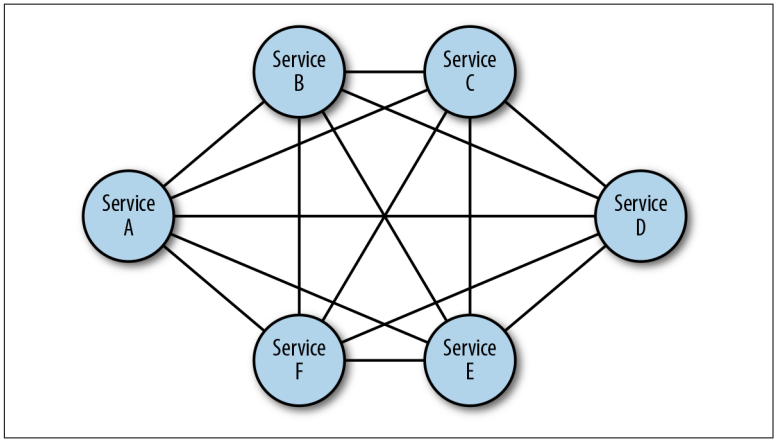


Figure 1-2. Mesh topology—fully connected network nodes

Service mesh network planes

Again, service mesh architectures typically employ data and control planes (see Figure 1-3). Service meshes typically consolidate the analogous physical network control and management planes into the control plane, leaving some observability aspects of the management plane as integration points to external monitoring tools. As in physical networking, service mesh data planes handle the actual inspection, transiting, and routing of network traffic, whereas the control plane sits out-of-band providing a central point of management and backend/underlying infrastructure integration. Depending upon which architecture you use, both planes might or might not be deployed.

A service mesh data plane (otherwise known as the *proxying layer*) intercepts every packet in the request and is responsible for health checking, routing, load balancing, authentication, authorization, and generation of observable signals. Service proxies are transparently inserted, and as applications make service-to-service calls, applications are unaware of the data plane's existence. Data planes are responsible for intracluster communication as well as inbound (ingress) and outbound (egress) cluster network traffic. Whether traffic is entering the mesh (ingressing) or leaving the mesh (egressing), application service traffic is directed first to the service proxy

for handling. In Istio's case, traffic is transparently intercepted using iptables rules and redirected to the service proxy.

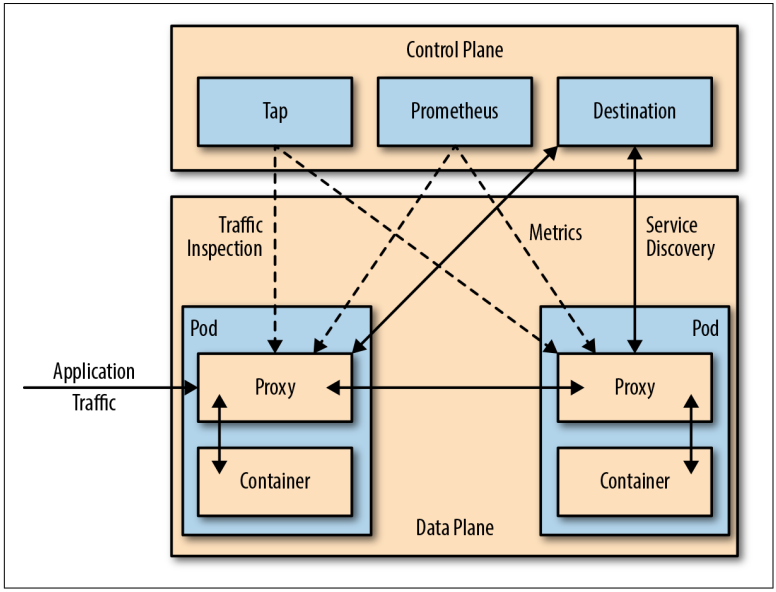


Figure 1-3. An example of service mesh architecture. In Conduit's architecture, control and data planes divide in-band and out-of-band responsibility for service traffic

A service mesh control plane is called for when the number of proxies becomes unwieldy or when a single point of visibility and control is required. Control planes provide policy and configuration for services in the mesh, taking a set of isolated, stateless proxies and turning them into a service mesh. Control planes do not directly touch any network packets in the mesh. They operate out-of-band. Control planes typically have a command-line interface (CLI) and user interface with which to interact, each of which provides access to a centralized API for holistically controlling proxy behavior. You can automate changes to the control plane configuration through its APIs (e.g., by a continuous integration/continuous deployment pipeline), where, in practice, configuration is most often version controlled and updated.

NOTE

Proxies are generally considered stateless, but this is a thought-provoking concept. In the way in which proxies are generally informed by the control plane of the presence of services, mesh topology updates, traffic and authorization policy, and so on, proxies cache the state of the mesh but aren't regarded as the source of truth for the state of the mesh.

Reflecting on Linkerd (pronounced "linker-dee") and Istio as two popular open source service meshes, we find examples of how the data and control planes are packaged and deployed. In terms of packaging, Linkerd contains both its proxying components (`linkerd`) and its control plane (`namerd`) packaged together simply as "Linkerd," and Istio brings a collection of control-plane components (`Mixer`, `Pilot`, and `Citadel`) to pair by default with Envoy (a data plane) packaged together as "Istio." Envoy is often labeled a service mesh, inappropriately so, because it takes packaging with a control plane (we cover a few projects that have done so) to form a service mesh. Popular as it is, Envoy is often found deployed more simply standalone as an API or ingress gateway.

In terms of control-plane deployment, using Kubernetes as the example infrastructure, control planes are typically deployed in a separate "system" namespace. In terms of data-plane deployment, some service meshes, like Conduit, have proxies that are created as part of the project and are not designed to be configured by hand, but are instead designed for their behavior to be entirely driven by the control plane. Although other service meshes, like Istio, choose not to develop their own proxy; instead, they ingest and use independent proxies (separate projects), which, as a result, facilitates choice of proxy and its deployment outside of the mesh (standalone).

Why Do I Need One?

At this point, you might be thinking, "I have a container orchestrator. Why do I need another infrastructure layer?" With microservices and containers mainstreaming, container orchestrators provide much of what the cluster (nodes and containers) need. Necessarily so, the core focus of container orchestrators is scheduling, discovery, and health, focused primarily at an infrastructure level (Layer 4 and below, if you will). Consequently, microservices are left with unmet,

service-level needs. A service mesh is a dedicated infrastructure layer for making service-to-service communication safe, fast, and reliable, often relying on a container orchestrator or integration with another service discovery system for operation. Service meshes often deploy as a separate layer atop container orchestrators but do not require one in that control and data-plane components may be deployed independent of containerized infrastructure. As you'll see in Chapter 3, a node agent (including service proxy) as the data-plane component is often deployed in non-container environments.

As noted, in microservices deployments, the network is directly and critically involved in every transaction, every invocation of business logic, and every request made to the application. Network reliability and latency are at the forefront of concerns for modern, cloud-native applications. A given cloud-native application might be composed of hundreds of microservices, each of which might have many instances and each of those ephemeral instances rescheduled as and when necessary by a container orchestrator.

Understanding the network's criticality, what would you want out of a network that connects your microservices? You want your network to be as intelligent and resilient as possible. You want your network to route traffic away from failures to increase the aggregate reliability of your cluster. You want your network to avoid unwanted overhead like high-latency routes or servers with cold caches. You want your network to ensure that the traffic flowing between services is secure against trivial attack. You want your network to provide insight by highlighting unexpected dependencies and root causes of service communication failure. You want your network to let you impose policies at the granularity of service behaviors, not just at the connection level. And, you don't want to write all this logic into your application.

You want Layer 5 management. You want a services-first network. You want a service mesh.

Value of a Service Mesh

Service meshes provide visibility, resiliency, traffic, and security control of distributed application services. Much value is promised here, particularly to the extent that much is begotten without the need to change your application code (*or much of it*).

Observability

Many organizations are initially attracted to the uniform observability that service meshes provide. No complex system is ever fully healthy. Service-level telemetry illuminates where your system is behaving sickly, illuminating difficult-to-answer questions like why your requests are slow to respond. Identifying when a specific service is down is relatively easy, but identifying where it's slow and why, is another matter.

From the application's vantage point, service meshes largely provide black-box monitoring (observing a system from the outside) of service-to-service communication, leaving white-box monitoring (observing a system from the inside—reporting measurements from inside-out) of an application as the responsibility of the microservice. Proxies that comprise the data plane are well-positioned (transparently, in-band) to generate metrics, logs, and traces, providing uniform and thorough observability throughout the mesh as a whole, as seen in Figure 1-4.

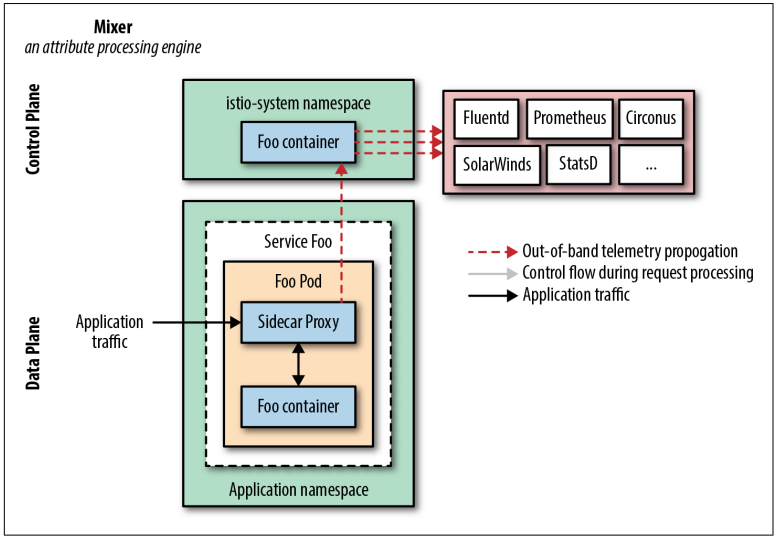


Figure 1-4. Istio's Mixer is capable of collecting multiple telemetric signals and sending those signals to backend monitoring, authentication, and quota systems via adapters

You are probably accustomed to having individual monitoring solutions for distributed tracing, logging, security, access control, and so

on. Service meshes centralize and assist in solving these observability challenges by providing the following:

Logging

Logs are used to baseline visibility for access requests to your entire fleet of services. Figure 1-5 illustrates how telemetry transmitted through service mesh logs include source and destination, request protocol, endpoint (URL), associated response code, and response time and size.

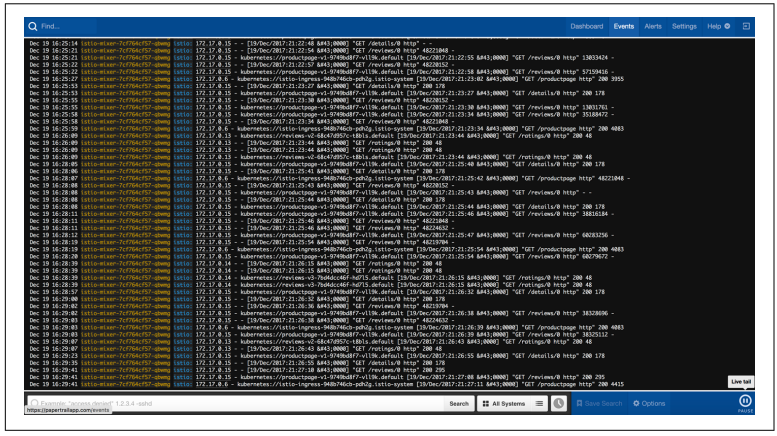


Figure 1-5. Request logs generated by Istio and sent to Papertrail™. (© 2018 SolarWinds Worldwide, LLC. All rights reserved.)

Metrics

Metrics are used to remove dependency and reliance on the development process to instrument code to emit metrics. When metrics are ubiquitous across your cluster, they unlock new insights. Consistent metrics enables automation for things like autoscaling, as an example. Example telemetry emitted by service mesh metrics include global request volume, global success rate, individual service responses by version, source and time, as shown in Figure 1-6.

Tracing

Without tracing, slow services (versus services that simply fail) are most difficult to debug. Imagine manual enumeration of all of your service dependencies being tracked in a spreadsheet. Traces are used to visualize dependencies, request volumes, and failure rates. Imagine manual enumeration of all of your dependencies being tracked in a spreadsheet. With automatically gen-

erated span identifiers, service meshes make integrating tracing functionality *almost* effortless. Individual services in the mesh still need to forward context headers, but that's it. In contrast, many application performance management (APM) solutions require manual instrumentation to get traces out of your services. Later, you'll see that in the sidecar proxy deployment model, sidecars are ideally positioned to trace the flow of requests across services.

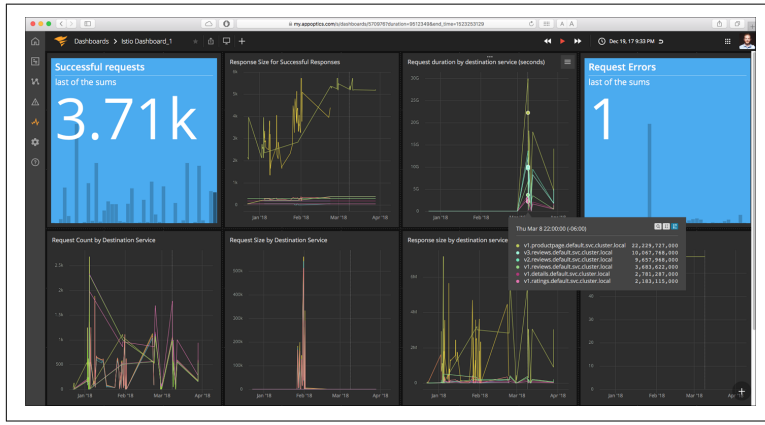


Figure 1-6. Request metrics generated by Istio and sent to AppOptics™ (© 2018 SolarWinds Worldwide, LLC. All rights reserved.)

Traffic control

Service meshes provide granular, declarative control over network traffic to determine where a request is routed to perform canary release, for example. Resiliency features typically include circuit breaking, latency-aware load balancing, eventually consistent service discovery, retries, timeouts, and deadlines.

Timeouts provide cancellation of service requests when a request doesn't return to the client within a predefined time. Timeouts limit the amount of time spent on any individual request, commonly enforced at a point in time after which a response is considered invalid or too long for a client (user) to wait. *Deadlines* are an advanced service mesh feature in that they facilitate the feature-level timeouts (a collection of requests) rather than independent service timeouts, helping to avoid retry storms. Deadlines deduct time left to handle a request at each step, propagating elapsed time with each downstream service call as the request travels through the mesh.

Timeouts and deadlines, illustrated in Figure 1-7, can be considered as enforcers of your Service-Level Objectives (SLOs).

When a service times-out or is unsuccessfully returned, you might choose to retry the request. Simple *retries* bear the risk of making things worse by retrying the same call to a service that is already under water (retry three times = 300% more service load). Retry *budgets* (aka maximum retries), however, provide the benefit of multiple tries but with a limit so as to not overload what is already a load-challenged service. Some service meshes take the elimination of client contention further by introducing jitter and an exponential back-off algorithm in the calculation of timing the next retry attempt.

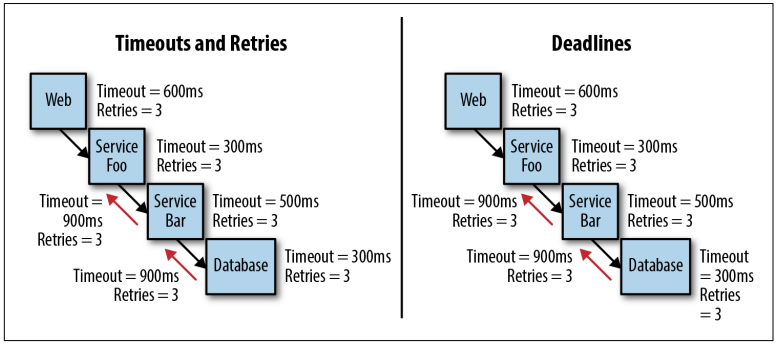


Figure 1-7. Deadlines, not ubiquitously supported by different service meshes, set feature-level timeouts

Instead of retrying and adding more load to the service, you might elect to fail fast and disconnect the service, disallowing calls to it. *Circuit breaking* provides configurable timeouts (failure thresholds) to ensure safe maximums and facilitate graceful failure commonly for slow-responding services. Using a service mesh as a separate layer to implement circuit breaking avoids undue overhead on applications (services) at a time when they are already oversubscribed.

Rate limiting (throttling) is used to ensure stability of a service so that when one client causes a spike in requests, the service continues to run smoothly for other clients. Rate limits are usually measured over a period of time, but you can use different algorithms (fixed or sliding window, sliding log, etc.). Rate limits are typically operationally focused on ensuring that your services aren't oversubscribed.

When a limit is reached, well-implemented services commonly adhere to IETF RFC 6585, sending 429 Too Many Requests as the response code, including headers, such as the following, describing the request limit, number of requests remaining, and amount of time remaining until the request counter is reset:

```
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 1372016266
```

Rate limiting protects your services from overuse by limiting how often a client (most often mapped to a user access token) can call your service(s), and provides operational resiliency (e.g., service A can handle only 500 requests per second).

Subtlety distinguished is *quota management* (or *conditional rate limiting*) that is primarily used for accounting of requests based on business requirements as opposed to limiting rates based on operational concerns. It can be difficult to distinguish between rate limiting and quota management, given that these two features can be implemented by the same service mesh capability but presented differently to users.

The canonical example of a quota management is to configure a policy setting a threshold for the number of client requests allowed to a service over the course of time, like user Lee is subscribed to the Free service plan and allowed only 10 requests per day. Quota policy enforces consumption limits on services by maintaining a distributed counter that tallies incoming requests often using an in-memory datastore like Redis. Conditional rate limits are a powerful service mesh capability when implemented based on a user-defined set of arbitrary attributes.

Conditional Rate Limiting Example: Implementing Class of Service

In this example, let's consider a "temperature-check" service that provides a readout of the current temperature for a given geographic area, updated on one-minute intervals. The service provides two different experiences to clients when interacting with its API: an unentitled (free account) experience, and an entitled (paying account) experience like so:

- If the request on the temperature-check service is unauthenticated, the service limits responses to a given requester (client) to one request every 600 seconds. Any unauthenticated user is restricted to receiving an updated result at 10-minute intervals to spare the temperature-check service's resources and provide paying users with a premium experience.
- Authenticated users (perhaps, those providing a valid authentication token in the request) are those who have active service subscriptions (paying customers) and therefore are entitled to up-to-the-minute updates on the temperate-check service's data (authenticated requests to the temperature-check service are not rate limited).

In this example, through conditional rate limiting, the service mesh is providing a separate class of service to paying and nonpaying clients of the temperature-check service. There are many ways in which class of service can be provided by the service mesh (e.g., authenticated requests are sent to a separate service, "temperature-check-premium").

Generally expressed as rules within a collection of policies, traffic control behavior is defined in the control plane and pushed as configuration to the data plane. The order of operations for rule evaluation is specific to each service mesh, but it is often evaluated from top to bottom.

Security

Most service meshes provide a certificate authority to manage keys and certificates for securing service-to-service communication. Certificates are generated per service and provided unique identity of that service. When sidecar proxies are used (discussed later in Chapter 3, they take on the identity of the service and perform life-cycle management of certificates (generation, distribution, refresh, and revocation) on behalf of the service. In sidecar proxy deployments, you'll typically find that local TCP connections are established between the service and sidecar proxy, whereas mutual Transport Layer Security (mTLS) connections are established between proxies, as demonstrated in Figure 1-8.

Encrypting traffic internal to your application is an important security consideration. No longer are your application's service calls kept inside a single monolith via localhost; they are exposed over the net-

work. Allowing service calls without TLS on the transport is setting yourself up for security problems. When two mesh-enabled services communicate, they have strong cryptographic proof of their peers. After identities are established, they are used in constructing access control policies, determining whether a request should be serviced. Depending on the service mesh used, policy controls configuration of the key management system (e.g., certificate refresh interval) and operational access control used to determine whether a request is accepted. White and blacklists are used to identify approved and unapproved connection requests as well as more granular access control factors like time of day.

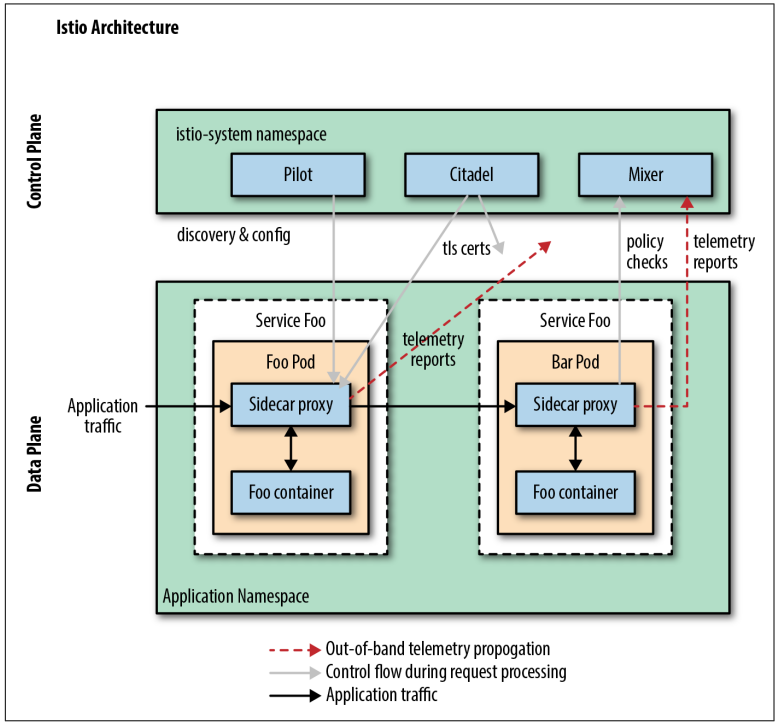


Figure 1-8. An example of service mesh architecture. Secure communication paths in Istio

Delay and fault injection

The notion that your systems will fail must be embraced. Why not preemptively inject failure and verify behavior? Given that proxies sit in line to service traffic, they often support protocol-specific fault injection, allowing configuration of the percentage of requests that

should be subjected to faults or network delay. For instance, generating HTTP 500 errors helps to verify the robustness of your distributed application in terms of how it behaves in response.

Injecting latency into requests without a service mesh can be a tedious task but is probably a more common issue faced during operation of an application. Slow responses that result in an HTTP 503 after a minute of waiting leaves users much more frustrated than a 503 after six seconds. Arguably, the best part of these resilience testing capabilities is that no application code needs to change in order to facilitate these tests. Results of the tests, on the other hand, might well have you changing application code.

Using a service mesh, developers invest much less in writing code to deal with infrastructure concerns—code that might be on a path to being commoditized by service meshes. The separation of service and session-layer concerns from application code manifests in the form of a phenomenon I refer to as a *decoupling at Layer 5*.

Decoupling at Layer 5

Service meshes help you to avoid bloated service code, fat on infrastructure concerns.

Duplicative work is avoided in making services production-ready by way of singularly addressing load balancing, autoscaling, rate limiting, traffic routing, and so on. Teams avoid inconsistency of implementation across different services to the extent that the same set of central control is provided for retries and budgets, failover, deadlines, cancellation, and so forth. Implementations done in silos lead to fragmented, non-uniform policy application and difficult debugging.

Service meshes insert a dedicated infrastructure layer between dev and ops, separating what are common concerns of service communication by providing independent control over them. The service mesh is a networking model that sits at a layer of abstraction above TCP/IP. Without a service mesh, operators are still tied to developers for many concerns as they need new application builds to control network traffic, shaping, affecting access control, and which services talk to downstream services. The decoupling of dev and ops is key to providing autonomous independent iteration.

Decoupling is an important trend in the industry. If you have a significant number of services, you nearly certainly have both of these two roles: developers and operators. Just as microservices is a trend in the industry for allowing teams to independently iterate, so do service meshes allow teams to decouple and iterate faster. Technical reasons for having to coordinate between teams dissolves in many circumstances, like the following:

- Operators don't necessarily need to involve Developers to change how many times a service should retry before timing out.
- Customer Success teams can handle the revocation of client access without involving Operators.
- Product Owners can use quota management to enforce price plan limitations for quantity-based consumption of particular services.
- Developers can redirect their internal stakeholders to a canary with beta functionality without involving Operators.

Microservices decouple functional responsibilities within an application from one another, allowing development teams to independently iterate and move forward. Figure 1-9 shows that in the same fashion, service meshes decouple functional responsibilities of instrumentation and operating services from developers and operators, providing an independent point of control and centralization of responsibility.

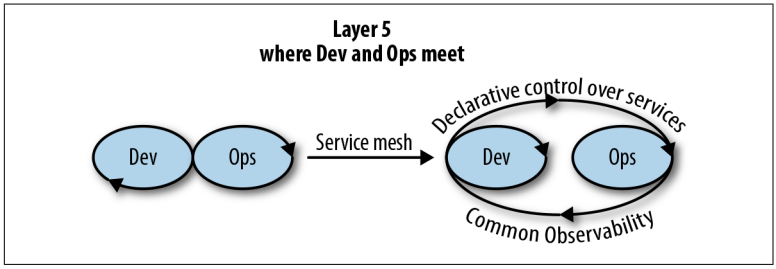


Figure 1-9. Decoupling as a way of increasing velocity

Even though service meshes facilitate a separation of concerns, both developers and operators should understand the details of the mesh. The more everyone understands, the better. Operators can obtain uniform metrics and traces from running applications involving

diverse language frameworks without relying on developers to manually instrument their applications. Developers tend to consider the network as a dumb transport layer that really doesn't help with service-level concerns. We need a network that operates at the same level as the services we build and deploy.

Essentially, you can think of a service mesh as surfacing the session layer of the OSI model as a separately addressable, first-class citizen in your modern architecture.

Conclusion

The data plane carries the actual application request traffic between service instances. The control plane configures the data plane, provides a point of aggregation for telemetry, and also provides APIs for modifying the mesh's behavior.

Decoupling of dev and ops avoids diffusion of the responsibility of service management, centralizing control over these concerns into a new infrastructure layer: Layer 5.

Service meshes makes it possible for services to regain a consistent, secure way to establish identity within a datacenter and, furthermore, do so based on strong cryptographic primitives rather than deployment topology.

With each deployment of a service mesh, developers are relieved of their infrastructure concerns and can refocus on their primary task (of creating business logic). More seasoned software engineers might have difficulty in breaking the habit and trusting that the service mesh will provide, or even displacing the psychological dependency on their handy (but less capable) client library.

Many organizations find themselves in the situation of having incorporated too many infrastructure concerns into application code. Service meshes are a necessary building block when composing production-grade microservices. The power of easily deployable service meshes will allow for many smaller organizations to enjoy features previously available only to large enterprises.

CHAPTER 2

Contrasting Technologies

How do service meshes contrast to one another? How do service meshes contrast to other related technologies?

You might already have a healthy understanding of API gateways, ingress controllers, container orchestrators, client libraries, and so on. How are these technologies related to, overlapping with, or deployed alongside service meshes? Where do service meshes fit in?

Different Service Meshes (and Gateways)

Let's begin by characterizing different service meshes. Some service meshes support a variety of underlying platforms, whereas some focus solely on layering on top of container orchestrators. All support integration with service discovery systems. The subsections that follow provide a brief survey of offerings within the current technology landscape.

NOTE

This list is neither exhaustive nor intended to be a detailed comparative. See *https://layer5.io/landscape* for community-maintained contrasting of service meshes and related technologies.

Linkerd

Hosted by the Cloud Native Computing Foundation (CNCF) and built on top of Twitter Finagle. Linkerd (pronounced "linker-dee")

includes both a proxying data plane and the Namerd ("namer-dee") control plane all in one package.

- Open source. Written primarily in Scala.
- Data plane can be deployed in a node proxy model or in a proxy sidecar. Proven scale, having served more than one trillion service requests.
- Supports services running within container orchestrators and as standalone virtual or physical machines.
- Service discovery abstractions to unite multiple systems.

Conduit

Conduit is a Kubernetes-native (only) service mesh announced as a project in December 2017. In contrast to Istio and in learning from Linkerd, Conduit's design principles revolve around a minimalist architecture and zero configuration philosophy, optimizing for streamlined setup.

- Open Source. From Buoyant. Written in Rust and Go.
- Data plane implemented in Rust. Purports sub-1-ms p99 traffic latency.
- Support for gRPC, HTTP/2, and HTTP/1.x requests plus all TCP traffic.

NOTE

Conduit is merging with Linkerd. Conduit 0.5.0 will be the last major release of the project under this name. Conduit is graduating (merging) into the Linkerd project to become the basis of Linkerd 2.0.

Istio

Announced as a project in May 2017, Istio is considered to be a "second explosion after Kubernetes" given its architecture and surface area of functional aspiration.

- Supports services running within container orchestrators and as standalone virtual or physical machines.

- Supports automatic injection of sidecars using Kubernetes Admission controller.
- nginMesh. Launched in September 2017, the nginMesh project deploys NGINX as a sidecar proxy in Istio.
- AspenMesh. A commercial offering built on top of Istio. Closed source.

Envoy

A modern proxy hosted by the CNCF. Many projects have sprung up to use Envoy, including Istio.

- Rotor. A control plane that provides service discovery (EC2, ECS, Kubernetes, DC/OS, Consul, and JSON/YAML files) and a log sink.
- Houston. A control plane for Envoy that provides management for use cases like application routing and releasing. Closed source. From Turbine Labs.
  - Houston provides service discovery integrations with Kubernetes, AWS, ECS, DC/OS, and Consul.
- Ambassador. An API gateway for microservices functioning as a Kubernetes Ingress Controller. Open Source. From Datawire. Primarily written in Python.
- Contour. A reverse proxy and load balancer deployed as a Kubernetes Ingress Controller.
  - Open Source. From Heptio. Written in Go.
- Consul Connect. Connect is a major new feature in Consul that provides secure service-to-service communication with automatic TLS encryption and identity-based authorization.
  - Open source. From HashiCorp. Primarily written in Go.
- Mesher. Layer 7 (L7) proxy that runs as a sidecar deployable on Huawei Cloud Service Engine.
  - Open source. Written primarily in Go. From Huawei.

Following are a couple of early service mesh–like projects, forming control planes around existing load-balancers:

SmartStack
: Comprising two components: Nerve for health-checking and Synapse for service discovery. Open source. From AirBnB. Written in Ruby.

Nelson
: Takes advantage of integrations with Envoy, Prometheus, Vault, and Nomad to provide Git-centric, developer-driven deployments with automated build-and-release workflow. Open source. From Verizon Labs. Written in Scala.

Service Mesh Linguistics

As the *lingua franca* of the cloud-native ecosystem, Go is certainly prevalent and you might expect most service mesh projects to be written in Go. By the nature of their task, data planes must be highly efficient in the interception, introspection, and rewriting of network traffic. Although Go certainly provides high performance, there's no denying that native code (machine code) is the Holy Grail of performance. As a data-plane component, Envoy is written in C++11 because it provides excellent performance (some say it provides a great developer experience). As an emerging language (and something of a C++ competitor), Rust has found its use within service meshes. Because of its properties around efficiency (outperforming Go) and memory safety (when written to be so) without garbage collection, Rust has been used for Conduit's data plane component and for nginMesh's Mixer module (see "Customizable Sidecars" on page 45).

Container Orchestrators

Why is my container orchestrator not enough? What if I'm not using containers?

What do you need to continuously deliver and operate microservices? Leaving Continuous Integration (CI) and their deployment pipelines aside for the moment, you need much of what the container orchestrator provides at an infrastructure level and what it doesn't at a services level. Table 2-1 takes a look at these capabilities.

Table 2-1. Container orchestration capabilities and focus versus service-level needs

Core capabilities	Missing service-level needs
• Cluster management* — Host discovery — Host health monitoring • Scheduling* • Orchestrator updates and host maintenance • Service discovery • Networking and load balancing • Stateful services • Multitenant, multiregion Additional key capabilities • Simple application health and performance monitoring • Application deployments • Application secrets	• Circuit breaking • L7 granular traffic routing — HTTP redirects — CORS handling • Chaos testing • Canary deploys • Timeouts, retries, budgets, deadlines • Per-request routing • Backpressure • Transport security (encryption) • Identity and access control • Quota management • Protocol translation (REST, GRPC) • Policy

* Must have this to be considered a container orchestrator

Service meshes are a dedicated layer for managing service-to-service communication, whereas container orchestrators have necessarily had their start and focus on automating containerized infrastructure, overcoming ephemeral infrastructure and distributed systems problems. Applications are why we run infrastructure, though. Applications have been and are still the North Star of our focus. There are enough service and application-level concerns that additional platforms/management layers are needed.

Container orchestrators like Kubernetes have different mechanisms for routing traffic into the cluster. Ingress Controllers in Kubernetes expose the services to networks external to the cluster. Ingresses can terminate Secure Sockets Layer (SSL) connections, execute rewrite rules, and support WebSockets and sometimes TCP/UDP, but they don't address the rest of service-level needs.

API gateways address some of these needs and are commonly deployed on a container orchestrator as an edge proxy. Edge proxies provide services with Layer 4 (L4) to L7 management, while using

the container orchestrator for reliability, availability, and scalability of container infrastructure. Let's consider API gateways in greater detail.

API Gateways

How do API gateways interplay with service meshes?

This is a very common question, the nuanced answer to which puzzles many, particularly given that within the category of API gateways lies a sub-spectrum. API gateways come in a few forms:

- Traditional (e.g., Kong)
- Cloud-hosted (e.g., Azure Load Balancer)
- L7 proxy used as an API gateway and microservices API gateways (e.g., Traefik, NGINX, HAProxy, or Envoy)

L7 proxies used as API gateways generally can be represented by a collection of microservices-oriented, open source projects, which have taken the approach of wrapping existing L7 proxies with additional features needed for an API gateway.

NGINX

As a stable, efficient, ubiquitous L7 proxy, NGINX is commonly found at the core of API gateways. It may be used on its own or wrapped with additional features to facilitate container orchestrator native integration or additional self-service functionality for developers. Examples of this include:

- Ambassador uses Envoy
- APIUmbrella uses NGINX
- Kong uses NGINX
- OpenResty uses NGINX

Additional differences between traditional API gateways and microservices API gateways revolve around which team is using the gateway: operators or developers (service owners). Operators tend to focus on measuring API calls per consumer to meter and disallow API calls when a consumer exceeds its quota. On the other hand,

developers tend to measure L7 latency, throughput, and resilience, limiting API calls when service is not responding.

With respect to service meshes, one of the more notable lines of delineation is that API gateways, in general, are designed for accepting traffic from outside of your organization/network and distributing it internally. API gateways expose your services as managed APIs, focused on transiting north/south traffic (in and out of the service mesh). They aren't as well suited for traffic management within the service mesh (east/west) necessarily, because they require traffic to travel through a central proxy and add a network hop. Service meshes are designed foremost to manage east/west traffic internal to the service mesh.

Given their complementary nature, API gateways and service meshes are often found deployed in combination. API gateways marry-up against other components of the API management ecosystem, such as API marketplace and API publishing portal.

API Management

API management solutions handle this as well as analytics, business data, adjunct provider services, and implementation of versioning control. Many of the API management vendors have moved API management systems to a single point of architecture, designing their API gateways to be implemented at the edge.

At the API management level, you can either use built-in interservice communication capabilities of the API gateway or an API gateway can call downstream services via service mesh by offloading application network functions to the service mesh.

API management capabilities distinct from that of a service mesh are oriented toward developer engagement in the following ways:

- Developers use a portal to self-serve:
  - Discover what APIs are available
  - API documentation and discovery
  - Developer notification (when the version of the API changes)
  - API testing/exercise code

- API analytics:
  - — API/KPI
  - — Generating reports; capturing and sharing metrics
  - — Usage trends/adoption trending
- API life-cycle management:
  - — Secure (allocate API keys to it)
  - — Promote or demote API
- Monetization:
  - — Tracking payment plans
  - — Enforcing quotas

Today, there's overlap and underlap between service mesh capabilities and API gateways and API management systems. As service meshes gain new capabilities, use cases overlap more and more.

Client Libraries

Client libraries (sometimes referred to as microservices frameworks) became very popular as microservices took foothold in modern application design as means to avoid rewriting the same logic in every service. Example frameworks include the following:

Twitter Finagle
: An open source remote procedure call (RPC) library built on Netty for engineers that want a strongly-typed language and be on the Java Virtual Machine (JVM). Finagle is written in Scala.

Netflix Hystrix
: An open source latency and fault tolerance library designed to isolate points of access to remote systems, services, and third-party libraries; stop cascading failure; and enable resilience. Hystrix is written in Java.

Netflix Ribbon
: An open source Inter-Process Communication (RPCs) library with built-in software load balancers. Ribbon is written in Java.

The problem with microservices frameworks is that there's too much infrastructure code in services, duplication of code, and inconsistency in what frameworks provide and how they behave.

Getting teams to update their frameworks can be an arduous process. When these distributed systems concerns are embedded into your service code, you need to chase your engineers to update and correct their libraries, of which there might be a few, used to varying degrees. Getting a consistent and recent version deployed can take some time. Enforcing consistency is challenging. These frameworks couple your services with the infrastructure, as seen in Figure 2-1.

When infrastructure code is written into the application, different services teams must get together to negotiate things like timeouts and retries. A service mesh control plane removes this. Service meshes are deployed as infrastructure that reside outside of your applications.

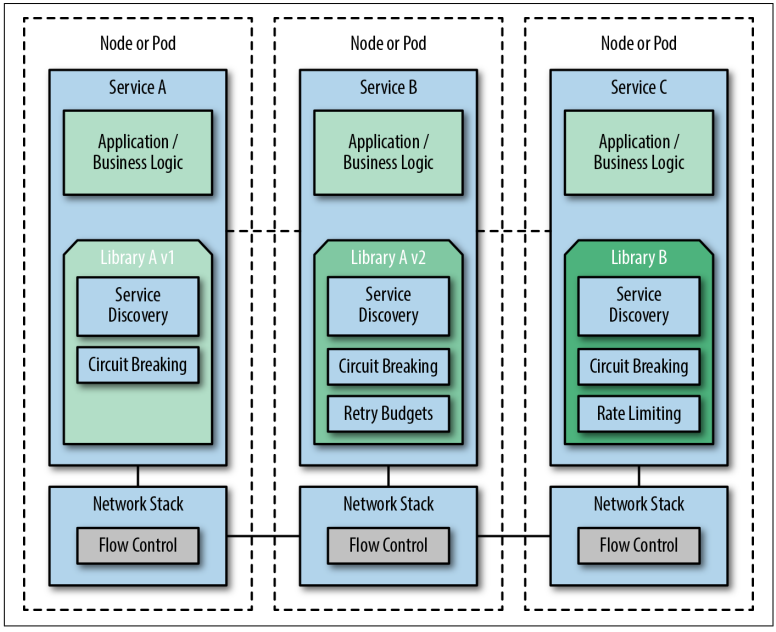


Figure 2-1. Services architecture using client libraries coupled with application logic

Conclusion

Container orchestrators have so many distributed systems challenges to address within lower-layer infrastructure that they've yet to holistically address services and application-level needs.

API gateways are the technology having the most overlap in functionality, but they are deployed at the edge (often once), not on every node or within every pod. It's primarily in their deployment model that API gateways and service meshes are visually viewed as complementary.

Microservices frameworks come with their set of challenges. Service meshes move these concerns into the service proxy and decouple them from the application code.

CHAPTER 3

Adoption and Evolutionary Architectures

What are practical steps to adopt a service mesh in my enterprise?

As organizations adopt service mesh architectures, they often do so in a piecemeal fashion, starting at the intersection of the most valuable (to them) feature and the lowest risk deployment.

Piecemeal Adoption

Desperate to gain an understanding of what's going on in their distributed infrastructure, many organizations seek to benefit from auto-instrumented observability first, taking baby steps in their path to a full service mesh after initial success and operational comfort have been achieved. Financial organizations might seek improved security with strong identity (per service certificates) and strong encryption (mTLS) between each service. Others begin with an ingress proxy as their entry to a service mesh deployment.

Consider an organization that has a thousand existing services running on virtual machines (VMs) external to the service mesh that have little to no service-to-service traffic. Nearly all of the traffic flows from the client to the service and back to the client. This organization can deploy a service mesh ingress (e.g., Istio Gateway) and begin gaining granular traffic control (e.g., path rewrites) and detailed service monitoring without immediately deploying a thousand sidecars.

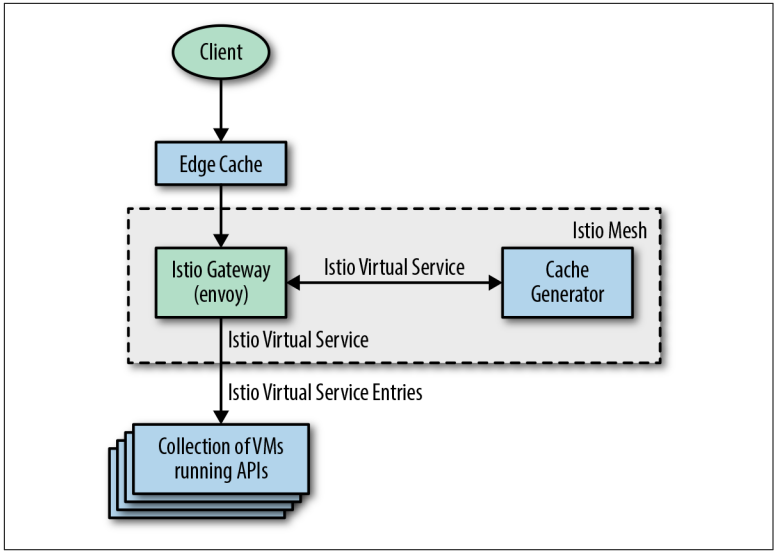


Figure 3-1. Simple service mesh deployment primarily using ingress traffic control.

Not everyone needs a full service mesh or needs to start with one.

Practical Steps to Adoption

Here are two common paths:

- Wholesale adoption of a service mesh, commonly while designing a new application (a greenfield project).
- Piecemeal adoption of some components and capabilities of a service mesh, but not others, commonly while working with an existing application (a brownfield project).

Let's walk through the various forms in which the second path takes shape, because it's the path that most will face (that is, those who have existing services) and is the approach most organizations take. Incremental steps are taken. When teams are comfortable with their understanding of the deployment, operational expertise gained, and value derived, often another step is taken toward a full mesh. Not all teams choose to take another step given that not all aspects of a full service mesh are valuable to teams based on their focus or current pain points.

Engineering maturity and skill set factors into the decision of which applications should be built from the ground up or converted with a new service mesh architecture. A palatable suggestion is that you don't have to use all of the features, just those that you need. Perhaps, the best approach is to mitigate risk, baby-step it, and show incremental victories and consider that some service meshes provide a path to partial adoption. Other service meshes are easily deployed and digestible in one motion. Although, even when this is the case, you might find that you enable only a portion of its capabilities.

Observability is why most organizations deploy a service mesh. Outside of metrics, logs, and traces, typically you get a *service dependency graph*. These graphs visually identify how much traffic is coming from one service and going to the next. Without a visual topology or service graph, you're likely to feel as if you're running blind.

Alternatively, it could be your current load balancer that is running blind. If you're running gRPC services, for example, and a load balancer that is ignorant of gRPC, treating this traffic the same as any other TCP traffic, you'll find most service mesh proxies very helpful. Modern service proxies will support HTTP/2 and, as such, might provide a gRPC bridge from HTTP/1.1 to HTTP/2.

Security

Generally, people get to security last. When you finally do, you might not want strong authentication and encryption. Although best practice is to secure everything with strongly authenticated and authorized services, most organizations don't implement this, which means that most microservices deployments have a soft center. Many of us are content to secure the edge of our network but would still like the observability and control from our service mesh.

Why don't we want to take advantage of service mesh's managed certificate authority? It's another thing to operate. Encryption takes resources (CPU cycles) and injects a couple of microseconds of latency. To this end, and to help with adoption, some service meshes prominently present two installation scripts: one with a certificate authority (CA) embedded and one without. Maybe you consider the gooey center of your mesh to be secure because there is little to no ingress/egress cluster traffic and access is provided only via VPN

into the cluster. Depending on workload, wallet, and sensitivity to latency, you might find that you don't want the overhead of running encryption between all of your services.

Maybe you are deploying monoliths, not microservices (don't need canary deploys), and are simply looking for authorization checks only. You already have API management and don't need any more monitoring integrations. Maybe you use IP addresses (subnets) for security—for network security zones. A service mesh can help you get rid of network partitions and firewalling on Layer 3 (L3) boundaries, using identities and encryption provided by the service mesh combined with authorization checks enforced by policy you define. Through policy enforcing authorizations checks across your monoliths, you can flatten your internal network, making services broadly reachable, granularly controlling which requests are authorized.

Retrofitting a Deployment

Recognize that although some greenfield projects have the luxury of incorporating a service mesh from the start, most organizations will have existing services (monoliths or otherwise) that they'll need to onboard to the mesh. Rather than a container, these services could be running in VMs or bare-metal hosts. Service meshes help with modernization, allowing organizations to renovate their services inventory via the following:

- Not having to rewrite their applications
- Adapting microservices and existing services using the same mesh architecture
- Facilitating adoption of new languages
- Facilitating moving to the cloud

Service meshes ease the insertion of facade services as a way of breaking down monoliths for those organizations that adopt a strangler pattern of building services around the legacy monolith to expose a more developer-friendly set of APIs.

Organizations are able to get observability (e.g., metrics, logs, and traces) support as well as dependency or service graphs for every one of their services (micro or not) as they adopt a service mesh. With respect to tracing, the only change required within the service is to forward certain HTTP headers. Service meshes are useful for

retrofitting uniform and ubiquitous observability tracing into existing infrastructures with the least amount of code change.

Evolutionary Architectures

Different phases of adoption provide multiple paths to service mesh architectures.

Client Libraries

Some consider libraries to be the first service meshes. Figure 3-2 demonstrates how the use of a library requires that your architecture has application code either extending or using primitives of the chosen library(ies). Additionally, your architecture must consider for use language-specific frameworks and/or application servers to run them.

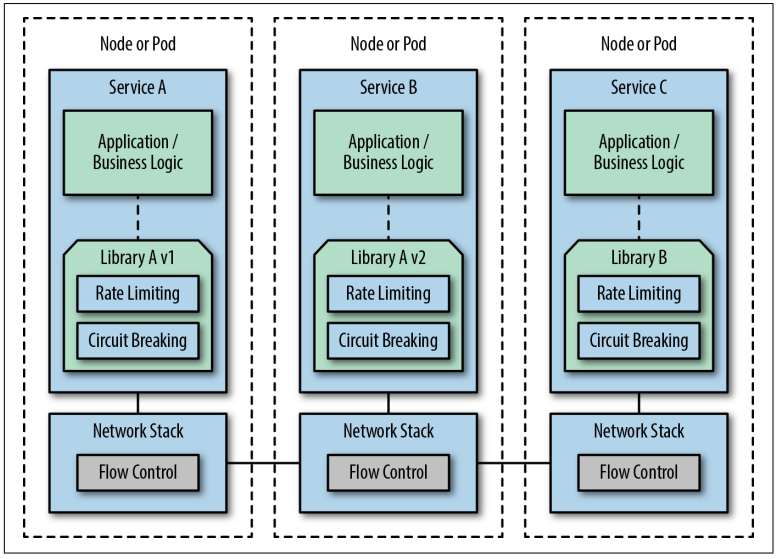


Figure 3-2. Services architecture using client libraries coupled with application logic.

Advantages:

- Resources are locally accounted for each and every service
- Self-service adoption for developers

Disadvantages:

- Strong coupling is a significant drawback
- Non-uniform; upgrades are challenging in large environments

Figure 3-3 illustrates how service meshes further the promise that organizations implementing microservices might finally realize the dream of using the best frameworks and language for their individual jobs without worrying about the availability of libraries and patterns for every single platform.

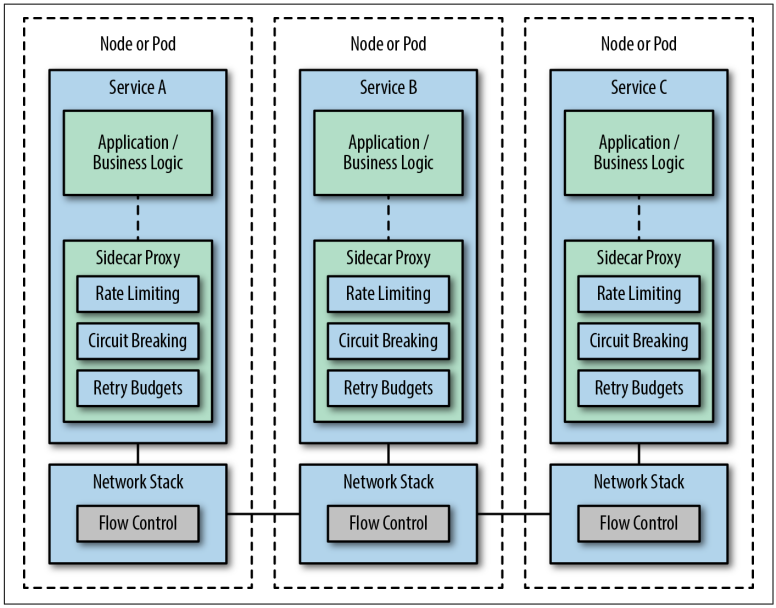


Figure 3-3. Services architecture using sidecar proxies decoupled from application logic.

NOTE

While testing various service mesh deployments, installing and uninstalling the mesh, do not uninstall by deleting the control plane first. Using Istio as an example, if you were to delete the `istio-system` namespace without applying manifests to uninstall the mesh, this could cause you a huge amount of grief because you might be left with a nonfunctional Kubernetes cluster where `kubectl` times-out when communicating with the Kubernetes API server. When Istio isn't cleaned up properly, especially when automatic sidecar injection is enabled in your Kubernetes cluster, proxies are left residual in the cluster and in an unmanaged state.

Ingress or Edge Proxy

Start with load-balancers (or gateways) and get scalability and availability. Strangle your monolith with a facade until you've slowly suffocated it entirely by incrementally routing all service traffic over to microservices that displace the monolith's functionality.

By starting with reverse proxying at the edge (see Figure 3-4), applications avoid the operational overhead of exposing each service via an independent endpoint and the tight coupling of internal business service interfaces. Starting an implementation of modern proxying technology at the edge provides business value in the form of improved observability, load-balancing, and dynamic routing. After an engineering team has gained operational expertise with operating a proxy technology at ingress, the benefits can be rolled inward toward ultimately creating an internal service mesh.

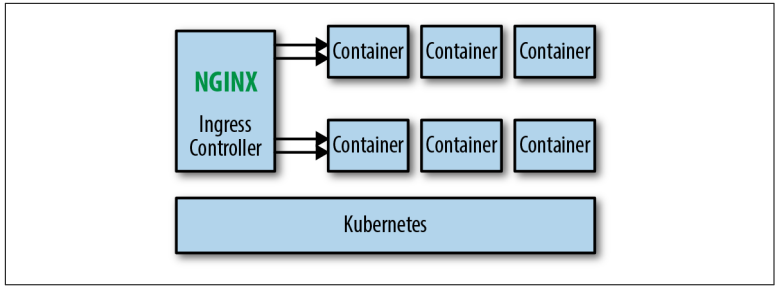


Figure 3-4. NGINX proxy as an ingress controller in Kubernetes.

Service meshes are also used to enforce policy about what egress traffic is leaving your cluster. Typically, this is accomplished in one of a couple ways:

- Registering the external services with your service mesh (so that they can traffic-match traffic against the external destination) and configure traffic control rules to both allow and govern external service calls (provide timeouts on external services, for example).
- Call external services directly without registering them with your service mesh, but configure your mesh to allow traffic destined for external services (maybe for a specific IP range) to bypass the service proxies.

Advantages:

- Works with existing services that can be broken down over time.

Disadvantages:

- Is missing the benefits of service-to-service visibility and control.

Router Mesh

Depicted in Figure 3-5, a router mesh performs service discovery and provides load balancing for service-to-service communication. All service-to-service communication flows through the router mesh, which provides circuit breaking through active health checks (measuring the response time for a service and when latency/timeout threshold is crossed, the circuit is broken) and retries.

Advantages:

- Good starting point for building a brand-new microservices architecture or for migrating from a monolith.

Disadvantages:

- When the number of services increase, it becomes difficult to manage.

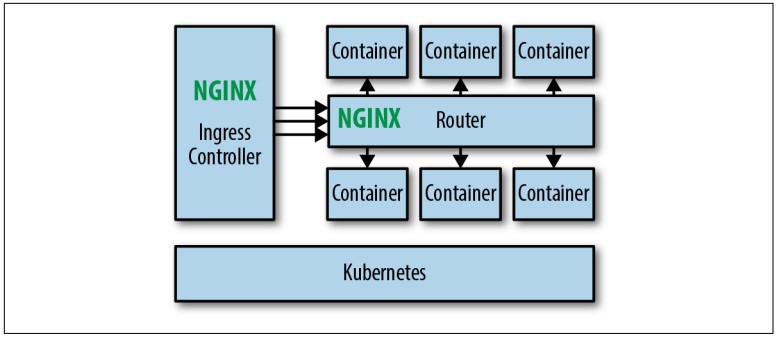


Figure 3-5. NGINX as an example router proxy deployed in Kubernetes.

Proxy per Node

Replacing the router mesh with per-host service proxies brings greater granularity of control to your services' deployment. Using Linkerd (1.x) as an example (Figure 3-6), in the per-host deployment model, one Linkerd instance is deployed per host (whether physical or virtual), and all application service instances on that host route traffic through this instance. It's not particularly well suited to be deployed as a sidecar given its memory resource needs.

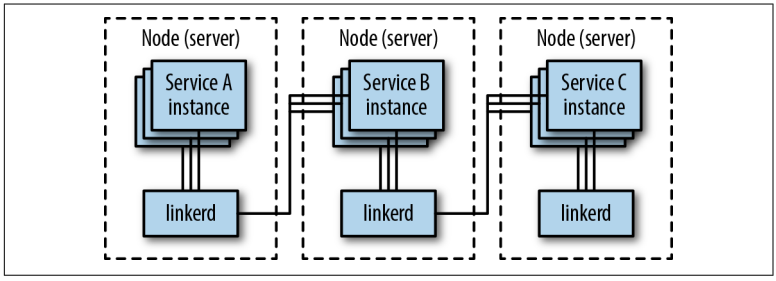


Figure 3-6. Proxy per node deployed using Linkerd (1.x).

Advantages:

- Less overhead (especially memory) for things that could be shared across a node.
- Easier to scale distribution of configuration information than it is with sidecar proxies (if you're not using a control plane).

- This model is useful for deployments that are primarily physical or virtual server based. Good for large monolithic applications.

Disadvantages:

- Coarse support for encryption of service-to-service communication, instead host-to-host encryption and authentication policies.
- Blast radius of a proxy failure includes all applications on the node, which is essentially equivalent to losing the node itself.
- Not a transparent entity, services must be aware of its existence.

Sidecar Proxies/Fabric Model

Although this is not a common deployment model, some organizations arrive here and quickly move onto deploying a control plane—a service mesh. This model, shown in Figure 3-7, is worth highlighting in so much as *sidecarring* is a useful pattern in general.

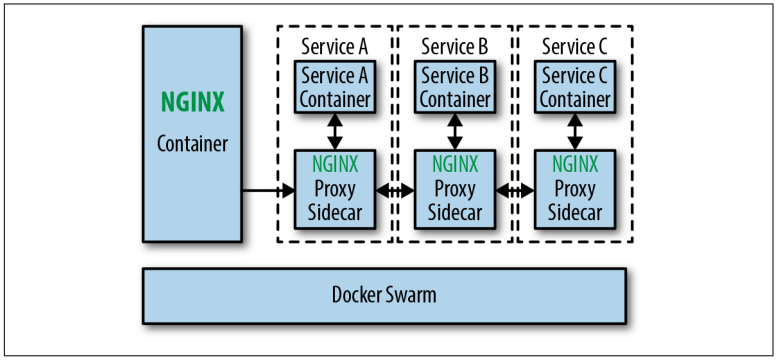


Figure 3-7. Proxy sidecars (fabric model) deployed in Docker Swarm.

The pattern and usefulness of sidecarring isn't constrained to service proxies; it is a generally applicable model of deploying components of an application into a separate container to provide isolation and encapsulation—to separate concerns. For example, you might deploy a logging sidecar alongside the application container to locally collect application logs and forward them to a centralized syslog receiver.

Nearly all proxies support (and microservice frameworks, for that matter) hot reloading of their configuration and hot upgrade of the

proxy themselves (their executable/processes). For some proxies, however, their configuration is not able to be updated on-the-fly without dropping active connections; instead, they need their process to be restarted in order to load new configurations. Given how frequently containers might be rescheduled, this behavior is suboptimal. Container orchestrators offer assistance for proxies that don't support hot reloading. These reloading and upgrading of these proxies can be facilitated through traffic shifting techniques rolling updates provide as traffic is drained and shifted from old containers to new containers.

NOTE

NGINX supports *dynamic reloads* and *hot reloads*. Upstreams (a group of servers or services that can listen on different ports) are dynamically reloaded without loss of traffic. Hence, new server instances that are attached or detached from a route can be handled dynamically. This is the most common case in Kubernetes deployments. Adding or removing new route locations requires a hot reload that keeps the existing workers around for as long as there is traffic passing through them. Frequent reloads of such configurations can exhaust the system memory in some extreme cases. Although there is an option that can accelerate the aging of workers, doing so will affect traffic.

As your number of sidecar proxies grow, so does the work of managing each independently. The next deployment model, sidecar, is a natural next step that brings a great deal more functionality and operational control to bear.

Advantages:

- Granular encryption of service-to-service communication.
- Can be gradually added to an existing cluster without central coordination.

Disadvantages:

- Lack of central coordination. Difficult to scale operationally.

Sidecar Proxies/Control Plane

Most service mesh projects and their deployment efforts promote and support this deployment model foremost. In this model, you provision a control plane (and service mesh) and get the logs and traces out of the service proxies. A powerful aspect of a full service mesh is that it moves away from thinking of proxies as isolated components and acknowledges the network they form as something valuable unto itself. In essence, the control plane is what takes service proxies and forms them into a service mesh. When you're using the control plane, you have a service mesh, as illustrated in Figure 3-8.

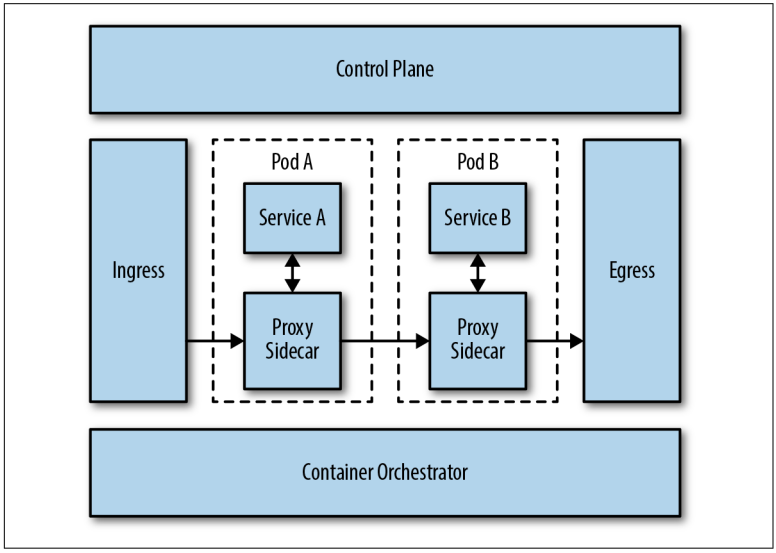


Figure 3-8. Service mesh

Service mesh implementations have evolved allowing deployment models to evolve in concert. For example, Linkerd, created by former Twitter engineers, built on top of Finagle and Netty to create Linkerd.

A number of service meshes that employ the sidecar pattern (shown in Figure 3-7) facilitate the automatic injection of sidecar proxies not only alongside their application container at runtime, but into existing container/deployment manifests, saving time on reworking manifests and facilitating retrofitting of existing containerized service deployments.

A convenient model for containerized service deployments, sidecars are commonly auto-injected or done so via command-line interface (CLI) utility. Using Kubernetes as an example, you can automatically add sidecars to appropriate Kubernetes pods using a mutating WebHook admission controller (in this case, code that intercepts and modifies requests to deploy a service, inserting the service proxy prior to deployment).

Example: Injecting a Conduit Service Proxy as a Sidecar

To onboard a service to the Conduit service mesh, the pods for that service must be redeployed to include a data-plane proxy in each pod. The `conduit inject` command accomplishes this as well as the configuration work necessary to transparently funnel traffic from each instance through the proxy.

Conduit:

```
$ conduit inject deployment.yml | kubectl apply -f -
```

Istio:

```
$ kubectl apply -f <(istioctl kube-inject -f
samples/sleep/sleep.yaml)
```

The `istioctl kube-inject` operation is not idempotent and should not be repeated on the output from a previous `kube-inject`. For upgrade purposes, if you're using manual injection, I recommend that you keep the original non-injected .yaml file so that the data-plane sidecars can be updated.

Advantages:

- App-to-sidecar communication is easier to secure than app-to-node proxy
- Resources consumed for a service is attributed to that service
- Blast radius of a proxy failure is limited to the sidecarred app

Disadvantages:

- Sidecar footprint—per service overhead of running a service proxy sidecar

Multicluster Deployments

Istio is an example of a service mesh that supports (currently with caveats) deployment of its service mesh across Kubernetes clusters. Multicluster deployments facilitate connection from proxies running in multiple clusters to one Istio control plane. Proxies (Envoys) can then communicate with the single Istio control plane and form a mesh network across multiple Kubernetes clusters.

Expanding the Mesh

Some service meshes support onboarding external services. External services running on infrastructure unmanaged by the mesh, like those running on separate VMs or bare-metal servers—onto the mesh. Most service meshes are able to talk to multiple service discovery backends, facilitating the intermingling of meshed and external services.

Example: Using istioctl register to Create a Service Entry

Service entries enable adding additional entries into Istio's internal service registry so that autodiscovered services in the mesh can access/route to these manually specified services. A service entry describes the properties of a service (DNS name, VIPs, ports, protocols, endpoints). These services could be external to the mesh (e.g., web APIs) or mesh-internal services that are not part of the platform's service registry (e.g., a set of VMs talking to services in Kubernetes).

Service entries are dynamically updatable; teams can change their endpoints at-will. Mesh-external entries represent services external to the mesh. Rules to redirect and forward traffic, to define retry, timeout, and fault injection policies are all supported for external destinations.

Some caveats apply:

- Weighted (version-based) routing is not possible, however, because there is no notion of multiple versions of an external service.

- mTLS authentication is disabled and policy enforcement is performed on the client-side, instead of on the usual server-side of an internal service request.

Conclusion

In many respects, deployment of a control plane is what defines a service mesh.

Service meshes support onboarding existing (noncontainerized) services onto the mesh. They support mesh deployment across multiple clusters. Both of these are areas of improvement for the service mesh projects and products that are available today.

As technology evolves capabilities are commoditized and pushed down the stack. Data-plane components will become mostly commoditized. Standards like TCP/IP incorporated solutions to flow control and many other problems into the network stack itself. This means that that piece of code still exists, but it has been extracted from your application to the underlying networking layer provided by your operating system.

It's commonplace to find deployments with north/south load balancers deployed external to the cluster in addition to the ingress/egress proxies within the service mesh.

CHAPTER 4

Customization and Integration

How do I fit a service mesh into my existing infrastructure, operational practices and observability tooling?

Some service meshes such as Conduit (soon to be Linkerd) are not designed to be customized; rather, they aim to focus on out-of-the-box functionality and ease of deployment. Istio is an example of a service mesh designed with customizability in mind. Its extensibility comes in two primary forms: swappable sidecar proxies and telemetry/authorization adapters. Consul's Connect is intended to be displaced as needed, as well.

Customizable Sidecars

Within Istio, though Envoy is the default service proxy sidecar, you can choose another service proxy for your sidecar. Although there are multiple service proxies in the ecosystem, outside of Envoy, only two have currently demonstrated integration with Istio: Linkerd and NGINX. Conduit is not currently designed as a general-purpose proxy; instead, it is focused on being lightweight, placing extensibility as a secondary concern by offering extension via gRPC plug-in. Consul uses Connect as the default proxy embedded into the same binary. Operating at Layer 4, Connect isn't meant to compete on features or performance with dedicated proxy solutions, but enables third-party proxies to integrate to provide the data plane with Consul operating as the control plane. As a control plane, Consul integrates with many data-plane solutions including Envoy, HAProxy,

NGINX, and exposes an API for integration with hardware load balancers like those from F5.

Why use another service proxy?

Linkerd
: Use this if you're already running Linkerd and want to begin adopting Istio control APIs like CheckRequest, which is used to get a thumbs-up/thumbs-down before performing an action.

NGINX
: Based on your operational expertise and need for battle-tested proxy, you might select NGINX. You might be looking for caching, web application firewall (WAF) or other functionality available in NGINX Plus, as well.

Connect
: You might choose to deploy Connect based on ease of deployment and simplicity of needs.

NOTE

I hear talk of Huawei's CSE Mesher considering integrating with Istio as an alternative to Envoy, but I consider its likelihood low.

The arrival of choice in service proxies for Istio has generated a lot of excitement. Linkerd's integration was created early in Istio's 0.1.6 release. Similarly, the ability to use NGINX as a service proxy through the nginMesh project (see Figure 4-1) was provided early in Istio release cycle.

NOTE

You might find this article on how to customize an Istio service mesh and its adjoining webcast helpful in further understanding Istio's extensibility with respect to swappable service proxies.

Without configuration, proxies are without instructions to perform their tasks. Pilot is the head of the ship in an Istio mesh, so to speak, keeping synchronized with the underlying platform by tracking and representing its services to `istio-proxy`. `istio-proxy` contains the proxy of choice (e.g. Envoy). Typically, the same `istio-proxy` Docker image is used by Istio sidecar and Istio ingress gateway,

which contains not only the service proxy but also the Istio Pilot agent. The Istio Pilot agent pulls configuration down from Pilot to the service proxy at frequent intervals so that each proxy knows where to route traffic. In this case, nginMesh's translator agent performs the task of configuring NGINX as the `istio-proxy`. Pilot is responsible for the life cycle of `istio-proxy`.

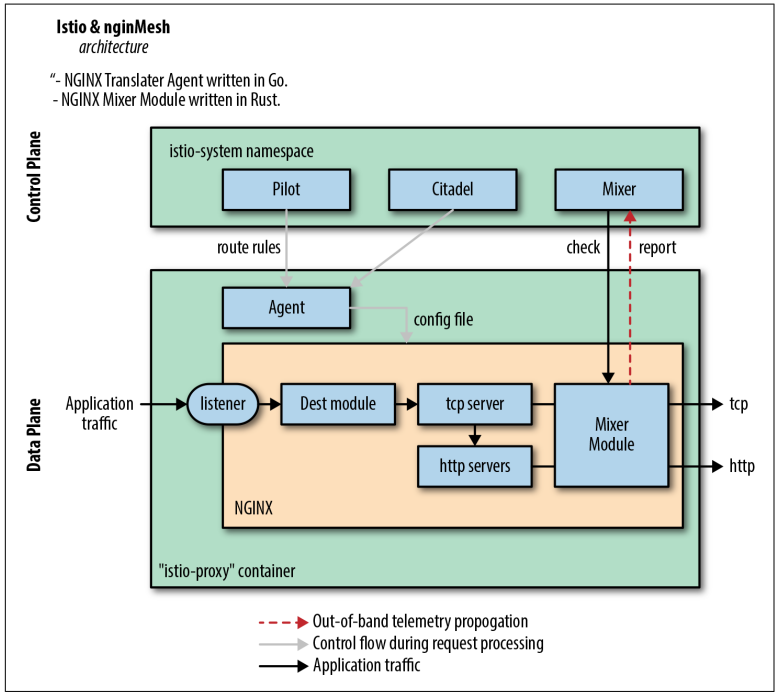


Figure 4-1. Example of swapping proxies—Istio + nginMesh.

Extensible Adapters

Istio's Mixer control plane component is responsible for enforcing access control and usage policies across the service mesh and collecting telemetry data from the sidecar proxy. Mixer categorizes adapters based on the type of data they consume.

Future extensibility might come in the form of secure key stores for Hardware Security Modules (HSMs), HashiCorp Vault integration, and better support for swapping out distributed tracing infrastructure backends.

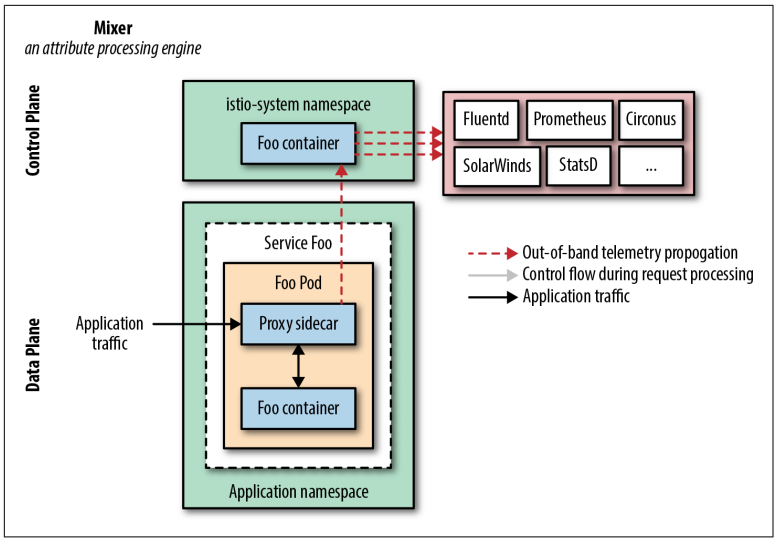


Figure 4-2. Istio Mixer is an example of an extension point in a service mesh. Mixer acts as an attribute processing engine, collecting, transforming, and transmitting telemetry.

Conclusion

The nginMesh project has drawn much interest in the use of NGINX as Istio's service proxy, as many organizations have broad and deep operational expertise built around this battle-tested proxy.

Ideally, your security posture evolves to that of a zero-trust environment.

CHAPTER 5
Conclusion

First container runtimes had their go, then container orchestrators had theirs. The next layer of infrastructure (Layer 5) will have its time. Whether you think of them as the Proxy Wars or not, 2018 is the year of the service mesh. There's much promise in the value service meshes will provide.

Although orchestrators don't bring all that you need, unfortunately, neither do service meshes. They do get you closer, however. Service meshes are missing or have nascent support for the following:

- *Distributed debugging*
- Provide nascent *topology and dependency graphs*, although projects like Kiali are out to improve this area.
- Participate in *application life cycle management*, but would benefit from shifting left to incorporate:
  - Deeper automated canary support with integration into Continuous Integration systems, which would improve the pipeline of many software projects
  - Automatic API documentation, perhaps, integrating with toolkits like swagger or readme.io
  - API function/interface discovery
- *White-box monitoring* to move beyond distributed tracing and into application performance monitoring.
- *Multitenancy* (multiple control planes running on the same platform).

- *Multicluster* such that each certificate authority shares the same root certificate and workloads can authenticate each other across clusters within the same mesh.
- Improve on the integration of load testing tools like slow_cooker, fortio, lago or others to identify ideal mesh *resiliency configurations* by facilitating load testing of your services' response times so that you can tune queue sizes, timeouts, retry budgets, and so on, accordingly. Meshes provide fault and delay injection. What are appropriate deadlines for your services under different load?
- Advanced circuit breaking with *fallback paths*, identifying alternate services to respond as opposed to 503 Service Unavailable errors.
- *Pluggable certificate authorities* component so that external CAs can be integrated.

To Deploy or Not to Deploy?

There are many paths to service mesh as well as a variety of deployment models. Which is right for you and your organization depends on where you are in your maturity curve (cloud-native skill set), number of services, underlying infrastructure, and how centric technology is to your business. So, should you deploy a service mesh? It depends. Table 5-1 has factors to consider:

Table 5-1. Factors to consider when opting for or against a service mesh

Consideration factor	Lightly consider a service mesh (external client focus)	Strongly consider a service mesh (internal/external client focus)
Service Communication	Low inter-service communication.	High inter-service communication.
Observability	Edge focus—metrics and usage are for response time to clients and request failure rates.	Edge plus internal focus—observability is key for understanding service behavior.
Client Focus	Strong separation of external and internal users. Focused on external API experience.	Equal and undifferentiated experience for internal and external users.
World-view of APIs	Primary client-facing interaction is through APIs are for clients only (separation of client-facing or service-to-service communication)	APIs are treated as a product; APIs are how your application exposes its capabilities.

Consideration factor	Lightly consider a service mesh (external client focus)	Strongly consider a service mesh (internal/external client focus)
Security Model	• Security at perimeter • Subnet zoning (firewalling) • Trusted internal networks (gooey center)	• Zero-trust mindset • authN and authZ between all services • Encryption between all services

I recommend starting small. As you roll out your service mesh, understand that failures in the environment can misplace blame on the service mesh. Understand what service meshes do *and what they don't*. Prepare for failures by removing a culture of blame. Learn from failures and outages. Familiarize yourself well with service mesh troubleshooting tools and built-in diagnostics of your service mesh; for example:

- `Use $ istioctl proxy-config` to inspect the current configuration of a given service proxy.
- Inspect Linkerd `l5d-error` headers, which annotate when requests fail helping you identify whether the failure is in your service or the service mesh.

Show immediate value. Observability signals are a great way of doing so and conduit makes this simple:

```
$ conduit stat deploy --from web
NAMESPACE  NAME    MESHED  SUCCESS RPS    LATENCY_P50 _P95 _P99
emojivoto  emoji      1/1  100.00% 2.0rps 1ms         2ms  2ms
emojivoto  voting     1/1   72.88% 1.0rps 1ms         1ms  1ms
```

When choosing a path, I recommend embracing an open platform that is adaptable to existing infrastructure investments and technology expertise you already have. Choose a project that embraces open standards and is API-driven to allow for automated configuration. Given that not all open source software is created equally, consider its community, including project governance, number and diversity of maintainers, and velocity. Recognize your comfort in having or not having a support contract. Understand where OSS functionality stops and "Enterprise" begins. Embrace diversity in your technology stack so that you're open to selecting best-fit technology and can experiment when needed. Realize the democratization of technology selection afforded by microservices and the high degree of control afforded by a service mesh. Account for whether your organization

has different skill sets with potentially differing subcultures. Understand that technology evolves and will change. Ensure you have the ability to change as all architectures are in a state of evolution.

About the Author

Lee Calcote is the head of technology strategy at Solarwinds, where he stewards strategy and innovation across the business. Previously, Calcote led software-defined datacenter engineering at Seagate, upleveling the systems portfolio by delivering new predictive analytics, telemetric, and modern management capabilities. Prior to Seagate, Calcote held various leadership positions at Cisco, where he created Cisco's cloud management platforms and pioneered new, automated, remote management services.

In addition to his role at Solarwinds, Calcote advises a handful of startups and serves as a member of various industry bodies, currently including the Cloud Native Computing Foundation (CNCF), and formerly, the Distributed Management Task Foundation (DMTF) and Center for Internet Security (CIS). As a Docker Captain and Cloud Native Ambassador, Calcote is an organizer of technology conferences, analyst, author, and frequent speaker.

You can reach Lee on Twitter as @lcalcote or at *gingergeek.com*.