O'REILLY

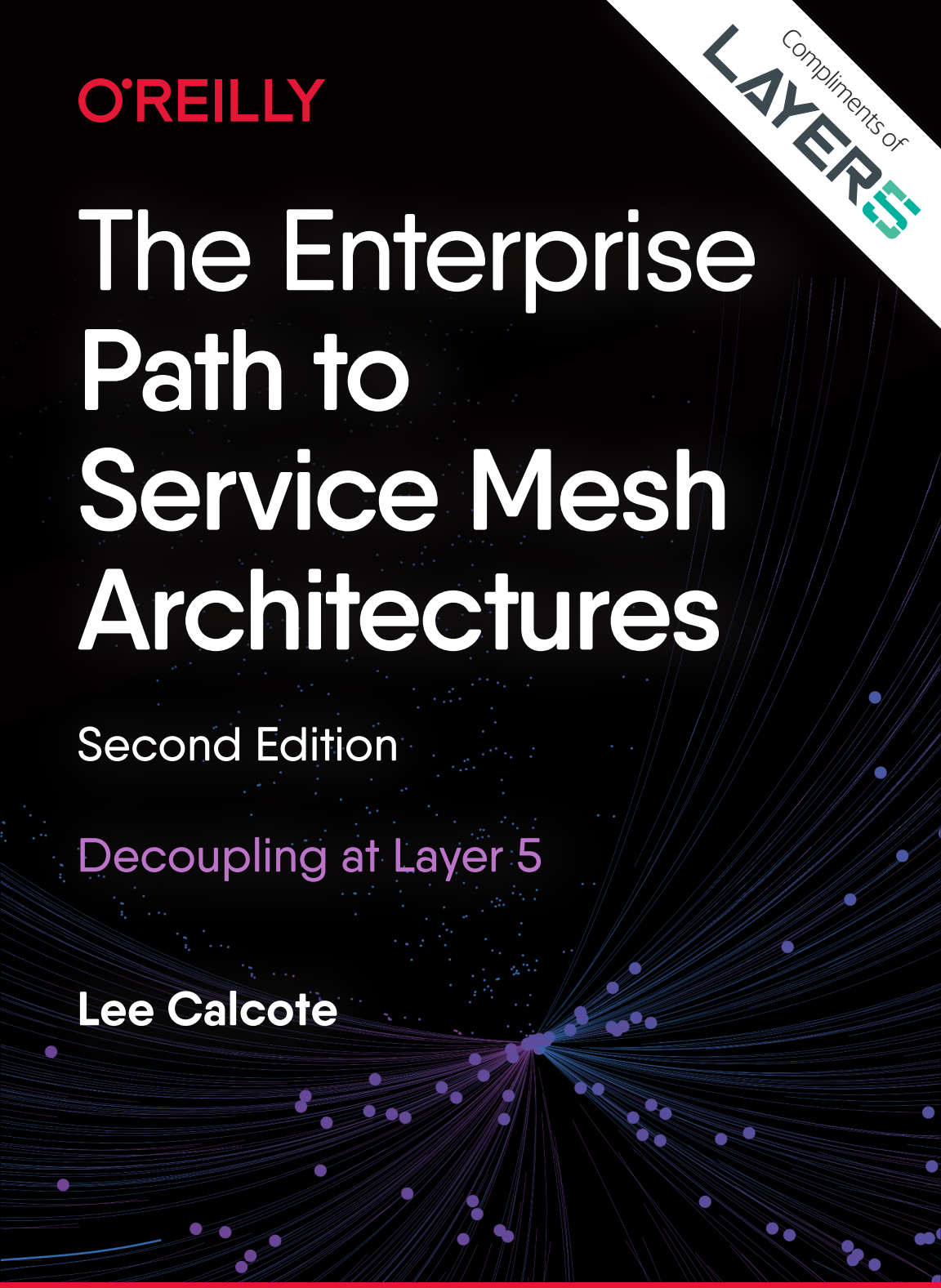
Compliments of LAYER 5

The Enterprise Path to Service Mesh Architectures

Second Edition

Decoupling at Layer 5

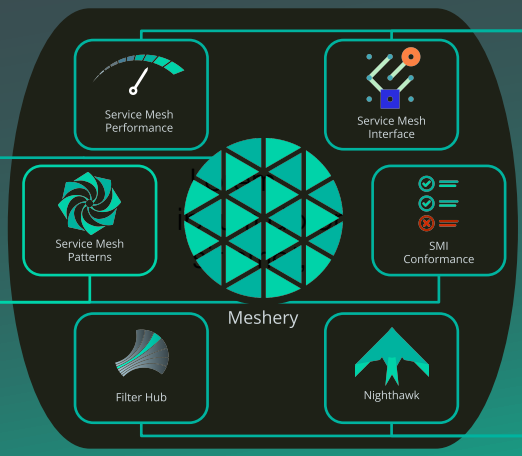
Lee Calcote



REPORT

LAYER5

expect more from your infrastructure



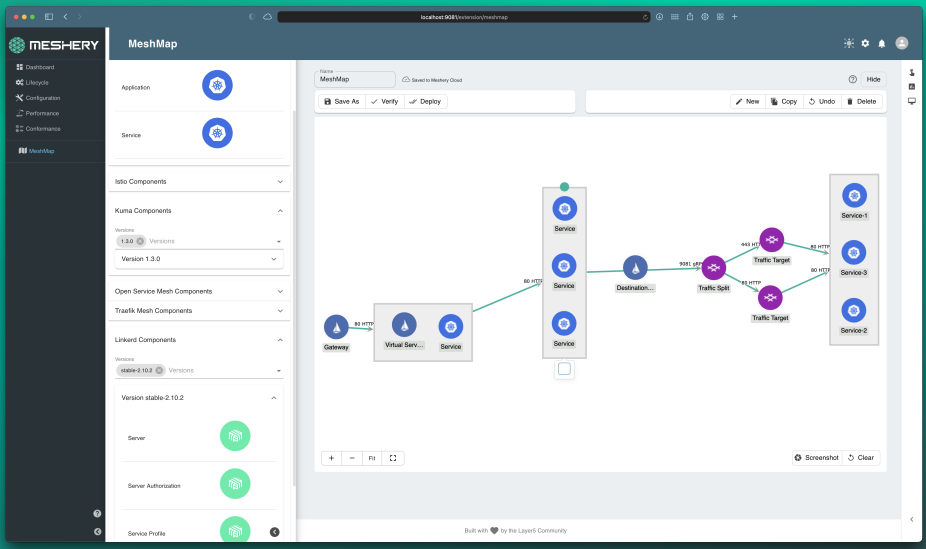
Define and Enforce Service Mesh Standards

The Only Openly Governed Service Mesh Manager

Defining Service Mesh Best Practices

Advanced Analysis and Service Mesh Intelligence

Try Layer5 MeshMap



bringing GitOps to service meshes as the world's only service mesh **designer**

Try at layer5.io/meshmap

SECOND EDITION

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*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Mary Preap
Development Editor: Gary O'Brien
Production Editor: Deborah Baker
Copyeditor: Piper Editorial, LLC
Proofreader: Abby Wheeler
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

November 2018: First Edition
November 2020: Second Edition

Revision History for the Second Edition
2020-11-09: First Release

978-1-492-08933-9

[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 are being layered into modern infrastructures ubiquitously, offering the ability to create and run robust and resilient 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 for a service mesh, 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

In this report, we'll discuss answers to the following questions:

- 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 should I adopt a service mesh?
  - What are popular deployment models and why?
  - What are practical steps for adopting 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 make an informed decision about selecting and investing in an architecture and deployment model. This will allow them to provide visibility, resiliency, traffic, and security control of their distributed application services.

Acknowledgments

Many thanks to Matt Turner, Ronald Petty, Karthik Gaekwad, Alex Blewitt, 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 look to the microservices and containers movement—the cloud-native approach to designing scalable, independently delivered services as a catalyst. 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 that create new features quickly as they iteratively and continuously deliver software (typically as a service). The decoupling of engineering teams and their increased speed is the most significant driver of microservices as an architectural model.

Operating Many Services

And, sure, the first couple of 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 3 or 100, the onset of a major technology challenge is inevitable. Different medicines are dispensed to alleviate microservices headaches; the 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 consider 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 fallacies that networks are reliable, are without latency, and have infinite bandwidth and that communication is guaranteed (it's worth reflecting on the fact that these same assumptions are held for service components using internal function calls). When you consider how critical the ability to control and secure service communication is to distributed systems that rely on network calls with every transaction every time an application is invoked, you begin to understand that you are under-tooled and see 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 intent-based networking for microservices describing the desired behavior of the network in the face of constantly changing conditions and network topology. At their core, service meshes provide:

- A services-first network
- A developer-driven network
- A network that is primarily concerned with removing the need for developers to build infrastructure concerns into their application code
- A network that empowers operators with the ability to declaratively define network behavior, node identity, and traffic flow through policy

- A network that enables service owners to control application logic without engaging developers to change its code
- 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 three planes: a *management plane*, a *control plane*, and a *data plane*. The concept of these three planes immediately resonates with network engineers by the analogous way 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.

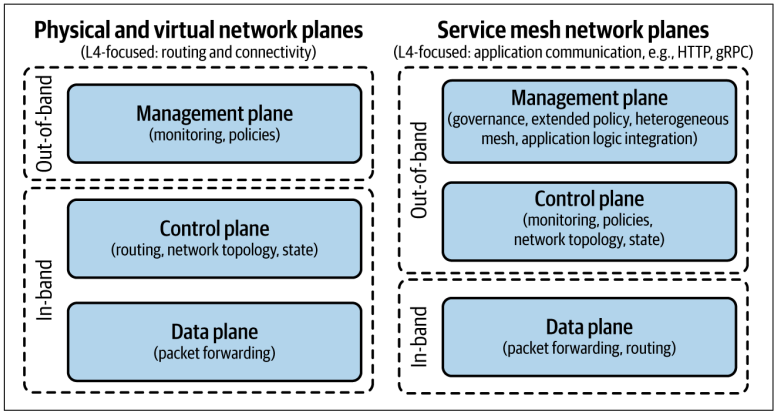


Figure 1-1. Physical networking versus software-defined networking planes

Network engineers also receive training in the OSI model. The OSI model is shown in Figure 1-2, as a refresher for those who have not seen it in some time. We will refer to various layers of this model throughout the book.

	Layer			
	Number	Name	Protocol data unit (PDU)	Function
Host layers	7	Application	Data	High-level APIs, including resource sharing and remote file access.
	6	Presentation		Translation of data between a networking service and an application; including character encoding, data compression, and encryption/deception.
	5	Session		Managing communication sessions, i.e., continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes.
	4	Transport	Segment, datagram	Reliable transmission of data segments between points on a network, including segmentation, acknowledgment, and multiplexing.
Media layers	3	Network	Packet	Structuring and managing a multinode network, including addressing, routing, and traffic control.
	2	Data link	Frame	Reliable transmitting of data frames between two nodes connected by a physical layer.
	1	Physical	Bit, symbol	Transmission and reception of raw bit streams over a physical medium.

Figure 1-2. Seven-Layer OSI model (source: Wikipedia)

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

Physical network planes

The physical network 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 delivered to end devices such as PCs and servers. Routers and switches use hardware chips—application-specific integrated circuits (ASICs)—to forward data plane traffic as quickly as possible. The physical networking data plane references a forwarding information base (FIB). A forwarding information base is a simple, dynamic table that maps a media access control address (MAC address) to a physical network port to transit traffic at wire speed (using ASICs) to the next device.

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. Physical network control planes operate in-band of network traffic, leaving them susceptible to denial-of-service (DoS) attacks that either directly or indirectly result in:

- Exhaustion of memory and/or buffer resources
- Loss of routing protocol updates and keepalives
- Slow or blocked access to interactive management sessions
- High CPU utilization
- Routing instability, interrupted network reachability, or inconsistent packet delivery

The physical networking *management plane* is the logical entity that describes the traffic used to access, manage, and monitor all of the network elements commonly using protocols like SNMP, SSH, HTTPS, and, heaven forbid, Telnet. 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. Separation of the management path from the data path has the desired effect of creating a security boundary. Restricting management plane access to devices on trusted networks is critical.

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 standardize and decouple. Open vSwitch and OpenDaylight are two examples of SDN projects. 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 1-3, 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 physical mesh networks when the networking medium is readily susceptible to line-of-sight, weather-induced, or other disruption, and, therefore, when 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, shown in Figure 1-3, is the design of choice for service mesh architectures.

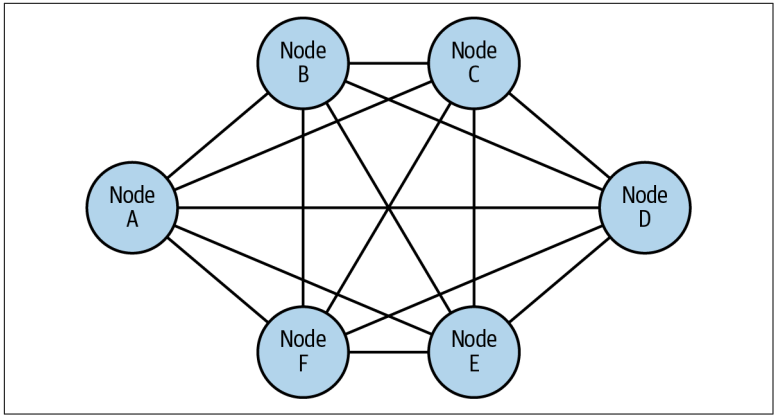


Figure 1-3. Mesh topology—fully connected network nodes

Service mesh network planes

Service mesh architectures typically employ the same three networking planes: data, control, and management (see Figure 1-4).

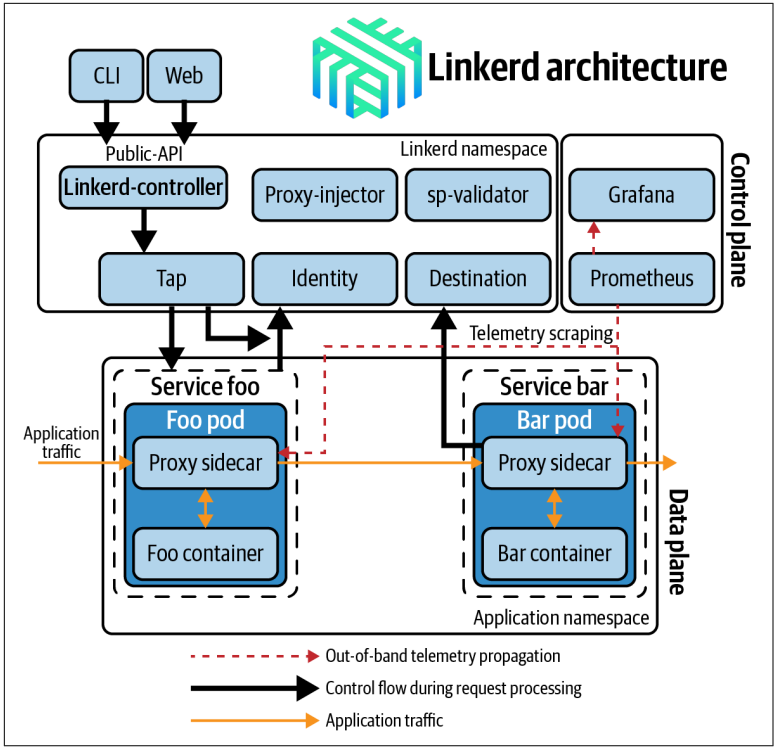


Figure 1-4. An example of service mesh architecture. In Linkerd's architecture, control and data planes divide in-band and out-of-band responsibility for service traffic.

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 intraservice communication as well as inbound (ingress) and outbound (egress) service mesh 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 prior to sending (or not sending) along to the application. To redirect traffic from the service proxy to the service application, traffic is transparently intercepted and redirected to the service proxy. The interception and redirection of traffic between the service proxy and service application places the service application's

container onto a network it would otherwise not be on. The service proxy sees all traffic to and from the service application (with small exception, this is the case in most service mesh architectures). Service proxies are the building blocks of service mesh data planes.

NOTE

Traffic Interception and Redirection

Service meshes vary in the technology used to intercept and redirect traffic. Some meshes are flexible as to whether a given deployment uses iptables, IPVS, or eBPF as the technology to transparently proxy requests between clients and service applications. Less transparently, other service mesh proxies simply require that application traffic be configured to direct their traffic to the proxy. The choice between each of these technologies affects the speed in which packets are processed and places environmental constraints on the type and kernel version of the operating system used for the service mesh deployment.

Envoy is an example of a popular proxy used in service mesh data planes. It is also often deployed more simply standalone as a load balancer or ingress gateway. The proxies used in service mesh data planes are highly intelligent and may incorporate any number of protocol-specific filters to manipulate network packets (including application-level data). With technology advances like WebAssembly, extending data plane capabilities means that service meshes are capable of injecting new logic into requests while simultaneously handling high traffic load.

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/or a user interface for you to interact with the mesh. Each of these commonly 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 [CI/CD] pipeline), where, in practice, configuration is most often version controlled and updated.

NOTE

Proxies are not regarded as the source of truth for the state of the mesh. 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 are generally considered stateless.

Linkerd (pronounced "linker-dee") and Istio (pronounced "Ist-tee-oh"), two popular, open source service meshes, provide examples of how the data and control planes are packaged and deployed. In terms of packaging, Linkerd v1 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 (`Galley`, `Pilot`, and `Citadel`) to pair by default with Envoy (a data plane) packaged together as "Istio." Envoy is often labeled a service mesh, but inappropriately so, because it takes packaging with a control plane to form a service mesh.

A service mesh management plane is a higher-order level of control, as shown in Figure 1-5. A management plane may provide a variety of functions. As such, implementations vary in their functionality, some focusing on orchestrating service meshes (e.g., service mesh life-cycle management) and mesh federation, providing insight across a collection of diverse meshes. Some management planes focus on integrating service meshes with business process and policy, including governance, compliance, validation of configuration, and extensible access control.

In terms of deployments of these planes, data planes, like that of Linkerd v2, have proxies that are created as part of the project and are not designed to be configured by hand but are instead designed so their behavior will be entirely driven by the control plane. 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). In terms of control plane deployment, using Kubernetes as the example infrastructure, control planes are typically deployed in a separate "system" namespace. Management planes are deployed both on and off cluster, depending on how deeply they integrate with noncontainerized workloads and a business's backend systems.

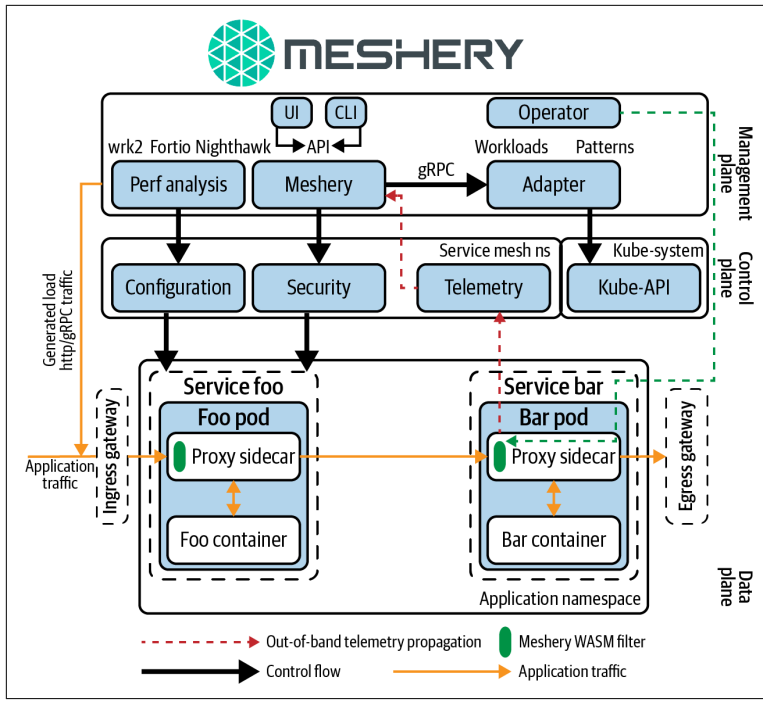


Figure 1-5. Architecture of Meshery, the service mesh management plane

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 (networking being a Layer 4 and below focus). 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 they do not require one because 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 noncontainer 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 may be rescheduled as 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 around 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 attacks. 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 of 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 given without the need to change your application code (or *much of it, depending*).

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 provide opaque monitoring of *service-to-service communication* (observing metrics and logs external to the application) and *code-level monitoring* (offering observable signals from within the application). Some service meshes work in combination with a distributed tracing library to deliver code-level monitoring, while other service meshes enable a deeper level of visibility through protocol-specific filters as a capability of their proxies. Comprising the data plane, proxies 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 1-6.

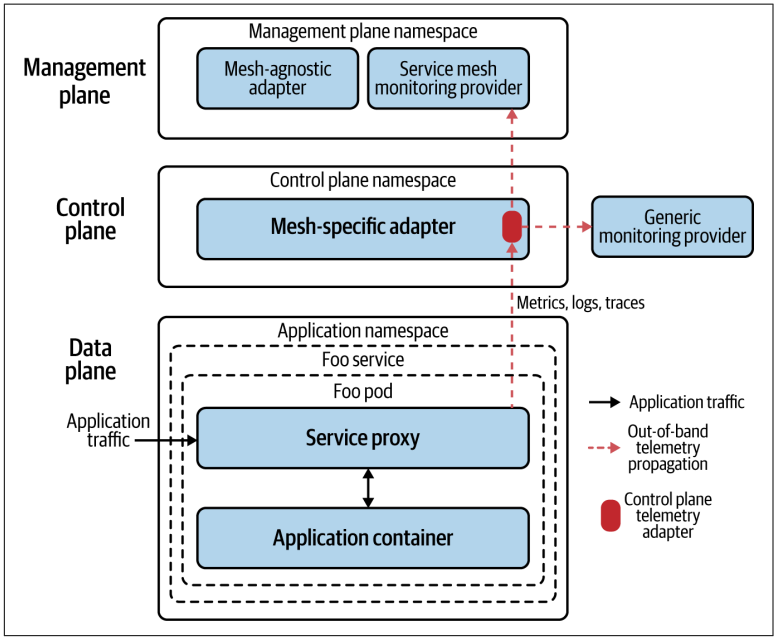


Figure 1-6. Among other things, a service mesh is an accounting machine. Service meshes can collect multiple telemetric signals and send them for monitoring and facilitating decisions such as those dealing with authorization and quota management.

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 1-6 illustrates how telemetry transmitted through service mesh logs includes source and destination, request protocol, endpoint (URL), associated response code, and response time and size.

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 enable automation for things like autoscaling, as an example. Example telemetry emitted by service mesh metrics includes global request volume, global success rate, individual service responses by version, and source and time, as shown in Figure 1-7.

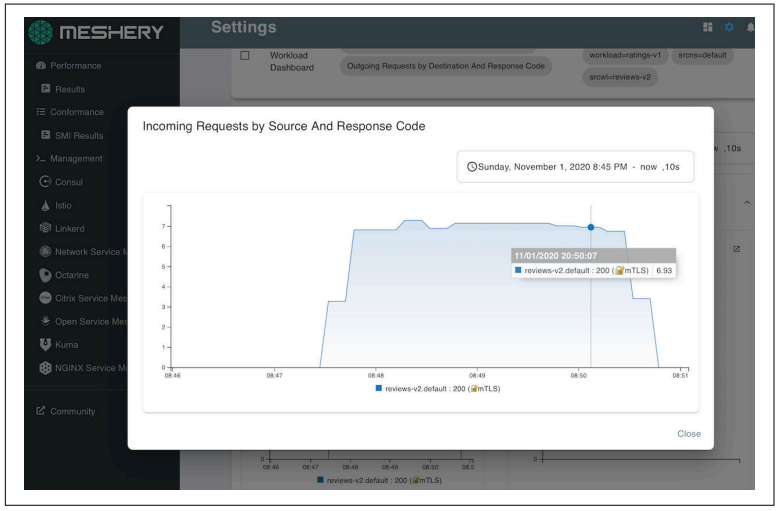


Figure 1-7. Request metrics generated by Istio and visible in Meshery

Tracing

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

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, timeouts, deadlines, and retries.

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 and are enforced at a point in time when 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 1-8, 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 into the calculation of timing the next retry attempt.

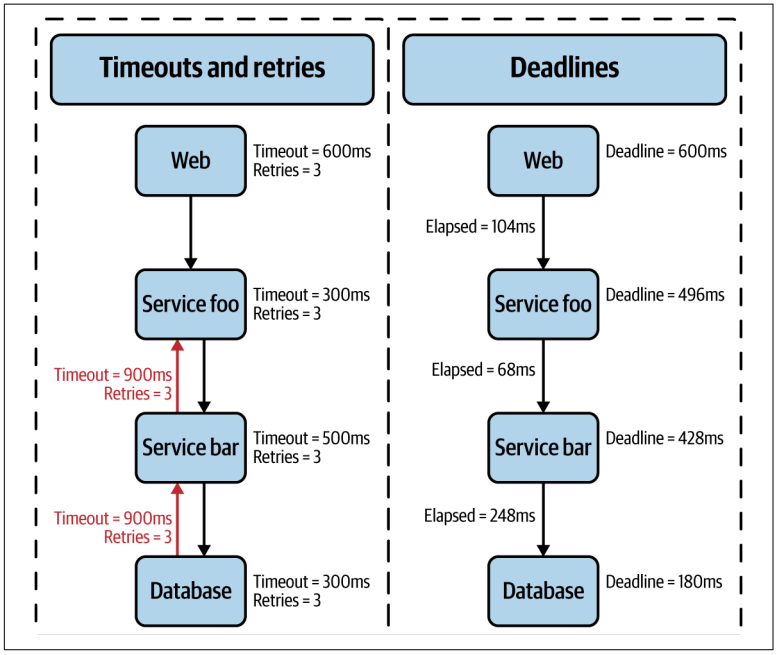


Figure 1-8. Deadlines, not ubiquitously supported by different service meshes, set feature-level timeouts

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 (or 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 will avoid excessive 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 identified by a user access token) can call your service(s), and provides operational resiliency (e.g., service A can handle only 500 requests per second).

A slightly different approach is *quota management* (or *conditional rate limiting*), which 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 authenticated but 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 temperature-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 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 disseminated 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 provide a 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 1-9.

Encrypting traffic internal to your application is an important security consideration. Your application's service calls are no longer kept inside a single monolith via localhost; they are exposed over the network. 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 are used to determine whether a request is accepted. Allow and blocklist are used to identify approved and unapproved connection requests as well as more granular access-control factors like time of day.

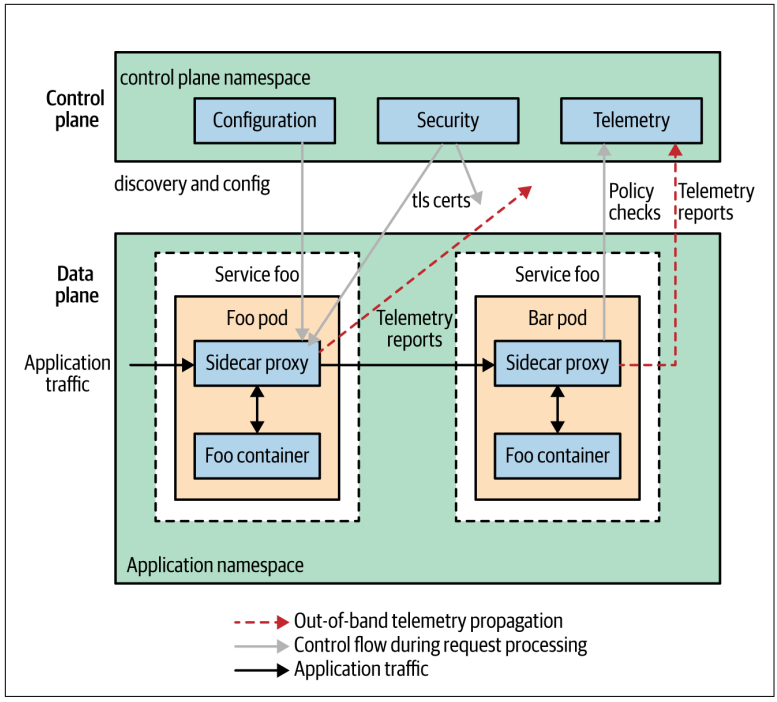


Figure 1-9. An example of service mesh architecture

Delay and fault injection

The notion that your networks and/or 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 leave users much more frustrated than a 503 after a few 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 avoid bloated service code, fat on infrastructure concerns.

You can avoid duplicative work in making services production-ready by 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, nonuniform 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, change authorization behavior, implement resiliency, and so on. 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 have at least these three roles:

developers, operators, and service owners (product owners). Just as microservices are 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 dissolve in many circumstances, like the following short list of examples:

- Operators don't necessarily need to involve developers to change how many times a service should retry before timing out or to run experiments (known as chaos engineering). They are empowered to affect service behavior independently.
- Customer success (support) 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.
- Security engineers can declaratively define authentication and authorization policies, enforced by the service mesh.
- Network engineers are empowered with an extraordinarily high degree of application-level control formerly unavailable to them.

Microservices decouple functional responsibilities within an application from each other, allowing development teams to independently iterate and move forward. Figure 1-10 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.

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. Because service meshes are

capable of deep packet inspection and mutation at the application level, service owners are empowered to bypass developers and affect business-level logic behavior without code change.

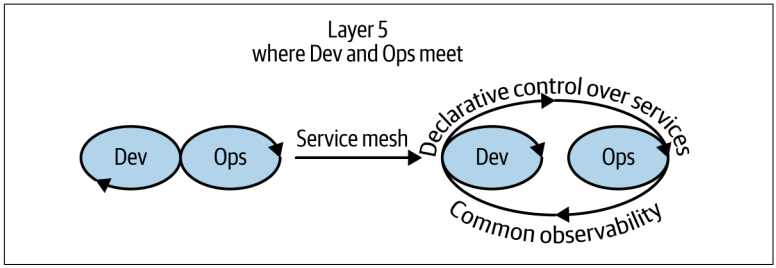


Figure 1-10. Decoupling as a way of increasing velocity

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. As a highly configurable workhorse, the service mesh is the secret weapon of cloud-native architectures, waiting to be exploited.

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. The management plane extends governance and backend systems integration and further empowers personas other than strictly operators while also significantly benefitting developers and product/service owners.

Decoupling of Dev and Ops avoids diffusion of the responsibility of service management, centralizing control over these concerns in a new infrastructure layer: Layer 5.

Service meshes make 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 (creating business logic). More-seasoned software engineers might have difficulty breaking the habit and trusting that the service mesh

will provide, or even displacing the psychological dependency on their client libraries.

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

You might already have a healthy understanding of the set of technology within the service mesh ecosystem, like API gateways, ingress controllers, container orchestrators, client libraries, and so on. But how are these technologies related to, overlapped with, or deployed alongside service meshes? Where do service meshes fit in, and why are there so many of them?

Let's begin with an examination of client libraries, predecessor to the service mesh.

Client Libraries

Client libraries (sometimes referred to as microservices frameworks) became very popular when microservices took a foothold in modern application design as a 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 who want a strongly typed language 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 (IPCs) library with built-in software load balancers. Ribbon is written in Java.

Go kit
: An open source toolkit for building microservices (or elegant monoliths) with gRPC as the primary messaging pattern. With pluggable serialization and transport, Go kit is written in Go.

Other examples include Dropwizard, Spring Boot, Akka, and so on.

NOTE

See the Layer5 service mesh landscape for a comprehensive perspective on and characterization of all popular client libraries.

Prior to the general availability of service meshes, developers commonly turned to language-specific microservices frameworks to uplevel the resiliency, security, and observability of their services. The problem with client libraries is that their use means embedding infrastructure concerns into your application code. In the presence of a service mesh, services that embed the same client library between themselves incorporate duplicative code. Services that embed different client libraries or different versions of the same client library fall prey to inconsistency in what one framework or one version of the same framework provides versus what another framework or framework version provides and how the libraries behave. Use of different client libraries is most often seen in environments with polyglot microservices.

Getting teams to update their client libraries can be an arduous process. When these infrastructure concerns are embedded into your service code, you need to chase down your developers to update and/or reconfigure these libraries, of which there might be a few in use and used to varying degrees. Aligning on the same client library, client library version, and client library configuration can consume

time unnecessarily. Enforcing consistency is challenging. These frameworks couple your services with the infrastructure, as seen in Figure 2-1.

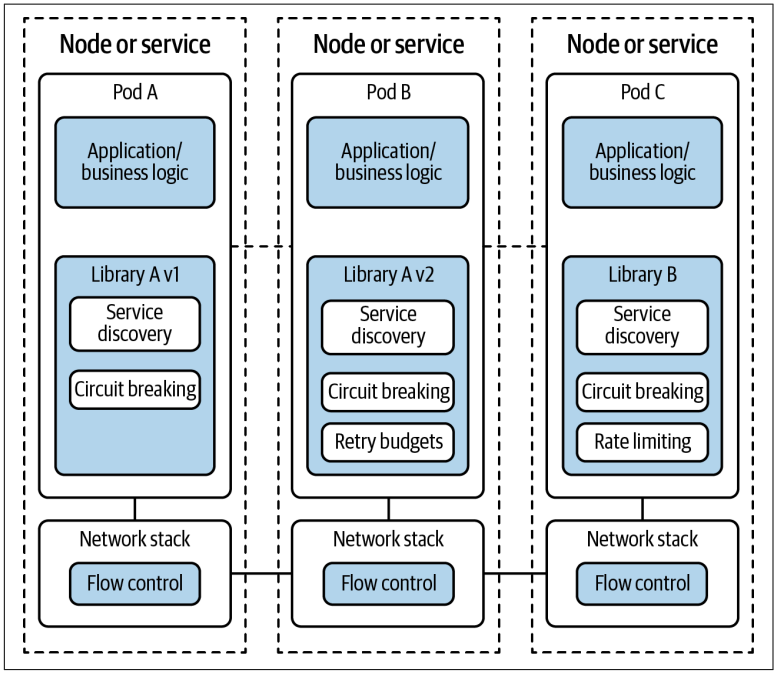


Figure 2-1. Services architecture using client libraries coupled with application logic

NOTE

Astute developers understand that service meshes not only alleviate many distributed systems concerns but also greatly enable runtime application logic, further removing the need for developers to spend time on concerns not core to the business logic they truly need to advance. The concept of service meshes providing and executing application and business logic is an advanced topic that is lightly covered in Chapter 4.

When infrastructure code is embedded in the application, different services teams must get together to negotiate things like timeouts and retries. A service mesh decouples not only infrastructure code from application code, but more importantly, it decouples teams. Service meshes are generally deployed as infrastructure that resides outside your applications, but with their adoption mainstreaming,

this is changing, and their use for influencing or implementing business logic is on the rise.

API Gateways

How do API gateways interplay with service meshes?

This is a very common question, and the nuanced answer puzzles many, particularly given that within the category of API gateways lies a subspectrum. 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. For example:

- API Umbrella uses NGINX
- Kong uses NGINX
- OpenResty uses NGINX

Envoy

The Envoy project has also been used as the foundation for API gateways:

Ambassador
: With its basis in Envoy, Ambassador is an API gateway for microservices that functions standalone or as a Kubernetes Ingress Controller.

- Open source. Written in Python. From Ambassador Labs. Ambassador is currently proposed to join the CNCF.

Contour
: Based on Envoy and deployed as a Kubernetes Ingress Controller. Hosted in the CNCF.

  - Open source. Written in Go. From Heptio. Contour joined the CNCF in July 2020.

Enroute
: Envoy Route Controller. API gateway created for Kubernetes Ingress Controller and standalone deployments.

  - Open source. Written primarily in Go. From Saaras.

Additional differences between traditional API gateways and microservices API gateways revolve around which team is using the gateway: operators or developers. 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) 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.

NOTE

North-south (N-S) traffic refers to traffic between clients outside the Kubernetes cluster and services inside the cluster, while east-west (E-W) traffic refers to traffic between services inside the Kubernetes cluster.

Given their complementary nature, API gateways and service meshes are often found deployed in combination. Service meshes are on a path to ultimately offering much, if not all, of the functionality provided by API gateways.

API Management

API gateways complement other components of the API management ecosystem, such as API marketplaces and API publishing portals—both of which are surfacing in service mesh offerings. API management solutions provide analytics, business data, adjunct provider services like single sign-on, and API 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.

An API gateway can call downstream services via service mesh by offloading application network functions to the service mesh. Some API management capabilities that are oriented toward developer engagement can overlap with service mesh management planes in the following ways:

- Developers use a portal to discover available APIs and review API documentation, and as a sandbox for exercising their code against the APIs.
- API analytics for tracking KPIs, generating reports on usage and adoption trending.
- API life-cycle management to secure APIs (allocate keys) and promote or demote APIs.
- Monetization for tracking payment plans and enforcing quotas.

Today, there's overlap and underlap among service mesh capabilities and API gateways and API management systems. The overlap is rapidly increasing as service meshes are deployed and API management functionality is brought into the service mesh—which makes intuitive sense. In the presence of a service mesh, why run API gateways and API management systems separately? As service meshes gain new capabilities, use cases (shown in Figure 2-2) will overlap more and more.

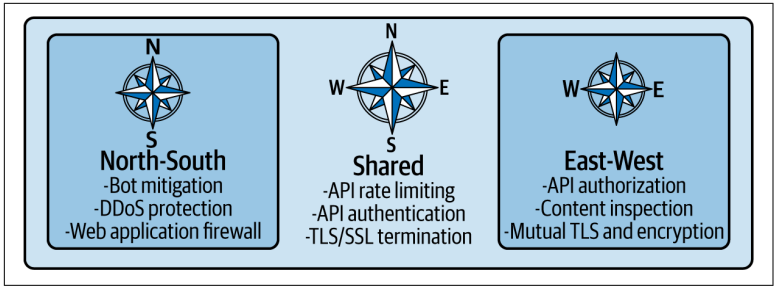


Figure 2-2. The underlap and overlap of API gateway and service proxy security functions by traffic direction

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 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[a]	Missing service-level needs
Cluster management	L7 granular traffic routing
—Host discovery	—HTTP redirects
—Host health monitoring	—CORS handling
Scheduling	Circuit breaking
Orchestrator updates and host maintenance	Chaos testing
Service discovery	Canary deploys
Networking and load balancing	Timeouts, retries, budgets, deadlines
Stateful services	Per-request routing
Multitenant, multiregion	Backpressure
	Transport security (encryption)
	Quota management
	Protocol translation (REST, gRPC)
	Policy

[a] Must have this to be considered a container orchestrator.

Additional key capabilities include simple application health and performance monitoring, application deployments, and application secrets.

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 and 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 to Layer 7 management while using the container orchestrator for reliability, availability, and scalability of container infrastructure.

Service Meshes

In a world of many service meshes, you have a choice when it comes to which service mesh(es) to adopt. For many of you, your organization will end up with more than one type of service mesh. Every organization I've been in has multiple hypervisors for running VMs, multiple container runtimes, different container orchestrators in use, and so on. Infrastructure diversity is a reality for enterprises. Diversity is driven by a broad set of workload requirements. Workloads vary from those that are process-based to those that are event-driven in their design. Some run on bare metal, while other workloads execute in functions. Others represent each and every style of deployment artifact (container, virtual machine, and so on) in between. Different organizations need different scopes of service mesh functionality. Consequently, different service meshes are built with slightly divergent use cases in mind, and therefore, the architecture and deployment models of service meshes differ between

them (see Figure 2-3). Driven by Cloud, Hybrid, On-Prem, and Edge, service meshes are capable of enabling each of these. Microservice patterns and technologies, together with the requirements of different edge devices and their function along with ephemeral cloud-based workloads, provide myriad opportunities for service mesh differentiation and specialization. Cloud vendors also produce or partner as they offer service mesh as a managed service on their platforms.

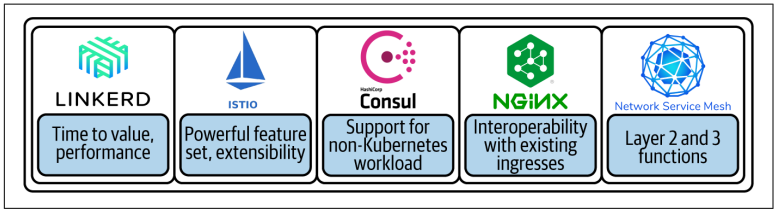


Figure 2-3. A comparative spectrum of the differences among some service meshes, based on their individual strengths

Open source governance and corporate interests dictate a world of multiple meshes, too. A huge range of microservice patterns drives service mesh opportunity. Open source projects and vendors create features to serve microservice patterns (they splinter the landscape and function differently), and some accommodate hybrid workloads. Noncontainerized workloads need to integrate and benefit from a service mesh as well.

As the number of microservices grows, so too does the need for service meshes, including meshes native to specific cloud platforms. This leads to a world where many enterprises use multiple service mesh products, whether separately or together.

Service Mesh Abstractions

Because there are any number of service meshes available, independent specifications have cropped up to provide abstraction and standardization across them. Three service mesh abstractions exist today:

Service Mesh Performance (SMP)
: This is a format for describing and capturing service mesh performance. Created by Layer5; Meshery is the canonical implementation of this specification.

Multi-Vendor Service Mesh Interoperation (Hamlet)
: This is a set of API standards for enabling service mesh federation. Created by VMware.

Service Mesh Interface (SMI)
: This is a standard interface for service meshes on Kubernetes. Created by Microsoft; Meshery is the official SMI conformance tool used to ensure that a cluster is properly configured and that its behavior conforms to official SMI specifications.

Service Mesh Landscape

With a sense now of why it's a multi-mesh world, let's begin to characterize different service meshes. Some service meshes support non-containerized workloads (services running on a VM or on bare metal), whereas some focus solely on layering on top of a container orchestrator, most notably Kubernetes. All service meshes support integration with service discovery systems. The subsections that follow provide a very brief survey of service mesh offerings within the current technology landscape.

NOTE

The list included here is neither exhaustive nor intended to be a detailed comparison but rather a simple overview of some of the available service meshes and related components. Many service meshes have emerged, and more will emerge after the publication of this book. See the Layer5 service mesh landscape for a comprehensive perspective and characterization of all of the service meshes, service proxies, and related tools available today. This landscape is community-maintained and contrasts service meshes so the reader can make the most informed decision about which service mesh best suits their needs.

Data Plane

Service proxies (gateways) are the elements of the data plane. How many are present is both a factor of the number of services you're running and the design of the service mesh's deployment model. Some service mesh projects have built new proxies, while many others have leveraged existing proxies. Envoy is a popular choice as the data plane element.

BFE
: BFE is a modern proxy written in Golang and is now hosted by the CNCF. It supports different load balancing algorithms and multiple protocols, including HTTP, HTTPS, SPDY, HTTP/2, WebSocket, TLS, and FastCGI. BFE defines its own domain-specific language in which users can configure rule and content-based routing.

Envoy
: Envoy is a modern proxy written in C++ and hosted by the CNCF. Envoy's ability to hot reload both its configuration and itself (upgrade itself in place while handling connections) contributed to its initial popularity. Any number of projects have been built on top of Envoy, including API gateways, ingress controllers, service meshes, and managed offerings by cloud providers. Istio, App Mesh, Kuma, Open Service Mesh, and other service meshes (discussed in the Control Plane section) have been built on top of Envoy.

Linkerd v2
: The Linkerd2-proxy is built specifically for the service mesh sidecar use case. Linkerd can be significantly smaller and faster than Envoy-based service meshes. The project chose Rust as the implementation language to be memory-safe and highly performant. This service proxy purports a sub-1ms p99 traffic latency. Open source. Written in Rust. From Buoyant.

NGINX
: Launched in September 2017, the nginMesh project deploys NGINX as a sidecar proxy in Istio. The nginMesh project has been set aside as of October 2018. Open source. Written primarily in C and Rust. From NGINX.

The following are a couple of early, and now antiquated, 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.

Control Plane

Control plane offerings include the following:

Consul
: Announced service mesh capable intention in v1.5. Became a full service mesh in v1.8. Consul uses Envoy as its data plane, offering multicluster federation.

  - Open and closed source. From HashiCorp. Primarily written in Go.

Linkerd
: Linkerd is hosted by the Cloud Native Computing Foundation (CNCF) and has undergone two major releases with significant architectural changes and an entirely different code base used between the two versions.

Linkerd v1
: The first version of Linkerd was built on top of Twitter Finagle. Pronounced "linker-dee," it includes both a proxying data plane and a control plane, Namerd ("namer-dee"), all in one package.

  - Open source. Written primarily in Scala.
  - Data plane can be deployed in a node proxy model (common) or in a proxy sidecar (not common). 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.

Linkerd v2
: The second major version of Linkerd is based on a project formerly known as Conduit, a Kubernetes-native and Kubernetes-only service mesh announced as a project in December 2017. In contrast to Istio and in learning from Linkerd v1, Linkerd v2's design principles revolve around a minimalist architecture and

zero configuration philosophy, optimizing for streamlined setup.

- Open Source. From Buoyant. Control plane written in Go. Hosted by the CNCF.
- Support for gRPC, HTTP/2, and HTTP/1.x requests, plus all TCP traffic. Currently only supports Kubernetes.

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.
- Was the first service mesh to promote the model of supporting automatic injection of service proxies as sidecars using Kubernetes admission controller.
- Many projects have been built around Istio—commercial, closed source offerings built around Istio include AspenMesh, VMware Tanzu Service Mesh, and Octarine (acquired by VMware in 2020).
- Many projects have been built within Istio. Commercial, closed source offerings built inside Istio include Citrix Service Mesh. To be built "within Istio" means to offer the Istio control plane with an alternative service proxy. Citrix Service Mesh displaces Envoy with CPX.
  - An open source, data plane proxy, MOSN released support for running under Istio as the control plane, while displacing Envoy as the service proxy.
- Mesher. Layer 7 (L7) proxy that runs as a sidecar deployable on HUAWEI Cloud Service Engine.
  - Open source. Written primarily in Go. From HUAWEI.

NGINX Service Mesh

Released in September 2020, NGINX Service Mesh is a more recent entrant into the service mesh arena. Using an NGINX Plus augmented to interface with Kubernetes natively as its data plane, supports ingress and egress gateways through NGINX Plus Kubernetes Ingress Controllers. Using the Service Mesh

Interface (SMI) specification as its API, NGINX Service Mesh presents its control plane as a CLI, `meshctl`.

- Open and closed source. From NGINX. Primarily written in C.

Other examples include Open Service Mesh, Maesh, Kuma, and App Mesh.

Many other service meshes are available. This list is intended to give you a sense of the diversity of service meshes available today. See the community-maintained, Layer5 service mesh landscape page for a complete listing of service meshes and their details.

Management Plane

The management plane resides a level above the control plane and offers a range of potential functions between operational patterns, business systems integration, and enhancing application logic while operating across different service meshes. Among its uses, a management plane can perform workload and mesh configuration validation—whether in preparation for onboarding a workload onto the mesh or in continuously vetting their configuration as you update to new versions of components running your control and data planes or new versions of your applications. Management planes help organizations running a service mesh get the most out their investment. One aspect of managing service meshes includes performance management—a function at which Meshery excels.

Meshery
: The service mesh management plane for adopting, operating, and developing on different service meshes, Meshery integrates business processes and application logic into service meshes by deploying custom WebAssembly (WASM) modules as filters in Envoy-based data planes. It provides governance, policy and performance, and configuration management of service meshes with a visual topology for designing service mesh deployments and managing the fine-grained traffic control of a service mesh.

  - Open source. Created by Layer5. Primarily written in Go. Proposed for adoption in the CNCF.

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 this language. By the nature of their task, data planes must be highly efficient in the interception, introspection, and rewriting of network traffic. As a data plane component, Envoy is written in C++11 because it provides excellent performance (surprisingly, 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 Linkerd v2's data plane component and for the former nginMesh's Mixer module (see "How to Customize an Istio Service Mesh") and is now being used in WebAssembly programs as data plane filters (see the Layer 5 page, "Learn How to Write WASM Filters for Envoy in Rust and Deploy with Consul").

Conclusion

Client libraries (microservices frameworks) come with their set of challenges. Service meshes move these concerns into the service proxy and decouple them from the application code. API gateways are the technology with the most overlap in functionality, but they are deployed at the edge, not on every node or within every pod. As service mesh deployments evolve, I'm seeing an erosion of separately deployed API gateways and in an inclusion of them within the service mesh. Between client libraries and API gateways, service meshes offer enough consolidation functionality to either diminish their need or replace them entirely with a single layer of control.

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. Service meshes offer a robust set of observability, security, traffic, and application controls beyond that of Kubernetes. Service meshes are a necessary layer of cloud native infrastructure. There are many service meshes available along with service mesh specifications to abstract and unify their functionality.

CHAPTER 3

Adoption and Evolutionary Architectures

What practical steps can be taken 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 deployment model with lowest risk as calculated based on their operating environment and their current operations skill set.

Piecemeal Adoption

Desperate to gain an understanding of what's going on across 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. Stepping into using a service mesh for its ability to provide enhanced observability is a high-value, relatively safe first step. First steps for others might be on a parallel path. A financial organization, for example, might seek improved security with strong identity (assignment of a certificate to each individual service) and strong encryption through mutual TLS between each service, while others might begin with an ingress proxy as their entryway to a larger service mesh deployment.

Consider an organization that has hundreds of existing services running on virtual machines (VMs) external to the service mesh that have little to no service-to-service traffic, rather 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 Ingress Gateway) and begin gaining granular traffic control (e.g., path rewrites) and detailed service monitoring without immediately deploying hundreds of service proxies (Figure 3-1).

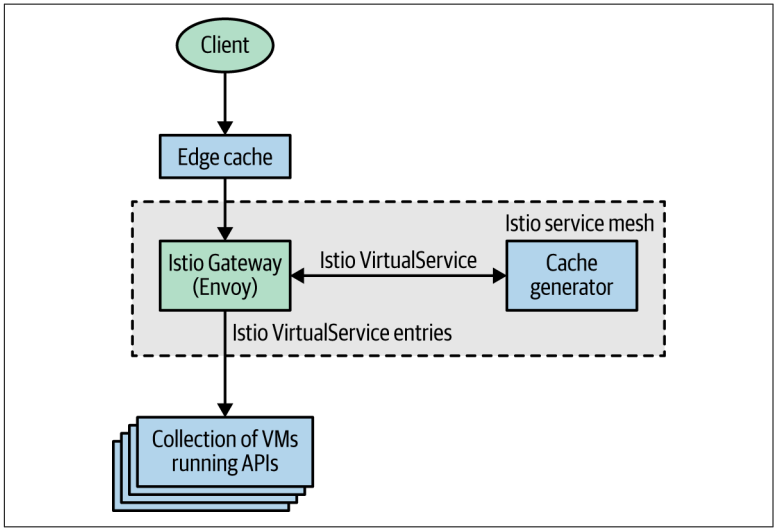


Figure 3-1. Simple service mesh deployment primarily using ingress traffic control

You can start with a full service mesh deployment from the get-go, or you can work your way up to 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 in this approach. When teams are comfortable with their understanding of the deployment, have gained operational expertise, and derived substantive value, then 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. Over time, though, this will change—full service mesh deployments will become ubiquitous. More than this, application developers and service (product) owners will begin to rely on the power of a service mesh to empower and satisfy their requirements as well.

Engineering maturity and skill set factor 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 you need. Perhaps the best approach is to mitigate risk, baby-step it, and show incremental victories considering that some service meshes provide a path to partial adoption. Some service meshes are easily deployed and digestible in one motion. Even when this is the case, though, you might find that you enable only a portion of its capabilities. Presence of a service mesh's capabilities is separate from whether those capabilities are actively engaged.

It's been my experience that observability rises to the top of why most organizations initially 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

Though not always the case, organizations generally get to security last. When they finally do, they might not want strong authentication and encryption. Although the best practice is to secure everything with strongly authenticated and authorized services, some organizations don't implement this, which means that their microservices deployments have a soft, gooey center, so to speak. Some teams are content to secure the edge of their network but would still like the observability and control from a service mesh.

NOTE

Needless to say, I recommend that you run workloads securely, using a service mesh to provide authentication and authorization between all service requests.

Why don't some organizations want to take advantage of service mesh's managed certificate authority? Because it is another thing to operate? Encryption takes resources (CPU cycles) and can inject a couple of microseconds of latency when connections are established. Given this, and to help with adoption, some service meshes offer a choice as to whether or not a deployment will include a certificate authority (CA). Maybe you consider the "gooey center" of your mesh to be secure because there is little to no ingress/egress traffic to/from the cluster, 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 (which 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 that enforces authorization checks across your monoliths, you can flatten your internal network, making services broadly reachable, granularly controlling which requests are authorized. There is significantly more flexibility afforded by the

power of service meshes to examine and reason over details of request traffic far beyond IP addresses and ports (Layer 3/4).

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. Fear not! Some service meshes squarely address such environments and help with modernization of such services, allowing organizations to renovate their services inventory by:

- Not having to rewrite their applications
- Adapting microservices and existing services using the same infrastructure architecture
- Facilitating adoption of new languages
- Facilitating moving to or securely connecting with services in the cloud (or on edge)

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 a 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 people consider libraries to be the first service meshes. Figure 3-2 illustrates 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 whether to use language-specific frameworks and (potentially) the application servers to run them.

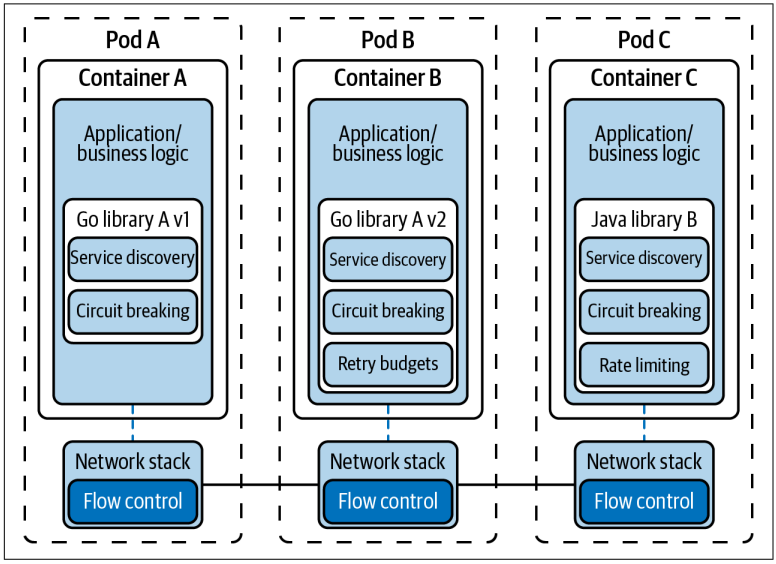


Figure 3-2. Services architecture using client libraries coupled with application logic

Using client libraries carries the following advantages and disadvantages:

Advantages

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

Disadvantages

- Strong coupling is a significant drawback
- Nonuniform functionality; upgrades are challenging in large environments (developers may need to be chased down with pitchforks to upgrade their libraries)

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.

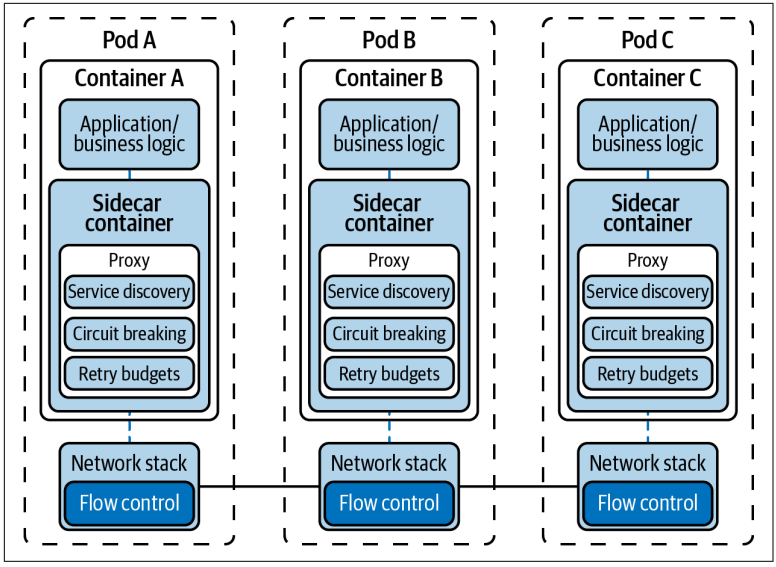


Figure 3-3. Services architecture using sidecar proxies decoupled from application logic

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, deleting the `istio-system` namespace without applying manifests to uninstall the mesh could cause you some 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.

Client Libraries as a Proxyless Service Mesh

Recognizing the merits of the service mesh design, gRPC—a high-performance, open source remote procedure call framework—has worked to support Envoy's xDS APIs such that gRPC can be dynamically (re)configured, as shown in Figure 3-4.

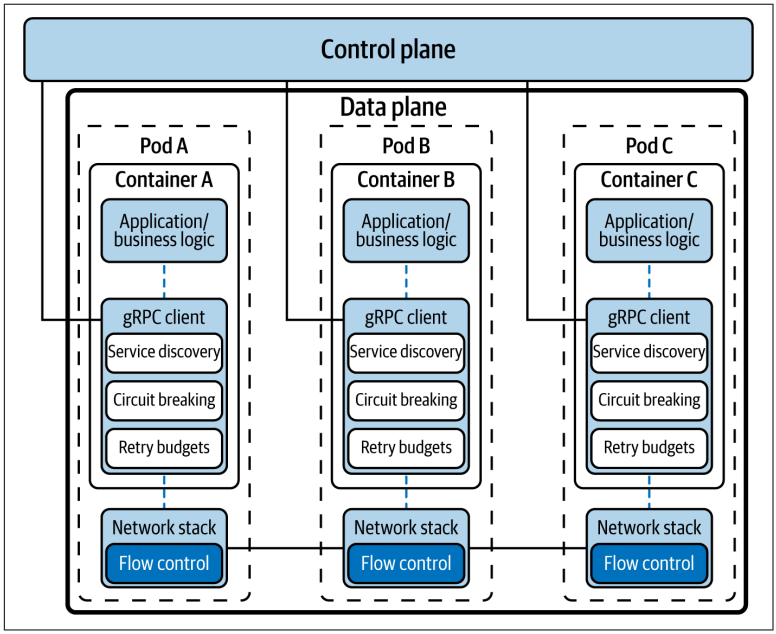


Figure 3-4. A service mesh with proxyless gRPC applications configured by a control plane

This carries the following advantages and disadvantages:

Advantages

- Enables xDS control planes to configure gRPC clients with service information such as endpoint address, health status, priority (based on proximity and capacity), and client-side traffic policies.

Disadvantages

- Continues to couple infrastructure logic with the application. Is not as capable or powerful as having a full-service proxy.

This proxyless model is an evolved form of client library usage. However, in general, use of a client library (proxyless or not) is

considered to be an early step on a path to a more robust and powerful service mesh model. For example, arguably the first service mesh, Linkerd v1, was created by former Twitter engineers, built on top of Finagle and Netty (client libraries).

Ingress or Edge Proxy

Start with load balancers (or gateways) and get scalability and availability. And just as importantly, get the ability to facilitate upgrades without service unavailability (depending on your application architecture). 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 3-5), 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 a full service mesh.

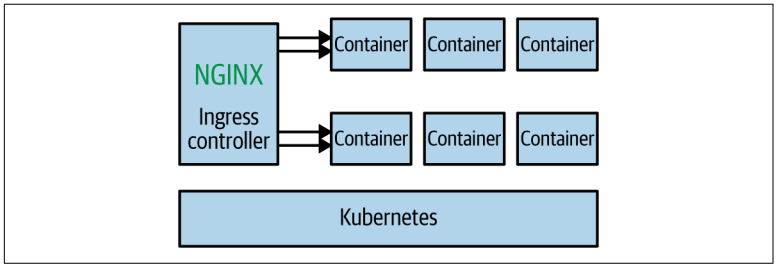


Figure 3-5. NGINX proxy as an ingress controller in Kubernetes

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 of ways:

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

Using NGINX architecture has the following advantages and disadvantages:

Advantages
- Works with existing containerized and noncontainerized services; additional proxies can be proliferated across the infrastructure over time

Disadvantages
- Lacks the benefits of service-to-service visibility and central control and backend systems integration

Router Mesh

Depicted in Figure 3-6, 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.

Given the following disadvantages, I generally recommend skipping this model, if you can:

Advantages
- A starting point for building a brand-new microservices architecture or for migrating from a monolith

Disadvantages
- When the number of services increases, it becomes difficult to manage
- A crutch on your path to a better architecture that can be overwhelmed with a single point of failure

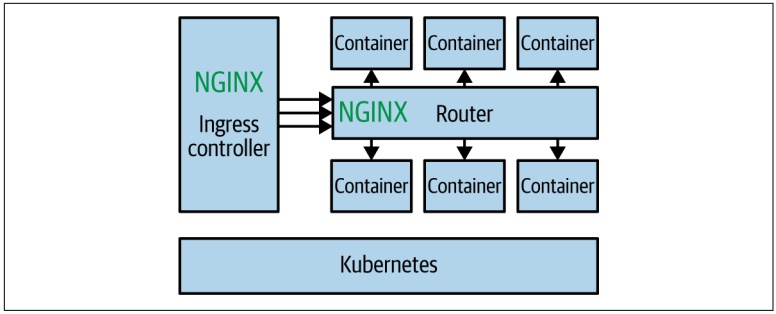


Figure 3-6. NGINX as an example router proxy deployed in Kubernetes

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 (see Figure 3-7), 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.

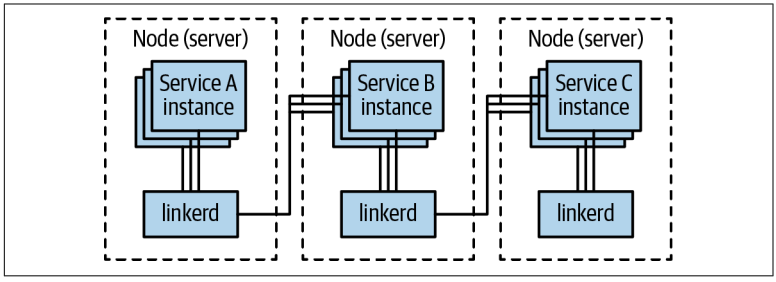


Figure 3-7. Proxy per node deployed using Linkerd (1.x)

Maesh is another, more recent example of a service mesh that has a proxy per node service mesh architecture. Like other service meshes that follow a proxy-per-node architecture, Maesh's architecture has the following advantages and disadvantages:

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 provided by host-to-host level 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 in a Fabric Model

Although this is not a common deployment model, some organizations evolve their deployments to this stage before moving onto deploying a control plane—a service mesh. This model (shown in Figure 3-8) is worth highlighting, as *sidecarring* is a useful pattern in general.

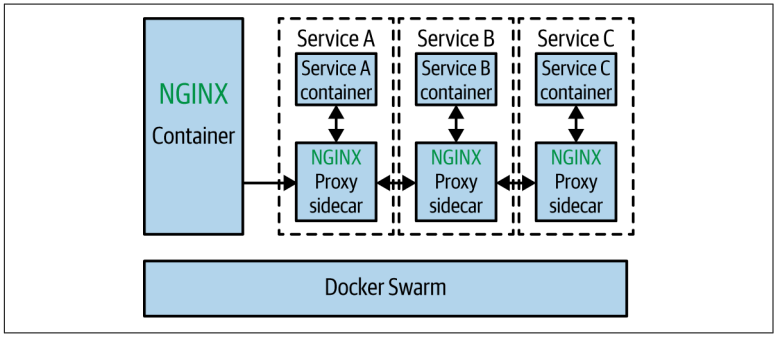


Figure 3-8. Proxy sidecars (fabric model) deployed in Docker Swarm

The pattern and usefulness of sidecarring isn't constrained to service proxies; it is a generally applicable model of deploying components of an application or necessary utilities 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 (and microservice frameworks, for that matter) support 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. The reloading and upgrading of these proxies can be facilitated through traffic-shifting techniques that 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 grows, so does the work of managing each independently. The next deployment model, sidecar proxy, is a natural next step that brings a great deal more functionality and operational control to bear, with these considerations:

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 with a 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 3-9.

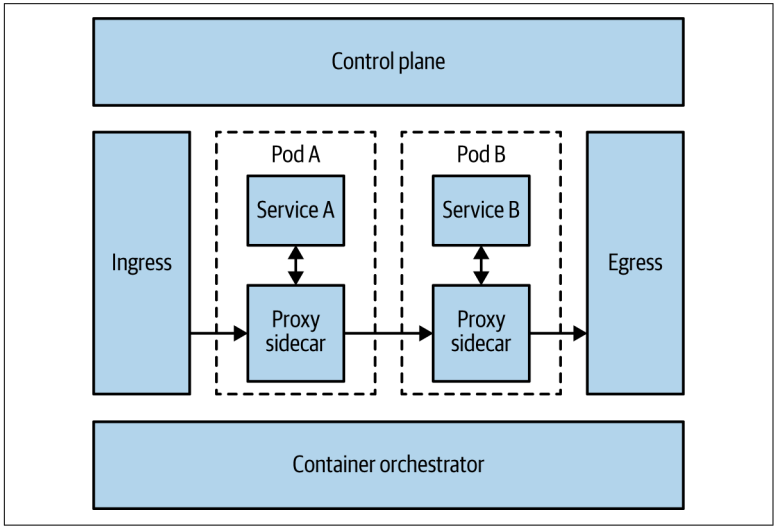


Figure 3-9. Service mesh

Service mesh implementations have evolved, allowing deployment models to evolve in concert. A number of service meshes that employ the sidecar pattern (shown in Figure 3-8) 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 automatically injected at the moment they are scheduled to run in the cluster. Alternatively, it's commonplace for a service mesh to allow you to manually inject the sidecar proxy into your application definition via a command-line interface utility

before deploying your application to the cluster. Using Kubernetes as an example, you can automatically add sidecar proxies 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: Manually Injecting a Service Proxy as a Sidecar

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

Alternatively, this command can also perform the full injection purely on the client side, by enabling the `--manual` flag.

This example shows Linkerd:

```
# Inject all the resources inside a folder and
  its sub-folders.
$ linkerd inject deployment.yml | kubectl apply -f -
```

And this example shows 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 noninjected .yaml file so that the data plane sidecars can be updated.

A number of advantages are realized in this architectural model. Its single greatest disadvantage is the concern of the Service Mesh Performance (SMP) standard and focus of MeshMark:

Advantages

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

Disadvantages

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

Multicluster and Cross-Cluster Deployments

Some service meshes support deployment across Kubernetes clusters. Istio multicluster deployments facilitate connections between service proxies (Envoys) running in different clusters to one Istio control plane or multiple Istio control planes with shared components. Services can then communicate with the central Istio control plane and form a service mesh network across multiple Kubernetes clusters. Advanced configurations of Istio's cross-cluster deployment include gateway-to-gateway communication via 15443/tcp. These deployments overload the Service Name Indicator (SNI) value by populating it with the remote Istio cluster name (including service version subset) so that when a service request is destined to a remote gateway on 15443/tcp, "zero configuration" routing is facilitated on the ingress gateway of the remote cluster. This configuration means that the identity of the client pod in the remote cluster reaches the current cluster service proxy sidecar so that the correct identities are available when authorization rules are applied.

Linkerd provides secure multicluster communication between Kubernetes clusters. Linkerd's service mirroring provides separate failure domains to avoid a single point of failure (SPOF) cluster, a unified security domain (identity is validated across clusters), and support for heterogeneous networks and clusters. In keeping with the project's design goals, Linkerd's cross-cluster communication is secure by default with mutual Transport Layer Security (mTLS) and is fully transparent to the application.

Consul supports WAN federation via mesh gateways, which requires that mesh gateways are exposed with routable addresses so that those addresses can front the mesh gateway pods with a single Kubernetes service and all traffic flows between datacenters through the mesh gateways. WAN federation is open source. Consul's WAN federation uses end-to-end mTLS between the source and destination services. The gateways function like Multiprotocol Label Switching (MPLS) Label Switch Routers (LSRs) in that they route packets based off of the Service Name Indicator (SNI) header but do not terminate TLS sessions.

Expanding the Mesh

Some service meshes support onboarding external services. Onboarding external services refers to services running on infrastructure unmanaged by the mesh, like those running on separate VMs or bare-metal servers, and bringing those 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 internal to the mesh and 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 and 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. Otherwise, what you have is an unmanaged collection of service proxies.

Service meshes support onboarding existing (noncontainerized) services onto the mesh. Service meshes can be deployed across multiple clusters. Federation of disparate service meshes is now being facilitated.

As technology evolves, capabilities are sometimes 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 load balancers deployed external to the cluster handling north-south traffic in addition to the ingress/egress proxies that handle east-west traffic within the service mesh. Over time, these two separate tiers of networking will look more and more alike.

CHAPTER 4
Customization and Integration

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

Some service meshes are designed with simplicity as their foremost design principle. While simple isn't necessarily synonymous with inflexible, there is a class of service meshes that are less extensible than their more powerful counterparts. Maesh, Kuma, Linkerd, and Open Service Mesh are designed with some customizability in mind; however, they generally focus on out-of-the-box functionality and ease of deployment. Istio is an example of a service mesh designed with customizability in mind. Service mesh extensibility comes in different forms: swappable sidecar proxies, telemetry and authorization adapters, identity providers (certificate authorities), and data plane filters.

Service mesh sidecar proxies are the real workhorse, doing the heavy lifting of traffic bytes and bits from one destination to the next. Their manipulation of packets prior to invoking application logic makes the transparent service proxies of the data plane an intriguing point of extensibility. Service proxies are commonly designed with extensibility in mind. While they are customizable in many different ways (e.g., access logging, metric plug-ins, custom authentication and authorization plug-ins, health-checking functions, rate-limiting algorithms, and so on), let's focus our exploration on the ways service proxies are customizable by way of creating custom traffic filters (modules).

The Power of the Data Plane

Control planes bring much-needed element management to operators. Data planes composed of any number of service proxies need control planes to go about the task of applying service mesh-specific use cases to their fleet of service proxies. Configuration management, telemetry collection, infrastructure-centric authorization, identity, and so on are common functions delivered by a control plane. However, their true source of power is drawn significantly from the service proxy. Users commonly find themselves in need of customizing the chain of traffic filters (modules) that service proxies use to do much of their heaving lifting. Different technologies are used to provide data plane extensibility and, consequently, additional custom data plane intelligence, including:

Lua
: A scripting language for execution inside a Just-In-Time compiler, LuaJIT

WebAssembly (WASM)
: A virtual stack machine as a compilation target for different languages to use as an execution environment

NGINX and Lua

NGINX provides the ability to write dynamic modules that can be loaded at runtime based on configuration files. These modules can be unloaded by editing the configuration files and reloading NGINX. NGINX supports embedding custom logic into dynamic modules using Lua.

Lua is a lightweight, embeddable scripting language that supports procedural, functional, and object-oriented programming. Lua is dynamically typed and runs by interpreting bytecode with a register-based virtual machine.

NGINX provides the ability to integrate dynamic Lua scripts using the `ngx_lua` module. Using NGINX with `ngx_lua` helps you offload logic from your services and hand their concerns off to an intelligent data plane. Leveraging NGINX's subrequests, the `ngx_lua` module allows the integration of Lua threads (or coroutines) into the NGINX event model. Instead of passing logic to an upstream server, the Lua script can inspect and process service traffic. `ngx_lua`

modules can be chained to be invoked at different phases of NGINX request processing.

Envoy and WebAssembly

WebAssembly, or WASM, is an open standard that defines a binary format for executable programs. Through WebAssembly System Interface (WASI), it also defines interfaces for facilitating interaction with host environments. The initial focus of these host environments was browsers and large web applications with the intention of securely running programs to improve performance. As an open standard, WASM is maintained by the W3C and has been adopted by all modern browsers. After HTML, CSS, and Javascript, WebAssembly is the fourth language to natively run in web browsers.

WASM support is coming to Envoy through the efforts of Google and Envoy maintainers embedding Google's open source high-performance JavaScript and WebAssembly engine, V8, into Envoy. Through the WebAssembly System Interface, Envoy exposes an Application Binary Interface (ABI) to WASM modules so that they can operate as Envoy filters. The way WASI works is straightforward. You write your application in your favorite languages, like Rust, C, or C++. Then, you build and compile them into a WebAssembly binary targeting the host environment. The generated binary requires the WebAssembly runtime to provide the necessary interfaces to system calls for the binary to execute. Conceptually, this is similar to JVM. If you have a JVM installed, you can run any Java-like languages on it. Similarly, with a runtime, you can run the WebAssembly binary.

Comparing Lua and WebAssembly

The introduction of WASM into service meshes leaves people pondering the merits of the use of a WebAssembly runtime. A Lua runtime can be as small as 4 kb with LuaJIT being surprisingly fast and only ~200 kb runtime.

From the perspective of the host software, the complexity is in the WebAssembly loader, not the runtime. In comparing the two, how do you measure the weight of GCC or LLVM in terms of making optimized C or C++ faster or slower than LuaJIT?

> The complexity involved in the WebAssembly runtime comes from it containing arch-specific optimizers and the intermediate representation to machine code translation stage that would normally be performed inside GCC or LLVM. Machine code can be generated once and then cached on nonvolatile storage until the hash on the WASM file changes (like the extracted contents of a tar file). Once the machine code is generated, the result is lighter than Lua because WASM has a comparable approach to sandboxing (which is to make the language/bytecode unable to describe accessing resources outside what are granted). WASM programs are compiled machine code and need no garbage collector or JIT engine.
>
> WebAssembly has the same flat, non-garbage-collected memory model that things like `malloc` and `free` expect. Garbage collection can be built into a WebAssembly application, by compiling `gc` to WebAssembly and running `gc` inside the sandbox. Forthcoming extensions like "opaque reference types" allow WebAssembly applications to interact with objects managed by a garbage collector external to the sandbox.

Envoy provides the ability to integrate additional filters in one of two ways:

- Natively, by incorporating your custom filter into Envoy's C++ source code and compiling a new Envoy version. The drawback is that you need to maintain your own version of Envoy, while the benefit is that your custom filter runs at native speed.
- Via WASM, by incorporating your custom filter as a WebAssembly binary writing in C++, Rust, AssemblyScript, or Go. The drawback is that WASM-based filters incur some overhead, while the benefit is that you can dynamically load and reload WASM-based filters in Envoy at runtime.

Envoy configuration is initialized via bootstrap on startup. Envoy's xDS APIs allow configuration to be loaded and reloaded dynamically during runtime, as shown in Figure 4-1. Envoy configuration has different sections (e.g., LDS, which is for configuring listeners, and CDS, which is for configuring clusters). Each of the sections can configure WASM plug-ins (programs).

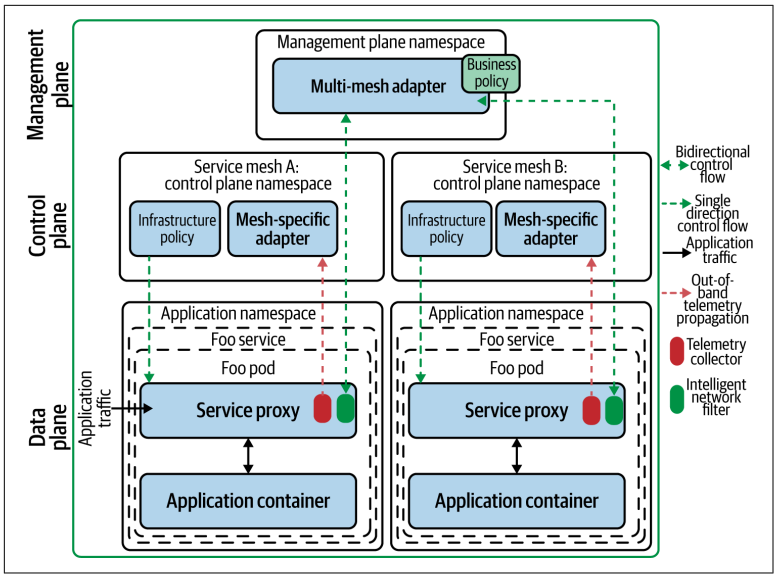


Figure 4-1. How the intelligence of the service mesh management plane and the power of the service mesh data plane combine to deliver application infrastructure logic

Dynamically (Re)loadable Intelligence

The fact that WASM programs can be dynamically loaded to inspect, rewrite, and reroute packets carrying application requests makes data planes powerful. With a management plane added to the mix, WASM programs can include business logic considerations when filtering application requests. Business logic can be implemented by the service mesh, including the service mesh implementing common application infrastructure logic:

Subscription plan enforcement
: Rate limiting requests based on user's subscription plan.

Class of service
: Directing requests to high-performance clusters based on user demographics or activity.

Multivariate testing
: Facilitating comparison of a high number of variables between deployments (service versions) and users.

A learning resource
: To try your hand at this functionality, you might try playing with the Image Hub—a sample application written in Rust to run on Consul for exploring WebAssembly modules used as Envoy filters.

WebAssembly is exciting in part because of its performance characteristics, which vary based on the type of program/filter being used. Some run between 10% to 20% overhead as compared to natively executed code for network filtering use cases (see "The Importance of Service Mesh Performance (SMP)" on page 68). WebAssembly bears some resemblance to Docker given its high degree of portability. Like the Java Virtual Machine (JVM), WASM's virtual stack machine is becoming a write once, run anywhere (WORA). WASM executables are precompiled with a healthy variety of languages supporting it as a compilation target—currently about 40 languages.

Swappable Sidecars

Functionality of the service mesh's proxy is one of the more important considerations when adopting a service mesh. From the perspective of a developer, much significance is given to a proxy's cloud-native integrations (e.g., with OpenTelemetry/OpenTracing, Prometheus, and so on). Surprisingly, a developer may not be very interested in a proxy's APIs. The service mesh control plane is the point of…well, control for managing the configuration of proxy. A developer will, however, be interested in a management plane's APIs. A top item on the list of developers' demands for proxies is protocol support. Generally, protocol considerations can be broken into two types:

TCP, UDP, HTTP
: Network team-centric consideration in which efficiency, performance, offload, and load balancing algorithm support are evaluated. Support for HTTP/2 often takes top billing.

gRPC, NATS, Kafka
: A developer-centric consideration in which the top item on the list is application-level protocols, specifically those commonly used in modern distributed application designs.

The reality is that selecting the perfect proxy involves more than protocol support. Your proxy should meet all key criteria:

- High performance and low latency
- High scalability and small memory footprint
- Deep observability at all layers of the network stack
- Programmatic configuration and ecosystem integration
- Thorough documentation to facilitate an understanding of expected proxy behavior

Any number of service meshes have adopted Envoy as their service proxy. Envoy is the default service proxy within Istio. Using Envoy's APIs, different projects have demonstrated the ability to displace Envoy as the default service proxy with the choice of an alternative.

Standardizing Data Plane APIs

Envoy's APIs are collectively known as xDS APIs. The Universal Data Plane API (UDPA) working group is pursuing a set of APIs to provide the de facto standard for L4/L7 data plane configuration (akin to the role played by OpenFlow at L2/L3/L4 in SDN). In combination with a well-defined, stable API versioning policy, the Envoy xDS APIs are being evolved to address service discovery, load-balancing assignments, routing discovery, listener configuration, secret discovery, load reporting, health check delegation, etc.

In early versions of Istio, Linkerd demonstrated an integration in which Istio was the control plane providing configuration to Linkerd proxies. Also, in more than a demonstration, NGINX hosted a project known as nginMesh in which, again, Istio functioned as the control plane while NGINX proxies ran as the data plane.

With many service proxies in the ecosystem, outside of Envoy only two have currently demonstrated integration with Istio. Linkerd is not currently designed as a general-purpose proxy; instead, it is focused on being lightweight, placing extensibility as a secondary concern by offering extensions via gRPC plug-in. Consul uses Envoy as its proxy. Why use another service proxy? Following are some examples:

NGINX
: While you won't be able to use NGINX as a proxy to displace Envoy (recall that the nginMesh project was set aside), based on your operational expertise, need for a battle-tested proxy, or integration of F5 load balancer, you might want to use NGINX. You might be looking for caching, web application firewall (WAF), or other functionality available in NGINX Plus, as well. An enhanced version of NGINX Plus that interfaces natively with Kubernetes is the service proxy used in the NGINX Service Mesh data plane.

CPX
: You might choose to deploy the Citrix Service Mesh (which is an Istio control plane with CPX data plane) if you have existing investment in Citrix's Application Delivery Controllers and have them across your diverse infrastructure, including new microservices and existing monoliths.

MOSN
: MOSN can deploy as an Istio data plane. You might choose to deploy MOSN if you need to highly customize your service proxy and are a Golang shop. MOSN supports a multiprotocol framework, and you access private protocols with a unified routing framework. It has a multiprocess plug-in mechanism, which can easily extend the plug-ins of independent MOSN processes through the plug-in framework and do some other management, bypass, and functional module extensions.

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 4-2) was provided early in the 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`.

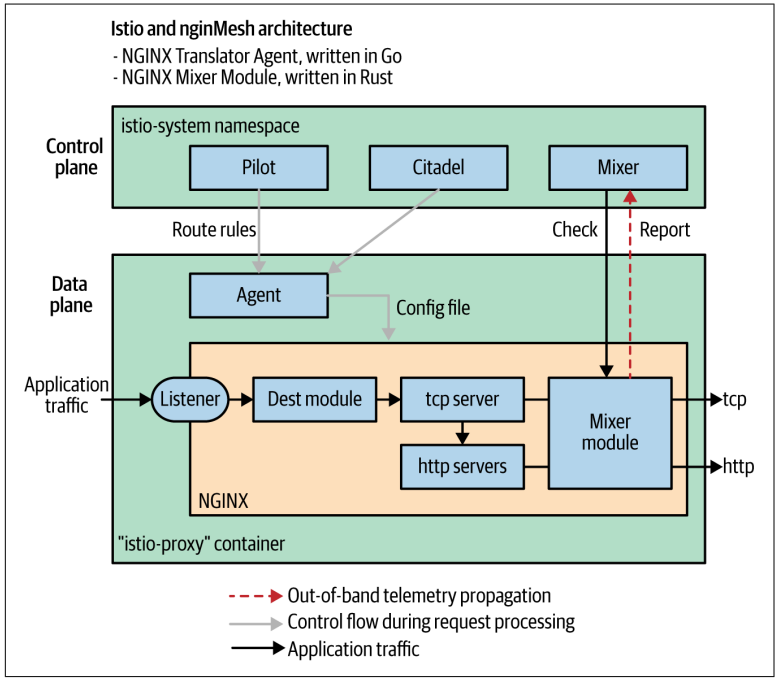


Figure 4-2. Example of swapping proxies—Istio and nginMesh.

Extensible Adapters

Management planes and control planes are responsible for enforcing access control, usage, and other policies across the service mesh. In order to do so, they collect telemetry data from service proxies. Some service meshes gather telemetry as shown in Figure 4-3. The service mesh control plane uses one or more telemetry adapters to collect and pass along these signals. The control plane will use multiple adapters either for different types of telemetry—races, logs, metrics—or for transmitting telemetry to external monitoring providers.

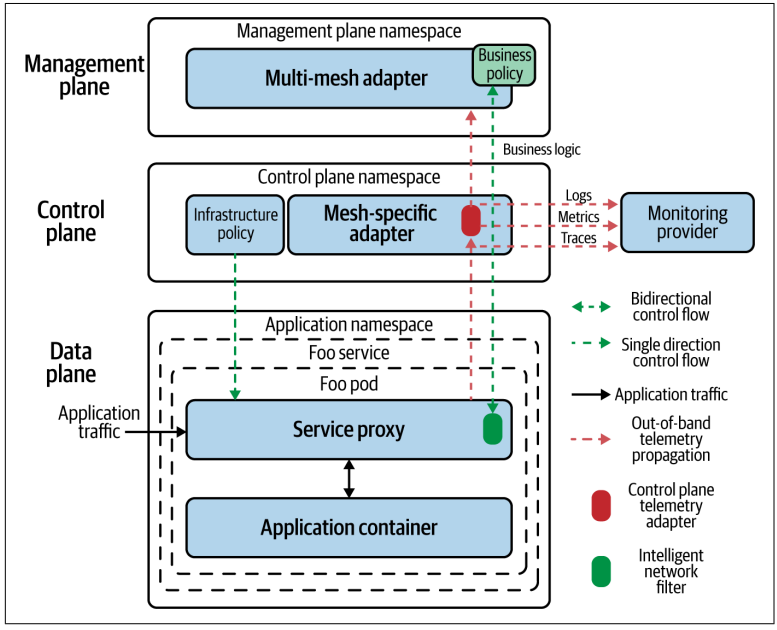


Figure 4-3. An example of a data plane proxy generating and directly sending telemetry to the control plane. Control plane adapters are points of extensibility, acting as an attribute processing engine, collecting, transforming, and transmitting telemetry.

Some service meshes gather telemetry through service mesh-specific network filters residing in the service proxies in the data plane, as shown in Figure 4-4. Here, we also see that management planes are capable of receiving telemetry and sending control signals to affect the behavior of the application at a business-logic level. The service

mesh control plane uses one or more telemetry adapters to collect and pass along these signals. The control plane will use multiple adapters either for different types of telemetry—traces, logs, or metrics, or for transmitting telemetry to external monitoring providers.

Later versions of Istio are shifting this model of adapter extensibility out of the control plane and into the data plane, as shown in Figure 4-4.

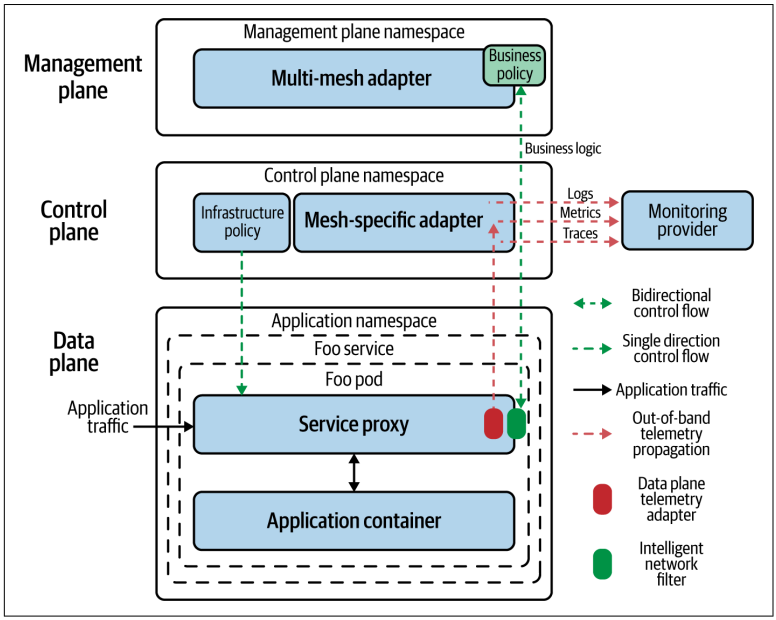


Figure 4-4. Data plane performing the heavy lifting to ensure efficient processing of packets and telemetry generation. "Mixerless Istio" is an example of this model.

The default model of telemetry generation in Istio 1.5 and onward is done through Istio proxy filters, which are compiled to WebAssembly (WASM) modules and shipped with Istio proxy. This architectural shift was primarily done for performance reasons. Performance management is a recurring theme in consideration of deploying and managing service meshes.

The Performance of the Data Plane

Considering the possibilities of what can be achieved when the intelligence of the management plane and the power of the data plane are combined, service meshes pack quite a punch in terms of facilitating configuration management, central identity, telemetry collection, traffic policy, application infrastructure logic, and so on. With this understanding, consider that the more value you try to derive from a service mesh, the more work you will ask it to do. The more work a service mesh does, the more its efficiency becomes a concern. While benefiting from a service mesh's features, you may ponder the question of what overhead your service mesh is incurring. This is one of the most common questions service mesh users have.

The Importance of Service Mesh Performance (SMP)

Answering this question is anything but simple. There are so many variables involved, and so many of those are specific to any given environment. So how do you know if you've struck the right balance between all that you're having your service mesh do, the time you save, the consistency and quality of traffic management you receive, and the cost of doing so? For every load-balancing decision made, for every rate limit enforced, for every A/B test performed, what is the incremental overhead incurred, and what is the incremental operational savings gained? The Service Mesh Performance specification enables its adopters to answer those very questions.

Carefully consider the quantifiable performance aspects of your service mesh in context of the value you are deriving from it. If you were to acquire this same functionality outside of a service mesh, what would it "cost" you in terms of time to implement and operational efficiency of not having a single point of control for all that the service mesh provides?

Conclusion

The NGINX Service Mesh and Citrix Service Mesh draw much interest as many organizations have broad and deep operational expertise built around these battle-tested proxies.

Service mesh management and control and data planes are highly extensible and customizable to your environment. Data plane intelligence can be extended by executing traffic filters (modules) using technologies like Lua and WebAssembly.

Do not underestimate the importance of data plane intelligence. More and more, developers can shed application logic responsibilities to the data plane, which means that they can deliver on core business logic more readily, and it means that operators and service/product owners are empowered with more control over application logic by virtue of simply configuring the service mesh.

CHAPTER 5
Conclusion

In many respects, the advent of cloud native is marked by the introduction of Docker. The popularization of containers brought the need for orchestrators. In turn, containers and orchestrators, coupled with the popularity of microservice architecture for their speed of development, smaller surface area to reason over, and decoupling of development teams, lead to service sprawl. The ability to run a number of distributed services brought the need for empowering developers, operators, and service owners with a service mesh.

Service meshes are one layer of your infrastructure and don't bring all that you need. They do give you the ability to bridge the divide between your infrastructure and your application in ways most people have not previously seen, however. Service meshes and where they layer into the infrastructure makes them ripe with possibility beyond what they are being used for today. We will see growth in the following areas of service mesh usage and see how they change how developers write applications and how product owners manage their services:

- Refine developer experiences by:
  - Offering *distributed debugging*.
  - Allowing developers to skip past building in many user/tenancy segregation considerations and instead rely on the service mesh to provide enforcement based on configuration, not application code.

- Provide compelling *topology and dependency graphs* that allow you to not only visualize the service mesh and its workloads but design them as well.
- 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 deployment pipelines of many software projects.
  - Automatic API documentation, perhaps, integrating with toolkits like Swagger or ReadMe.io.
  - API function and interface discovery.
- Participate in *service and product management* by enabling service owners to offload any number of examples of application logic to the service mesh, allowing developers to hand off "application infrastructure logic" to the layer just below the application—Layer 5:
  - Perform A/B testing directly with service users without the need for developer or operator intervention.
  - Control service pricing based on the accounting of service request traffic in the context that the tenant/user is making these requests.
  - Forgo requesting specific application logic changes from the development team and instead deploy a network traffic filter to reason over and control user and service behavior.
- *Deeper observability* to move beyond distributed tracing alone and into full application performance monitoring leverage deep packet inspection for business layer insights.
- *Multitenancy* to allow multiple control planes running on the same platform.
- *Multicluster* and *cross-cluster* such that each certificate authority shares the same root certificate and workloads can authenticate each other across clusters within the same mesh.
- Cross-mesh to allow interoperability between heterogeneous types of service meshes.
- Improve on the integration of performance management tools like Meshery 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 loads?

- 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.

Adopting a Service Mesh

There are many service meshes to choose from, 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? More and more, the answer is "yes." Service meshes are quickly becoming a ubiquitous layer in modern infrastructures. Table 5-1 shows the factors to consider.

Table 5-1. Factors in considering how strongly you need a service mesh

Concern	Begin considering a service mesh	Strongly consider a service mesh	Consider that..
Service communication	Low interservice communication.	High interservice communication.	The higher the volume of requests to internal and external services, the more insight and control you will need and the higher the return on investment your service mesh will deliver.
Observability	Edge focus: metrics and usage are for response time to clients and request failure rates.	Uniform and ubiquitous: observability is key for understanding service behavior.	You can bring much insight immediately with little effort.

Concern	Begin considering a service mesh	Strongly consider a service mesh	Consider that..
Perspective from which you think of your APIs	Strong separation of external and internal users. Focused on external API experience. APIs are used primarily for client-facing interaction. APIs are for clients only.	Equal and undifferentiated experience for internal and external users. APIs are treated as a product; APIs are how your application exposes its capabilities.	Service meshes are infusing API gateway and API management functionality. Deploy a mesh early to have your architecture ready for scaling.
Security model	Security at perimeter. Subnet zoning (firewalling). Trusted internal networks.	Zero-trust mindset. authN and authZ between all services. Encryption between all services.	The security characteristics are desirable qualities of any deployment. Apply defense-in-depth. Why not pull these into a single layer of control?
# of services	A couple of services.	A few services or more.	Deploy a service mesh early. Doing so lowers risk and affords you time to become confident with the operations of a service mesh.
Service reliability	Either don't need or are willing to hand code or bring in other infrastructure to provide resiliency guarantees.	Need strong controls over the resiliency properties of your services and to granularly control ingress, between, and egress service request traffic.	Resilient infrastructure and highly available services are ideal in any environment. Let the service mesh do the heavy lifting for you.

I recommend starting small, with the lowest-risk deployment model. I consider the lowest-risk deployment model to be one focused on observability, primarily because gathering additional telemetric signals is more about observing service and system behavior than it is about augmenting them. 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 Meshery to identify antipatterns and also to analyze the state and configuration of your service mesh against known best practices.
- Inspect service request headers with your service mesh and annotate when requests fail to help you identify whether the failure is in your workloads or the service mesh.

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

```
$ linkerd 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 the 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. Don't leave value on the service mesh table when you can expect more from your infrastructure. 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.

What if you only have one service? A lightweight service mesh with an ingress gateway and a service proxy can provide you with a high degree of control that you otherwise will have to deploy separate infrastructure for. Consider how likely it is that your deployment will grow beyond a single service. Having a service mesh present from the start means that you can avoid embedded infrastructure logic into your application code from the very beginning. It also means that you don't have to tear down other temporary infrastructure during your deployment in the interim until you arrive at a

more painful inflection point when you inevitably deploy a service mesh.

Generally speaking, my advice is to deploy a service mesh even in environments with only a couple of services and with only a few engineers and whether you are working on a homogenous application stack or not. Especially deploy a service mesh if you are not confident in the observability, not confident in the reliability, and not confident in the security of your application. Understand that the consideration of the size of your organization is orthogonal to the consideration of your need for a service mesh. Whether you run a service mesh yourself or seek a managed service solution is a consideration more heavily weighted by the size of your organization. Expertise and experience with successfully adopting open source infrastructure projects is another example of an orthogonal consideration as to whether you and your workloads will benefit from a service mesh. Again, this is more of a factor of whether to deploy and run a service mesh yourself or seek assistance from a vendor.

The fifth layer of distributed systems is here. There are plenty of options to choose from. It's time to expect more from your infrastructure. Take the third step in your cloud-native journey and deploy a service mesh. The untapped power of the service mesh might surprise you.

About the Author

Lee Calcote is the founder and CEO of Layer5, where the community helps organizations harness the value of service meshes as a maintainer of Meshery, Service Mesh Performance (SMP), and Service Mesh Interface (SMI). Previously, Calcote stewarded technology strategy and innovation across SolarWinds as head of CTO technology initiatives. He led software-defined data center engineering at Seagate, delivering predictive analytics and modern systems management. Calcote held various leadership positions at Cisco, where he created Cisco's cloud management platform and pioneered software-defined network orchestration and autonomic remote management services.

In addition to his role at Layer5, Calcote serves in various industry bodies chairing the Cloud Native Computing Foundation (CNCF) SIG Network, and formerly, in the Distributed Management Task Foundation (DMTF), delivering Redfish 1.0, and in the Center for Internet Security (CIS), delivering the Docker Benchmark 1.0.

He serves on Cisco's advisory board, and formerly advised startups Twistlock and Octarine, acquired by Palo Alto Networks and VMware, respectively. As a Docker Captain and Cloud Native Ambassador, he is a frequent speaker in the cloud native ecosystem. Calcote is the coauthor of *Istio: Up and Running* and the forthcoming *Service Mesh Patterns* (both O'Reilly), in addition to titles with other publishers. He holds a bachelor's degree in computer science, a master's degree in business administration, and retains a list of industry certifications.