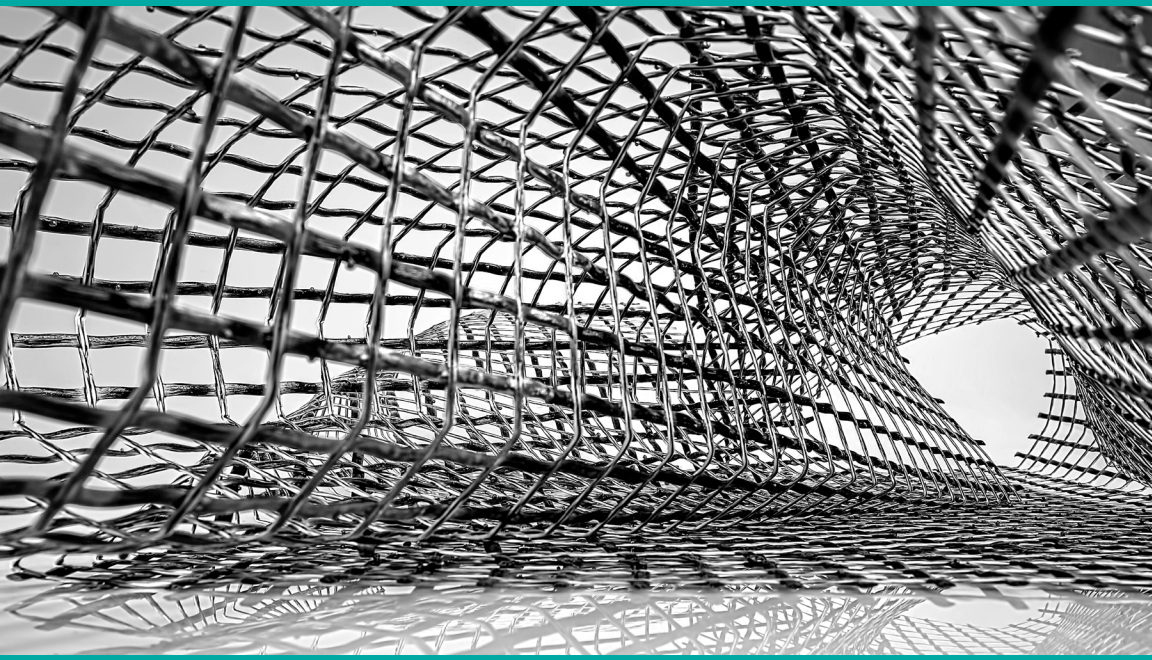
O'REILLY®

Compliments of BUOYANT

The Service Mesh

Resilient Service-to-Service Communication for Cloud Native Applications



George Miranda

The Service Mesh

Resilient Service-to-Service Communication for Cloud Native Applications

George Miranda

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

The Service Mesh
by George Miranda

Printed in the United States of America.

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

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

Acquisitions Editor: Nikki McDonald
Development Editor: Virginia Wilson
Production Editor: Melanie Yarbrough
Copyeditor: Octal Publishing Services
Proofreader: Sonia Saruba
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2018: First Edition

Revision History for the First Edition
2018-06-08: First Release

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

978-1-492-03129-1

[LSI]

Table of Contents

Preface

What Is a Service Mesh?

A service mesh is a dedicated infrastructure layer for handling service-to-service communication in order to make it visible, manageable, and controlled. The exact details of its architecture vary between implementations, but generally speaking, every service mesh is implemented as a series (or a "mesh") of interconnected network proxies designed to better manage service traffic.

If you're unfamiliar with the service mesh in general, a few in-depth primers can help jumpstart your introduction, including Phil Calçado's history of the service mesh pattern, Redmonk's hot take on the problem space, and (if you're more the podcast type) The Cloudcast's introductions to both Linkerd and Istio. Collectively, these paint a good picture.

Who This Book Is For

This book is primarily intended for anyone who manages a production application stack: developers, operators, DevOps practitioners, infrastructure/platform engineers, information security officers, or anyone otherwise responsible for supporting a production application stack. You'll find this book particularly useful if you're currently managing or plan to manage applications based in microservice architectures.

What You'll Learn in This Book

If you've been following the service mesh ecosystem, you probably know that it had a very big year in 2017. First, it's now an ecosystem! Linkerd crossed the threshold of serving more than one trillion service requests, Istio is now on a monthly release cadence, NGINX launched its nginMesh project, Envoy proxy is now hosted by the CNCF, and the new Conduit service mesh launched in December.

Second, that surge validates the "service mesh" solution as a necessary building block when composing production-grade microservices. Buoyant created the first publicly available service mesh, Linkerd (pronounced "Linker-dee"). Buoyant also coined the term "service mesh" to describe that new category of solutions and has been supporting service mesh users in production for almost two years. That approach has been deemed so necessary that 2018 has been called "the year of the service mesh". I couldn't agree more and am encouraged to see the service mesh gain adoption.

As such, this book introduces readers to the problems a service mesh was created to solve. It will help you understand what a service mesh is, how to determine whether you're ready for one, and equip you with questions to ask when establishing which service mesh is right for your environment. This book will walk you through the common features provided by a service mesh from a conceptual level so that you might better understand why they exist and how they can help support your production applications. Because I work for Buoyant (a vendor in this space), in this book I've intentionally focused on broader general context for the service mesh rather than on product-specific side-by-side feature comparisons.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Many thanks to Chris Devers, Lee Calcote, Michael Ducy, and Nathen Harvey for technical review and help with presentation of this material. Thanks to the wonderful staff at O'Reilly for making me seem like a better writer. And special thanks to William Morgan and Phil Calçado for their infinite patience and guidance onboarding me into the world of service mesh technology.

The Service Mesh

Basic Architecture

Every service mesh solution should have two distinct components that behave somewhat differently: a *data plane* and a *control plane*. Figure 1-1 presents the basic architecture.

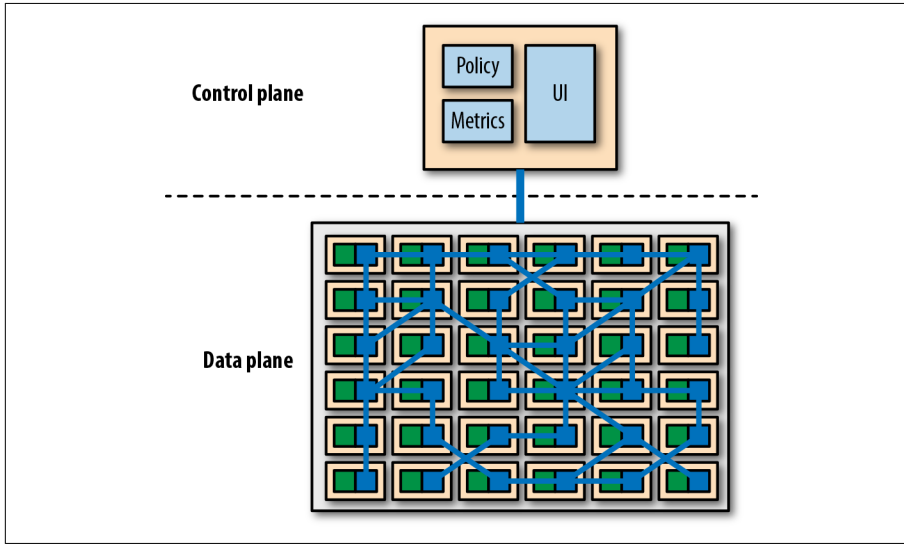


Figure 1-1. Basic service mesh architecture

The data plane is the layer responsible for moving your data (e.g., service requests) through your service topology in real time. Because this layer is implemented as a series of interconnected proxies, when your applications make remote service calls, they're typically unaware of the data plane's existence. Generally, no changes to your application code should be required in order to use most of the features of a service mesh. These proxies are more or less transparent to your applications. The proxies can be deployed several ways (one per physical

host, per group of containers, per container, etc.). But they're commonly deployed as one per communication endpoint. Just how "transparent" the communication is depends on the specific endpoint type you choose.

A service mesh should also have a control plane. When you (as a human) interact with a service mesh, you most likely interact with the control plane. A control plane exposes new primitives you can use to alter how your services communicate. You use the new primitives to compose some form of policy: routing decisions, authorization, rate limits, and so on. When that policy is ready for use, the data plane can reference that new policy and alter its behavior accordingly. Because the control plane is an abstraction layer for management, it's theoretically possible to not use one. You'll see why that approach could be less desirable later when we explore the features of currently available products.

That's enough to get started. Next, let's look at the problems that necessitate a service mesh.

The Problem

This section explores recurrent problems that developers and operators face when supporting distributed applications in production. These problems are highlighted by recent technology shifts.

There's a new breed of communication introduced by the shift to microservice architectures. Unfortunately, it's often introduced without much forethought by its adopters. This is sometimes referred to as the difference between the north-south versus east-west traffic pattern. Put simply, north-south traffic is server-to-client traffic, whereas east-west is server-to-server traffic. The naming convention is related to diagrams that "map" network traffic, which typically draw vertical lines for server-client traffic, and horizontal lines for server-to-server traffic. There are different considerations for managing server-to-server networks. Different considerations for the network and transport layers (L3/L4) aside, there's a critical difference happening in the session layer.

In most cases, monolithic applications are deployed in the same runtime along with all other services (e.g., a cluster of application servers). The applications initially deployed to that runtime are all contained in one cohesive unit. As applications evolve, they have a tendency to accumulate new functions and features. Over time, that glob of functions piled into the same app turns it into a monumental pillar that can become very difficult to manage.

One key value in the popularity of composing microservices is avoiding that management trap. New features and functions are instead introduced as new independent services that are no longer a part of the same cohesive unit. That's a very useful innovation. But it also means learning how to successfully create dis-

tributed applications. There are common mistaken assumptions that surface when programming distributed applications.

The Fallacies of Distributed Computing

The fallacies of distributed computing are a set of principles that outline the mistaken assumptions that programmers new to distributed applications invariably make.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

The architectural shift to microservices now means that service-to-service communication becomes the fundamental determining factor for how your applications will behave at runtime. Remote procedure calls now determine the success or failure of complex decision trees that reflect the needs of your business. Is your network robust enough to handle that responsibility in this new distributed world? Have you accounted for the reality of programming for distributed systems?

The service mesh exists to address these concerns and decouple the management of distributed systems from the logic in your application code.

A Pragmatic Problem Example

As a former system administrator, I tend to glom onto situations that require me to think about how I would troubleshoot things in production. To illustrate how the problem plays out in production, let's begin with a resonant problem: the challenge of visibility.

Measuring the health of service communication requests at any given time is a difficult challenge. Monitoring network performance statistics can tell you a lot about what's happening in the lower-level network layer (L3/L4): packet loss, transmission failures, bandwidth utilization, and so on. That's important data, but it's difficult to infer anything about service communications from those low-level metrics.

Directly monitoring the health of service-to-service requests means looking further up the stack, perhaps by using external latency monitoring tools like smokeping or by using in-band tools like tcpdump. Although either option provides either too much or too little helpful information, you can use them in tandem with another monitoring source (like an event-stream log) to triage and correlate the source of errors if something goes wrong.

For a majority of us who've managed production applications, these tools and tactics have mostly been good enough; investing time to create more elegant solutions to unearth what's happening in that hidden layer simply hasn't been worth it.

Until microservices.

When you start building out microservices, a new breed of communication with critical impact on runtime functionality is introduced and complexity is distributed. For example, when decomposing a previously monolithic application into microservices, that typically means that a three-tier architecture (presentation layer, application layer, and data layer) now becomes dozens or even hundreds of distributed microservices. Those services are often managed by different teams, working on different schedules, with different styles, and with different priorities. This means that when running in production, it's not always clear where requests are coming from and going to or even what the relationship is between the various components of your applications.

Some development teams solve for that blind spot by building and embedding custom monitoring agents, control logic, and debugging tools into their service as communication libraries. And then they embed those into another service, and another, and another (Jason McGee summarizes this pattern well).

The service mesh provides the logic to monitor, manage, and control service requests by default, everywhere. It pushes that logic into a lower part of the stack where you can more easily manage it across your entire infrastructure.

The service mesh doesn't exist to manage parts of your stack that already have sufficient controls, like packet transport and routing at the TCP/IP level. The service mesh presumes that a useable (even if unreliable) network already exists. The scope of the service mesh should be only to provide a solution that solves for the common challenges of managing service-to-service communication in production. Some products might begin to creep out of the session layer and into lower parts of the network stack. Because there are existing (nonservice mesh) solutions that manage those parts of the stack sufficiently, for the purposes of this book, when I talk about a "service mesh," I'm speaking only of the new functionality specifically geared for solving distributed service-to-service communication.

Creating a Reliable Application Runtime

To be sufficient for production applications, service communication for distributed applications must be resilient and secure. The management of the properties required to make the runtime visible, resilient, and secure should not be managed inside of your individual applications.

Historically, before the service mesh, any logic used to improve service communication had to be written into your application code by developers: open a socket, transmit data, retry if it fails, close the socket when you're done, and so on. The burden of programming distributed applications was placed directly on the shoulders of each developer, and the logic to do so was tightly coupled into every distributed application as a result.

To solve this in a developer-friendly way, network resiliency libraries were born. Simply include this library in your application code and let it handle the logic for you. It's worth noting that the service mesh is a direct descendant of the Finagle network library open-sourced by Twitter. In its earlier days, Twitter's need to massively scale its platform led down a path of engineering decisions that made it (along with other web-scale giants of the time) an early pioneer of microservice architectures in a pre-Docker world. To deal with the challenge of managing distributed services in production at scale, Finagle was developed as a management library that could be included in all Twitter services (presumably meaning that a service mesh should measure outages in units of fail whales). A description of the problems that led up to its creation is well covered in William Morgan's talk "The Service Mesh: Past, Present, and Future". In short, Finagle's aim was to make service-to-service communication (the fundamental factor determining how applications now ran in production) manageable, monitored, and controlled. But the network library approach still left that logic very much entangled with your application code.

The architecture of the service mesh provides an opportunity to create a reliable distributed application runtime but in a way that is instead entirely decoupled from your applications. The two most common ways of setting up a service mesh (today) are to either deploy one proxy on each container host or to deploy each proxy as a container sidecar. Then, whenever your containerized applications make external service requests, they route through the new proxy. Because that proxy layer now intercepts every bit of network traffic flowing between production services, it can (and should) take on the burden of ensuring a reliable runtime and relieve developers of codifying that responsibility.

To decouple that dependency, the service mesh abstracts that logic and exposes primitives to control service behavior on an infrastructure level. From a code perspective, now all your apps need to do is make a simple remote procedure call. The logic required to make those calls robust happens further down the stack.

That change allows you to more easily manage how communications occur on a global (or partial) infrastructure level.

For example, the service mesh can simplify how you manage Transport Layer Security (TLS) certificates. Rather than baking those certificates into every microservice application code base, you can handle that logic in the service mesh layer. Code all of your apps to make a plain HTTP call to external services. At the service mesh layer, you specify the certificate and encryption method to use when that call is transmitted over the wire, and manage any exceptions on a per-service basis. Whenever you inevitably need to update certificates, you handle that at the service mesh layer without needing to change any code or redeploy your apps.

The service mesh can both simplify your application code and provide more granular control. You push management of all service requests down into an organization-wide set of intermediary proxies (or a "mesh") that inherit a common behavior from a common management interface. The service mesh exists to make the runtime for distributed applications visible, manageable, and controlled.

Are You Ready for a Service Mesh?

If you're asking yourself whether you need a service mesh, the first sign that you do need one is that you have a lot of services intercommunicating within your infrastructure. The second is that you have no direct way of determining the health of that intercommunication, managing its resiliency, or managing it securely. Without a service mesh, you could have services failing right now and not even know it. The service mesh works for managing all service communication, but its value is particularly strong in the world of managing cloud-native applications given their distributed nature.

Observability

In distributed applications, it's critical to understand the traffic flow that now defines your application's behavior at runtime. It's not always clear where requests are coming from or where they're going. When your services aren't behaving as expected, troubleshooting the cause shouldn't be an exercise in triaging observations from multiple sources and sleuthing your way to resolution. What we need in production are tools that reduce cognitive burden, not increase it.

> An observable system is one that exposes enough data about itself so that generating information (finding answers to questions yet to be formulated) and easily accessing this information becomes simple.
>
> —Cindy Sridharan

Let's examine how the service mesh helps you to create an observable system.

Because this is a relatively new category of solutions—all using the same "service mesh" label—with a sudden surge of interest, there can be some confusion around where and how things are implemented. There is no universal "service mesh" specification (nor am I suggesting that there should be), but we can at least nail down basic architectural patterns so that we can reach some common understandings.

First, let's examine how its components come together so that we can better understand where and how observability works in the service mesh.

How the Data and Control Planes Interact

A full-featured service mesh should have both a proxying layer where communication is managed (i.e., a data plane) and a layer where humans can dictate management policy (i.e., a control plane). To create that cohesive experience, some implementations use separate products in those layers. For example, Istio (a control plane) pairs with Envoy (a data plane) by default. Envoy is sometimes called a service mesh, although the project is a self-described "universal data plane." Envoy does offer a robust set of APIs on top of which users could build their own control plane or use other third-party add-ons such as Houston by Turbine Labs.

Some service mesh implementations contain both a data plane and a control plane using the same product. For example, Linkerd contains both its proxying components (linkerd) and namerd (a control plane) packaged together simply as "Linkerd." To make things even more confusing, you can do things like use the Linkerd proxy (data plane) with the Istio mixer (control plane).

There are different combinations of products that you can make work together as a service mesh, and committing to a specific number would likely make this book stale by publishing time. Succinctly, the takeaway is that every service mesh solution needs both a data plane and a control plane.

Where Observability Constructs Are Introduced

The data plane isn't just where the packets that comprise service-to-service communication are exchanged, it's also where telemetry data around that exchange is gathered. A service mesh gathers descriptive data about what it's doing at the wire level and makes those stats available. Exactly which data is gathered varies between proxying implementations, and the precise set of metrics that matter to an organization varies. But your organization should care about certain "top-line" service metrics that most profoundly affect the business. It's important to collect a significant number of bottom-line metrics to triage events, but what you want surfaced are the metrics that tell you something you care about is wrong right now.

For example, a bottom-line metric might be something like CPU or memory usage. If there's an outage that occurs or an anomaly in the system, it helps to be able to correlate that with those types of resource consumption patterns. But just because CPU usage is temporarily abnormal, that doesn't mean you want to be woken up about it at 4 A.M. What you *do want* to be woken up for are things like a massive drop in service request success rates. That's a real failure having a real impact.

Some metrics are useful for debugging. Others are useful for proactively predicting system failures or triggering alerts when failures occur. Observability is a broad topic and this is only the tip of the iceberg. But in the context of a service mesh, things to look for are how well a solution exposes things like latency, request volume, response times, success/failure counts, retry counts, common error types, load balancing statistics, and more. To be most useful, a service mesh should not only contain that data, it should surface presentation of significant changes to those top-line metrics so that the data can be processed and you can take action.

With a service mesh, external metrics-collection utilities can directly poll the data plane for aggregation. In some solutions, the control plane might act as an intermediary ingestion point by aggregating and processing that data before sending it to backends like Prometheus, InfluxDB, or statsd. Some contributors are also creating custom adapters to pipe out that data to a number of sources, for example, the SolarWinds adapter. That data then can be presented in any number of ways, including the popular choice of displaying it visually via dashboards.

Where You See the Data

Dashboards help humans visualize trends when troubleshooting by presenting aggregated data in easily digestible ways. In a service mesh, helpful dashboards are often included as a component somewhere in the product set. That inclusion can sometimes be confusing to new users. Where do dashboard components fit into service mesh architecture?

For reference, let's look at how a couple of different service mesh options handle dashboards. Envoy is a data plane and it supports using Grafana. Istio is a control plane and it supports using Grafana. And Linkerd, which contains both a data plane and a control plane, also supports using Grafana. That doesn't help make things any clearer. Are dashboards part of the data plane or the control plane?

The truth is, they're not strictly a part of either. When you (as a human) interact with a service mesh, you typically interact with the control plane. So, it often makes sense to bolt on dashboards to the place where humans already are. But dashboards aren't a requirement in the service mesh.

We see that in practice using the earlier examples. Envoy presumes that you already have your own metrics-collection backend set up elsewhere; Istio includes that backend as the Istio dashboard add-on component; Linkerd provides that with the linkerd-viz add-on component; and Conduit bundles them in by default.

Any dashboard, no matter where it's implemented, is reading data that was observed in the data plane. That's where observations occur, even if you notice the results somewhere else.

Beyond Service Metrics with Tracing

Beyond service health metrics, distributed tracing in the service mesh provides another useful layer of visibility. Distributed tracing is a component that you can implement separately, but a service mesh typically integrates its use.

Figure 1-2 shows what a typical service request might look like in a distributed system. If a request to any of the underlying services fails, the issuing client knows only that its request to the profile service failed, but not where in the dependency tree or why. External monitoring exposes only overall response time and (maybe) retries, but not individual internal operations. Those operations might be scattered across numerous logs, but a user interacting with the system might not even know where to look. As Figure 1-2 shows, if there's an intermittent problem with the audit service, there's no easy way to tie that back to failures seen via the profile service unless an engineer has intrinsic knowledge of how the entire service tree operates. Then, the issue still requires triaging separate data sources to determine the source of any transient issues.

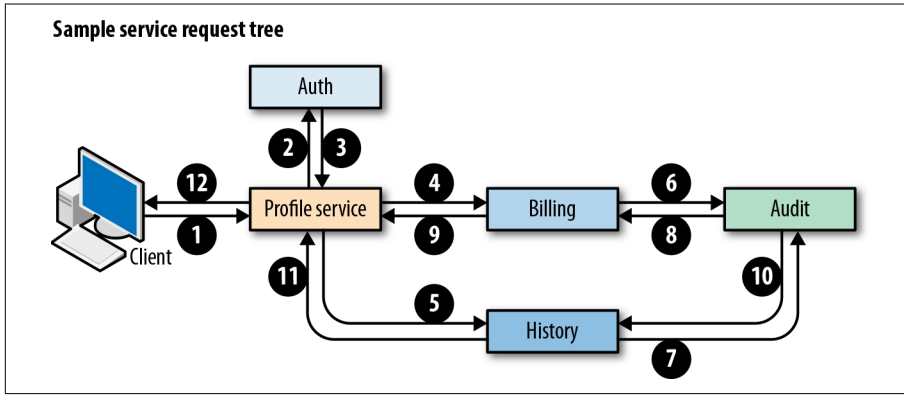


Figure 1-2. Sample service request tree

Distributed tracing helps developers and operators understand the behavior of service requests and their dependencies in microservice architectures. In the service mesh, requests routed by the data plane can be configured to trace every step

(or "span") they take when attempting to fulfill successfully. Because the service mesh handles all service traffic, it's in the right layer to observe all requests and report back everything that happened in each span to help assemble a full trace of what occurred.

Figure 1-2 shows how these various services fit together. But it doesn't show time durations, parallelism, or dependent relationships. There's also no way to easily show latency or other aspects of timing. A full trace allows you to instead visualize every step required to fulfill a service request by correlating them in a manner like that shown in Figure 1-3.

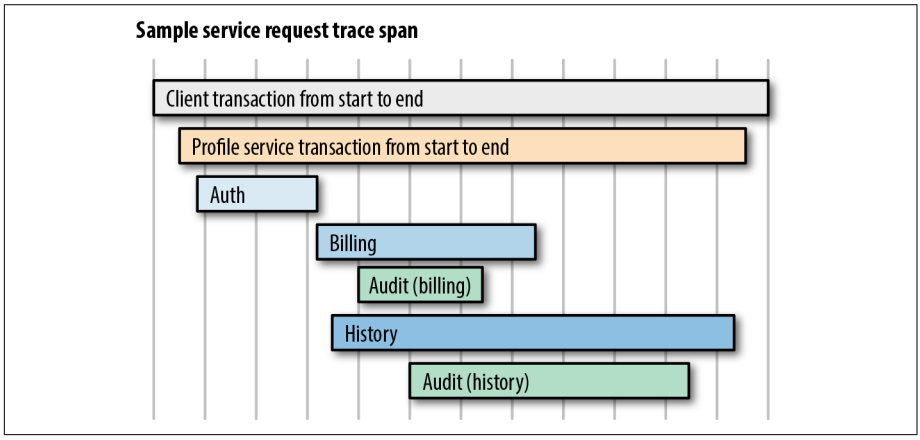


Figure 1-3. Sample service request trace span

Each span corresponds to a service call invoked during the execution of the originating request. Because the service mesh data plane is proxying the calls to each underlying service, it's already gathering data about each individual span like source, destination, latency, and response code.

The service mesh is in a position to easily provide instrumentation for and produce richer data about the individual spans. Combined with another system like Zipkin or Jaeger, you can combine and assemble this data into a full trace to provide more complete observability of a distributed system. The specifics of how that trace is assembled depends on the underlying implementation, which varies between service mesh products. However, the net effect is that without prerequisite knowledge of the system, any developer or operator can more easily understand the dependencies of any given service call and determine the exact source of any issues presented.

Although the hooks exist to capture this data, you should note that application code changes are (currently) required in order to use this functionality. Tracing works only with HTTP protocols. Your apps need to propagate and forward the required HTTP headers so that when the data plane sends span information to

the underlying telemetry, the spans can be reassembled and correlated back into a contiguous single trace.

Visibility by Default

Just by deploying a service mesh to your infrastructure, you should realize immediate out-of-the-box visibility into service health metrics without any application code changes required. You can achieve more detailed granularity to see otherwise obscured steps performed by each request by making the header modifications required to use distributed tracing. Most service mesh products give you the option to take those metrics and plug them into some sort of external data processing platform, or you can use their bundled dashboards as a start.

Resiliency

Managing applications in production is complicated. Especially in a cloud-native world, your applications are built on fundamentally unreliable systems. When the underlying infrastructure breaks (as it inevitably does) your entire application may or may not survive. Faster recovery times and isolated failures are a start. But depending on where in your architecture those small isolated failures occur—for example, a critical service relied upon by hundreds of smaller services—they can quickly escalate into cascading global failures if handled improperly.

Building resilient services is a complex topic with many different consideration vectors. For the purposes of this book, we're not going to cover what happens at the infrastructure layer or the containerized application layer. An entire field of practice is devoted to making containerized infrastructure robust and resilient with management platforms like Kubernetes, DC/OS, or others. To examine the service mesh, we'll focus on the service-to-service communication layer.

As covered in "Beyond Service Metrics with Tracing" on page 9, dependencies between distributed services can introduce complexity because it's not always clear where requests are coming from or where they're going to. These dependent relationships between services can introduce fragility that needs to be understood and managed.

If a service with many underlying dependencies is updated, do all of the dependent services need to be updated? In a true service-oriented architecture, it shouldn't matter. But in practice, it's not uncommon to run into practical questions and challenges when running in production. Can multiple versions of the same service run in parallel and, if so, how can you control which applications use which version of the service? Can you stage proposed new changes to a service and route only certain segments of traffic to test functionality, or do you need to deploy straight to production and hope for the best?

These are common challenges in a cloud-native world. The service mesh is built to help address these types of common situations. The next logical step after adding a layer of observability where one didn't previously exist is to also insert and expose a number of primitives to help developers and operators build more resilient applications at the service communication layer by dealing with failures gracefully.

Let's examine what common service mesh features can do to create resiliency.

Managing Failed Requests Gracefully

Unlike the previous generation of network management tools, the service mesh is hyper-focused on improving the production-level quality of remote procedure calls (RPCs). Therefore, it is specifically built to more closely examine and use session data like request status codes. You can configure the service mesh to recognize whether a particular type of request is idempotent, where it should be sent for fulfillment, how long it should be retried, or whether it should be throttled to prevent system-wide failures. Generally speaking, features for a service mesh include things like timeouts, retries, deadlines, load balancing, and circuit breaking. Although the implementation details of those features vary between products, we can at least cover the core concepts behind them.

Timeouts help you to predict service behavior. By setting the maximum allowed time before a service request is considered failed, you can take reliable action when system performance is degraded. It's worth noting that you should consider the maximum timeout values for both parent and child requests. A bottleneck in several child services could easily exceed aggressively set timeout values. Typically, you can manage timeouts both globally and on a per-request basis.

The service mesh is primarily written with RPC protocols in mind, although all TCP connections can be passed through most available options. Because the service mesh operates at the session layer, management of RPC protocols includes the ability to examine response codes to determine outcomes. If the service mesh recognizes a request as idempotent, you can configure it to safely and automatically retry failed calls. The settings are what you'd expect: which calls should be retried and for how long? However, some retries also can be configured for things like "jitter" settings, or time delays between retries designed to smooth out spikes caused by transient failures and avoid overloading services.

Transient failures in distributed systems can quickly escalate into cascading failures. If a momentary blip occurs and fails to resolve within several seconds, you could end up in a situation in which several dependent services queue retries while they wait for the failure to resolve. That retry queue ties up system resources while it works to resolve itself. If a service falls into a lengthy retry loop, the resource demand required to resolve the queue could be great enough to also cause it to fail. That secondary failure then causes other dependent services to fall

into lengthy retry loops, also causing tertiary failures, then another failure, then another, and so on.

A handy way to mitigate lengthy retry loops is to set request deadlines. Deadlines are maximum allotted time windows for a request and its multiple retries to complete. If the time window has expired and no response was received, it's no longer considered useful to receive a successful response. Regardless of allowable retries remaining, the request and its entire operation are failed.

A more sophisticated way of managing resource constraints in lengthy retry loops is to use a "retry budget." A retry budget is expressed as a percentage of requests that can be retried during a particular time window. For example, suppose that your retry budget is set to 25% of requests within a 2-second window. If 200 requests were issued in the last 2 seconds, only 50 requests (maximum) will be allowed to issue a retry, whereas the 150 others instead receive a hard failure. Although it's ideal for 100% of all requests to always succeed, the pragmatic approach for some environments might be to degrade performance to ensure overall system stability. Retry budgets exist to ensure that all calling services receive a response (whether success or failure) within a predictable timeframe. Setting a predictable timeframe can also provide for a better user experience by allowing you to set up workflows that force and respond to quick failures rather than waiting for prolonged slow failures.

Circuit breaking is another construct that exists to isolate service failures predictably by preventing client connections to known failed instances. Following electrical principles, a "circuit" is considered closed when traffic flows through it and open when traffic is stopped. A healthy service has a circuit that is closed by default. In the service mesh, unhealthy services can be detected at both the connection and the request level. When a service is deemed unhealthy, the circuit is flipped to open (or "broken") to stop further requests from even being issued. As seen in the earlier examples, managing failures consumes system resources. Circuit breaking minimizes the amount of time spent routing requests to failed services. Any requests attempting to call a broken circuit instantly receive a hard failure response. Later, when the failed service is deemed as once again healthy, the circuit is closed and connections resume as normal.

Load Balancing and Distributing Requests

Any scalable system of distributed services requires some form of load balancing. Load balancing exists to distribute traffic intelligently across numerous dynamic endpoints to keep the overall system healthy, even when those endpoints have degraded performance or fail entirely. Hardware load balancing most frequently manages traffic at Layers 3 and 4 (though some also manage session traffic). Load balancing at the session layer is more commonly managed with software. For

managing service communication, a service mesh can offer some advantages over other load-balancing options.

Service mesh load balancers offer the types of routing options you'd expect in a software load balancer: round robin, random, or weighted. They also observe network heuristics and route requests to the most performant instances. But unlike other load balancers, those for the service mesh operate at the RPC layer. Rather than observing heuristics like LRU or TCP activity, they can instead measure queue sizes and observe RPC latencies to determine the best path. They optimize request traffic flow and reduce tail latencies in microservice architectures.

Service mesh load balancers can also distribute load based on dynamic rules. Service mesh operators can compose policies that describe how they want to manipulate service request routing. In practice, that's done by using aliased service names for routing (similar to DNS). The alias introduces a distinction between the service destination (e.g., the foo service) and the concrete destination (e.g., the version of the foo service running in zone bar). Your applications can then be configured to address requests to that new alias and become agnostic to the implementation details of the environment.

That aliasing construct allows operators to arbitrarily target specific segments of load and route them to new destinations. For example, Linkerd uses a flexible naming strategy—delegation tables, or "dtabs"—that allows you to apply changes to a percentage of traffic, allowing you to shift traffic in incremental and controlled ways—granularly on a per-request basis. You can use them to shift or copy traffic from production to staging, from one version of a service to another, or from one datacenter to another. That kind of traffic shifting enables things like canary deployments, blue–green releases as part of a Continuous Improvement/Continuous Delivery pipeline or cross-datacenter failovers.

Istio manages that same type of approach by introducing the concept of a service version, which subdivides service instances by versions (v1, v2) or environment (staging, production) to represent any iterative change to the same service.

The specific level of granularity and use of logic varies between products, and an entire other book could be written around configuration and edge case usage for request routing and load balancing. Suffice it to say, complexity can run pretty deep here: with great power comes great responsibility. Optimization of traffic is very application specific and you should closely compare service mesh features if you already know what those patterns are like in your environment to find the solution that's right for you.

Lastly, it's worth noting that functionality for how load balancing occurs is typically implemented in a control plane, although the work happens in the data plane. If your approach is mixing and matching separate products, functionality

must be common to both layers. As of this writing, it's not uncommon for some product combinations to limit functionality available in the data plane by not presenting it in the control plane, or vice versa.

Resiliency at the Application Level

The shift to microservices highlights several challenges in managing service communication. Managing the complexity of service dependencies can make your distributed applications fragile. The service mesh aims to expose primitives not only to introduce visibility, but also to improve resiliency when something in any particular service tree fails. New primitives exist to handle failures gracefully, mitigate the risk of cascading failures, and grant you the control to manage traffic flow at a level that's optimized for how service requests actually flow through your applications.

Security

Any application running in production must also be reasonably secure. By pushing management of all communication into the service mesh data plane, it's possible to more effectively ensure things like encrypted network transmission by default and enforce role-based access controls (RBACs).

Information security is a complex topic with wide-ranging concerns. For the purposes of this book, "security" discussions will focus on securing data in motion. The service mesh is responsible for the exchange of all bits flowing through your network, and there's an opportunity to help your apps do the right thing by default.

Encrypting Service Communication

A basic architectural approach for encrypting service communication happens at the network level via firewalls. Presume that all external traffic is untrusted, but trust internal sources. You should always encrypt ingress and egress traffic. This should be the first step in many lines of defense, although (distressingly) sometimes it's the only step. After data is transmitted through trusted networks, encryption standards can be more lax or even nonexistent. You can enhance network security via segmentations that occur at the Virtual Private Cloud (VPC), Virtual Local-Area Network (VLAN), Virtual Private Network (VPN), or other virtual (or physical) level with necessary access control lists (ACLs). But often, when inside those layers, communication can all too easily be intercepted by a third party that has gained access past those barriers.

If tech history teaches us anything, it's that no system should ever be considered safe. It's imperative to implement safeguards as layers upon layers upon layers of security throughout your entire stack. The reality is that the layered approach is a

series of trade-offs in terms of time, maintenance, complexity, and so on. Some organizations punt on the layered approach because of inherent complexity and a perceived low return on investment (ROI) on tackling it. That additional work is low ROI until you become the next Equifax, Sony PlayStation, eBay, Target, Yahoo, or any of the myriad companies that have joined the public data-breach club. Some companies only realize that ROI after it's far too late.

But it doesn't need to be that way. The service mesh can be a part of that layered strategy by easing the maintenance and complexity burden of managing secure communication by default. Rather than managing TLS certificates and encrypted communications per instance at the application level, you can push that into the infrastructure layer to manage it globally.

Mutual Transport Layer Security by Default

Most languages have robust Transport Layer Security (TLS) encryption libraries available that are fairly trivial to use. The difficulty for most teams is dealing with third-party certificates, private keys, and managing secrets. There are no foolproof secrets management solutions on the market that protect you against any attack vector. Every solution has trade-offs, and you must implicitly trust some party somewhere for the entire thing to work.

As a result, many app teams will integrate trusted certificates into their application code for distribution. But when dealing with microservice architectures, that means duplication of secrets among a wide variety of applications with challenges multiplied by common factors like polyglot applications or distributed development teams. In theory, this situation wouldn't be problematic if any time a secret needed rotation (due to compromise, schedule, or any other routine reason), every development team in your stack could update its code and immediately redeploy its production apps. In practice, that's not always so easy.

Historically, that's been an app-level concern because we trust our deployment scheme and app code. With the service mesh, there's another trusted layer that can decouple that dependency from your applications to simplify management.

Because it proxies all traffic, you can use the service mesh to wrap service calls by originating and terminating TLS at both ends without needing to modify any application code. Each proxy is configured with the appropriate certificates, and updating settings is largely a matter of changing configurations in the control plane. How that's done varies between implementations, but, as with any secrets management approach, it's still mostly a question of how to distribute certificates where they're needed.

Each endpoint needs a certificate and key for its own service(s) as well as the root certification authority (CA) certificate to validate the identity of other services. Some platforms, like Kubernetes, include their own secrets-management mecha-

nisms, or you might instead use an external mechanism (e.g., Hashicorp's Vault). Istio, for example, has built-in features to help manage secrets in the control plane. When using Kubernetes, a per-cluster Istio CA automates the key and certificate management process by generating key and certificate pairs, distributing them to the correct recipients, and rotating/revoking those credentials as necessary. For non-Kubernetes platforms, the Istio CA relies on use of node agents to carry out some of these functions. Again, the trust relationship should be closely considered. But this ease of management can outweigh some of the potential risks to this type of automated approach.

Verifying Service Roles

Istio also relies on Kubernetes service accounts to add an identity to running services. The identity (or role) a workload runs represents its privileges to potentially control what it can access. This approach sets the stage to later add more robust control mechanisms, and it will be interesting to see what develops in this arena. This approach is still young but promising. Other projects, like Conduit, are also exploring this area as the next logical step for authentication.

The Service Mesh in Practice

Next, let's look at real customer use cases to examine the types of problems solved by running the service mesh in production. As of this writing, Buoyant is the only vendor supporting production service mesh users. The following studies came during my time at Buoyant, working with Linkerd users.

Use Case 1: Unnoticed Outages in Production

Our first anonymous case study involves a small vignette from an online banking company. The company has been running containerized applications using Kubernetes and Linkerd in production since 2016.

Although the company experiences substantial user traffic, managing volume is less important to them than managing reliability. As a finance company, every service request represents a real monetary transaction. Failed service requests have a measurable dollar value attached.

Although general service mesh features like observability and management are used by the company, it has paid substantially more attention to implementing proper failure handling and retry logic. The company keeps a close eye on availability metrics to ensure failures aren't costing it money.

The company introduced code changes that were causing intermittent failures to core underlying services. However, those breaking changes went unnoticed for several days in production because Linkerd was managing retries successfully. The user experience was unaffected, services continued to work, and failure rates

were unchanged because of successful retries. In other words, the service mesh was doing its job a little *too* well. It obscured defective code because everything still remained in operation.

Although amusing (to the Linkerd authors, anyway) when discovered, obviously that type of obscurity is not what you want in production. The company later added alerts that trigger when spikes in retry rates occur, even if the overall success rate remains unchanged.

Use Case 2: Resiliency Creatively Used for Local Development

The next anonymous case study is from a privately held company that develops B2B ecommerce software and processes more than five billion dollars in sales. The company supports several high-volume transactional products in its product suite. When developing a new additional application, it decided to greenfield the design. Given the nature of its business, it was critical that customers be supported with a system that was safe, resilient, and performant.

The company decided to design its application using a modern microservice architecture. It opted to use Nomad to schedule both its containerized and virtual workloads, gRPC as the protocol to handle service-to-service communication, and Linkerd as the service mesh solution. It has been running in production since April 2017.

The resiliency features provided by Linkerd were a primary selling point of the service mesh for this implementation. The gRPC protocol has many advantages like authentication, bidirectional streaming and flow control, and cancellations or time-out. But the company also needed an additional level of resiliency using complementary service mesh features like load balancing, circuit breaking, and dynamic routing.

Because Nomad maintains a small scope of functionality, it purposefully does not implement features like load balancing, which are common to heavier-weight workload management platforms like Kubernetes. When Linkerd is configured to manage load balancing, it bypasses any existing load-balancing system (if present) and instead uses your workload management platform to read the correct configuration for destination instances. In the case of Kubernetes, Linkerd references the configuration of Kubernetes services. With Nomad, Linkerd reads the configuration data from Consul.

The company has run large-scale distributed systems in production for many years. But (as covered earlier) even the most resiliently designed distributed systems can still be prone to system-wide outages via cascading failures. This company has been a primary user and developer of the circuit-breaking features in Linkerd. As such, it has put a lot of work into managing per-request routing using Linkerd's delegation tables (or "dtabs").

An interesting side benefit was using dtabs to enable local development and testing. A big challenge with test-driven local development is, of course, the tests. Because no level of testing can truly mimic conditions encountered when running in production, an entire subfield of practice exists around the art of stubbing, mocking, and otherwise simulating real responses to edge-case scenarios. It's a delicate balance between shamanistic incantations and recognizing diminishing returns. But for this company, using dtabs allowed it to test changes made on local development instances by routing a subset of calls to real production services to test operability, without affecting other production traffic.

Dtabs allow users to specify per-request routing rules. In other words, when this one request enters the stack, you can configure the output to route to your local development instance rather than its typical production-based target. You can specify dtabs generally, or you can use them to granularly manage specific requests and also inherit from other existing rulesets. Think of them like a sophisticated version of iptables operating at the request level. That construct allows this company to test staging versions of services (even services buried deep within the stack) in meaningful ways without affecting normal production traffic.

Use Case 3: Enabling Cloud Migration

A large-scale Enterprise Resource Planning (ERP) software company was seeking to modernize its systems by running them in the cloud. Typical ERP systems contain sensitive information like personnel data used by Human Resources. As a Software as a Services (SaaS)–based offering, the company also handles payment card information. That sensitive data combination had created a regulatory compliance policy that prohibited storing customer data off-premises, citing security concerns.

In 2016, the company decided to work to find a way to both modernize its infrastructure and address its information security concerns. After it found a provider that could comply with its need for encrypting and securing data at rest, it was then left with the challenge of securing customer data in motion.

In modernizing its infrastructure, the company had decided to decompose a monolithic application into microservices. It had started running Kubernetes and reorganizing development teams to manage different parts of the stack. But a frequent point of contention between the development teams and information security was the inconsistent use of Transport Layer Security (TLS) to transmit data securely: some teams used TLS diligently and others didn't. When development teams did use TLS diligently, they'd still have to scramble to patch the Secure Sockets Layer (SSL) vulnerabilities that were all too frequent that year. Some teams would be patched and running current, whereas others would take weeks

or months to get there. Those inconsistencies in the remediation time frame were prohibitive and threatened to stop the cloud migration project altogether.

The company introduced the Linkerd service mesh later that same year. All development teams were then instructed to remove TLS bindings from their applications and instead use regular HTTP calls to external services. Encryption would no longer occur at the application level. The company deployed Linkerd instances at every host endpoint to ensure that all over-the-wire service calls were being encrypted by default.

The company runs Linkerd as a Kubernetes DaemonSet, running one Linkerd pod on each node of the cluster. Kubernetes applications then route all network traffic through the Linkerd running on their node. When a service request spans across nodes, Linkerd automatically upgrades the connection to use TLS. When any change to TLS protocols needs to be made, those changes can be made globally at the service mesh level.

The company was able to address the needs of development, operations, and information security to enable a migration of its applications to the cloud. The company has been running a cloud-based deployment using Kubernetes with Linkerd in production since July 2017.

Use Case 4: An Incremental Approach to Stack Modernization

A large-scale payment processing enterprise was also seeking to modernize its application stack in 2016. However, this large enterprise has many interdependent legacy systems, and redesigning the entire distributed enterprise stack simultaneously was prohibitive. If it was going to modernize its stack, it would need to do it one component at a time and support running in hybrid mode between legacy and modern applications.

As the modernizing initiative began, the company decided to decompose its applications into microservices and then move data. It used Kubernetes to manage the new microservices. Many of the legacy distributed systems were Java Virtual Machine (JVM)–based applications that used Zookeeper for service discovery. To address the challenge of maintaining connectivity between applications in Kubernetes and applications used elsewhere, the enterprise implemented Linkerd as a solution.

Linkerd has a pluggable *namerd* interface; a namer binds a concrete name to a physical address for service discovery. One of the supported namers uses ZooKeeper ServerSets. The enterprise deployed new apps to Kubernetes but configured them to continue using Zookeeper for service discovery utilizing Linkerd. By doing so, services can communicate with one another regardless of whether they're running inside Kubernetes. That approach has allowed it to weave together old-world systems with state-of-the-art infrastructure. The company has

been running Kubernetes with Linkerd and Zookeeper for many months and is making incremental progress toward modernizing the rest of the stack.

Flexibility and Control

These are just some of the ways that users have applied the primitives within the service mesh to solve problems inherent to running modern applications in production. Linkerd has been production-ready since early 2016, so these use cases all center around how it has been used. As the ecosystem expands and usage of other solutions enters into production, we should all expect to see a wide variety of use cases that have yet to be considered.

Choosing What to Implement

Let's look a few practical tips to help you flesh out considerations when evaluating which service mesh is right for you.

Readiness

First and foremost is readiness. Will a service mesh provide enough usefulness to justify introducing an additional tool in your production environment?

If you're managing a monolithic application (even one running in a container) that makes relatively few and predictable service requests, you might not see a tremendous level of usefulness for any service mesh. Benefits such as additional visibility and resilience are great, but what pains are you feeling today? Introducing any new tool into your production stack carries risk, time, and energy. Are the benefits discussed in earlier sections going to be enough to outweigh the additional overhead required to implement, learn, and maintain this new solution?

If you manage applications that make many service calls with many underlying service dependencies, you're probably ready for a service mesh.

Existing Capabilities

How will any service mesh product you are evaluating interact with your environment when you use it?

If you're already managing microservices in production and you aren't using a service mesh, you might have already built some of these capabilities into your apps. How well is that solution working for you? Are the maintenance and management burdens for your existing solution worth the effort to migrate?

How easy is it to add service management to the mesh incrementally? Are you required to use all of the features of this product all at once? How will those fea-

tures interact with your applications as you begin decoupling them from service management logic? Is this a gradual migration or a big-bang replacement?

Obviously, the choice is easier for greenfield projects. Even then you should consider your needs for things like connecting legacy apps and infrastructure or onboarding future applications.

Team Dynamics and Structure

Following Conway's law, we build systems that resemble our organizational structures. How opinionated is the product you're choosing about dictating how you work? For example, if features like our earlier TLS-by-default example are managed globally, how does that compare with your expectations for which team will manage which part of your stack? You might be in a team structure that has clear ownership boundaries between infrastructure and app code. Or you might be in a flatter organization where perhaps every development team in your organization needs to be able to alter service mesh settings as necessary.

This is where use of a control plane and its capabilities begins to matter. More to the point, do you need centralized or decentralized management of configuration?

In my admittedly limited experience, organizations tend to favor centralized control for things like infrastructure configurations, authorization, and information security. My suspicion is that, even in a DevOps world, a majority of organizations will favor the use of a centralized control plane from which data plane configurations can be accessed. But that's not going to work for everyone.

There are use cases for managing distributed configuration among all of your different proxy endpoints. Typically, apps with distributed management already have some mechanism for distributing changes throughout the fleet. If that's the case, managing proxy settings is simply another configuration endpoint. In these types of scenarios, there's probably already a metrics collection mechanism across these distributed machines. You can add data plane metrics for collection, presuming built-in support from your service mesh choice. In essence, you already have the core of what a control plane provides within your infrastructure. So, you might find yourself devaluing the role of a control plane or possibly foregoing one entirely, depending on the complexity of your deployment.

Supportability

How are you going to support a service mesh in production?

The additional overhead required for any new tool means obvious things like performance constraints and managing resource consumption. But it also means things like cognitive burden and the learning curve required to use this tool. The more complex a product, the more time and effort it takes to understand the var-

ious components, how they work, and how to troubleshoot them when things go wrong. The pragmatist in me can't overstate the value of being able to clearly reason about production systems. Unpredictable failure situations *occur all the time*. When they do, how well equipped are you to solve the mystery and stop it from happening again?

Of course, there's a trade-off here. Sometimes, additional complexity is worthwhile if you gain meaningful benefits as a result. Again, it's a matter of clearly understanding the pains you feel today and judging whether the juice is worth the squeeze.

Lastly, there's also the question of commercial (or enterprise) support. Many existing service mesh solutions are open source and rely on community-based support. Is that enough for your organization or do you require more?

Rapidly Evolving Technology

The service mesh is a relatively new technology. First, consider your own readiness and the state of your infrastructure today. Do you need to jump in now and keep up with new changes? Do you have time to wait before your production support needs are eminent and there's more clarity in the field of options?

If you're ready today, consider the maturity of the tool you're evaluating along with other pragmatic factors like production readiness, support options, development community, currently available features, feature development roadmap, complexity, and overall supportability, including experience supporting production workloads.

This is not an exhaustive list of considerations, but hopefully this prompts you to think about the types of circumstances that you should be considering to evaluate your fit against the service mesh options available today.

Conclusions

It's an early and exciting time for the service mesh ecosystem, with a lot of buzz around development for various products with differing philosophies and scopes. The service mesh has proven itself as a necessary building block in the modern application stack. Because the service mesh adds visibility, resilience, and security to your applications, it's seen a surge of interest in the past few months that is probably only going to continue to increase for the foreseeable future.

With that recent surge, options in the ecosystem might not always be obvious. Unfortunately, hype has a way of undermining usefulness. Hopefully, this book has been useful to help you more clearly see what a service mesh does, why it matters, how it's used, and how you should think about finding what's right for your organization.

Like any new technology, the ecosystem moves fast. Things are always changing, and one of the biggest challenges in writing this book is picking what to say in a way that will still be valid a few months from now. As such, this book keeps its focus fairly high level. So, if you find yourself wanting to drill down deeper, I encourage you to reach out to me via Twitter (@gmiranda23). My DMs are always open.

About the Author

George Miranda is the Head of Marketing for Buoyant, Inc., authors of the Linkerd and Conduit service meshes. For over 15 years, he has made a career of managing distributed systems in various web development and operations roles at a variety of startups and large enterprises, mostly in the finance and video game sectors. Since then, he has shifted to working with vendors creating open source infrastructure solutions that help improve life for people doing the same jobs he performed in the past. Prior to Buoyant, he also worked at Chef Software.