O'REILLY®

Istio

Up & Running

Using a Service Mesh to Connect, Secure, Control, and Observe

Early Release

RAW & UNEDITED

Lee Calcote & Zack Butcher

b. Use Cases
c. Choosing a topology
d. Cross-Cluster or Multi-Cluster?
e. Multicluster
f. Cross-Cluster

Istio: Up and Running

Using a Service Mesh to Connect,
Secure, Control, and Observe

Lee Calcote and Zack Butcher

Istio: Up and Running

by Lee Calcote and Zack Butcher

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*.

Editors: Nikki McDonald and Eleanor Bru

Production Editor: Deborah Baker

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

August 2019: First Edition

Revision History for the First Edition

- 2019-04-17: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492043782 for release details.

978-1-492-04371-3

[LSI]

Preface

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.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/oreillymedia/Istio_Up_and_Running.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Istio: UP and Running* by Lee Calcote and Zack Butcher (O'Reilly). Copyright 2019 Matt Baldwin, Zack Butcher, and Lee Calcote, 978-1-492-04378-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

> **NOTE**
>
> For almost 40 years, *O'Reilly* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video

from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

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)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://www.oreilly.com/catalog/9781492043782.

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

Chapter 1. Cloud Native Approach to Uniform Observability

Driven by the need for speed of delivery, potential global scale, and judicious resource utilization, cloud native applications run as immutable, isolated, ephemeral packages on what is typically shared infrastructure.

What Does It Mean to Be Cloud Native?

Cloud native as an umbrella term manifests in various forms as a combination of both *technology* and *process*. Driven by the need for both machine and human efficiency, cloud native *technology* spans application architecture, packaging and infrastructure, while cloud native *process* embodies the full software lifecycle, often, but not always, reducing what are historically multiple, separate organizational functions and lifecycle steps (e.g. architecture, QA, security, documentation and so on) to two functions: *development* and *operations*. Cloud native software is commonly, but not always, continuously delivered as a service.

Cloud native applications typically run in a public or private cloud. Minimally, they run on top of programmatically addressable infrastructure. That said, lifting and shifting an application into a cloud doesn't quite make it cloud native.

Characteristics of cloud native applications:

- Run on programmatically-addressable infrastructure.
- Distributed and decentralized with the focus often being about how the application behaves, not on where its running.
- Dynamic and decoupled from physical resources by one or more layers of abstraction across compute, network and storage resources. Often allow for rolling updates to smoothly upgrade services without service disruption.
- Resilient and scalable in that they are intended to run without guarantee underlying infrastructure, designed without single points of failure and run

redundantly.

- Observable through their own instrumentation and that provided by underlying layers. With their dynamic nature, distributed systems are relatively harder to inspect and debug, and must account for their observability.

PATH TO CLOUD NATIVE

For most of you, the path to cloud native is an evolutionary act of applying cloud native principles to existing services whether through retrofit or rewrite. Others of you are fortunate enough to have started projects after cloud natives principles and tools were generally available and accepted. Whether your journey calls for dealing with an existing service or writing a new collection of them, service meshes offer much value - *value that increases as you increase the number of services you own and run*. Service meshes are the next logical step after a container orchestration deployment. Figure 2.1 outlines various cloud native paths. Depending upon which service mesh you deploy, you may need a certain number of microservices to make its deployment worthwhile, because some service meshes are easier to deploy than others and because some service meshes offer more value than others.

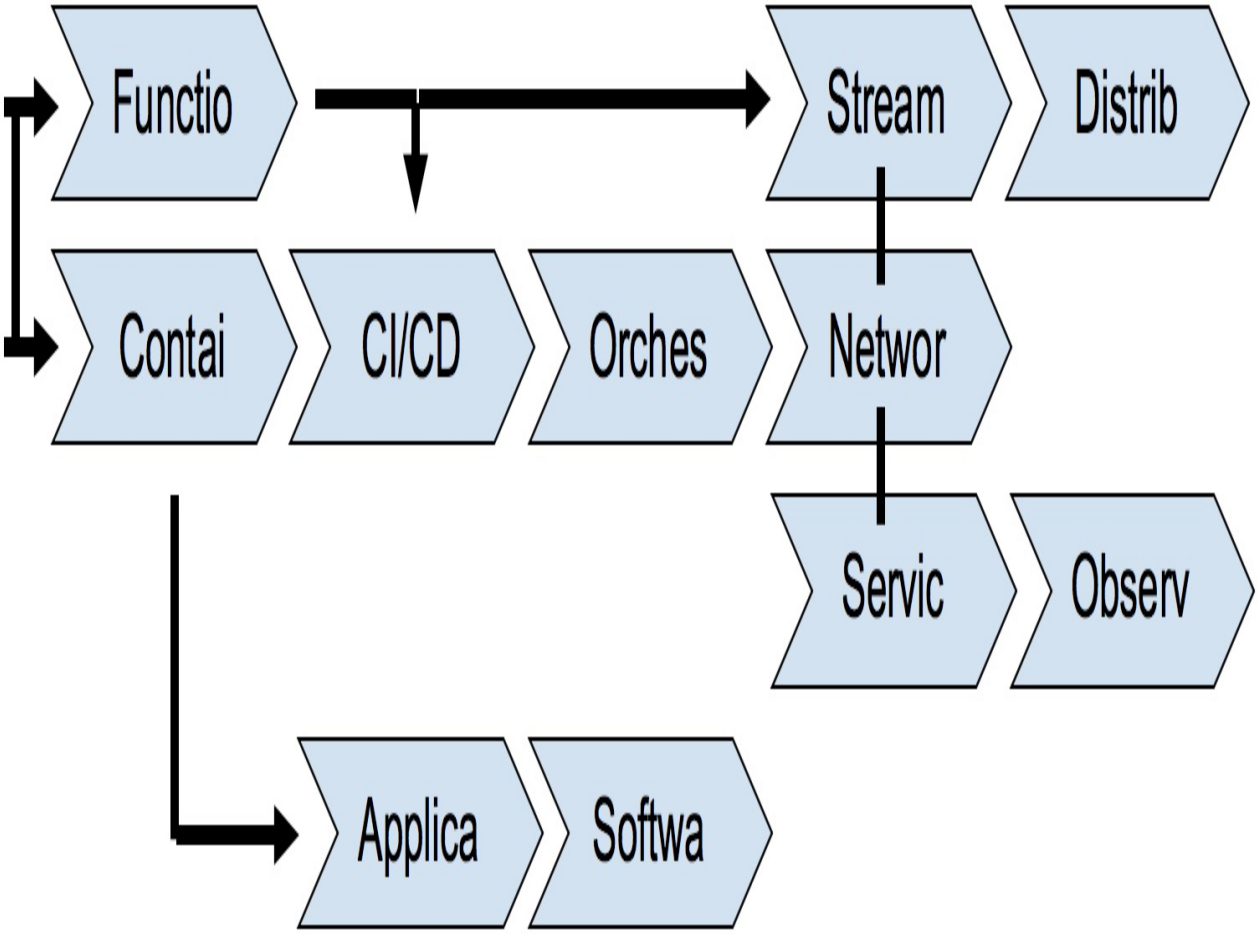


Figure 1-1. Figure 2.1 Paths taken to cloud native are varied and replete with choice.

Depending upon your teams' collective experience and your specific projects, your path to cloud native will use different combinations of software development process, operational practices, application architecture, packaging and runtimes, and application infrastructure. Applications and teams exhibiting cloud native characteristics use one, a combination of, or all of the approaches highlighted in red in Figure 2.2.

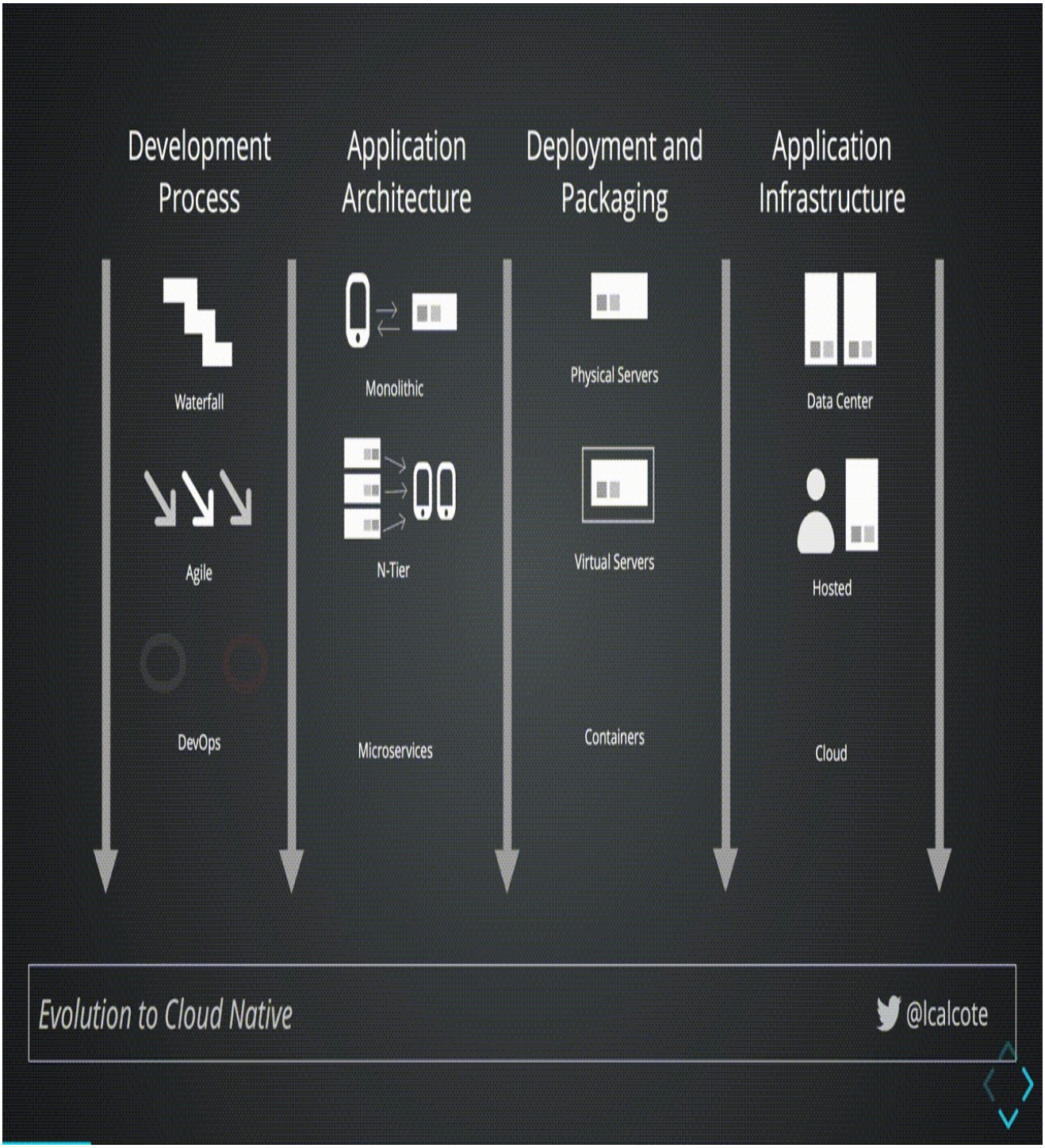


Figure 1-2. Figure 2.2 The evolution to cloud native through process and technology (architecture, packaging and infrastructure).

PACKAGING AND DEPLOYMENT

Cloud native technology often takes the shape of microservices (engaged through service endpoints) built in containers (engaged through a scheduler) and augmented by functions (engaged through event notifications). Evolutionary shifts in the patterns of packaging are driven by the need for efficient utilization of machines and speed of delivery by engineers. The path to cloud native pushes to smaller and smaller units of deployment, which is enabled through high levels of resource isolation. Isolation

through virtualization and containerization provide higher levels of efficiency as smaller packages make for more tightly bin-packed servers.

Each phase of our packaging evolution from bare metal servers to virtual machines to containers to unikernels to functions has enjoyed varying degrees of use when measured by the number of deployments using that form of packaging in the wild. Some package types provide better guarantees of portability, interoperability, isolation, efficiency and so on. As an example, work around containers has delivered higher degrees of portability and interoperability than virtual machines achieved. While lightweight and isolated, immediately and conceptually infinitely scalable, functions suffer in regard to portability as exhibiting possibly the highest degree of lock-in amongst the types of packaging. Irrespective of your chosen packaging--whether you deploy your services directly on host operating system, in a virtual machine, in a container, as a unikernel, or a function--service meshes can provide connection, control, observability, and security.

APPLICATION ARCHITECTURE

Arguably, more important than their taken form, are the characteristics exhibited by cloud native application architecture. Characteristics centric to cloud native include ephemerality, active scheduling workloads, loosely-coupled with dependencies explicitly described, event-driven, horizontally-scaled, and have clean separation of stateless and stateful services. Cloud native applications often exemplify an architectural approach that is commonly declarative and incorporating resiliency, availability, observability, as upfront design concerns. With these concerns central to their design, cloud native applications are commonly loosely-coupled, distributed systems that are horizontally-scalable, independent, and small (have service-bounded concerns).

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Within these environments, cloud native applications are centered around declarative APIs to interface with the infrastructure. These techniques enable loosely coupled systems that are resilient, manageable, and observable. Istio and other open source service meshes deliver the next generation of networking designed for cloud native applications.

DEVELOPMENT AND OPERATIONS PROCESS

Developer and operator experience is also centric to the philosophy of cloud native design and process, which fosters code and component reuse, high degrees of automation and . When married with infrastructure being treated as code, operators set forth ruthlessly automating the methods by which cloud native applications and their infrastructure are deployed, monitored and scaled. When combined with robust automation, microservices allow engineers to make high-impact changes frequently and predictably with minimal toil typically using multiple continuous integration and continuous delivery pipelines to build and deploy microservices.

High levels of granular observability is a key focus of systems and services that site reliability engineers monitor and manage. Istio generates metrics, logs, and traces pertaining to requests sent across the mesh, meaning that it facilitates instrumentation of services so that the creation of metrics and generation of logs and traces is done without need for code change (save for context propagation in traces). Istio, and service meshes in general, insert a dedicated infrastructure layer between dev and ops, separating common concerns of service communication by providing independent control over services. Without a service mesh, operators are still tied to developers for many concerns as they need new application builds to control network traffic, shaping, affecting access control and which services talk to downstream services. The decoupling of dev and ops is key to providing autonomous independent iteration.

INFRASTRUCTURE

Public, hybrid and private cloud are clearly core to the definition of what it means to be *cloud* native. In a nutshell, cloud is software-defined infrastructure. The use of APIs as the primary interface to infrastructure is a concept central to cloud. Natively integrated workloads leverage these APIs (or abstractions of these APIs) as opposed to non-native workloads that are ignorant of their infrastructure. As the definition of cloud native advances, so does cloud services themselves. Broadly, cloud services have evolved from infrastructure as a service (IaaS) to managed services to serverless offerings. Given that most function as a service (FaaS) compute systems execute inside of a container, these FaaS platforms can run on a service mesh and benefit from *uniform observability*.

CONCLUSION

When leveraged, cloud native technology and process radically improves machine efficiency and resource utilization while reducing the cost associated with maintenance and operations and significantly increasing the overall agility and maintainability of applications.

While leveraging a container orchestrator addresses a layer of infrastructure needs, it does not meet all application or service-level requirements. Service meshes provide a layer of tooling for the unmet service-level needs of cloud native applications.

What is Observability?

Newfangled words in our tech industry need proper classification. Their taxonomies are important not only to facilitate common nomenclature (and understanding), but avoid contentious debates. The notion of a system being observable vs. being monitored has been of some discussion in our industry. To distinguish, let's identify *monitoring* (a verb) as a function performed - an activity, while *observability* (an adjunct noun) as an attribute of a system. When speaking to the observability of a system, you describe how well and in what way the system provides signals to monitor. While monitoring is the action of observing and checking the behavior and outputs of a system and its components over time.

Much debate may subside were we to use "monitoring" and "monitorability". This verb and its noun adjunct may be left alone in synonymous company with its sister "observing" and "observability". That said, then we'll need to introduce a new term as a sacrificial offering for vendors to claim, coin, wield and posture around. If we didn't have a term and definition to debate, it would be like a quinceanera without a pinata to beat up.

If your software is observable, it's typically because its instrumented to capture and expose information (telemetry - measurements). Observability is for software engineers, who want the ability to reason or complex software. Monitoring is the process of checking and evaluating system state. Your ability to monitor a system is improved by its number of observable attributes (its observability). Monitoring is for operators, who want to assert if a state is true or not true. Observability for developers

and monitoring for operators? Maybe… or at least until service meshes arrived, introducing a decoupled management layer - layer 5.

Consider monitorability (a noun) as the condition of being monitorable; the ability to be monitored. Monitoring is being on the lookout for failures, typically through polling observable endpoints. Simplistically, early monitoring systems target uptime as a key metric to measure resilience. Modern monitoring tooling is oriented toward top-level services metrics like latency, errors (rate of requests that fail), traffic volume (by requests per second for web service or transactions retrievals per second for a key-value store), and saturation (a measurement of how utilized a resource is). Modern monitoring systems are often infused with analytics for identifying anomalous behavior, predicting capacity breaches and so on. Service meshes bridge observability and monitoring by providing some of both by way of generating, aggregating and reasoning over telemetry.

Different service meshes incorporate monitoring tooling as a capability or easy add-on.

PILLARS OF TELEMETRY

The medium of observability includes logs often in the form of events and errors, traces often in the form of spans and annotations, and metrics often in the form of histograms, gauges, summaries and counters.

Three Pillars of Observability

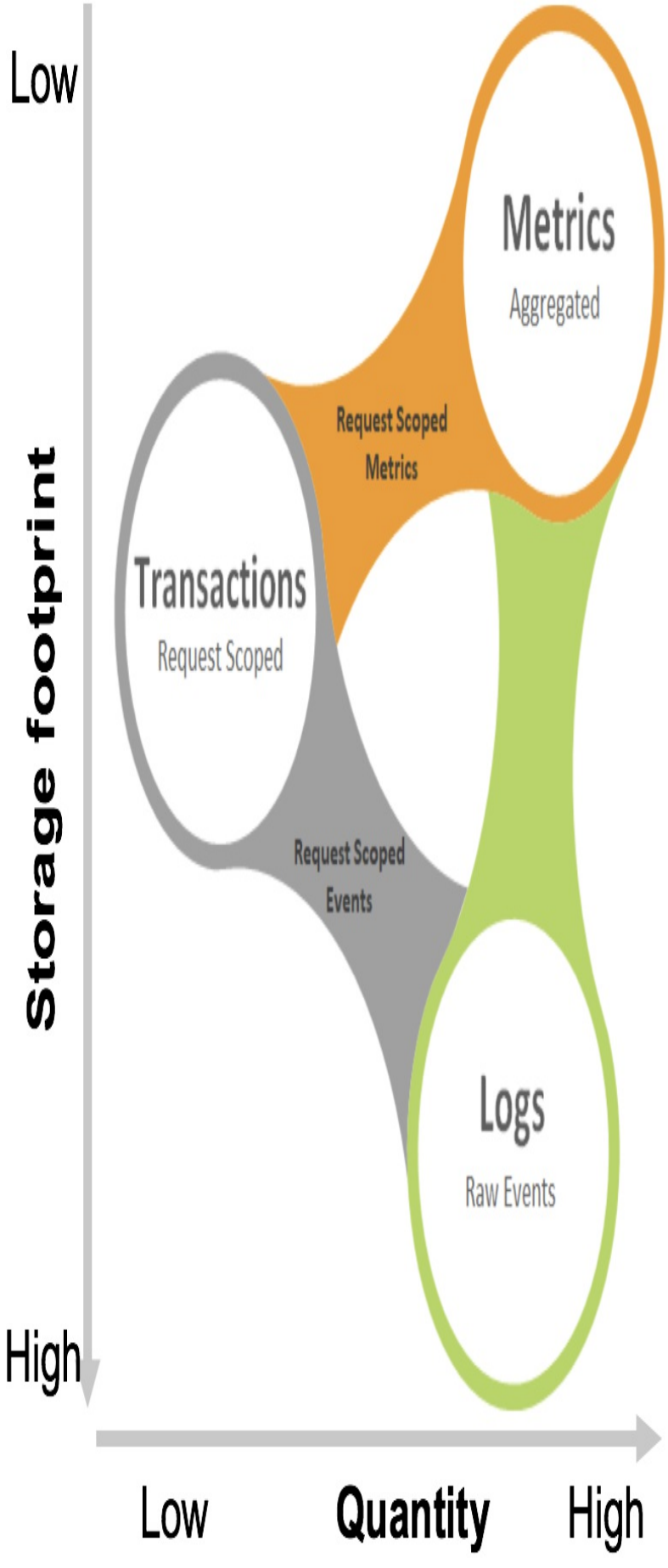


Figure 1-3. Figure 2.3 Three pillars of observability: key types of telemetry for internal observability

LOGS

Logs provide additional context to data like metrics and are well-suited for debugging, making it difficult to strike the right balance in tuning which logs to centrally persist vs. which to allow to eventually rotate out. However, logs are also costly in terms of performance as they are generally-speaking of the highest volume and require the most storage. Albeit structured logging doesn't suffer from some of the downsides inherent in pure string-based log manipulation, it's still far more voluminous to store as well as slower to query and process, as evident from the pricing models of log-based monitoring vendors.

METRICS

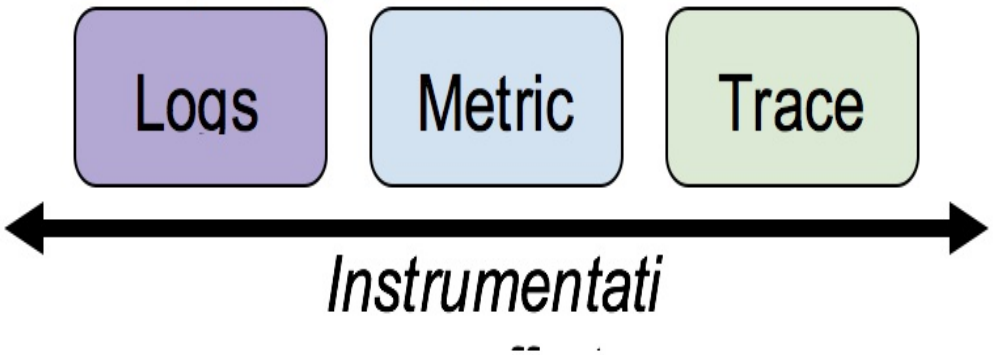
In contrast, metrics have a constant overhead and are good for alerts. Taken together, logs and metrics are good for insights into individual systems, but make it difficult to see into the lifetime of a request that has traversed multiple systems. This is pretty common in distributed systems. Metrics can be powerful and when aggregated quite insightful - good for identifying known-unknowns. The high rate of compression on metrics only pushes them to have lower footprint, considering that they are optimized for storage (a good Gorilla implementation can get a sample down to 2.37 bytes) and enable historical trends through long-term retention.

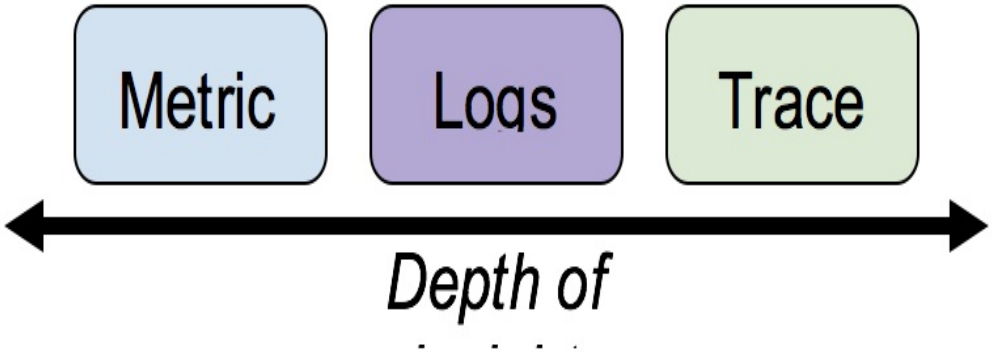
TRACES

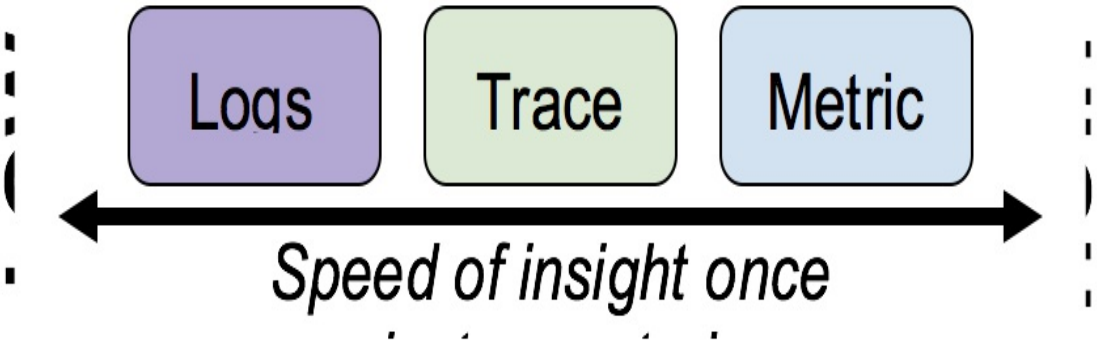
Tracing offers the ability to track a request as it moves through various systems. It is hard to introduce later, one reason being third party libraries that are used by the application also need to be instrumented. Traces are sampled to reduce overhead and storage costs. Sampling here means reducing the amount of information collected. Some best practices for logging include enforcing quotas and dynamic rate of adjustment of log generation. they are to process and store.

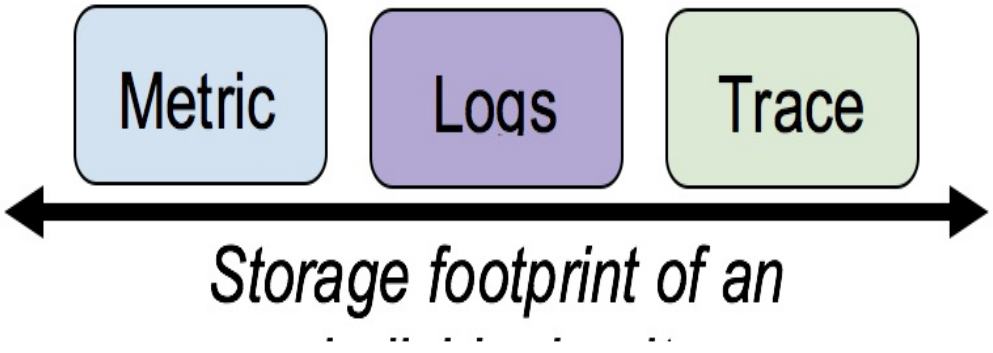
A maximally observable system includes use of each internal signal. A maximally monitored system includes synthetic checks, end user experience monitoring (real user monitoring) and tooling for distributed debugging. Blackbox testing (synthetic checks) are still needed as they are end-to-end validation of everything you may not have

observed.


Logs
Metric
Trace
Instrumentati


Metric
Logs
Trace
Depth of


Logs
Trace
Metric
Speed of insight once


Metric
Logs
Trace
Storage footprint of an

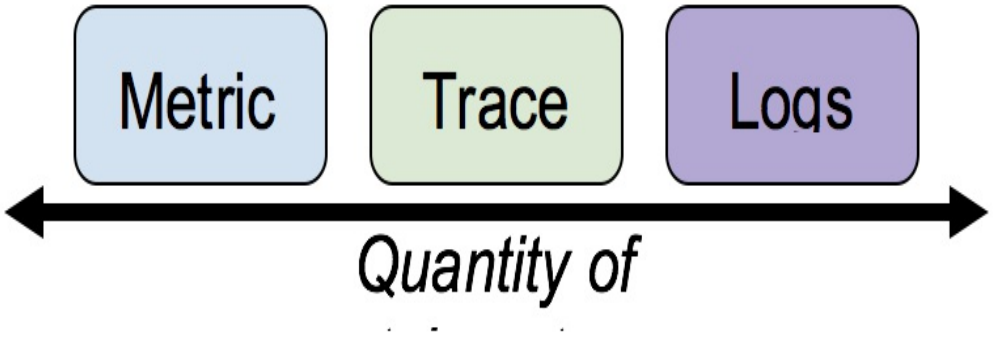
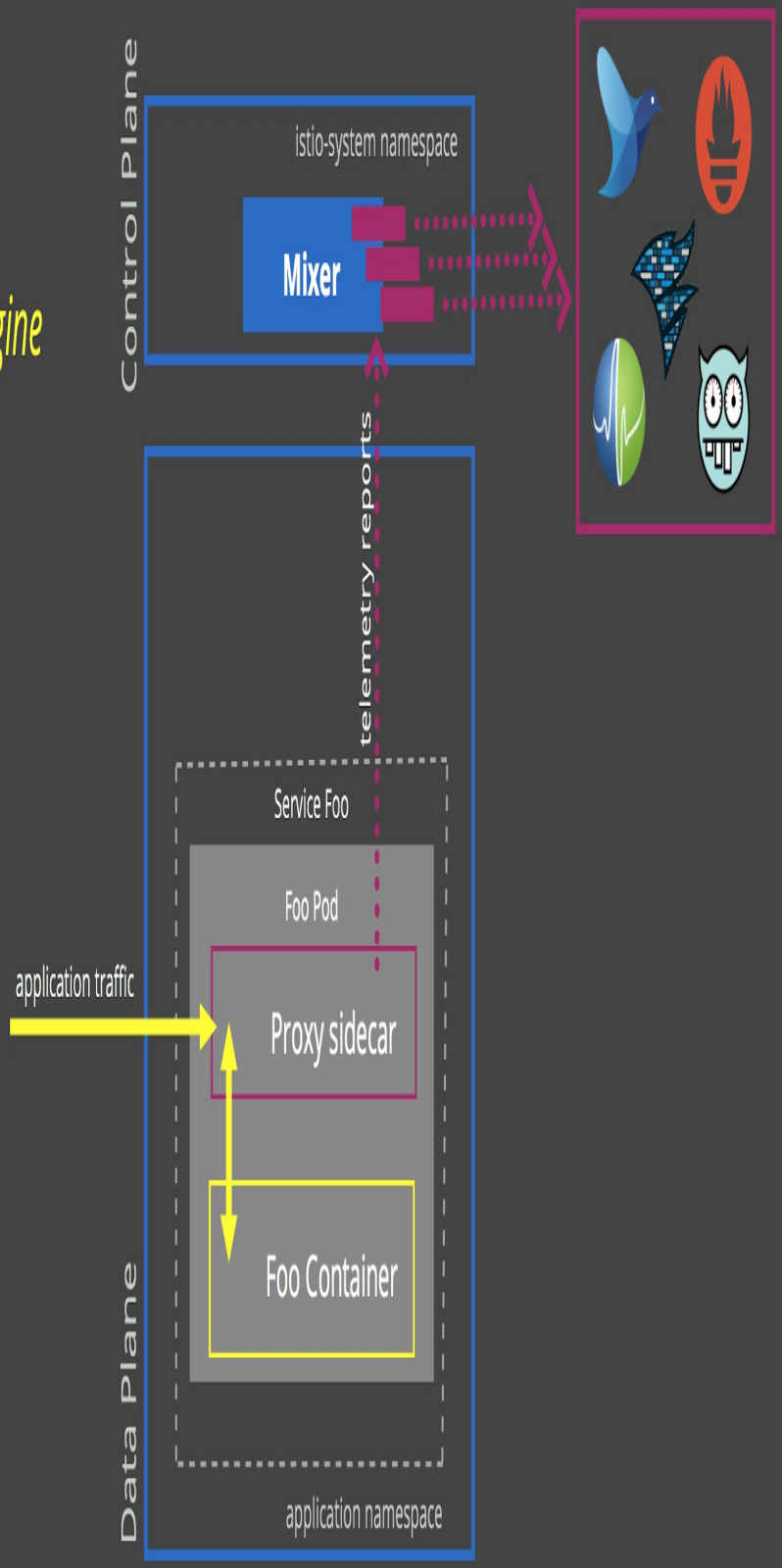

Metric
Trace
Logs
Quantity of

Figure 1-4. Figure 2.4 Comparative spectrum of value provided by and cost of each pillar

Many of you are accustomed to having individual monitoring solutions for distributed tracing, logging, security, access control etc. Service meshes centralize and assist in solving these observability challenges. Istio, specifically, generates and sends multiple telemetric signals based on requests send into the mesh.

Mixer

an attribute processing engine



layer5.io

Out-of-band telemetry propagation

Control flow during request processing

application traffic

@lcalcote

Figure 1-5. Figure 2.5 Istio's Mixer is capable of collecting multiple, telemetric signals and sending to backend monitoring, authentication, quota systems via adapters.

WHY IS OBSERVABILITY KEY IN DISTRIBUTED SYSTEMS?

The ability to aggregate and correlate logs, metrics, and traces together when running distributed systems is key to your ability to reason over what's happening within your application across the disparate infrastructure upon which is runs. When running a distributed system, you understand that failures *will* happen and can account for some percentage of these known-unknowns. You must also understand that you will not know beforehand all the ways in which failure will occur (unknown-unknowns). Therefore, your system must be granularly observable (and debuggable), so that you can ask new questions and reason over the behavior of your application (in context of its infrastructure). Of the multitude of signals available, which are the most critical to monitor? Taking the union of signals common across popular methodologies of USE, RED, and Google SRE's four golden signals you find requests, saturation, errors and latency.

As a service owner, you need to explore these complex and interconnected systems and explain anomalies based on telemetry delivered from your instrumentation. It's through a combination of internal observables and external monitoring that service meshes come to bear on illuminate service operation, where you might otherwise be blind.

CONCLUSION

Monitoring is an activity you perform. It's simply observing the state of a system over a period of time. Observability (the condition of being observable) is a measure of how well the internal states of a system can be inferred from knowledge of its external outputs; a measure of the extent to which something is observable.

It's argued that metrics provide the highest value for smallest investment. Given that some service meshes facilitate distributed tracing, I'll argue that distributed tracing provides the highest value and can be achieved at low investment. Ideally, your instrumentation allows you to dial back the verbosity levels and sampling rate when your service is operating normally.

Rather than attempting to overcome distributed systems concerns by writing infrastructure logic into application code, you can manage these challenges with a service mesh. A service mesh helps ensure that the responsibility of service management is centralized, avoiding redundant instrumentation, and making observability ubiquitous and uniform across services.

Uniform observability with a service mesh

Insight (observability) is the number one reason that people deploy a service mesh. Not only do service meshes provide a level of immediate insight, but they also do so *uniformly* and *ubiquitously*. Many of you are accustomed to having individual monitoring solutions for distributed tracing, logging, security, access control, metering and so on. Service meshes centralize and assist in consolidating these separate panes of glass by generating metrics, logs and traces of requests transiting the mesh and by acting as a layer 7 firewall to provide granular traffic control. Leveraging automatically-generated span identifiers from the data plane, Istio provides a baseline of distributed tracing in order to visualize dependencies, requests volumes, and failure rates. By default, Istio emits metrics for global request volume, global success rate, individual service responses by version, source and time. When metrics are ubiquitous across your cluster, they unlock new insights, and also remove dependency on developers to instrument code to emit these metrics.

The importance of ubiquity and uniformity of insight (and control over request behavior) is well-illustrated through use of client libraries.

CLIENT LIBRARIES

Client libraries (sometimes referred to as microservices frameworks) are today's go-to tooling for developers looking to infuse resilience into their microservices. There are a number of popular language-specific, client libraries that offer resiliency features like timing out a request or backing off and retrying when a service isn't responding in a timely fashion.

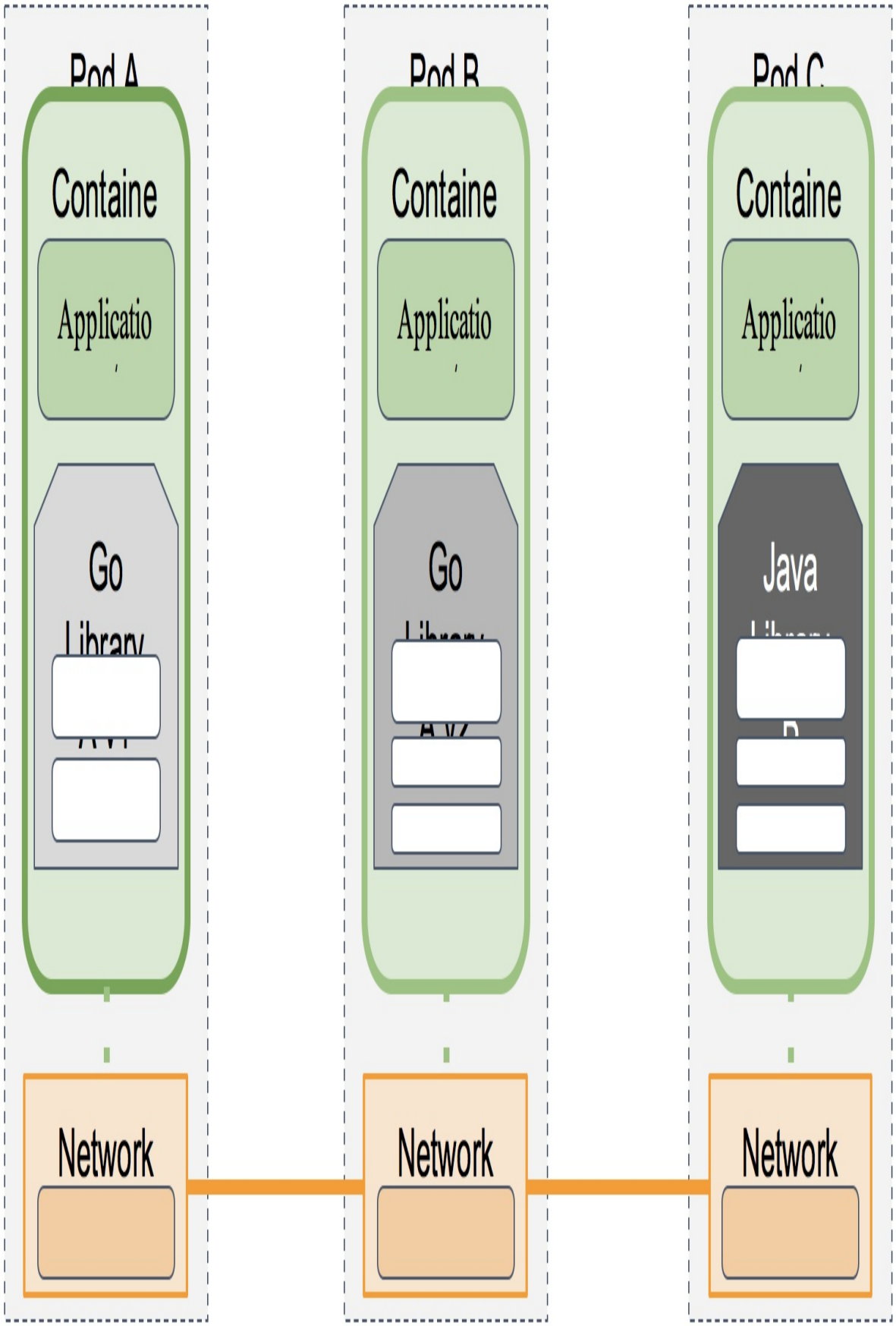


Figure 1-6. Figure 2.6 Applications tightly coupled with infrastructure control logic.

Client libraries became very popular as microservices took foothold in cloud native

application design as a means to avoid rewriting the same infrastructure and operational logic in every service. A problem with microservices frameworks is that they couple those same concerns - infrastructure and operational in nature - with your code. Their use leads to duplication of code across your services and inconsistency in what different libraries provide and how they behave. As shown in Figure 2.2, when running multiple versions of the same library or different libraries, getting service teams to update their libraries can be an arduous process. When these distributed systems concerns are embedded into your service code, you have to chase your engineers to update and correct their libraries of which there may be a few, used to varying degrees. Getting a consistent and recent version deployed can take some time. Achieving and enforcing consistency is challenging.

INTERFACING WITH MONITORING SYSTEMS

From the application's vantage point, service meshes largely provide blackbox monitoring (observing a system from the outside) of service-to-service communication, leaving whitebox monitoring (observing a system from the inside - reporting measurements from inside-out) of an application as the responsibility of the microservice. Proxies that comprise the data plane are well-positioned (transparently, in-band) to generate metrics, logs and traces providing uniform and thorough observability throughout the mesh as a whole. Istio provides adapters to translate this telemetry and transmit to your monitoring system(s) of choice.

CONCLUSION

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

Is your application easy to monitor in production? Many applications are, but sadly, some are designed with observability as an afterthought. The ability to monitor is one of many important factors of running apps at scale, and just like backups, security, auditability, and the like, it should ideally be considered in advance. In this way, you can make the tradeoffs consciously instead of accidentally. If you didn't, then deploy a service mesh.

Chapter 2. Istio at a Glance

As our organizations mature in their operations of container deployments, many will come to find value in and install a service mesh within their environment. The topic of service mesh is a significant buzz within the cloud native ecosystem. Currently, many administrators, operators, architects and developers are currently seeking an understanding and guidance, so let's have a glance at Istio.

As you learned in chapter 2, Istio like other service meshes introduces a new layer into modern infrastructure, creating the potential to implement robust and scalable applications with granular control over them. If you're running microservices these challenges are exacerbated as the more microservices deployed, the more aggravated these challenges become. You may not be running microservices, however. While the value of a service mesh shines most brightly in microservices deployments, Istio also readily accounts for services running directly on the operating system running in your virtual machine/bare metal server.

Service Mesh Architecture

At a high level, service mesh architectures are commonly comprised of two planes - a *control* and *data* plane. Istio's architecture adheres to this paradigm. Divisions of concern by planes are shown in Figure 3-1.

Separate Software
(monitoring, auditing, inventory, metering)

Out-of-band

Management Plane
(monitoring, policies, metering)

Service Mesh Planes
(L7-focused - application communication e.g. HTTP, gRPC..)

Out-of-band

Control Plane
(monitoring, policies)
(network topology / state)

In-band

Data Plane
(packet forwarding)
(routing)

Figure 2-1. Figure 3-1 Istio (and other service meshes) is composed of two planes. A third management plane depicts where incumbent/infrastructure systems may reside.

Planes

The **data plane** in Istio intercepts every packet in the request and is responsible for health checking, routing, load balancing, authentication, authorization and generation of observable signals. Operating in-band, 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 intra-cluster communication as well as inbound (ingress) and outbound (egress) cluster network traffic. Whether traffic is entering the mesh (ingressing) or leaving the mesh (egressing), application service traffic is directed first to the service proxy for handling. In Istio's case, traffic is transparently intercepted using iptables rules and redirected to the service proxy.

The Istio data plane:

- touches every packet/request in the system.
- is responsible for service discovery, health checking, routing, load balancing, authentication, authorization, and observability.

The **control plane** in Istio provides a single point of administration for service proxies, which need programmatic configuration based on the need to efficiently manage many of them and the need to have their configuration updated in real-time as services are rescheduled across your environment (i.e. container cluster). 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 and user interface with which to interact, each of which provides access to a centralized API for holistically controlling proxy behavior. Changes to control plane configuration can be automated through its APIs (e.g. by a CI/CD pipeline), where in-practice, configuration is most often version-controlled and updated.

The Istio control plane:

- Provides policy and configuration for services in the mesh.
- Takes a set of isolated stateless sidecar proxies and turns them into a service mesh.
- Does not touch any packets/requests in the system.

ISTIO CONTROL PLANE COMPONENTS

Pilot

Pilot is the head of the ship in an Istio mesh, so to speak, Pilot keeps in-sync with the underlying platform (e.g. Kubernetes) by tracking and representing the state and location of running services to the data plane, (you will learn about istio-proxy as a data plane component later in this chapter). Pilot interfaces with your environment's service discovery system, and produces configuration for the data plane service proxies (you will learn about istio-proxy as a data plane component later in this chapter).

As Istio evolves, more of Pilot's focus will be the scalable serving of proxy configuration and less on interfacing with underlying platforms. It serves Envoy-compatible configuration by coalescing configuration and endpoint information from various sources and translating into xDS objects. Another component, Galley, will eventually subsume the responsibility for interfacing directly with underlying platforms.

Galley

Galley is Istio's configuration aggregation and distribution component. As its role evolves, it will insulate the rest of Istio's components from underlying platform and user-supplied configuration by ingesting and validating configuration. Galley implements the Mesh Configuration Protocol (MCP) as a mechanism for serving and distributing configuration.

Mixer

Capable of standing alone, Mixer is a control plane component designed to abstract infrastructure backends from the rest of Istio, where infrastructure backends are things like Stackdriver or New Relic. Mixer bears these responsibilities for precondition

checking, quota management, and telemetry reporting. Mixer is:

Enables platform & environment mobility

Responsible for policy evaluation and telemetry reporting in that is provides granular control over operational policies and telemetry

Has a rich configuration model

Intent-based config abstracts most infrastructure concerns

Service proxies and gateways invoke Mixer to perform precondition checks in order to determine whether a request should be allowed to proceed (check) whether based on communication between the caller and the service is allowed or has exceeded quota, and to report telemetry once a request has completed (report). Mixer interfaces to infrastructure backends through a set of native and third-party adapters. Adapter configuration determines which telemetry is sent to which backend at what time. Service mesh operators are able to use Mixer's adapters as the point of integration and intermediation with their infrastructure backends in that it operates as an attribute processing and routing engine.

Citadel

Citadel provides strong service-to-service and end-user authentication using mutual TLS, with built-in identity and credential management. The certificate authority (CA) component of Citadel is responsible for approving and signing certificate signing requests (CSRs) sent by Citadel agents, and performs key and certificate generation, deployment, rotation and revocation. Citadel has the (optional) ability to interact with an Identity Directory during the certificate approval process.

Citadel has a pluggable architecture in which different certificate authorities (CAs) may be used so that Citadel is not using its self-generated, self-signed signing key and certificate to sign the workload certificates. The CA pluggability of Istio enables and facilitates:

Integration with your organization's existing Public Key Infrastructure (PKI) system.

Secure communication between Istio services and non-Istio legacy services (by sharing the same root of trust)

Protection of the CA signing key by storing it in a well-protected environment (e.g. Vault + HSM)

SERVICE PROXY

Service mesh proxies gate network traffic in its flow into the cluster, between services and out of the cluster. access to another object. Istio uses proxies between services and clients. The service proxy uses proxies that are usually deployed as sidecars in pods. It's the proxy-to-proxy communication truly forms "the mesh". Inherently, it follows that in order for an application to be onboarded to the mesh, a proxy must be placed between the application and the network. A sidecar adds behavior to a container without changing it. In that sense, the sidecar and the service behave as a single enhanced unit. The pods host the sidecar and service as a single unit.

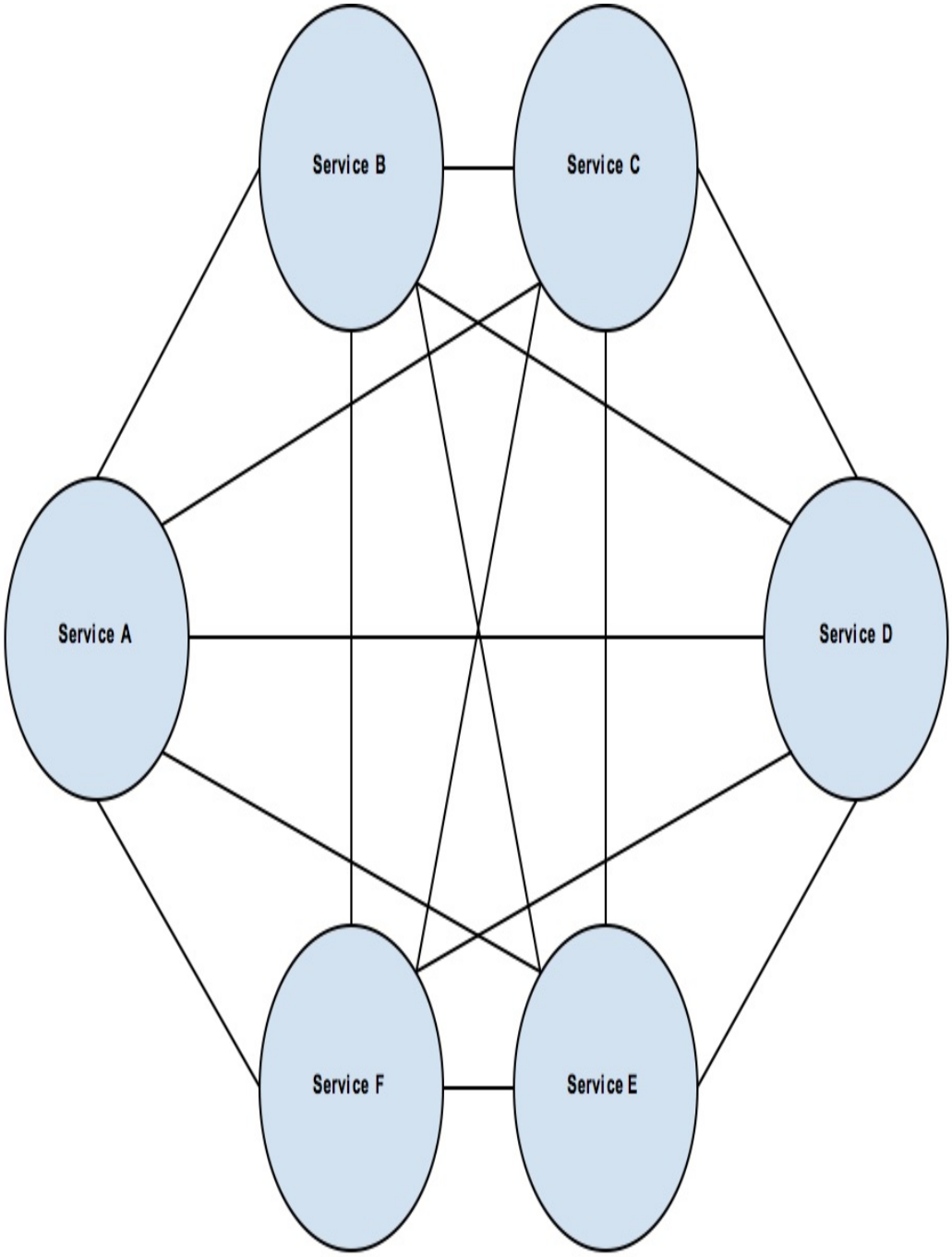


Figure 2-2. Figure 3-2 Fully interconnected service proxies are what form "the mesh."

ISTIO DATA PLANE COMPONENTS

Envoy

Istio uses an extended version of the Envoy proxy, a high-performance proxy developed in C++, to mediate all inbound and outbound traffic for all services in the service mesh. Istio uses Envoy's features such as dynamic service discovery, load balancing, TLS termination, HTTP/2 and gRPC proxying, circuit breakers, health checks, staged rollouts with %-based traffic split, fault injection, and rich metrics.

Envoy is deployed as a sidecar to the relevant service in the same Kubernetes pod.

This allows Istio to extract a wealth of signals about traffic behavior as attributes, which in turn it can use in Mixer to enforce policy decisions and be sent to monitoring systems to provide information about the behavior of the entire mesh.

INJECTION

The sidecar proxy model also allows you to add Istio capabilities to an existing deployment with no need to re-architect or rewrite code. This is a significant attraction to using Istio. The promises of an immediate view of top level service metrics, of detailed control over traffic and of automated authentication and encryption between all services without having to either: change your application code or change your deployment manifests.

Using the canonical sample application BookInfo you can see how service proxies come into play and form a mesh. Figure 3-3 pictures the BookInfo application seen without the service proxies.

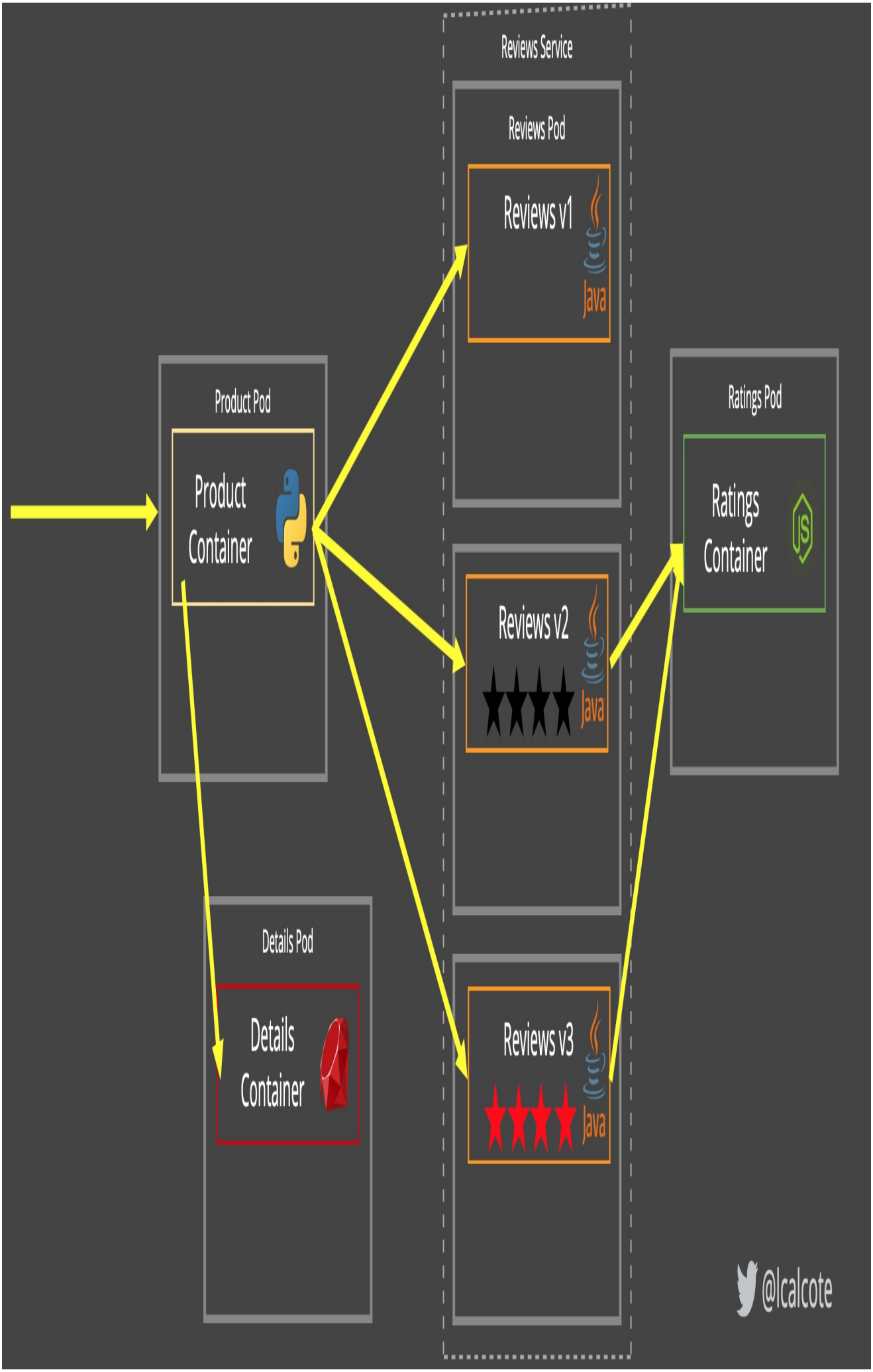

Reviews Service
Reviews Pod
Reviews v1
Java
Product Pod
Product
Container
Ratings Pod
Ratings
Container
JS
Reviews v2
Java
Details Pod
Details
Container
Reviews v3
Java
@lcalcote

Figure 2-3. Figure 3-3 Istio's sample application, BookInfo, shown without service proxies

In Kubernetes, automatic proxy injection is implemented as a webhook using a Kubernetes Mutating Webhook. It is stateless depending only on the injection template and mesh configuration configmaps as well as the to-be-injected pod object. As such, it can be easily horizontally scaled, either manually via the deployment object, or automatically via a Horizontal Pod Autoscaler.

Injection of the sidecar proxy into a newly created pod takes an average 1.5us (per micro benchmark) for the webhook itself to execute. Total injection time will be higher when accounting for network latency and api-server processing time.

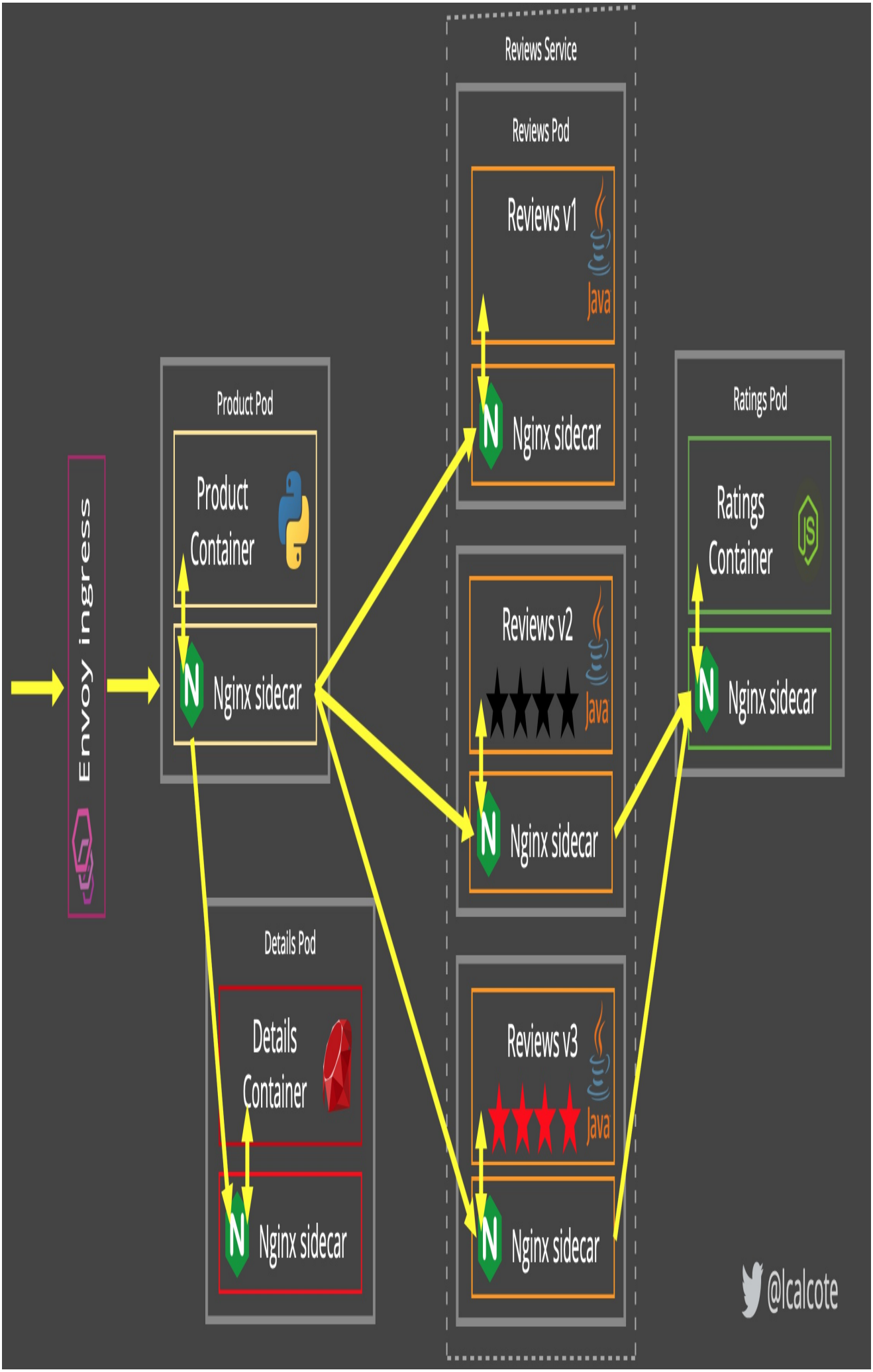

Reviews Service
Reviews Pod
Reviews v1
java
Nginx sidecar
Product Pod
Product
Container
Nginx sidecar
Ratings Pod
Ratings
Container
Nginx sidecar
Envoy ingress
Reviews v2
java
Nginx sidecar
Details Pod
Details
Container
Nginx sidecar
Reviews v3
java
Nginx sidecar
@lcalcote

Figure 2-4. Figure 3-4 Istio's sample application, BookInfo, shown with service proxies.

Istio strikes at the heart of one well-known distributed systems challenges of not having homogeneous, reliable, unchanging networks. It does so through deployment of lightweight proxies deployed between your application containers and the network. In this way,

As you can see in Figure 3-x, the full architecture of Istio includes the control and data planes with each of their internal components. A full mesh deployment also includes ingress and egress service gateways.

GATEWAYS:

Istio 0.8 introduced the concept of ingress and egress gateways. Symmetrically similar, ingress and egress gateways act as reverse and forward proxies, respectively, for traffic entering and exiting the mesh. Like other Istio components, the behavior of Istio Gateways is defined and controlled through configuration giving you explicit control over which traffic to allow in and out of the service mesh, at what rate, where it should, and so on.

Configuration includes a set of ports that should be exposed, the type of protocol to use, Server Name Indicator (SNI) and so on.

Ingress

Configuration of ingress gateways allow you to define traffic entryways into the service mesh for incoming traffic to flow through. Consider that ingressing traffic into the mesh is a reverse proxy situation - akin to traditional web server load balancing. Configuration for egressing traffic out of the mesh is a forward proxy situation (akin to traditional) where you will identify which traffic to allow out of the mesh and where to route it.

For example, the following Gateway configuration sets up a proxy to act as a load balancer exposing port 80 and 9080 (http), 443 (https), and port 2379 (TCP) for ingress. The gateway will be applied to the proxy running on a pod with labels app: my-gateway-controller. While Istio will configure the proxy to listen on these ports, it

is the responsibility of the user to ensure that external traffic to these ports are allowed into the mesh.[1]

```
apiVersion: networking.istio.io/v1alpha3

kind: Gateway

metadata:

name: my-gateway

spec:

selector:

app: my-gatweway-controller

servers:

- port:

number: 80

name: http

protocol: HTTP

hosts:

- uk.bookinfo.com

- eu.bookinfo.com

tls:

httpsRedirect: true # sends 301 redirect for http
requests
```

```
- port:

number: 443

name: https

protocol: HTTPS

hosts:

- uk.bookinfo.com

- eu.bookinfo.com

tls:

mode: SIMPLE #enables HTTPS on this port

serverCertificate: /etc/certs/servercert.pem

privateKey: /etc/certs/privatekey.pem

- port:

number: 9080

name: http-wildcard

protocol: HTTP

hosts:

- "*"

- port:

number: 2379 # to expose internal service via external
```

```
port 2379

name: mongo

protocol: MONGO

hosts:

- "*"
```

Egress

Traffic can egress an Istio mesh in two ways - directly from the sidecar or funnelled through and egress gateway.

> **TIP**
>
> By default, Istio-enabled applications are unable to access URLs outside the cluster.

Direct from service proxy

If you want traffic destined to an external source to bypass the egress gateway, you may provide configuration to the ConfigMap of the istio-sidecar-injector. Set the following configuration in the sidecar injector, which will identify cluster-local networks and keep traffic destined locally within the mesh, while forwarding traffic for all other destinations externally.

```
--set global.proxy.includeIPRanges="10.0.0.1/24"
```

Once applied and istio proxies updated with this configuration, external requests bypass the sidecar and route directly to the intended destination. The Istio sidecar will only intercept and manage internal requests within the cluster.

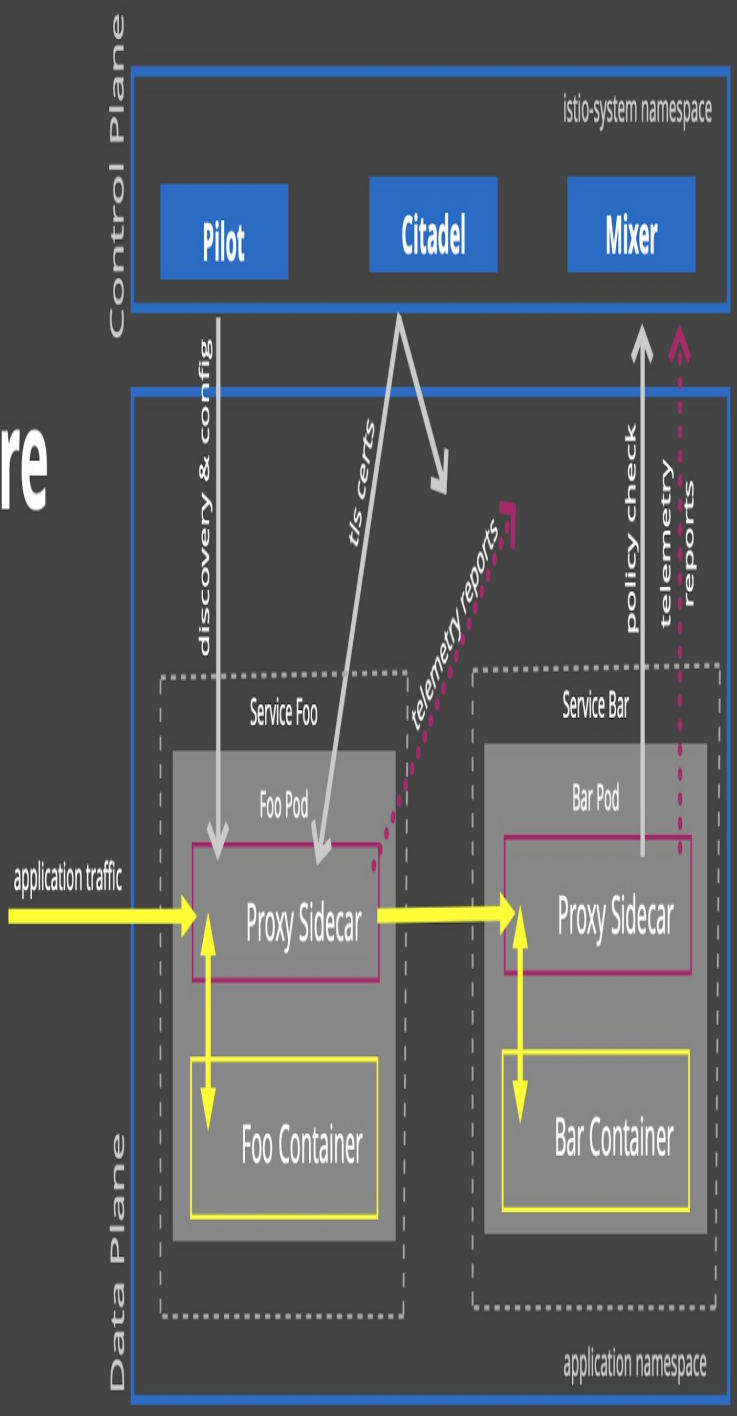
Route through an egress Gateway

You may need to use an egress gateway in order to provide connectivity from your cluster’s private IP address space, monitoring or cross-cluster connectivity. We will

discuss cross-cluster as an advanced topology in Chapter 14.

An egress gateway allows Istio monitoring and route rules to be applied to traffic exiting the mesh. It also facilitates communication between applications running in a cluster where the nodes do not have public IP addresses, preventing applications in the mesh from accessing the Internet. Defining an egress gateway, directing all the egress traffic through it and allocating public IPs to the egress gateway nodes allows the nodes (and applications running on them) to access external services in a controlled way.

Istio Architecture



Out-of-band telemetry propagation

Control flow during request processing

Application traffic

@lcalcote

Figure 2-5. Figure 3-5 Istio's architecture and all components

Why use Istio Gateways and not Kubernetes Ingresses? In general, the Istio v1alpha3 APIs leverage Gateways for richer functionality as Kubernetes Ingress has proven insufficient for Istio applications.

One consideration compared to Kubernetes Ingress is that Istio Gateways can operate as a pure L4 TCP proxy and supports all protocols that Envoy supports.

Another consideration is the separation of trust domains between organizational teams. Kubernetes Ingress API merges specification for L4 to L7, which makes it difficult for different teams in organizations with separate trust domains (like SecOps, NetOps, ClusterOps and Developers) to own Ingress traffic management.

Extensibility

While not an explicit goal for some service meshes, Istio is designed to be customized. As an extensible platform, its integrations comes in two primary forms - swappable sidecar proxies and telemetry / authorization adapters.

CUSTOMIZABLE SIDECARS

Within Istio, though Envoy is the default service proxy sidecar, you may choose another service proxy for your sidecar. While there are multiple service proxies in the ecosystem, outside of Envoy, only two have currently demonstrated integration with Istio: Linkerd and Nginx. Conduit is not currently designed as a general-purpose proxy, rather focused on being lightweight, placing extensibility as a secondary concern by offering extension via gRPC plugin.

While it's more likely that you would choose to run a different service mesh altogether, you many want to use Istio with an alternative service proxy for the following reasons:

- Linkerd: If you're already running Linkerd and want to start adopting Istio control APIs like CheckRequest, which is used to get a thumbs-up/thumbs-down before performing an action.
- Nginx: Based on your operational expertise and need for battle-tested proxy, you may select Nginx. You may be looking for caching, WAF or other

functionality available in Nginx Plus as well.

- Connect: Choose to deploy Connect based on ease of deployment and simplicity of needs.

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 the ability to use Nginx as a service proxy through the nginMesh project was provided early in Istio release cycle.

While the nginMesh project is currently paused, you may 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.

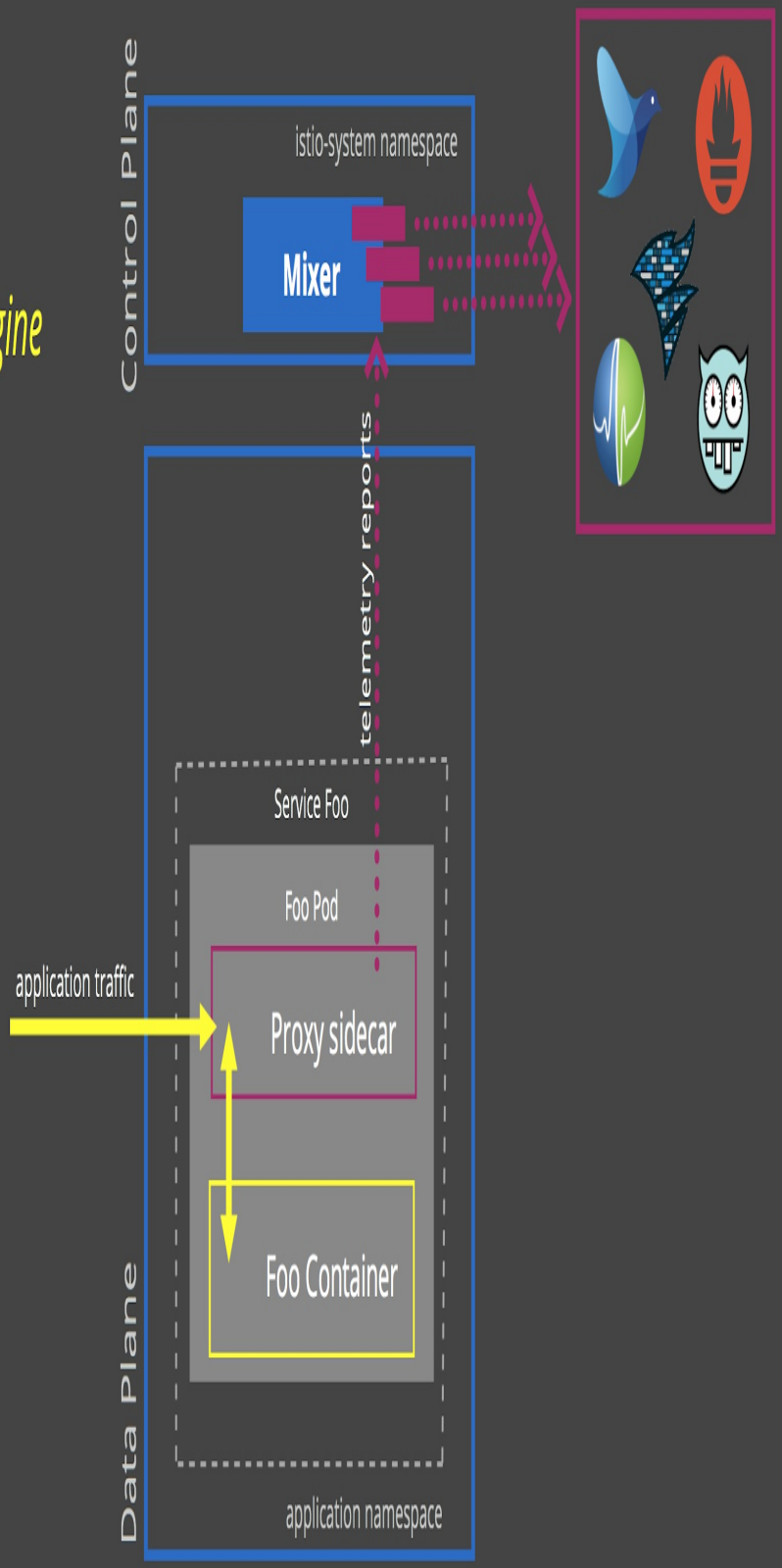
Without configuration, proxies are without instructions to perform their tasks. Pilot is the head of the ship in an Istio mesh, so to speak, keeping in-sync with the underlying platform by tracking and representing its services to istio-proxy. As the default proxy, -proxy contains the proxy of choice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

EXTENSIBLE ADAPTERS

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

Mixer

an attribute processing engine



layer5.io

Out-of-band telemetry propagation

Control flow during request processing

application traffic

@lcalcote

Figure 2-6. Figure 3-6 Istio Mixer is an example of an extension point in a service mesh. Mixer acts as an attribute processing engine, collecting, transforming and transmitting telemetry.

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

Scale and Performance

Like many, you may have been saying to yourself, "These features are great, but what's the overhead of running a service mesh?". It's true that these features come at a cost. Running a proxy per service and intercepting each packet takes a certain amount of continual overhead. Costs can be analyzed by data and control planes.

Answers to performance questions always start with "it depends". And so, depending upon whether your using all of Istio features, recommendations on resources needed vary. Current recommendations when using all Istio features are[2]:

1 vCPU per peak thousand requests per second for the sidecar(s) with access logging (which is on by default) and 0.5 without, fluentd on the node is a big contributor to that cost as it captures and uploads logs.

Assuming typical cache hit ratio (>80%) for mixer checks: 0.5 vCPU per peak thousand requests per second for the mixer pods.

Latency cost/overhead is approximately 10 millisecond for service-to-service (2 proxies involved, mixer telemetry and checks) as of 0.7.1, we expect to bring this down to a low single digit ms.

Mutual TLS costs are negligible on AES-NI capable hardware in terms of both CPU and latency.

DATA PLANE

The overhead of the service proxy is a common question in the minds of those adopting a service mesh. Given that Envoy has first-class support for HTTP/2 and SSL in either direction, single, long-lived TCP connections are leveraged by multiplexing requests

and responses across it. In this way, HTTP/2 achieves lower latency than its predecessor.

Maintainers of Envoy understand that this service proxy is in the critical path and have worked to tune its performance. While the project maintainers do not currently publish any official benchmarks, they encourage users to benchmark Envoy in their own environments with a configuration similar to what you plan on using in production. To fill this void, tools like Meshery have cropped up within the open source community.

Meshery is a multi-mesh performance benchmark and playground. As a performance benchmark, Meshery will provision different service meshes, sample applications or point its load generator to your service endpoints, collect and display performance metrics of the mesh and nodes in the cluster. It also acts as a multi-mesh playground to facilitate learning about functionality of different service meshes. Meshery incorporates a visual interface for manipulating traffic routing rules.

Without leveraging such a tool, to maximize performance of the service proxies, you can install an Envoy on client and server and manually configure them. Then, tune Envoy's configuration to ensure maximum performance.

CONTROL PLANE

As your service proxy (Envoy) fleet grows, the central role the control plane plays can become a source of bottlenecking. configuration in a control plane that implements Envoy's xDS APIs. Pilot produces xDS configurations and Envoy polls for these.

Fortio is a fast, small, reusable, embeddable go library as well as a command line tool and server process, the server includes a simple web UI and graphical representation of the results (both a single latency graph and a multiple results comparative min, max, average and percentiles graphs).

Fortio is also 100% open-source and with no external dependencies beside go and gRPC so you can reproduce all our results easily and add your own variants or scenarios you are interested in exploring. Fortio is complemented by other load-generation tools commonly used for service mesh performance testing.

Deployments

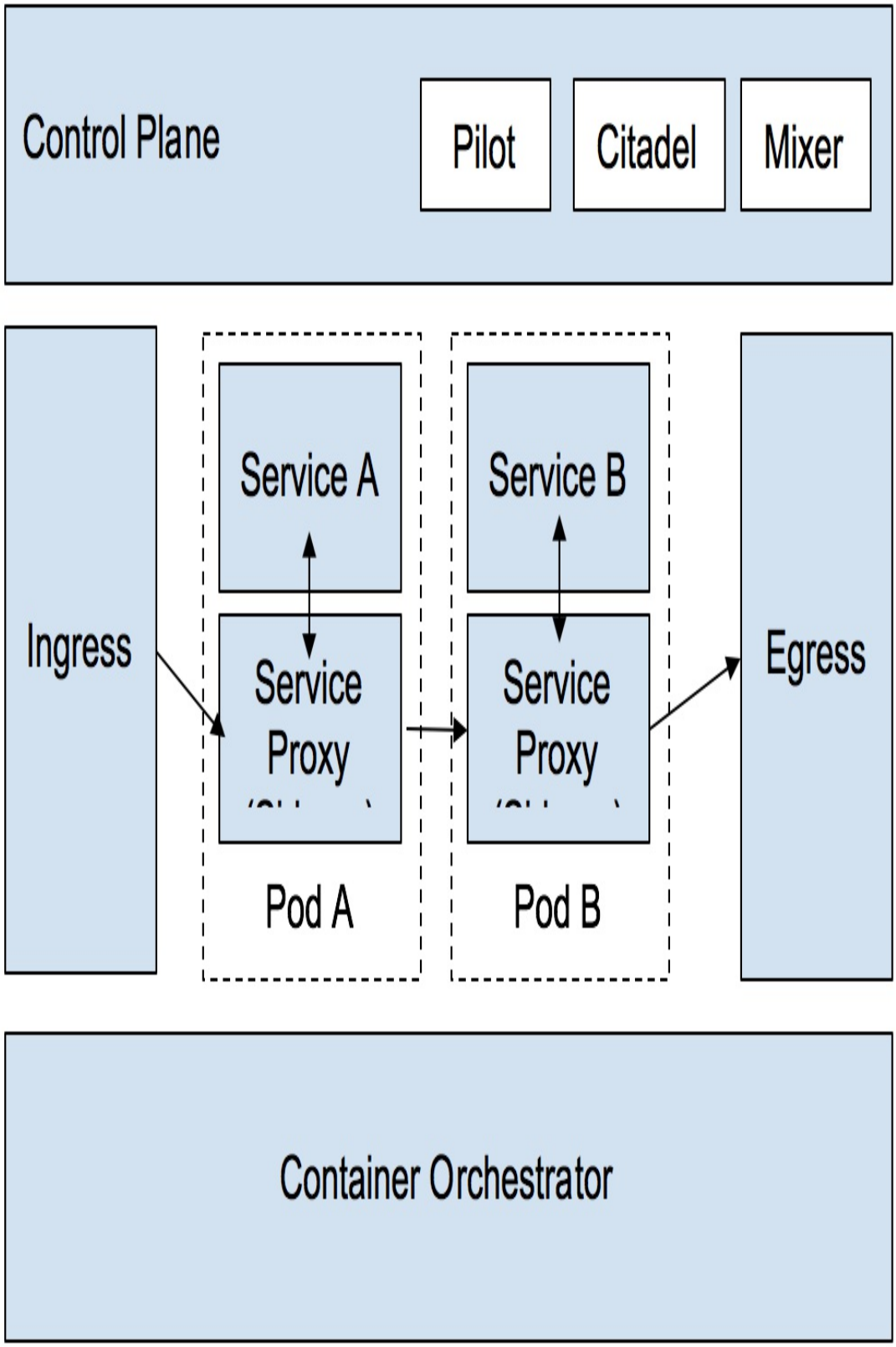


Figure 3-x - Istio shown as deployed on Kubernetes

Figure 2-7. Figure 3-7 Istio shown as deployed on Kubernetes

Conclusion

You've come to learn more about Istio, a service mesh, because you're ready to layer up - you're ready for enhanced service management. Istio aims to directly address service management challenges by providing a new layer of visibility, security and control.

The Istio control plane provides a single point of visibility and control, while the data plane facilitates traffic routing.

Istio is an example of a service mesh designed with customizability in mind.

Decoupling of control and unification of responsibility over microservices as a new layer of infrastructure, layer 5, avoids finger-pointing between development and operations teams.

---

1 https://istio.io/docs/reference/config/istio.networking.v1alpha3/#Gateway

2 https://istio.io/docs/concepts/performance-and-scalability/

Chapter 3. Security and Identity

Overview

Application security has been focused on the network for a long time. We build hard outer shells (firewalls, VPNs, etc) to prevent attackers, but once the outer shell is penetrated it can be easy for an attacker to access lots of systems. So we build defence-in-depth, and apply these networking isolation concepts within our own trust domain as well, requiring security admins to punch holes in network policy and set things up *just so*: these network identities (IP addresses) assigned here, with access there, through these ports, etc, just so our own applications can talk to each other. This network approach to security works well when the rate of change of the system is low: when changes happen over the course of days it's easy to take manual steps, or automate, the setup and maintenance of these networks.

The rate of change of container based systems isn't over the course days, changes occur by the second. In this highly dynamic environment, this traditional network security model starts to break down. The key problem is that traditional network security puts the emphasis on the only identity available to the network: an IP address. An IP address is not a strong indication of the application, and because dynamic environments like Kubernetes can freely reuse IP addresses for different workloads over time, they're not sufficient to use for policy or security.

To help address this problem, one of Istio's key features is the ability to issue an identity to every workload in the service mesh. These identities are tied to the workload, not to some particular host (or some particular network identity). This means you can write policy about service to service communication that's robust to changes in deployment and the topology of the system, not bound to the network.

In this chapter we'll talk through the concepts of Identity, Authorization, and Authentication as they relate to service to service communication and dig in to how Istio provides identity to workloads, performs authentication of those identities at runtime, and uses them to authorize service to service communication.

Authentication and Authorization

The fundamental question that access control systems answer is: "can **You** perform **Action** on **Object**?" where **You** is the *identity* in question, which we call the *principal*; **Action** is some operation the system defines, e.g. the Unix filesystem defines the actions "read", "write", and "execute" on files; and **Object** is the thing being acted upon by the principal. Authentication is all about the principal: it's the process of taking some credential and checking that the credential is valid (i.e. that the credential is authentic), ensuring that the **You** in the access control question contains the identity that is in fact you. Authorization is all about what actions **You** can and cannot perform on **Object**.

Authentication (abbreviated *authn*, pronounced "auth-in") is the act of taking a credential from a request and ensuring it's authentic. In Istio's case, the credential that services use when they communicate with each other is an X.509 certificate.The sidecars are able to authenticate the identity of the calling service (and the client is able to authenticate the server) by validating the X.509 certificate provided by the other party using the normal certificate validation process. If the certificate is valid, then the identity encoded within the certificate is considered authenticated. Once we've performed authentication we say that the principal is *authenticated*, calling it an *authenticated principal* to differentiate it from a *principal*, which may or may not be authenticated.

Authorization (abbreviated *authz*, pronounced "auth-zee") is the act of answering the question "can **You** perform **Action** on **Object**?". For example, to run a shell script on Unix, the system checks that the current user has the execute permission on the script file. That user already authenticated themselves by logging in. In Istio, authorization of service to service communication is configured with Role Based Access Control (RBAC) policies, which we'll cover in detail later in the chapter.

> **NOTE**
>
> We'll also use the abbreviation "auth" to refer to authn and authz collectively.

When we think about the access control question, "can **You** perform **Action** on **Object**?", it becomes clear that both authentication *and* authorization are required and one without the other is useless. If we only authenticate credentials then any user can perform any action on any object; all we've done is assert they're them while they do it! Similarly, if we only authorize requests then any user can pretend to be any other user and perform actions on that user's objects; all we've done is make sure *someone* has permission to perform the action in question, not necessarily the caller! One final thing to note: the question Istio auth answers is a little more specific than "can **You** perform **Action** on **Object**?"; instead, Istio answers the question: "can **ServiceA** perform **Action** on **ServiceB**?". In other words, both **You** and **Object** are identities of services in the mesh.

With the concepts of authentication and authorization in hand, the natural next questions are: what are the identities, the principals, that Istio gives to services; how does the system manage those identities at runtime; how do you write policy about the actions one service can perform on another; and how does Istio enforce those policies at runtime? We'll answer each through the rest of the chapter.

Identity

People are continually trying to define what a service mesh is, what makes something a mesh and something else not a mesh, and what the boundary of a mesh is. Typically answers are about the "administrative domain" - "all the stuff that's configured by one operator is one mesh"; or about "communication" - "everything that can communicate is part of the same mesh". Both are popular answers. In our opinion, these types of answers fall short. For example, multiple teams may administer different segments of a mesh and different parts of a mesh may not in fact be allowed to communicate. Instead, we believe the best answer is "a service mesh is a single identity domain." In other words, a single namespace that every service in the system is allocated an identity from. Identity is perhaps the most fundamental feature a service mesh provides. Everything else in the system builds on top of identity. Communication stems from identity: without knowing who you're talking to it is impossible to communicate meaningfully. Telemetry stems from identity: without knowing what you're metering, metrics are useless data. Configuring the flow of traffic stems from

Istio implements the *Secure Production Identity Framework for Everyone* (SPIFFE) spec to issue identities. In short, Istio creates an X.509 certificate which sets the certificate's Subject Alternative Name (SAN) to a URI that describes the service. Istio defers to the platform for identity. In a Kubernetes deployment Istio uses a pod's service account as the pod's identity, encoding it into a URI like: "spiffe://**ClusterName**/ns/**Namespace**/sa/**ServiceAccountName**".

Note that in Kubernetes a pod will use the "default" service account for the namespace it's deployed in if the ServiceAccount field is not set in the pod (or deployment) resource. This means all services in the same namespace will share a single identity if service accounts aren't already set up for each service.

SPIFFE

SPIFFE, Secure Production Identity Framework for Everyone, is a specification for a framework that can bootstrap and issue identities. SPIRE is the SPIFFE community's reference implementation, and Citadel - Istio auth - is a second implementation. The SPIFFE spec describes three concepts:

An Identity, as a URI, used by services to communicate

A standard encoding of that Identity into a SPIFFE Verifiable Identity Document (SVID)

An API for issuing and retrieving SVIDs (the Workload API)

SPIFFE requires that a service's identity be encoded as a URI with the scheme "spiffe", like: "spiffe://trust-domain/path". The trust domain is the root of trust of the identity, e.g. an organization, an environment, a team, etc. The trust domain is the URI's authority field (specifically the host section of the authority). The spec allows the path section of the URI to be anything - a UUID, a trust hierarchy, or nothing at all. On Kubernetes, Istio encodes a service's ServiceAccount using the local cluster's name as the trust domain, and creates a path using the service account's name and namespace. For example, the default service account is encoded as "spiffe://cluster.local/ns/default/sa/default" ("ns" for namespace, "sa" for service account).

SPIFFE also describes how to encode this identity into an X.509 SPIFFE Verifiable Identity Document (SVID) - an X.509 certificate that can be verified to prove the identity. The spec says the Identity URI is encoded as the certificate's Subject Alternative Name (SAN) field. Then to validate the SVID, first we perform normal X.509 validation, then we validate that the certificate is *not* a signing certificate (signing certificates cannot be used for identification per the spec) and that there is exactly one SAN in the cert with the "spiffe" scheme.

SPIFFE defines a Workload API, an API for issuing and retrieving SVIDs. This is a key point where Istio diverges from SPIFFE: Istio implements certificate provisioning using ACME instead. Both APIs accomplish a similar goal (prove some information about the workload to receive an identity for it) but ACME is a more widely used protocol and has a better defined security model.

Finally, while the SPIFFE spec does not require it, both SPIRE and Istio issue X.509 SVIDs that are short lived - they expire on the order of an hour after issuance. This is in contrast with traditional usage of X.509 certificates, which tend to be used for HTTPS TLS termination and frequently expire a year or more after issuance. The benefit of this is bounding attacks in time without requiring certificate revocation (and making revocation easy if you do choose to use it). Suppose an attacker compromises a workload and steals the workload's SVID; it's only valid across the rest of the trust domain for a short time. If their attack requires an extended period to execute, they must continually extract a valid credential from the workload. Once you become aware of the attack you can use policy to prohibit that identity from accessing other services, you can stop re-issuing certificates for that identity, and you could even put that certificate into a revocation list. Because the certificates are ephemeral, managing that revocation list is easy - it's a standard practice to remove expired certificates from a CRL. The list stays small because the certs expire quickly.

This use of short lived certificates comes with one big disadvantage, though: it's hard to issue and rotate certificates on every workload across the fleet at a short interval. We'll talk about how Istio solves this problem in the next section.

Key Management Architecture

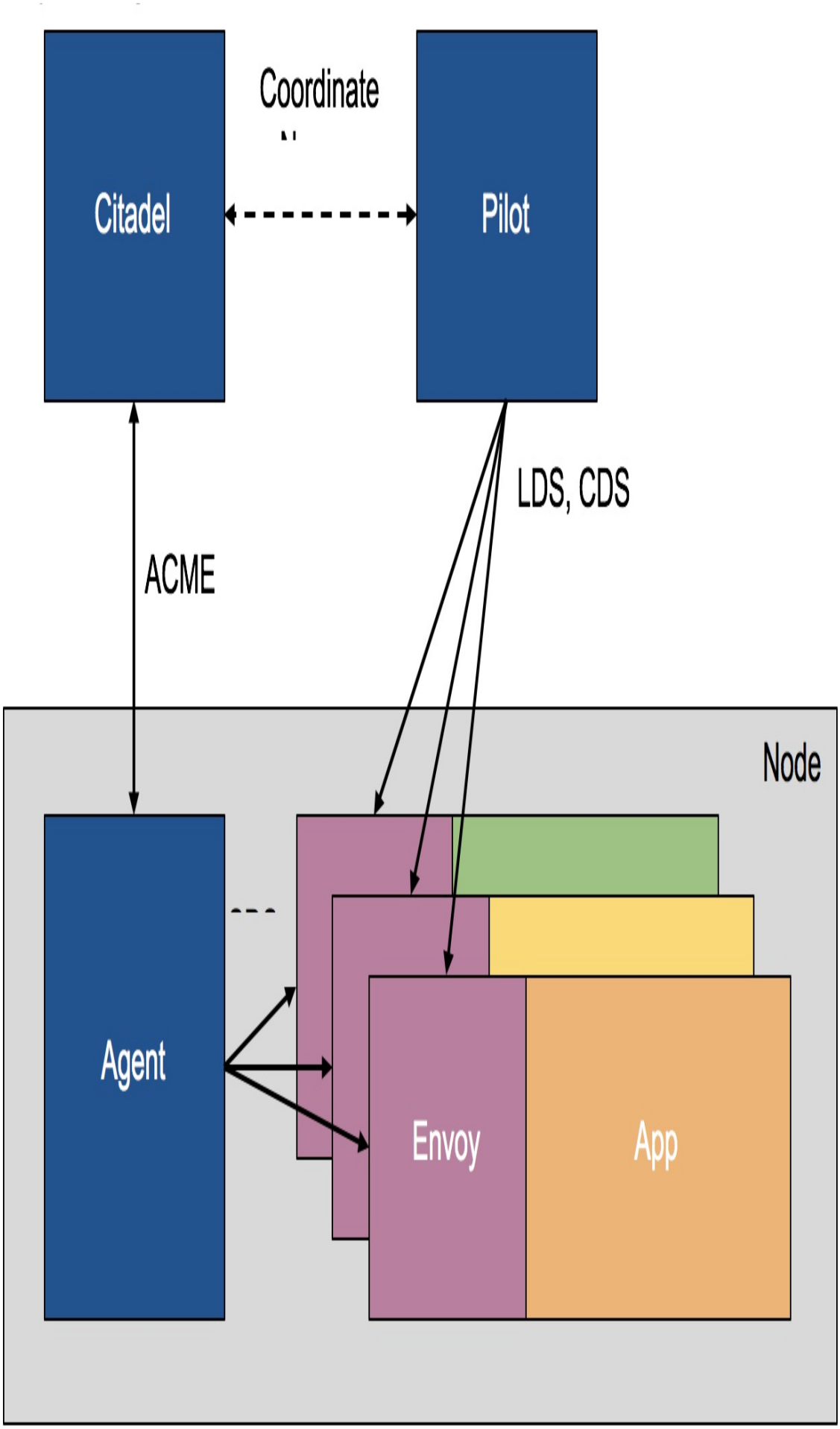


Figure 3-1.

Three components participate together to issue and rotate SVIDs across an Istio deployment:

- Citadel, which issues identities to workloads across the deployment by acting as a CA, signing certificate requests which form X.509 SVIDs.
- The Node Agent, a trusted agent deployed on each node. This agent acts as a broker between Citadel and the Envoy sidecars deployed on the node.
- Envoy, which speaks to the Node Agent locally to retrieve an identity and presents that identity to other parties at runtime.

CITADEL

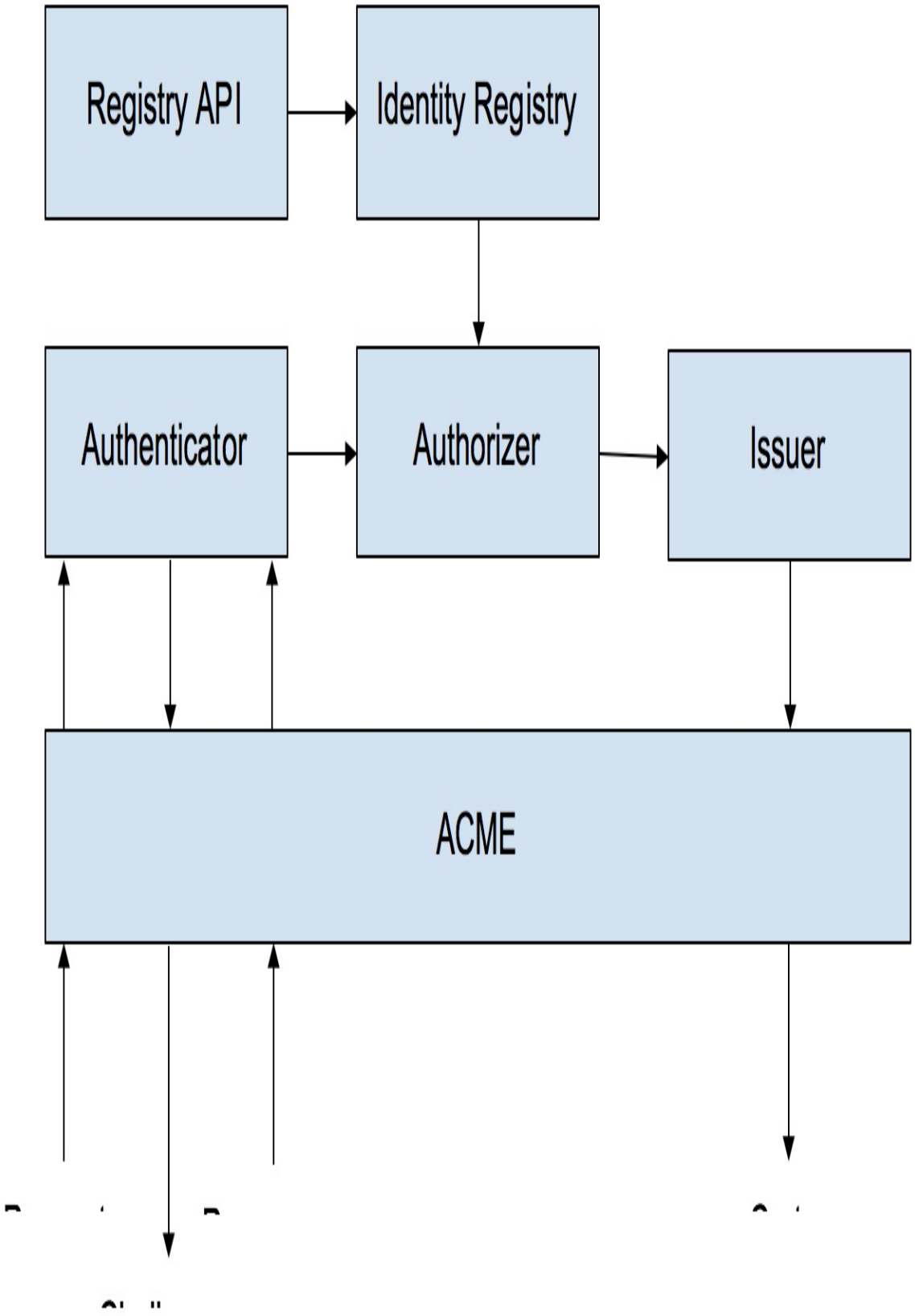


Figure 3-2.

Citadel is responsible for accepting requests for identity, authenticating them, authorizing them, and ultimately issuing a certificate for the identity in question. Citadel itself is composed of several logical component. From bottom to top, left to right in the diagram above:

Citadel exposes ACME as its public API to identity requestors. To request an identity, a caller performs ACME, ultimately sending a *Certificate Signing Request* (CSR) to Citadel, which Citadel signs, transforming the CSR into a certificate (an X.509 SVID).

Upon receiving a request, the request is fed to an *authenticator*, which generates a set of challenges. The requestor much satisfy these challenges to authenticate itself.

Upon satisfying all required challenges, the request is authenticated and the *authorizer* determines if the requested identity is valid for the authenticated principal to receive. The authorizer consults an Identity Registry, which maps workloads via their authenticated principal to identities, to perform authorization.

After a workload is authorized to receive an identity, we need to actually issue it by signing a certificate. The authorizer calls the *issuer* to generate a certificate and make it available to the requestor. Issuers in Citadel today include an in-memory certificate authority as well as Vault.

ACME, the Automated Certificate Management Environment protocol, is a certificate issuance protocol developed by Let's Encrypt to make it easy to programmatically provision certificates. Istio implements this protocol and extends it with a custom set of *Challenges* for verifying service identity. (ACME's specification allows for the creation of custom challenges to prove different types of identity. Due ACME's origin in the Let's Encrypt's project, which focuses on certificates for HTTPS, ACME's default challenges are focused only on issuing certificates for DNS hostnames.)

NODE AGENT

The Node Agent is deployed on every node that has workloads that Citadel will issue identities to. It has two responsibilities. First, it acts a simple protocol adapter between Envoy and Citadel. Envoy consumes a SecretDiscoveryService (SDS) API which configures the secrets Envoy will serve at runtime. This API is great for issuing key material - the certs themselves - but does not support verification. In other words, Citadel cannot use the SDS API to authenticate ownership of an identity. Instead Citadel uses ACME to authenticate requests, as we described in the previous section. The Node Agent bridges SDS and ACME on behalf of the workloads deployed on the

node. This brings us to the second key responsibility of the Node Agent: to be a trusted agent on the node able to validate a workload's environment for Citadel on behalf of the workload. When Citadel issues ACME challenges to authenticate a request for an identity, some of those challenges pertain to the workload itself, but other challenges are about the environment in which a workload is running. We wouldn't want to issue a production identity to a service running on a developer's laptop, afterall! The Node Agent is trusted to be able to accurately provide information about the environment its deployed in (we call this *environmental attestation*). In the case of a deployment on Kubernetes, the node agent is configured with nearly all of the information required to prove challenges during deployment.

The Node Agent keeps the secrets its retrieved from Citadel in memory, and when they near expiration (say, have 25% of their TTL left) the agent will contact Citadel and attempt to refresh the certificate. We leave a little wiggle room in case Citadel is temporarily inaccessible. When the Node Agent dies and is restarted by the orchestrator it will attempt to retrieve fresh credentials for all workloads on the node; in this way it remains stateless. Upon receiving a fresh SVID from Citadel, the Node Agent will push it to Envoy via SDS. This triggers Envoy to establish new connections to destination workloads; Envoy then drains the current connections into the new connections and terminates the connections which were using the old (and now potentially expired) certs[1].

Relying on the Node Agent in this way is justified in Istio's security model by the fact that the agent is already handling all secrets on the node to begin with; it *must* operate at a higher level of trust than other components on the node for exactly that reason. Therefore, extending that trust to additionally force the agent to validate the execution environment is reasonable. Even if we had Envoy communicate directly with Citadel, we could not trust environmental answers that Envoy provides because it runs in the same trust domain as the application itself (so an attacker compromising the application could just as easily confuse Envoy into giving different or wrong answers to environmental attestation challenges).

Finally, this architectural decision is to have workloads communicate with the local Node Agent and have that Node Agent communicate with Citadel is important for

keeping Citadel scalable. It forces the number of connections to Citadel instances in the system to grow with the number of nodes in the deployment, not the number of workloads. In environments like Kubernetes, there can be a substantially larger number of workloads than nodes, so this provides real benefits.

PILOT

Pilot also plays a minor role in key management. When Pilot pushes configuration to Envoys, including configuration about destination services and how to receive traffic, it has to reference certificates. It references these certs by name, therefore it has to coordinate with SDS, which is provided by the Node Agent! It would not be desirable to force the Node Agents to talk to Pilot in addition to Citadel. Instead, Istio components plan on a common naming scheme for secrets ahead of time so that Pilot can unambiguously refer to secrets provisioned by SDS. Further, the primary certificate on all Envoys, the identity SVID, resides at a well-known location (in /etc/certs/).

ENVOY

The final participant in this whole dance is the Envoy sidecar for the application. Envoy is configured to communicate with the local Node Agent as its source for the SDS API ahead of time by resolving some address to the local node, or more commonly by communicating with the Node Agent over Unix Domain Sockets (UDS). This configuration, where to locate the SDS server, can be served to Envoy by Pilot dynamically, but we typically configure this information statically via the deployment system as it's less error-prone at runtime[2].

With the SVID in hand, Envoy uses that certificate when it initiates connections to other services in the mesh. Two workloads in the mesh will perform *mutual TLS* (mTLS) when they communicate, which assures both the client and server of the identity of the other party, allows both to perform authentication and authorization of the communication being initiated, and provides for encryption in transit. mTLS alone isn't enough though, since we still need something to perform authorization of communication between two identities! We'll cover both mTLS and authorization of communication in the remainder of the chapter.

Mutual TLS

With all of the identity certificates (SVIDs) distributed to workloads across the system, how do we actually use them to verify the identity of the servers we're talking to and perform authentication and authorization? This is where mutual TLS comes in to play. First, though, a bit of background.

When we think of TLS or SSL (TLS is the new version of SSL), the use case that typically comes to mind is HTTPS. A user wants to use their browser to connect to some web server, e.g. wikipedia.org. Their browser (or the operating system) will perform a DNS lookup to determine an IP address for wikipedia.org, and the browser will fire off an HTTPS request to that address and wait for a response from the server. When the client attempts to connect to the server, the server responds by presenting a of certificate with the identity "wikipedia.org" signed by some root of trust that the client trusts; the client validates that certificate, authenticating the identity of the server and allowing the connection to be established. Then a set of keys are generated for the connection to enable encryption of the data sent by both the client and the server. In other words, TLS is how the client knows it can trust the server, that the server really is controlled by wikipedia.org, and that no one is eavesdropping or otherwise tampering with the data sent by the server.

Mutual TLS is TLS where both parties, client and server, present certificates to the each other. This allows the client to verify the identity of the server, like normal TLS, but also allows the server to verify the identity of the client attempting to establish the connection. We use mutual TLS in Istio, where both parties present their SVID to each other. This allows both parties to authenticate the SVID provided by the other and to perform authorization on the connection. (In practice in Istio we only perform authorization on the server side; this makes sense given how you write authorization policy, which we'll cover in the next section.)

Configuring Istio Auth Policies

Istio splits authentication and authorization policy into two sets of configuration. The first, authentication policy, controls how proxies in the mesh communicate with each other (whether they require an SVID or not). The second, Authorization Policy, requires authentication policy first and configures which identities are allowed to communicate.

AUTHENTICATION POLICY: CONFIGURING MUTUAL TLS

Adopting mTLS into an existing deployment is challenging since you need to make sure both client and server are provisioned with certificates at the same time (traditional TLS is far simpler since it only required coordinating the server deployment). As a result, Istio provides a few knobs to take a deployment that's not using mTLS and gradually enable it without causing outages for clients.

Policy (authentication.istio.io/v1alpha1.Policy) is the primary CRD we use to configure how services in the mesh communicate with each other. It allows us to require, make optional, or disable mTLS on a service-by-service, namespace-by-namespace basis. A cluster scoped variant, MeshPolicy, applies a default Policy to every namespace and service in the mesh.

To enable mTLS for a single service we create a Policy in that service's namespace with that service as the target, requiring mTLS:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
name: foo-require-mtls
namespace: default
spec:
targets:
- name: foo.default.svc.cluster.local
peers:
- mtls:
mode: STRICT
```

This Policy applies to the "default" namespace and marks TLS as required for talking to service foo. Note that because the default mTLS configuration *is* strict mode, we can simplify this configuration a bit by omitting the redundant fields:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
name: foo-require-mtls
namespace: default
spec:
```

```
targets:
- name: foo.default.svc.cluster.local
peers:
- mtls: {}
```

Many of the Policy examples on istio.io take this form, omitting the mTLS object since the behavior if the mTLS field is absent is to require STRICT mode.

Of course, just creating this config in a cluster could cause outages if the clients don't already have certificates to perform mTLS with. That's why Istio includes a PERMISSIVE mTLS mode, which allows clients to connect in *either* clear text *or* via mTLS. The following configuration allows clients to contact the bar service using both mTLS and clear text:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
name: bar-optional-mtls
namespace: default
spec:
targets:
- name: bar.default.svc.cluster.local
peers:
- mtls:
mode: PERMISSIVE
```

Similarly, we could make mTLS optional across an entire namespace by omitting the targets field:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
name: default-namespace-optional-mtls
namespace: default
spec:
peers:
- mtls:
mode: PERMISSIV
```

This configuration allows workloads in the mesh to contact any service in the default namespace using either mTLS or clear text. We can also enable or disable mTLS per-

port on a service. This is valuable in contexts like health checking; e.g. in Kubernetes it is not easy to provision the kube master's health checking service with certificates for mTLS, making health checking hard in mTLS enabled systems. By writing two Policy objects, we can exclude the health check port from mTLS, while requiring it for all others, making integration with existing systems easier:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
name: bar-require-mtls-no-port-81
namespace: default
spec:
targets:
- name: bar.default.svc.cluster.local
peers:
- mtls:
mode: STRICT
---
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
name: bar-require-mtls-no-port-81
namespace: default
spec:
targets:
- name: bar.default.svc.cluster.local
port:
name: http-healthcheck
peers:
- mtls:
mode: PERMISSIVE
```

We can use the same principals described above to exclude the "http-healthcheck" port from mTLS across the namespace by omitting the "name" in the target.

To apply the same configuration across all namespaces, we use the MeshPolicy resources. This is identical to the Policy resource in schema, but exists at the cluster level. Also note that the default MeshPolicy *must* be named "default" or Istio will not recognize it correctly.

```
apiVersion: authentication.istio.io/v1alpha1
kind: MeshPolicy
metadata:
name: mesh-wide-optional-mtls
spec:
peers:
- mtls:
mode: PERMISSIVE
```

And of course, we can make mTLS required across the mesh by setting “mode: STRICT” or by omitting the mTLS object entirely:

```
apiVersion: authentication.istio.io/v1alpha1
kind: MeshPolicy
metadata:
name: mesh-wide-mtls
spec:
peers:
- mtls: {}
```

Istio also supports performing end-user authentication via JSON Web Tokens (JWTs). Istio’s authentication policy supports setting a rich set of restrictions about the data in the JWT, allowing you to validate nearly all of the fields of the JWT. The following Policy configures Envoy to require mTLS, but also require end-user credentials stored as a JWT in the “x-goog-iap-jwt-assertion” header, issued by Google (“https://securetoken.google.com”), verified against Google’s public keys (“https://www.googleapis.com/oauth2/v1/certs”), for the audience “bar”:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
name: end-user-auth
namespace: default
spec:
target:
- name: bar
peers:
- mtls: {}
origins:
- jwt:
issuer: "https://securetoken.google.com"
```

```
audiences:
- "bar"
jwksUri: "https://www.googleapis.com/oauth2/v1/certs"
jwt_headers:
- "x-goog-iap-jwt-assertion"
principalBinding: USE_ORIGI
```

AUTHORIZATION POLICY: CONFIGURING WHO CAN TALK TO WHOM

With authentication policy in place, we want to use the identities across the system to control which services can communicate. In other words, we want to describe a service to service communication policy. Istio's authorization policy is described using a Role Based Access Control (RBAC) system. Like most RBAC systems, it defines two objects that are used together to write policy: ServiceRole, which describes a set of actions that can be performed on a set of services by any principal with the role; and ServiceRoleBinding, which assigns roles to a set of principals. In this context the principals are the service identities Istio issues; recall in a Kubernetes deployment these identities are Kubernetes service accounts.

First, we have to create an RBACConfig object which turns on RBAC in Istio:

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: RBACConfig
metadata:
name: default
namespace: istio-system
spec:
mode: ON
```

This configuration enables RBAC to service to service communication across the entire mesh. Like enabling mTLS, this is a potentially dangerous to do in a live system so Istio supports enabling RBAC for service to service communication incrementally by changing the RBACConfig's mode. Istio supports four modes: OFF - no RBAC required for communication (if no RBACConfig object exists, this is the default behavior of the system); ON - RBAC policies are required for communication and communication not allowed by a policy is forbidden; ON_WITH_INCLUSION - RBAC policies are required for communicating with any service in the set of namespaces listed in the policy; and ON_WITH_EXCLUSION - RBAC policies are

required for communicating with any service in the mesh, *except* for services in the set of namespaces listed in the policy.

So to roll out RBAC incrementally across the system, we can enable RBAC in the ON_WITH_INCLUSION mode and as we define policies for each service or namespace we can add that service or namespace to the inclusion list, allowing us to enable RBAC service-by-service (or namespace-by-namespace):

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: RBACConfig
metadata:
name: default
namespace: istio-system
spec:
mode: ON_WITH_INCLUSION
inclusion:
services:
- bar.bar.svc.cluster.local
namespaces:
- default
```

The policy above requires RBAC policies to communicate with any service in the default namespace, and for the bar service in the foo namespace. No other communication requires a policy defined. At some point, more namespaces and services in our system will have RBAC policies than those without; at that point we can swap to an ON_WITH_EXCLUSION policy.

With RBAC enabled, we need to write policies. We start by picking a namespace or service and describing the roles that exist for that service. For our example, we'll create a ServiceRole that allows read access (HTTP GET requests) to the foo service:

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
name: foo-viewer
namespace: default
spec:
rules:
- services:
```

```
- foo.default.cluster.local
methods:
- GET
```

We can then use a ServiceRoleBinding to assign that role to the service account that the bar service uses, allowing it to call the foo service:

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRoleBinding
metadata:
name: bar-foo-viewer-binding
namespace: default
spec:
subjects:
- properties:
# the SPIFFE ID of the bar service account in the bar
namespace
source.principal: "cluster.local/ns/bar/sa/bar"
roleRef:
kind: ServiceRole
name: "foo-viewer"
```

The ServiceRoleBinding

Summary

---

1 We don't *have* to re-establish connections using the new credentials immediately - after all, the TLS session remains valid so long as it was *initiated* before the certificate expires. An established connection will continue to use an expired certificate happily. We choose re-establish connections to mitigate certain types of credential hijacking attacks. Plus a little jitter tends to be good for a system!

2 Envoy won't overwrite static configuration with configuration from the API, so you can't accidentally push configuration to Envoy that makes it unable to talk to the configuration server.

Chapter 4. Traffic Management

In this chapter we'll discuss traffic management in Istio. We'll start with some background on how requests flow through Istio and cover Istio's networking APIs. Then we'll show how those APIs can be used to configure traffic flow, enabling you to do things like canary new deployments, set timeout and retry policies that are consistent across all of your services, and test your application's failure modes with controllable, repeatable fault injection.

Understanding How Traffic Flows in Istio

To understand how Istio's networking APIs work, it's important to understand how requests actually flow through Istio. Pilot, which we talked about in the previous chapter, understands the topology of the service mesh. It uses that knowledge, alongside additional Istio's networking configuration you provide, to configure the mesh's Envoy sidecars. We cover in detail the kind of configuration Pilot pushes to sidecars in the previous chapter.

At runtime the sidecar Envoy intercepts all incoming and outgoing requests. This interception is done via iptables rules or a BPF program which routes all network traffic, in and out, through Envoy. Envoy inspects the request and uses the request's hostname, SNI, or service virtual IP address to determine the request's *target* - the service the client was trying to send a request to. Then Envoy applies that target's routing rules to determine the request's *destination* - the service the sidecar is actually going to send the request to. With the destination in hand Envoy applies the destination's rules, including a load balancing strategy, to pick an endpoint. This endpoint is the address of a workload in the destination service. Finally, Envoy forwards the intercepted request to the endpoint.

There are a few important things to note here. First, it's desirable to have your applications speak cleartext (communicate without encryption) to the sidecar and let the sidecar handle transport security. For example, your application can speak HTTP to

the sidecar and let the sidecar handle the upgrade to HTTPS. This allows the sidecar to gather L7 metadata about requests, which allows Istio to generate L7 metrics and apply L7 policy. Otherwise Istio can only generate metrics for and apply policy on the L4 parts of the request. Secondly, we get to perform client-side load balancing rather than relying on traditional load balancing via reverse proxies. Client side load balancing means we can establish network connections directly from clients to servers while still maintaining a resilient, well-behaved system. That, in turn, enables more efficient network topologies with fewer hops than traditional systems that depend reverse proxies.

Typically Pilot has detailed endpoint information about services in the registry, which it pushes directly to the sidecars. This means that, unless you configure it to do otherwise, at runtime Envoy selects an endpoint from a static set pushed to it by Pilot and does not perform dynamic address resolution (e.g. via DNS) at runtime. Therefore, the only things Istio can route traffic to are hostnames in Istio's service registry. In the next section we'll discuss *hostnames*, which are the core of Istio's networking model, and how Istio's networking APIs allow you to create hostnames to describe workloads and control how traffic flows to them.

Understanding Istio's Networking APIs

Applications address dependencies via names (e.g. via a hostname resolved via DNS), therefore Istio's network configuration has adopted a *name-centric model*: Gateways expose names, VirtualServices configure and route names, DestinationRules describe how to communicate with the workloads behind a name, and ServiceEntries enable the creation of new names.

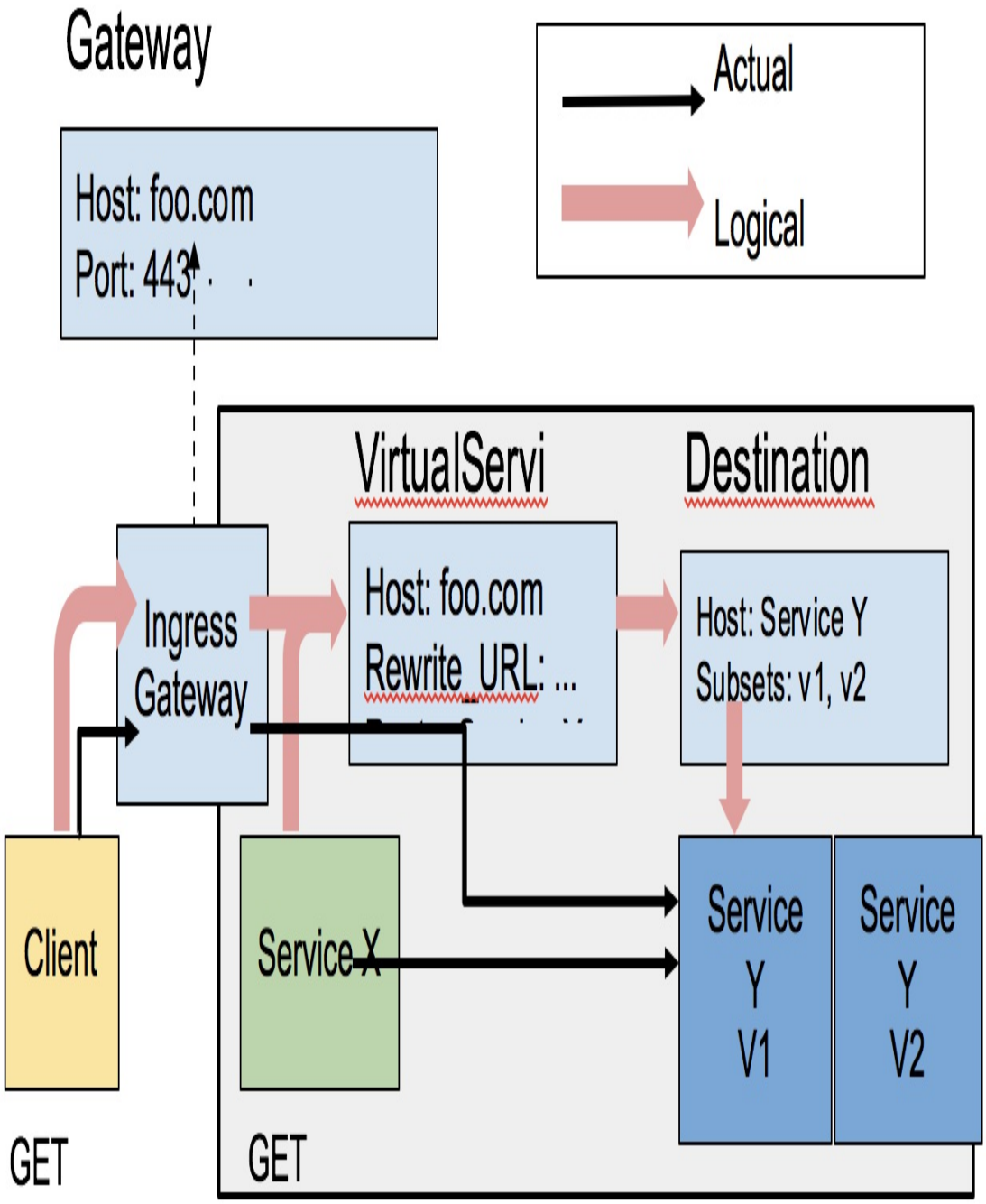


Figure 4-1.

ServiceEntry

ServiceEntries are how you configure Istio's service registry, creating new names. These names can receive traffic and be targeted by other Istio configuration. At their simplest, they can be used to give a name to an IP address:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
```

```
metadata:
name: http-server
spec:
hosts:
- some.domain.com
ports:
- number: 80
name: http
protocol: http
resolution: STATIC
endpoints:
- address: 2.2.2.2
```

Sidecars in the mesh will forward requests to “some.domain.com” to the IP address 2.2.2.2. ServiceEntries can be used to elevate a name in DNS into Istio:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
name: external-svc-dns
spec:
hosts:
- foo.bar.com
location: MESH_EXTERNAL
ports:
- number: 443
name: https
protocol: HTTP
resolution: DNS
endpoints:
- address: baz.com
```

Sidecars in the mesh will forward any requests to “foo.bar.com” to “baz.com”, using DNS to discover endpoints. Because the we declare that the service is outside of the mesh (location: MESH_EXTERNAL) sidecars won’t attempt to use mTLS to communicate with it.

All service registries that Istio integrates with (Kubernetes, Consul, Eureka, etc) work by transforming their data into a ServiceEntry. For example, a Kubernetes Service with one pod (and therefore one endpoint) map directly into a ServiceEntry with a host and an IP address endpoint:

```
apiVersion: v1
kind: Service
metadata:
name: my-service
spec:
selector:
app: MyApp
ports:
- protocol: TCP
port: 80
---
apiVersion: v1
kind: Endpoints
metadata:
name: my-service
subsets:
- addresses:
- ip: 1.2.3.4
ports:
- port: 80
```

Becomes the ServiceEntry:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
name: k8s-my-service
spec:
hosts:
# the names an application can use in k8s to target this
service
- my-service
- my-service.default
- my-service.default.svc.cluster.local
ports:
- number: 80
name: http
protocol: HTTP
resolution: STATIC
endpoints:
- address: 1.2.3.4
```

Note that ServiceEntries created by platform adapters don’t appear directly in Istio’s

configuration (i.e. you cannot `istioctl get` them). It's also important to note that Istio does not populate DNS entries based on ServiceEntries. This means that the above example giving the address 2.2.2.2 the name "some.domain.com" will not allow an application to resolve "some.domain.com" to 2.2.2.2 via DNS. This is a departure from systems like Kubernetes, where declaring a Service also creates DNS entries for that service that an application can use at runtime.

Finally, ServiceEntries can be used to create virtual IP addresses (VIPs), mapping an IP address to a name that you can configure via Isto's other networking APIs:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
name: http-server
spec:
hosts:
- my-tcp-service.internal
addresses:
- 1.2.3.4
ports:
- number: 975
name: tcp
protocol: TCP
resolution: DNS
endpoints:
- address: foo.com
```

This declares 1.2.3.4 as a VIP with the name "my-tcp-service.internal." All traffic to that VIP on port 975 will be forwarded to an IP address for foo.com resolved via DNS. Of course we can configure the endpoints for a VIP just like any other ServiceEntry, deferring to DNS or configuring a set of addresses explicitly. Other Istio configs can use the name "my-tcp-service.internal" to describe traffic for this service. Again though, note that Istio will not set up DNS entries so that "my-tcp-service.internal" resolves to the address "1.2.3.4" for applications; you must configure DNS to do that, or the application must address "1.2.3.4" directly itself.

DestinationRule

DestinationRules, a little counterintuitively, are really all about configuring clients.

They allow a service operator to describe how a client in the mesh should call their service, including: subsets of the service (e.g. v1 and v2), the load balancing strategy the client should use, the conditions to use to mark endpoints of the service as unhealthy, L4 and L7 connection pool settings, and TLS settings for the server. We cover client side load balancing, load balancing strategy, and outlier detection in detail in the Resiliency section of this chapter.

CONNECTION POOL SETTINGS

With DestinationRules we can configure low level connection pool settings like the number of TCP connections allowed to each destination host, the maximum number of outstanding HTTP1, HTTP2, or gRPC requests allowed to each destination host, and the maximum number of retries that can be outstanding across all of the destination's endpoints.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
name: foo-default
spec:
host: foo.default.svc.cluster.local
trafficPolicy:
connectionPool:
tcp:
maxConnections: 4
http:
http2MaxRequests: 1000
```

The above DestinationRule allows a maximum of four TCP connections per destination endpoint, with a maximum of 1000 concurrent HTTP2 requests over the four TCP connections.

TLS SETTINGS

DestinationRules can describe how a sidecar should secure the connection with a destination endpoint. Four modes are supported:

- disabled, which disables TLS for the TCP connection
- simple, which originates a TLS connection to the destination endpoint

- mutual, which establishes a mutual TLS connection to the destination endpoint
- Istio mutual, which is mutual TLS using Istio-provisioned certificates

Enabling mTLS across the mesh via Istio's mesh configuration is a shorthand for setting Istio mutual TLS as the value for all destinations in the mesh.

For example, we can use a DestinationRule to allow connecting to a HTTPS website outside of the mesh:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
name: google.com
spec:
host: "*.google.com"
trafficPolicy:
tls:
mode: SIMPLE
```

Or we can describe connecting to another server with mTLS:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
name: remote-a-ingress
spec:
host: ingress.a.remote.cluster
trafficPolicy:
tls:
mode: MUTUAL
clientCertificate: /etc/certs/remote-cluster-a.pem
privateKey: /etc/certs/client_private_key_cluster_a.pem
caCertificates: /etc/certs/rootcacerts.pem
```

A DestinationRule like the one above together with a ServiceEntry for "ingress.a.remote.cluster" can be used to route traffic across trust domains (e.g. separate clusters) over the internet, securely, with no VPN or other overlay networks. We cover zero-VPN networking and other topics in the Advanced Use Cases chapter of this book.

SUBSETS

Finally, DestinationRules allow you to split a single service into subsets based on labels. All of the features we described above that a DestinationRule allows you to configure can also be configured separately for each subset. For example, we could split a service into two subsets based on version and use a VirtualService to perform a canary traffic to the new version, gradually shifting all traffic to the new version. We'll cover VirtualServices in more detail below, and traffic steering and routing in the next section.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
name: foo-default
spec:
host: foo.default.svc.cluster.local
subsets:
- name: v1
labels:
version: v1
trafficPolicy:
loadBalancer:
simple: ROUND_ROBIN
- name: v2
labels:
version: v2
trafficPolicy:
loadBalancer:
simple: LEAST_CON
```

VirtualService

A VirtualService describes how traffic addressed to a name flows to a set of destinations.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-identity
spec:
hosts:
```

```
- foo.default.svc.cluster.local
http:
- route:
- destination:
host: foo.default.svc.cluster.loca
```

The VirtualService above just forwards traffic addressed to foo.default.svc.cluster.local to the destination (a name we have a ServiceEntry for) foo.default.svc.cluster.local. Pilot generates a VirtualService like the one above implicitly for every service it has a ServiceEntry for.

Of course, we can do much more interesting things with VirtualServices than that. For example, we can define HTTP endpoints for a service and have Envoy deliver 404s client side, without calling the remove server, for invalid paths:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-apiserver
spec:
hosts:
- foo.default.svc.cluster.local
http:
- match:
- uri:
prefix: "/api"
route:
- destination:
host: apiserver.foo.svc.cluster.local
```

Clients calling foo.default.svc.cluster.local/api/... will be directed to a set of API servers at the destination apiserver.foo.svc.cluster.local, and any other URI will result in Envoy not finding a destination in that request and responding to the application with a 404 (this is why Pilot creates an implicit VirtualService for every ServiceEntry).

VirtualServices can be used to target very specific segments of traffic and direct them to different destinations. For example, a VirtualService can match requests by header values, the port a caller is attempting to connect to, or the labels on the client's workload (e.g. labels on the client's pod in Kubernetes) and send matching traffic to a

different destination (e.g. a new version of a service) than all the unmatched traffic. We cover these use cases in detail in Traffic Steering and Routing section of this chapter. A simple example is sending a fraction of traffic to the new version of a service, allowing a quick rollback in the case of a bad deployment.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-apiserver
spec:
hosts:
- foo.default.svc.cluster.local
http:
- match:
- uri:
prefix: "/api"
route:
- destination:
host: apiserver.foo.svc.cluster.local
subset: v1
weight: 90
- destination:
host: apiserver.foo.svc.cluster.local
subset: v2
weight: 10
```

It's important to note that within a VirtualService, the match conditions are checked at runtime in the order that they appear. This means that the most specific match clauses should appear first, and less specific clauses later. For safety, a "default" route, with no match conditions, should be provided; a request that does not match any condition of a VirtualService will result in a 404 for the sender (or some connection refused error for non-HTTP protocols).

HOSTS

We say that a VirtualService *claims* a name: a hostname can appear in at most one VirtualService, though a VirtualService can claim many hostnames. This can cause problems when a single name, like apis.foo.com, is used to host many services that route by path, e.g. apis.foo.com/bars or apis.foo.com/bazs, because many teams must

edit a single VirtualService “apis.foo.com”. One solution to this problem is to use a set of tiered VirtualServices. The top level VirtualService splits up requests into logical services by path prefix, and is a resource shared by every team (similar to a Kubernetes Ingress resource). Then a VirtualService for each of the logical services in the top level VirtualService can describe traffic for that block of requests. This pattern can be applied repeatedly to delegate management of smaller and smaller segments of traffic.

For example, a shared VirtualService with business logic for multiple teams, like the one below:

```
>apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-apiserver
spec:
hosts:
- apis.foo.com
http:
- match:
- uri:
prefix: "/bars/newMethod"
route:
- destination:
host: bar.foo.svc.cluster.local
subset: v2
- match:
- uri:
prefix: "/bars"
route:
- destination:
host: bar.foo.svc.cluster.local
subset: v1
- match:
- uri:
prefix: "/bazs/legacy/rest/path"
route:
- destination:
host: monolith.legacy.svc.cluster.remote
retries:
```

```
attempts: 3
perTryTimeout: 2s
- match:
- uri:
prefix: "/bazs"
route:
- destination:
host: baz.foo.svc.cluster.local
```

can be decomposed into separate VirtualServices owned by the appropriate teams:

```
>apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-svc-shared
spec:
hosts:
- apis.foo.com
http:
- match:
- uri:
prefix: "/bars"
route:
- destination:
host: bar.foo.svc.cluster.local
- match:
- uri:
prefix: "/bazs"
route:
- destination:
host: baz.foo.svc.cluster.local
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
Name: foo-bars-svc
spec:
hosts:
- bar.foo.svc.cluster.local
http:
- match:
- uri:
prefix: "/bars/newMethod"
```

```
route:
- destination:
host: bar.foo.svc.cluster.local
subset: v2
route:
- destination:
host: bar.foo.svc.cluster.local
subset: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
Name: foo-bazs-svc
spec:
hosts:
- baz.foo.svc.cluster.local
http:
- match:
- uri:
prefix: "/bazs/legacy/rest/path"
route:
- destination:
host: monolith.legacy.svc.cluster.remote
retries:
attempts: 3
perTryTimeout: 2s
route:
- destination:
host: baz.foo.svc.cluster.local
```

Finally, VirtualServices can claim a set of hostnames described by a wildcard pattern. In other words, a VirtualService can claim a host like “*.com”. When choosing which configuration to use, the most specific host will always apply: for a request to “baz.foo.com”, the VirtualService for “baz.foo.com” applies, and the VirtualServices for “*.foo.com” and “*.com” are ignored. Note, though, that no VirtualService can claim the wildcard host “*”.

Gateway

Gateways are all about exposing names over trust boundaries. Suppose you have a webserver.foo.svc.cluster.local service deployed in your mesh that serves your

website, foo.com. You can expose that webserver to the public internet using a Gateway to map from your internal name, webserver.foo.svc.cluster.local, to your public name foo.com. We also need to know what port to expose the public name on, and the protocol to expose it with.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
name: foo-com-gateway
spec:
selector:
app: gateway-workloads
servers:
- hosts:
- foo.com
port:
number: 80
name: http
protocol: HTTP
```

Just mapping between the names isn't enough though: a Gateway has to be able to prove to callers that it owns the name too. You can do this by configuring the Gateway to serve a certificate for foo.com.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
name: foo-com-gateway
spec:
selector:
app: gateway-workloads
servers:
- hosts:
- foo.com
port:
number: 443
name: https
protocol: HTTPS
tls:
mode: SIMPLE #enables HTTPS on this port
serverCertificate: /etc/certs/foo-com-public.pem
```

```
privateKey: /etc/certs/foo-com-privatekey.pem
```

Both foo-com-public.pem and foo-com-privatekey.pem in our example above are long-lived certs for foo.com, that you'd get from a CA like Let's Encrypt. Unfortunately Istio doesn't handle these types of certificates today, so you need to mount any certificates that a Gateway must serve into the workload's file system. Also note that we updated both the port and protocol to match; we could keep serving foo.com on over HTTP on port 80 if we wanted to:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
name: foo-com-gateway
spec:
selector:
app: gateway-workloads
servers:
- hosts:
- foo.com
port:
number: 80
name: http
protocol: HTTP
- hosts:
- foo.com
port:
number: 443
name: https
protocol: HTTPS
tls:
mode: SIMPLE #enables HTTPS on this port
serverCertificate: /etc/certs/foo-com-public.pem
privateKey: /etc/certs/foo-com-privatekey.pem
```

But we're better off configuring our Gateway to perform HTTPS upgrade:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
name: foo-com-gateway
spec:
```

```
selector:
app: gateway-workloads
servers:
- hosts:
- foo.com
port:
number: 80
name: http
protocol: HTTP
tls:
httpsRedirect: true # sends 301 redirect for http
requests
- hosts:
- foo.com
port:
number: 443
name: https
protocol: HTTPS
tls:
mode: SIMPLE #enables HTTPS on this port
serverCertificate: /etc/certs/foo-com-public.pem
privateKey: /etc/certs/foo-com-privatekey.pem
```

Note that while all of our examples above use HTTP(S) and ports 80 and 443, Gateways can expose any protocol over any port. When Istio can control the workload implementing a Gateway's behavior (i.e. it's an Envoy proxy; more on that below), the workload will listen to all ports listed in the Gateway.

But none of these Gateways map foo.com to any service in our mesh yet! For that, we need to *bind* a VirtualService to our Gateway:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-com-virtual-service
spec:
hosts:
- foo.com
gateways:
- foo-com-gateway
http:
```

```
- route:
- destination:
host: webserver.foo.svc.cluster.local
```

We'll cover the rules for binding VirtualServices to Gateways in a later section, but this raises an important point: Gateways configure L4 behavior, not L7 behavior. What we mean by that is that Gateway describes ports to bind to, what protocols to expose on those ports, and the names (and proof of those names, certs) to serve on those ports. Only VirtualServices describe L7 behavior: how to map from some name to different applications and workloads.

This decoupling of L4 from L7 behavior was a primary design goal. This allows patterns like providing a single Gateway that many teams re-use:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
name: foo-com-gateway
spec:
selector:
app: gateway-workloads
servers:
- hosts:
- *.foo.com
port:
number: 80
name: http
protocol: HTTP
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-com-virtual-service
spec:
hosts:
- api.foo.com
gateways:
- foo-com-gateway
http:
- route:
- destination:
```

```
host: api.foo.svc.cluster.local
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-com-virtual-service
spec:
hosts:
- www.foo.com
gateways:
- foo-com-gateway
http:
- route:
- destination:
host: webserver.foo.svc.cluster.local
```

More importantly, this decoupling of L4 and L7 behavior means a Gateway can be used to model network interfaces in Istio, e.g. network appliances or non-flat L3 networks. Finally, Gateways can be used to build mTLS tunnels between parts of a mesh deployed on separate L3 networks. For example, they can be used to establish secure connections between Istio deployments across separate cloud provider availability zones, over the public internet, without the need for a VPN. We cover that topic and others in detail in the Advanced Use Cases chapter of this book.

Finally, an implication of the fact that Gateways can be used to model arbitrary network interfaces in Istio is that Istio cannot always control the workload implementing a Gateway's behavior. For example, if you're using a Gateway to model an internal load balancer in your cloud deployment, Istio configuration cannot affect the decisions that that load balancer makes. The workloads that belong to a Gateway are described by the "selector" field on the Gateway object; any workloads with labels matching the selector will be treated like Gateways in Istio. When Istio controls the workload implementing a Gateway's behavior (i.e. the Gateway is an Envoy), we can *bind* VirtualServices to the Gateway to take advantage of VirtualService features at ingress and egress points in our cluster.

BINDING VIRTUALSERVICES TO GATEWAYS

We say a VirtualService *binds* to a Gateway if:

The VirtualService lists the Gateway's name in its gateways field

At least one host *claimed* by the VirtualService is *exposed* by the Gateway

The hosts in a Gateway's configuration are similar to the ones in a VirtualService, with a few subtle differences. Importantly, Gateways do not claim hostnames like VirtualServices do. Instead, a Gateway *exposes* a name, allowing a VirtualService to configure traffic for that name by binding to that Gateway. For example, any number of Gateways can exist exposing the name "foo.com", but a single VirtualService must configure traffic for them across all Gateways. The host field of a Gateway accepts wildcard hostnames in the same way the VirtualService does, but Gateways *do* allow the wildcard hostname "*".

Suppose we have the following two Gateways:

Gateways

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
name: foo-gateway
spec:
selector:
app: my-gateway-impl
servers:
- hosts:
- foo.com
port:
number: 80
name: http
protocol: HTTP
```

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
name: wildcard-gateway
spec:
selector:
app: my-gateway-impl
servers:
- hosts:
- *.com
port:
number: 80
name: http
protocol: HTTP
```

Then we can see how the following VirtualServices bind to those Gateways (or don't, as the case may be):

VirtualService	Comment
apiVersion: networking.istio.io/v1alpha3	
kind: VirtualService	
metadata:	
name: foo-default	
spec:	
hosts:	**Binds** to "foo-gateway" because the Gateway name in the VirtualService matches, and because the VirtualService claims the host "foo.com" which is exposed by "foo-gateway". This means requests to "foo.com" received on port 80 by this Gateway will be routed to port 7777 of the "foo" service in namespace "default".
- foo.com	
gateways:	Does **not bind** to "wildcard-gateway": the hosts match but the VirtualService does not list the Gateway "wildcard-gateway" as a target.
- foo-gateway	
http:	
- route:	
- destination:	
host: foo.default.svc.cluster.local	

apiVersion:
networking.istio.io/v1alpha3

kind:
VirtualService

metadata:

name: foo-default

spec:

hosts:

- foo.com

Binds to “foo-gateway” because the Gateway name in the VirtualService matches, and because the VirtualService claims the host “foo.com” which is exposed by “foo-gateway”. Only the name “foo.com” is visible to callers of the Gateway even though the VirtualService claims the name “foo.super.secret.internal.name” too.

- foo.super.secret.internal.name

Does **not bind** to “wildcard-gateway”: the hosts match but the VirtualService does not list the Gateway “wildcard-gateway” as a target.

gateways:

- foo-gateway

http:

- route:

-
destination:

host:
foo.default.svc.cluster.local

apiVersion:

networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: foo-internal

spec:

hosts:

- foo.super.secret.internal.name

Does **not bind** to either Gateway: while the VirtualService lists both Gateways, the hostname the VirtualService claims, “foo.super.secret.internal.name”, is not exposed by either Gateway so the Gateways will not accept requests for those names.

gateways:

- foo-gateway

- wildcard-gateway

http:

- route:

- destination:

host: foo.default.svc.cluster.local

---

apiVersion: networking.i

stio.io/v1alp
ha3

kind:
VirtualServi
ce

metadata:

name: foo-
internal

spec:

hosts:

Binds to “foo-gateway” because the Gateway name in the VirtualService matches, and because the VirtualService claims the host “foo.com” which is exposed by “foo-gateway”.

- foo.com

gateways:

Binds to “wildcard-gateway” because the Gateway name in the VirtualService matches, and because the VirtualService claims the host “foo.com” which is exposed by “wildcard-gateway” (because “foo.com” matches the wildcard “*.com”).

**- foo-
gateway**

**- wildcard-
gateway**

http:

- route:

-
destination:

host:
foo.default.s
vc.cluster.lo
cal

THE MESH GATEWAY

There’s a special, implicit Gateway in every Istio deployment called the “mesh”

Gateway. It represents a Gateways whose workloads are every sidecar in the mesh and exposes the wildcard host on every port. When a VirtualService does not list any Gateways, it automatically applies to the mesh gateway, i.e. all of the sidecars in the mesh. A VirtualService always binds to *either* the mesh gateway *or* the gateways listed in its "gateways" field. A common tripping hazard using VirtualServices is to update a VirtualService being used inside of the mesh to bind to a specific Gateway too. Upon pushing that resource, its configuration no longer applies to sidecars, causing errors. When doing updates like that, make sure to include the "mesh" gateway specifically in the list of Gateways to bind to.

Traffic Steering and Routing

We can use the APIs described above in a ton of different ways to affect traffic flow in our deployment. In this section we'll cover some of the most common use cases: using VirtualServices to make routing decisions based on request attributes like the URI, headers, the request's scheme, or its target port, and to implement canary and blue/green deployment strategies.

ROUTING WITH REQUEST METADATA

One of Istio's most powerful features is its ability to perform traffic routing based on request metadata like the request's URI, its headers, the source or destination IP addresses, and other metadata about the request. The one key limitation is that Istio will not perform routing based on the *body* of the request.

In the VirtualService section above we covered routing based on URI prefixes extensively. Similar routing can be performed both on exact URI matches and regexes:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
Name: foo-bars-svc
spec:
hosts:
- bar.foo.svc.cluster.local
http:
- match:
```

```
- uri:
exact: "/assets/static/style.css"
route:
- destination:
host: webserver.frontend.svc.cluster.local
- match:
- uri:
# match requests like "/foo/132:myCustomMethod"
regex: "/foo/\\d+:myCustomMethod"
route:
- destination:
host: bar.foo.svc.cluster.local
subset: v3
- route:
- destination:
host: bar.foo.svc.cluster.local
subset: v
```

We can also route based on headers or cookie values:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
Name: dev-webserver
spec:
hosts:
- webserver.company.com
http:
- match:
- headers:
cookie:
environment: "dev"
route:
- destination:
host: webserver.dev.svc.cluster.local
- route:
- destination:
host: webserver.prod.svc.cluster.local
```

And of course, Istio supports routing requests for TCP services as well, using L4 request metadata like destination subnet and target port. For TLS TCP services, the SNI can be used to perform routing just like the Host header in HTTP.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
Name: dev-api-server
spec:
hosts:
- api.company.com
tcp:
- match:
- port: 9090
destinationSubnets:
- 10.128.0.0/16
route:
- destination:
host: database.test.svc.cluster.local
- match:
- port: 9090
route:
- destination:
host: database.prod.svc.cluster.local
tls:
- match:
- sniHosts:
- example.api.company.com
route:
- destination:
host: example.prod.svc.cluster.local
```

See Istio's website for a full reference on all available match conditions and their syntax.

BLUE/GREEN DEPLOYMENTS

A blue/green deployment is one in which two versions, old and new, of an application are deployed side-by-side, and user traffic is flipped from the old set to the new. This allows for a quick fallback to the previously working version if something goes wrong, since all that's required is reverting user traffic back to the old set from the new (as opposed to a deployment strategy like rolling update, where to rollback to the previous version we have to re-deploy the previous version's binary).

Istio's networking APIs make it pretty easy to do blue/green deployments. We'll

declare two *subsets* for our service using a DestinationRule, then we'll use a VirtualService to direct traffic to one subset of the other.

Note that rather than use blue/green terminology in our DestinationRule, we'll refer to subsets by the version of the application they represent. This is both easier for developers to understand (since it talks about parts of their service in terms of versions they control) and less prone to errors (avoid "Hey, before I deploy, Is blue or green the active set?" style outages). This phrasing also makes it easier to transition to other deployment strategies like canaried deployments.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
name: foo-default
spec:
host: foo.default.svc.cluster.local
subsets:
- name: v1
labels:
version: v1
- name: v2
labels:
version: v2
```

Then we can write a VirtualService that directs all traffic in the cluster targeting our service to a single subset of the service:

```
>apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-blue-green-virtual-service
spec:
hosts:
- foo.default.svc.cluster.local
http:
- route:
- destination:
host: foo.default.svc.cluster.local
subset: v1
```

To flip to the other set, you simply update the VirtualService to target subset v2:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-blue-green-virtual-service
spec:
hosts:
- foo.default.svc.cluster.local
http:
- route:
- destination:
host: foo.default.svc.cluster.local
subset: v2
```

Of course this can be combined with Gateways as well so you can perform blue/green deployments for users consuming your service via a Gateway in addition to the services in your mesh.

CANARY DEPLOYMENTS

A canary deployment is the practice of sending a small portion of traffic to newly deployed workloads, gradually ramping up until all traffic flows the new workloads. The goal is to verify a new workload is healthy (up, running, not returning errors) before sending all traffic to it. It's similar to a Blue/Green deployment in that it allows a fast fallback to known-healthy workloads, but improves by only sending only a portion of traffic, rather than all of it, to the new workloads. This overall reduces the amount of error budget you may spend performing a deployment.

Canary based deployments also tend to require resource capacity for updates; a true Blue/Green deployment requires double the resource capacity of a standard deployment (to have both a full blue and a full green deployment). Canaries can be combined with in-place binary rollout strategies to get the rollback safety of a Blue/Green deployment while only requiring a constant amount of additional resources (spare capacity to schedule just a small number of additional workloads).

A new workload can be canaried in a variety of ways; you can use the full set of matches outlined in the Routing with Request Metadata section above to send small

portions of traffic to a new backend. However, the simplest canary is a percentage based traffic split. We can start by sending 5% of traffic to the new version, gradually pushing new VirtualService configurations ramping traffic up to 100% to the new version.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-canary-virtual-service
spec:
hosts:
- foo.default.svc.cluster.local
tcp:
- route:
- destination:
host: foo.default.svc.cluster.local
subset: v2
weight: 5
- destination:
host: foo.default.svc.cluster.local
subset: v1
weight: 95
```

Another common pattern is to canary a new deployment to a set of “trusted testers,” e.g. the development team, or a set of customers that have opted in to experimental features. For example, we may set a “trusted-tester” cookie, which we can use at routing time to send requests in that session to different backends than normal requests:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-canary-virtual-service
spec:
hosts:
- foo.default.svc.cluster.local
http:
- match:
- headers:
cookie:
trusted-tester: "true"
route:
```

```
- destination:
host: foo.default.svc.cluster.local
subset: test
- route:
- destination:
host: foo.default.svc.cluster.local
subset: v1
```

Of course, take care when using caller-supplied values (like cookies) to perform routing: ideally all services in your cluster should perform authentication and authorization on all requests. This ensures that even if a caller fakes data to trigger routing behavior, they cannot access data they wouldn't be able to otherwise (and in fact, implementing authentication and authorization via Istio is a powerful way to ensure all services in your cluster do this correctly).

Resiliency

A resilient system is one that can maintain good performance for its users (i.e. staying inside of its SLOs) while coping with failures in downstream systems it depends on. Istio provides a lot of features to help build more resilient applications, most importantly client side load balancing, circuit breaking via outlier detection, automatic retry, and request timeouts. Istio also provides tools to inject faults into applications, allowing you to build programmatic, reproducible tests of your system's resiliency.

Load Balancing Strategy

Client side load balancing is an incredibly valuable tool for building resilient systems. By allowing clients to communicate directly with servers without going through reverse proxies, we remove points of failure while still keeping a well behaved system. Further, it allows clients to adjust their behavior dynamically based on responses from servers, e.g. to stop sending requests to endpoints that return more errors that other endpoints of the same service (we cover that feature, Outlier Detection, more in the next section). DestinationRules allow you to define the load balancing strategy clients use to select backends to call. For example, we can configure clients to use a simple round robin load balancing strategy:

```
apiVersion: networking.istio.io/v1alpha3
```

```
kind: DestinationRule
metadata:
name: foo-default
spec:
host: foo.default.svc.cluster.local
trafficPolicy:
loadBalancer:
simple: ROUND_ROBIN
```

The above destination rule will send traffic round robin across the endpoints of the service “foo.default.svc.cluster.local”. A ServiceEntry defines what those endpoints are (or how to discover them at runtime, e.g. via DNS). It’s important to note that a DestinationRule only applies to hosts in Istio’s service registry: if a ServiceEntry does not exist for a host, the DestinationRule is ignored.

More complex load balancing strategies, such as consistent hash based load balancing, are also supported. The following DestinationRule configures load balancing based on a hash of the caller’s IP address (HTTP headers and cookies can also be used to with consistent load balancing):

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
name: foo-default
spec:
host: foo.default.svc.cluster.local
trafficPolicy:
loadBalancer:
consistentHash:
useSourceIp: true
```

Outlier Detection

Circuit breaking is a pattern of protecting calls, e.g. network calls to a remote service, behind a “circuit breaker”. If the protected call returns too many errors we “trip” the circuit breaker and return errors to the caller without executing the protected call. It can be used to mitigate several classes of failure, including cascading failures. In load balancing, lame ducking an endpoint is the practice of removing that endpoint from the “active” load balancing set so that no traffic is sent to if for some period of time. Lame

ducking is one method we can use to implement the circuit breaking pattern.

Outlier detection is a means of triggering lame ducking of endpoints that're sending bad responses. We can detect when an individual endpoint is an outlier compared to the rest of the endpoints in our "active" load balancing set (i.e. returning more errors than other endpoints of the service) and remove the bad endpoint from our "active" load balancing set.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
name: foo-default
spec:
host: foo.default.svc.cluster.local
trafficPolicy:
outlierDetection:
consecutiveErrors: 5
interval: 1m
baseEjectionTime: 3m
```

The DestinationRule above configures the sidecar to eject any endpoint that has had five consecutive errors from the load balancing set for at least three minutes. Every minute the sidecar will scan the set of all endpoints to decide if any endpoints in the load balancing set should be ejected or if ejected endpoints can be returned back to the load balancing set. It's important to remember that outlier detection is per client, since any given server could return bad results to only a specific client (e.g. if there's a network partition between them, but not between the server and its other clients).

Retries

Every system has transient failures: network buffers overflow, a server shutting down drops a request, a downstream system fails, etc. We use retries, sending the same request to a different endpoint of the same service, to mitigate the impact of transient failures. However, poor retry policies are a frequent secondary cause of outages: "something went wrong, and client retries made it worse" is a common refrain. Often this is because retries are hard-coded into applications (e.g. as a for loop around the network call) and therefore are hard to change. Istio gives you the ability to configure retries globally for all services in your mesh. More significantly, it allows you to

control those retry strategies at runtime via configuration, so you can change client behavior on the fly.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-default
spec:
hosts:
- foo.com
gateways:
- foo-gateway
http:
- route:
- destination:
host: foo.default.svc.cluster.local
retries:
attempts: 3
perTryTimeout: 500ms
```

The retry policy defined in a VirtualService works in concert with the connection pool settings defined in the destination's DestinationRule to control the total number of concurrent outstanding retries to the destination. Read more about that in the DestinationRule section above.

Timeouts

Timeouts are important for building systems with consistent behavior. By attaching deadlines to requests we're able to abandon requests taking too long and free server resources. We're also able to control our tail latency much more finely, because we know the longest we'll wait for any particular request in computing our response for a client.

A timeout can be attached to any HTTP Route in a VirtualService:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-default
spec:
```

```
hosts:
- foo.com
gateways:
- foo-gateway
http:
- route:
- destination:
host: foo.default.svc.cluster.local
timeout: 1s
```

When used with a retry, the timeout represents the total time the client will spend waiting for a server to return a result; the retry's per-try-timeout controls the timeout of each individual attempt.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-default
spec:
hosts:
- foo.com
gateways:
- foo-gateway
http:
- route:
- destination:
host: foo.default.svc.cluster.local
timeout: 2s
retries:
attempts: 3
perTryTimeout: 500ms
```

The VirtualService above configures our client to wait at most two seconds, retrying three times at 500ms timeouts each. We add in some slack time to allow for randomized waits between retries.

Fault Injection

Fault injection is an incredibly powerful way to test and build reliable distributed applications. Companies like Netflix have taken this to the extreme, coining the name "chaos engineering" to describe the practice of injecting faults into running production

systems to ensure that they're built to be reliable and tolerant of failures in their environment.

Istio allows you to configure faults for HTTP traffic, injecting arbitrary delays or returning specific response codes (e.g. 500) for some percentage of traffic.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-default
spec:
hosts:
- foo.default.svc.cluster.local
http:
- route:
- destination:
host: foo.default.svc.cluster.local
fault:
delay:
fixedDelay: 5s
percentage: 100
```

For example, the VirtualService above injects a five second delay for all traffic calling the foo service. This is a great way to reliably test things like how a UI behaves on a bad network when its backends are far away. It's also valuable for testing that applications set timeouts on their requests.

Replying to clients with specific response codes, like 429s or 500s, is also great for testing. For example, it can be challenging to programmatically test how your application behaves when a third party service it depends on starts to fail. Using Istio, you can write a set of reliable end to end tests of your application's behavior in the presence of failures of its dependencies.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: foo-default
spec:
hosts:
```

```
- foo.default.svc.cluster.local
http:
- route:
- destination:
host: foo.default.svc.cluster.local
fault:
abort:
httpStatus: 500
percentage: 10
```

For example, we can simulate 10% of requests to some backend failing at runtime with a 500 response code.

Ingress and Egress

Gateways represent trust boundaries; we use them to model ingress/egress. When Istio controls the gateways, we can do policy/routing/VS stuff at these points. Use VS in the mesh to direct traffic to egress point; apply policy there

Gateways represent *network trust boundaries* in a deployment. In other words, we typically use Gateways to model proxies on the edge of the network that control ingress and egress of traffic into and out of the network (in an environment like Kubernetes, which provides a flat network to pods, the network spans the entire cluster). Together Gateways and VirtualServices can precisely control how traffic enters and exits the mesh. Even better, when Istio is deployed with Policy enabled, they allow you to apply policy to traffic as it enters or leaves the mesh.

INGRESS

The Gateway section above covers how hostnames are "exposed" over a Gateway by *binding* a VirtualService to that Gateway. Once a VirtualService is bound to a Gateway, all of the normal VirtualService functionality we describe in the previous sections, such as retries, fault injection, or traffic steering, can be applied to traffic at ingress. In many ways, the ingress Gateway acts as the "client-side sidecar" proxy.

One thing Istio can't control, though, is how client traffic gets *to* the ingress proxies. A common pattern in Kubernetes environments is to model Istio's ingress proxies as a

NodePort service, then let the platform handle provisioning public IP addresses, DNS records, etc.

EGRESS

In the same way we think about ingress proxies as a sort of "client-side sidecar," egress proxies act as a sort of "server-side sidecar." With a combination of ServiceEntries, DestinationRules, VirtualServices, and Gateways, we can trap outbound traffic and redirect it to egress proxies where we're free to apply policy.

As a prerequisite, we assume that Istio has been deployed with an egress proxy named "istio-egressgateway.istio-system.svc.cluster.local".With that in place, we can start by modelling the destination we're trying to exit the mesh to reach, for example https://wikipedia.org, as a ServiceEntry:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
name: https-wikipedia-org
spec:
hosts:
- wikipedia.org
ports:
- number: 443
name: https
protocol: HTTPS
location: MESH_EXTERNAL
resolution: DNS
endpoints:
- address: istio-egressgateway.istio-
system.svc.cluster.local
ports:
http: 443
```

Then we can configure out egress Gateway to accept traffic for wikipedia.org:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
name: https-wikipedia-org-egress
```

```
spec:
selector:
istio: egressgateway
servers:
- port:
number: 443
name: https-wikipedia-org-egress-443
protocol: TLS # Mark as TLS as we are passing HTTPS
through.
hosts:
- wikipedia.org
tls:
mode: PASSTHROUGH
```

We have a problem though: we want our egress gateway to use DNS to get an address for wikipedia.org and forward the request there, but we've configured all of the proxies in the mesh to resolve wikipedia.org to the egress gateway (so the proxy will forward the message back to itself, or drop it). To fix this, we'll have to take advantage of the fact that VirtualServices can be bound to Gateways and route traffic going to wikipedia.org to a fake name we create, e.g. egress-wikipedia-org.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: egress-wikipedia-org
spec:
hosts:
- wikipedia.org
gateways:
- https-wikipedia-org-egress
tls:
- match:
- ports: 443
sniHosts:
- wikipedia.org
route:
- destination:
host: egress-wikipedia-org
```

Then we'll use a ServiceEntry to resolve egress-wikipedia-org via DNS as wikipedia.org:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
name: egress-https-wikipedia-org
spec:
hosts:
- egress-wikipedia-org
ports:
- number: 443
name: https
protocol: HTTPS
location: MESH_EXTERNAL
resolution: DNS
endpoints:
- address: wikipedia.org
ports:
http: 443
```

With this in place, we force traffic to an external site through a dedicated Egress Gateway deployment. Keep in mind that by default Istio will not route traffic to destinations that do not have a ServiceEntry, by services outside of the mesh must be whitelisted by creating service entries for them. To enable egress to an external service without going through an egress proxy, just create an identity ServiceEntry for it:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
name: egress-https-wikipedia-org
spec:
hosts:
- wikipedia.org
ports:
- number: 443
name: https
protocol: HTTPS
location: MESH_EXTERNAL
resolution: DNS
endpoints:
- address: wikipedia.org
ports:
http: 443
```

Summary

We've seen the full power of Istio's networking APIs, and there's a ton of features - it can be overwhelming to start with. The important thing to remember is that you can approach things incrementally: pick up one feature that's valuable to you today, start to configure it for your service, and get comfortable with it. *Then* reach for the next feature that solves a problem.

Chapter 5. Telemetry

Critical to running microservices is the ability to reason over their behavior. Not only does this include the triumphant of logs, metrics and traces, but critically needed visualization, troubleshooting and debugging as well. In Chapter 2, we covered how service meshes, in general, and Istio, specifically, provide for uniform observability. We'll get to troubleshooting and debugging Chapter 12.

As described in Chapter 9, Mixer plays a key role in the collecting and coalescing telemetry generated by service proxies. Service proxies generate telemetry based on the traffic it processes, buffering telemetry before flushing to Mixer. Operators have choice of the number and type of adapters deployed - Mixer supports have multiple of the same type simultaneously enabled (e.g. two logging adapter to send logs to two different backends).

> **NOTE**
>
> Istio's sample application, BookInfo, is designed to showcase aspects of the value proposition of service meshes. The Bookinfo sample application is used as the example application throughout this chapter. Let take a moment to familiarize with this application.

The Bookinfo sample application is used as the example application throughout this chapter. Let take a moment to familiarize with this application.

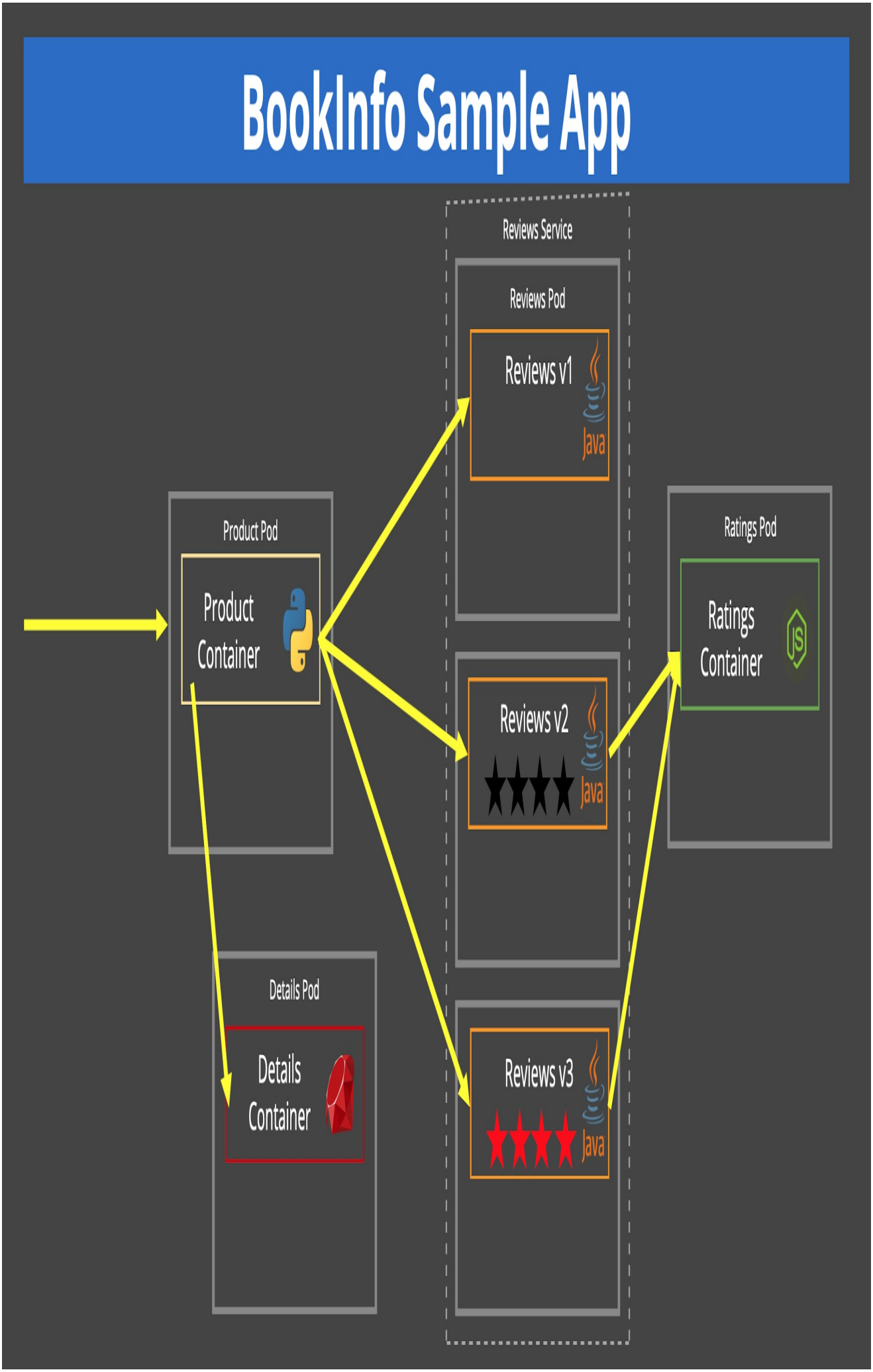

BookInfo Sample App
Reviews Service
Reviews Pod
Reviews v1
Java
Product Pod
Product
Container
Ratings Pod
Ratings
Container
JS
Reviews v2
Java
Details Pod
Details
Container
Reviews v3
Java

Figure 5-1. Figure 11.1 Istio's canonical sample application, BookInfo, with four separate microservices demonstrates various Istio features.

From left to right, users call the productpage microservice, which in turn calls the details and reviews microservices to populate the page. The details microservice contains book information. The reviews microservice contains book reviews and subsequently calls the ratings microservice to retrieve reviews. The ratings microservice contains book ranking in the form of a 1 to 5 star book review. The reviews microservice has three versions:

- reviews v1 has no ratings (does not call the ratings service).
- reviews v2 has ratings of 1 to 5 black stars (calls the ratings service).
- reviews v3 has ratings of 1 to 5 red stars (calls the ratings service).

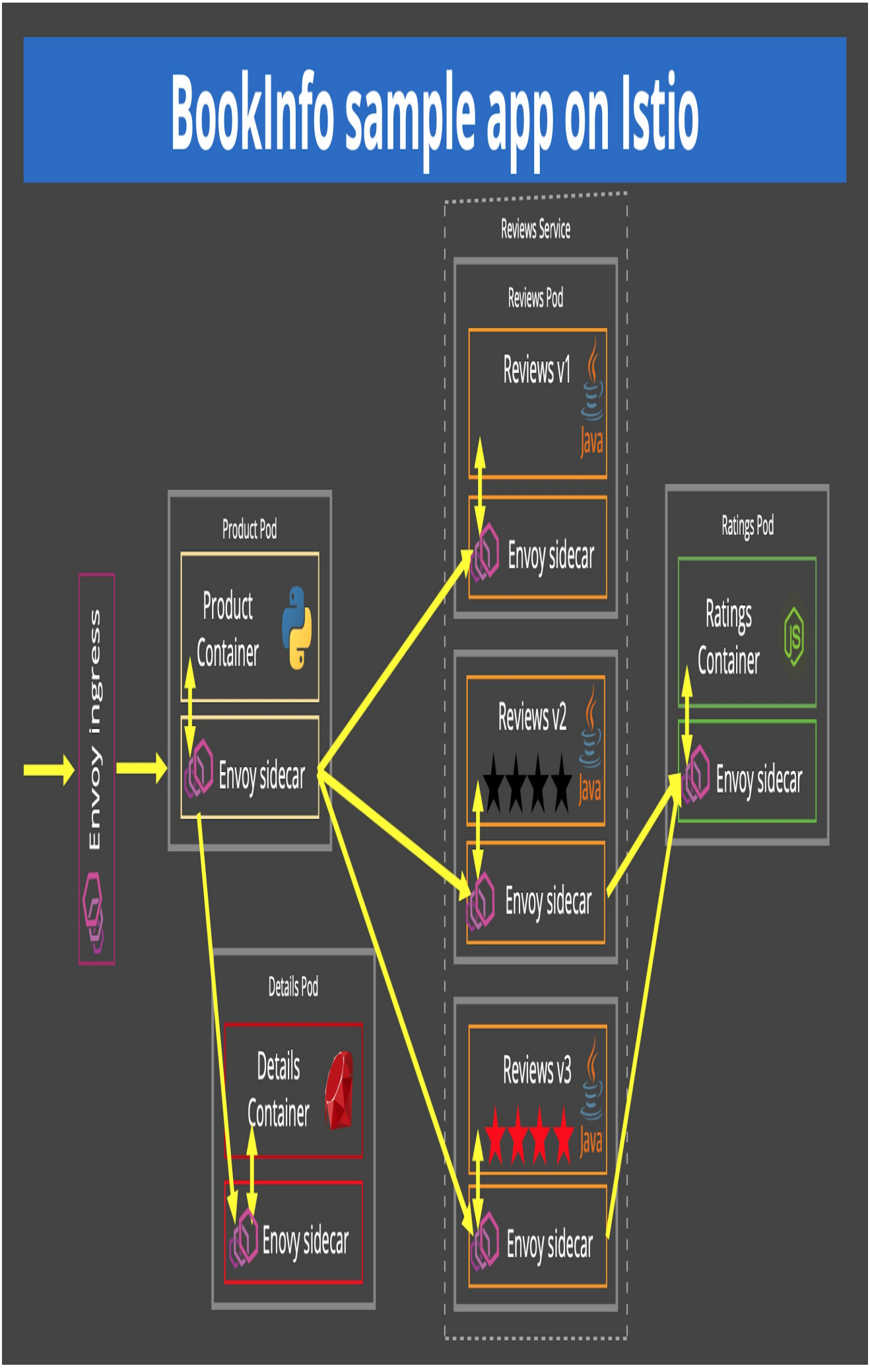

BookInfo sample app on Istio
Reviews Service
Reviews Pod
Reviews v1
Java
Envoy sidecar
Product Pod
Product Container
Envoy sidecar
Envoy ingress
Ratings Pod
Ratings Container
Envoy sidecar
Reviews v2
Java
Envoy sidecar
Details Pod
Details Container
Enovy sidecar
Reviews v3
Java
Envoy sidecar

Figure 5-2. Figure 11.2 BookInfo deployed on an Istio mesh with service proxies sidecarred.

To run the sample with Istio requires no changes to the application itself. Instead, we simply need to configure and run the services in an Istio-enabled environment, with Envoy sidecars injected alongside each service.

```
$ kubectl apply -f
samples/bookinfo/platform/kube/bookinfo.yaml
```

Adapter Models

As described in Chapter 9, adapters integrate Mixer with different infrastructure backends that deliver core functionality, such as logging, monitoring, quotas, access control list checking, and more.

Two of the same type of adapter can be deployed.

Example of having logs sent to two different backends.

Types of Telemetry

As a refresher from Chapter 2, Istio supports three forms of telemetry (metrics, logs, traces), which may transmit a diverse set of insights between them.

Reporting

Reports are contain *attributes*. See chapter 9 for a description of attributes. Context attributes provide the ability to distinguish between http and tcp protocols within policies.

Telemetric attributes are generated by service proxies. Attributes sent at three different points in time:

- when the connection is established (initial report).
- periodically while the connection is alive (periodical report).
- when the connection is closed (final report).

The default interval for periodical report is 10 seconds. It's recommended that this interval should not be subsecond.

Metrics

Service metrics collected by the sidecar proxies . . . .

Istio telemetry service runs the Mixer process which comes bundled with several adapters. Any Mixer adapter which is based off of the metric metric adapter template can be used to collect and process metrics forwarded to it by Istio mixer. Prometheus Mixer adapter, amongst those, is an obvious choice for most Istio newbies.

To use the Prometheus Mixer adapter we need to have a Prometheus instance deployed either on the same Kubernetes cluster or elsewhere which is capable of scraping metrics from the Prometheus Mixer adapter. There are many ways to deploy Prometheus either on Kubernetes or outside, the details of which are outside the scope of this book.

To configure and use Prometheus Mixer adapter we have to:

Create a metric instance which configures metrics that Istio will generate and collect

Configure a Prometheus handler to collect the metrics, assign appropriate labels and make it available for a Prometheus instance to scrape

Update Prometheus to scrape metrics from the Prometheus mixer adapter

Create Istio rules which will forward the metrics collected by Istio Mixer to the Prometheus Mixer adapter with the configured labels

Here is a sample configuration:

```
apiVersion: "config.istio.io/v1alpha2"
kind: handler
metadata:
name: prometheus
namespace: istio-system
```

```
spec:
compiledAdapter: prometheus
params:
metrics:
- name: requests_total
instance_name: requestcount.metric.istio-system
kind: COUNTER
label_names:2
- reporter
- source_app
- source_namespace
- source_principal
- source_workload
- source_workload_namespace
- source_version
- destination_app
- destination_namespace
- destination_principal
- destination_workload
- destination_workload_namespace
- destination_version
- destination_service
- destination_service_name
- destination_service_namespace
- request_protocol
- response_code
- connection_mtls
- name: request_duration_seconds
instance_name: requestduration.metric.istio-system
kind: DISTRIBUTION
label_names:
- reporter
- source_app
- source_namespace
- source_principal
- source_workload
- source_workload_namespace
- source_version
- destination_app
- destination_namespace
- destination_principal
- destination_workload
- destination_workload_namespace
```

```
- destination_version
- destination_service
- destination_service_name
- destination_service_namespace
- request_protocol
- response_code
- connection_mtls
buckets:
explicit_buckets:
bounds: [0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1,
2.5, 5, 10]
- name: request_bytes
instance_name: requestsize.metric.istio-system
kind: DISTRIBUTION
label_names:
- reporter
- source_app
- source_namespace
- source_principal
- source_workload
- source_workload_namespace
- source_version
- destination_app
- destination_namespace
- destination_principal
- destination_workload
- destination_workload_namespace
- destination_version
- destination_service
- destination_service_name
- destination_service_namespace
- request_protocol
- response_code
- connection_mtls
buckets:
exponentialBuckets:
numFiniteBuckets: 8
scale: 1
growthFactor: 10
- name: response_bytes
instance_name: responsesize.metric.istio-system
kind: DISTRIBUTION
label_names:
```

```
- reporter
- source_app
- source_namespace
- source_principal
- source_workload
- source_workload_namespace
- source_version
- destination_app
- destination_namespace
- destination_principal
- destination_workload
- destination_workload_namespace
- destination_version
- destination_service
- destination_service_name
- destination_service_namespace
- request_protocol
- response_code
- connection_mtls
buckets:
exponentialBuckets:
numFiniteBuckets: 8
scale: 1
growthFactor: 10
- name: tcp_sent_bytes_total
instance_name: tcpbytesent.metric.istio-system
kind: COUNTER
label_names:
- reporter
- source_app
- source_namespace
- source_principal
- source_workload
- source_workload_namespace
- source_version
- destination_app
- destination_namespace
- destination_principal
- destination_workload
- destination_workload_namespace
- destination_version
- destination_service
- destination_service_name
```

```
- destination_service_namespace
- connection_mtls
- name: tcp_received_bytes_total
instance_name: tcpbytereceived.metric.istio-system
kind: COUNTER
label_names:
- reporter
- source_app
- source_namespace
- source_principal
- source_workload
- source_workload_namespace
- source_version
- destination_app
- destination_namespace
- destination_principal
- destination_workload
- destination_workload_namespace
- destination_version
- destination_service
- destination_service_name
- destination_service_namespace
- connection_mtls
- name: tcp_connections_opened_total
instance_name: tcpconnectionsopened.metric.istio-system
kind: COUNTER
label_names:
- reporter
- source_app
- source_namespace
- source_principal
- source_workload
- source_workload_namespace
- source_version
- destination_app
- destination_namespace
- destination_principal
- destination_workload
- destination_workload_namespace
- destination_version
- destination_service
- destination_service_name
- destination_service_namespace
```

```
- connection_mtls
- name: tcp_connections_closed_total
instance_name: tcpconnectionsclosed.metric.istio-system
kind: COUNTER
label_names:
- reporter
- source_app
- source_namespace
- source_principal
- source_workload
- source_workload_namespace
- source_version
- destination_app
- destination_namespace
- destination_principal
- destination_workload
- destination_workload_namespace
- destination_version
- destination_service
- destination_service_name
- destination_service_namespace
- connection_mtls
---
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
name: promhttp
namespace: istio-system
spec:
match: (context.protocol == "http" || context.protocol ==
"grpc") && (match((request.useragent | "-"), "kube-
probe*") == false)
actions:
- handler: prometheus
instances:
- requestcount.metric
- requestduration.metric
- requestsize.metric
- responsesize.metric
---
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
```

```
name: promtcp
namespace: istio-system
spec:
match: context.protocol == "tcp"
actions:
- handler: prometheus
instances:
- tcpbytesent.metric
- tcpbytereceived.metric
```

Figure 11.x - a metric Mixer adapter configuration

CONFIGURING MIXER TO COLLECT METRICS

SETTING UP METRICS COLLECTION AND QUERYING FOR METRICS

SEND METRICS TO PROMETHEUS, STATSD

Traces

JAEGER

Distributed traces are arguably the most insightful of the telemetry information gleaned from the service mesh, giving you insight into hard to answer questions like “Why is my service slow?”

OPENTRACING

Generating Trace Spans

The Istio service proxy, Envoy, is responsible for generating the initial trace headers. The x-request-id header is used by Envoy to uniquely identify a request as well as perform stable access logging and tracing.

Propagating Trace Headers

This is one area that when someone initially learns of Istio’s capabilities, that they may be oversold. Given that the service proxy is a sidecar to your application, your application will need a thin client library is needed to collection and propagate HTTP headers, including the following specifically:

- x-request-id
- x-b3-traceid
- x-b3-spanid
- x-b3-parentspanid
- x-b3-sampled
- x-b3-flags
- x-ot-span-context

Figure 11.x presents a simple Go program with function to listen to HTTP requests, extract the trace headers and print them to stdout.

```
package main import ( "fmt" "log" "net/http" ) func
tracingMiddleware(next http.HandlerFunc) http.HandlerFunc
{ incomingHeaders := []string{ "x-request-id", "x-b3-
traceid", "x-b3-spanid", "x-b3-parentspanid", "x-b3-
sampled", "x-b3-flags", "x-ot-span-context", } return
func(w http.ResponseWriter, r *http.Request) { for _, th
:= range incomingHeaders { w.Header().Set(th,
r.Header.Get(th)) } next.ServeHTTP(w, r) } } func main()
{ http.HandleFunc("/", tracingMiddleware(func(w
http.ResponseWriter, r *http.Request) { fmt.Fprintf(w,
"Hello headers, %v", r.Header) }))
log.Fatal(http.ListenAndServe(":8081", nil)) }
```

Figure 11.x A simple Go program to print trace headers

Each of the services in our sample application, BookInfo is instrumented to propagate these HTTP trace headers.

DISABLING TRACING

The sampling of request traces does bear cost in terms of performance overhead. There is significant difference between sampling traces at a rate of 1% vs at rate of 100%.

<insert performance graph>

If you would like to run your Istio mesh without tracing being enabled at all, the easiest

way to ensure that it is disabled is by not enabling it when installing the service mesh. In your Helm chart:

--set tracing.enabled=fasle

istio-demo.yaml or istio-demo-auth.yaml or don't enable it when installing Istio.

If you deployed with tracing on and now wish to disable tracing, assuming your control plane is installed in the istio-system namespace, run:

Remove reference of the Zipkin URL from the Mixer deployment:

Now, manually remove instances of trace_zipkin_url from the file and save it.

Logs

FLUENTD

Service access logs play a crucial role in recording particulars of service access information. Mixer's native logentry adapter…. *<does something described here>*. Community contributed Mixer adapters that are based on the logentry Mixer adapter template can be used to collect and process logs forwarded to it by Istio Mixer. Amongst those, Fluentd Mixer adapter is a popular one.

To use the Fluentd Mixer adapter we need a Fluentd daemon to be running and listening either on the same Kubernetes cluster or elsewhere. There are many ways to deploy Fluentd either on Kubernetes or outside, the details of which are outside the scope of this book.

As with other Mixer adapters, to configure and use Fluentd mixer adapter we have to:

Create a logentry instance which configures a log stream that Istio will generate and collect

Configure a Fluentd handler to pass the collected logs to a listening Fluentd daemon

Create an Istio rule which will forward the log stream collected by Istio Mixer to the

Fluentd Mixer adapter

The sample configuration in Figure 11.x assumes a fluentd daemon is available on localhost:24224.

```
apiVersion: "config.istio.io/v1alpha2"
kind: logentry
metadata:
name: istiolog
namespace: istio-system
spec:
severity: '"warning"'
timestamp: request.time
variables:
source: source.labels["app"] | source.service | "unknown"
user: source.user | "unknown"
destination: destination.labels["app"] |
destination.service | "unknown"
responseCode: response.code | 0
responseSize: response.size | 0
latency: response.duration | "0ms"
monitored_resource_type: '"UNSPECIFIED"'
---
# Configuration for a fluentd handler
apiVersion: "config.istio.io/v1alpha2"
kind: fluentd
metadata:
name: handler
namespace: istio-system
spec:
address: "localhost:24224"
integerDuration: n
---
# Rule to send logentry instances to the fluentd handler
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
name: istiologtofluentd
namespace: istio-system
spec:
match: "true" # match for all requests
actions:
```

```
- handler: handler.fluentd
instances:
- istiolog.logentry
```

Figure 11.x - a logging Mixer adapter configuration

The default logging level set in the configuration for instances: https://github.com/istio/istio/blob/master/install/kubernetes/helm/subcharts/mixer/templates/config.yaml#L223https://github.com/istio/istio/blob/master/install/kubernetes/helm/subcharts/mixer/templates/config.yaml#L223

If you don't want to see those logs at all, you can delete the following rule: https://github.com/istio/istio/blob/master/install/kubernetes/helm/subcharts/mixer/templates/config.yaml#L311-L326

If you want to see logs only for non-200 responses, you could edit the match condition here: https://github.com/istio/istio/blob/master/install/kubernetes/helm/subcharts/mixer/templates/config.yaml#L322

> **NOTE**
>
> Istio has tentatively started adding sampling support (and experimented with things like errors-only logging), but we haven't turned that into any real recommendation at this point... maybe an action item for 1.1 patch releases or 1.2.

METRICS

Top line service performance is easily gleaned from the metrics visualized through charting and dashboarding tools. Grafana is a popular open source, metrics visualization, query, ze, alert on and understand your metrics no matter where they are stored. Create, explore, and share dashboards with your team and foster a data driven culture.Grafana is not deployed as a Mixer adapter, but is included in the default Istio deployment as an add-on and is configured to read metrics from Prometheus. This deployment of Grafana comes with predefined dashboards. The Istio dashboards in Grafana are dependent upon Prometheus running in the environment. Currently, the

packaged dashboards include:

A Mesh Summary View. This section provides Global Summary view of the Mesh and shows HTTP/gRPC and TCP workloads in the Mesh.

Individual Services View. This section provides metrics about requests and responses for each individual service within the mesh (HTTP/gRPC and TCP). Also, give metrics about client and service workloads for this service.

Individual Workloads View: This section provides metrics about requests and responses for each individual workloads within the mesh (HTTP/gRPC and TCP). Also, give metrics about inbound workloads and outbound services for this workload.

Visualization

As one of the more insightful telemetric capabilities (a little like taking the blinders off), topology visualization is a key aspect to understanding your deployment. As a project Istio comes with a rudimentary solution for this need called, ServiceGraph. To enable ServiceGraph in your deployment:

It exposes the following endpoints:

- /force/forcegraph.html is an interactive D3.js visualization.
- /dotviz is a static Graphviz visualization.
- /dotgraph provides a DOT serialization.
- /d3graph provides a JSON serialization for D3 visualization.
- /graph provides a generic JSON serialization.

Focused on real-time traffic flow, Vistio is an application that helps you visualize the traffic of your cluster from Prometheus data.

TOPOLOGY - KIALI

Summary

Istio doesn’t provide visibility of source traffic that flows into ingress gateways,

making visualizations less insightful.

Projects like, ServiceGraph and Kiali help visualize either configuration or traffic flows through Istio.

Chapter 6. Advanced Scenarios

For most users, a basic topology in which you have deployed Istio into a single Kubernetes cluster using either the official Helm Chart or the raw YAML is sufficient. Yes, while we're all excited to see the day when microservices rule the world and monolithic applications are relegated to the dusty pages of history, we're not there, yet. The Istio project understands this, and as such, supports a variety of deployment and configuration models. In this chapter, we will touch on a handful of the more useful advanced topologies. Advanced topologies are those that you will likely need either to facilitate adoption of microservices and a service mesh or to take advantage of a distributed service mesh across regions or providers.

Types of Advanced Topologies

While there is a multitude of possible topology configurations, let's highlight three foundational topologies as an overview.

SINGLE CLUSTER MESHES

One type of advanced topology is that of *mesh expansion*, which is where you include traditional, non-microservice workloads -- so monolithic apps -- running on bare-metal or virtual machines or both, into your Istio service mesh. While these apps don't receive all benefits offered by the service mesh, this does allow you to begin to gain insight into how these services are communicating with each other and lays the groundwork for a migration into a cloud native architecture or for divvying workload across Kubernetes and non-Kubernetes nodes. We'll dig into this a bit more later on in this chapter. The point here is to associate *mesh expansion* with onboarding brownfield applications into a mesh where you can observe the traffic, and even more importantly, affect change through route rules to test these services when they're not running on the VM or host machine anymore.

MULTIPLE CLUSTER MESHES

Single mesh across multiple clusters that can span multiple clusters as long as network

connectivity no IP address range overlap. Just works out of the box in Istio 1.0. You extend Istio 1.0 to non-flat networks, where there is IP address overlap between clusters by adding VPNs and NATs or extend to support mesh federation, but this requires a lot of additional tweaking and configuration within Istio.

In Istio 1.1, flat networking is no longer required. Two path to multi-cluster management are provided: Split Horizon EDS or Gateway to Gateway.

Title: Cross-cluster Load Balancing still improving

Generally, you will ideally have requests for a given service load balanced to other replica pods within the same cluster, not inefficiently sent to replicas in another cluster. You want to keep traffic local.

Even with 1.1's enhancements on multi-cluster support, note that only weighted load balancer. Weighted round-robin is the only supported . Default is 50/50. Would ideally incorporate latency as a factor.

How does Istio support Multiple ingress - one for TLS and one for not TLS?

We are overloading the SNI (to include the target service and the IP address that is trying to be accessed) and in this way, the Istio Gateway has sufficient information on how to route to the right gateway. To separate services by Gateway, Istio will need to be extended.

In addition to cross-cluster load balancing, single cluster load balancing is still improving as we acknowledge that while Envoy supports several sophisticated load balancing algorithms, Istio currently allows three load balancing modes: round robin, random, and weighted least request.

In order to have mTLS working correctly across clusters, must use a Shared Root CA.

Two other types of advanced topologies are that of *Istio Multicluster* and *Istio Cross-Cluster*. We put these two topologies into the same group is because, in essence, they intend to solve the same problem. They provide for communication between disparate

Kubernetes clusters running individual service meshes. But, this is where language gets a bit tricky and confusion comes in as both solve this problem in diametrically opposed ways.

To keep this simple we'll summarize these two approaches.

Default autoscaling for a Gateway with minimum of 1 and maximum of 5 as the default.

Policy checks are part of the data flow (policy needs to be checked to verify whether a flow is allowed). 1 minute cache in local sidecar. Mixer v2 pushes much of the policy checking down to the sidecar, allowing for distributed definition of policy.

Webassembly in Envoy will allow Mixer filter to be created.

Istio Multi-cluster (single mesh)

Istio Multi-cluster (single mesh) is a centralized approach for connecting multiple service meshes into a single service mesh. This is done by selecting a cluster that serves as the centralized component with the others being remotes. A variety of requirements apply which we dig into later on.

Single logical view of all the services in the cluster. The implementation of this could be with two control planes that are synchronized with each other (perhaps via GitOps) or one control plane that operates as one single service mesh; a set of clusters that are part of the same mesh.

As the name implies, single mesh combines multiple clusters into one unit, managed by one Istio control plane. It can be implemented as one "physical" control plane or as a set of control planes all synchronized with replicated configuration. This would typically use additional tooling, driven by shared CI/CD pipelines and/or GitOps practices.

Istio Cross-cluster (mesh federation)

Istio Cross-cluster (mesh federation) is a decentralized approach for unifying service meshes. Each cluster is running its own control plane and data plane. You can have two or more clusters participating in the service mesh regardless of their region or cloud

provider.

Cross-cluster deployments facilitate for relatively different service meshes under different administrative domains running in different regions. With on those individual administrative domains in mind, an advantage to this mesh federation pattern is that connections between clusters can be selectively made, and in turn, so can the exposure of one cluster's services to other clusters.

Use Cases

As you can see, already with the advanced configurations we've touched on we've opened up a slew of use cases. For those still scratching their heads, here's the elevator pitch for exploring whether or not you should adopt these more advanced approaches. Let's keep in mind our mantra that you should be able to secure, observe, control, and connect your services regardless of where they are running or what they are running on -- public, private or hybrid cloud.

HIGH AVAILABILITY (CROSS-REGION)

With both multi-cluster and cross-cluster, you can enable a cross-region story. This means you can have a Kubernetes cluster on deployed in two separate regions with service traffic being routed, securely, across those two regions. It is also possible to do failover between these regions with cross-cluster setups, meaning when one region drops your application doesn't necessarily need to drop.

CROSS-PROVIDER

Expanding on cross-region, both topologies can support multi-cloud setups between providers; however, the requirements to do this and, for that matter, cross-region differ significantly. We'll dig into that further into this chapter. But, simply put Istio will allow you to achieve multi-cloud easily.

DEPLOYMENT STRATEGIES

With a multiple cluster setup, you can easily put a canary online on lower cost instances at a lower cost provider. Imagine spinning up a canary at DigitalOcean that you can pass 1% of traffic to from your production environment running on Amazon.

This is possible with both topologies. The same carries through for A/B and blue/green.

DISTRIBUTED TRACING FOR THE MONOLITH

Using service mesh expansion, your monolithic app becomes less opaque. Once you've expanded your mesh to include traditional VMs or metal, you can gather tracing and telemetry information and more.

MIGRATION

A few migration scenarios open up. With cross-cluster you can now migrate your service across regions or across providers. It's all about the routing of your service traffic. But, one of the more interesting migration scenarios is the ability to take brownfield applications and transition them piecemeal into Kubernetes. This helps give your brownfield application a bit of cloud native varnish to it. Now, your brownfield can communicate, if you wanted it to, with new services you're deploying within the cluster.

All of these scenarios will begin to make sense as we dig deeper into each further into this chapter. Once you have finished the exercises you should have a basic understanding of how to set each up and how each works. We will be using the same demo application throughout to allow us to highlight the differences. We leverage the common bookinfo app to illustrate some of this.

Choosing a topology

The design of each deployment topology comes with implementation concessions. It's highly likely that the approach you select will be and probably should be dictated by where your compute lives. If you're using only clusters in the public cloud, then cross-cluster might make better sense. If you're running some on-prem clusters with a cluster or two in the public clouds, then multi-cluster might make sense. That's not to say you can't use cross-cluster between on-premise and the public cloud providers, especially as on-premise begins to model itself on how public cloud is delivered -- see solutions like NetApp HCI, AzureStack, GKE on-premise, and more.

First, the calling workload (a.k.a. client) needs to resolve the remote workload's name

to a network endpoint. This would typically be done using DNS in Kubernetes, though other discovery systems (such as Consul) can also be used. To successfully resolve a name to an endpoint, the service must somehow be registered in the client's local DNS server or registry.

Given a network endpoint, the client makes an outgoing call and sends a request on it. These are intercepted by the Envoy sidecar proxy using information it received from Pilot (which, in turn, received it from Galley). The connection and request are mapped to an upstream and a specific endpoint and then routed to the remote endpoint. Depending on network topology and security requirements, the client-side Envoy may connect directly to the remote endpoint, or the connection might need to be routed through Istio's egress and/or ingress gateways.

The remote (or server-side) proxy accepts the connection and validates the identities using mutual TLS exchange. Implicit in this is the need for certificates to share a common root of trust, even when signed by different Citadels.

An access check may be needed, in which case identities from different clusters are sent to the Mixer and matched against the set policies. Once the response is returned, operational information, such as latency, path, return codes, etc., may be collected and logged. We may want to add cluster information to this data to allow determination of where calls originated and completed. Without having this context, it could be difficult to determine if the high latency observed is indicative of an issue (such as a failure or load problem) or it could be the result of making a long distance call to another cluster.

Cross-Cluster or Multi-Cluster?

Let's dive into the deep end of the pool. As we explained above, the best way to think about Istio *multi-cluster* vs *cross-cluster* is to think centralized versus decentralized control planes, respectively. Overtime, these have become the two dominant approaches to connecting multiple Kubernetes clusters running a service mesh together. Both have their advantages and disadvantages. We'll take you through those pros and cons.

First, let's begin with *Istio Multi-cluster*.

Figure 14.x shows you where the control plane and data plane components live. Each remote service mesh must be able to reach the central control plane for its management components such as Pilot and Mixer. It will also need access for stats and telemetry along with tracing. All clusters participating in the service mesh must have a unique CIDR range and be routable amongst themselves. The high bar here is commonly to do this across providers or across regions is to stand-up a private tunnel between the clusters. Depending on your setup this could be a VPN between on-premise and the cloud provider or, if across regions. In AWS for example, you might use something VPC peering.

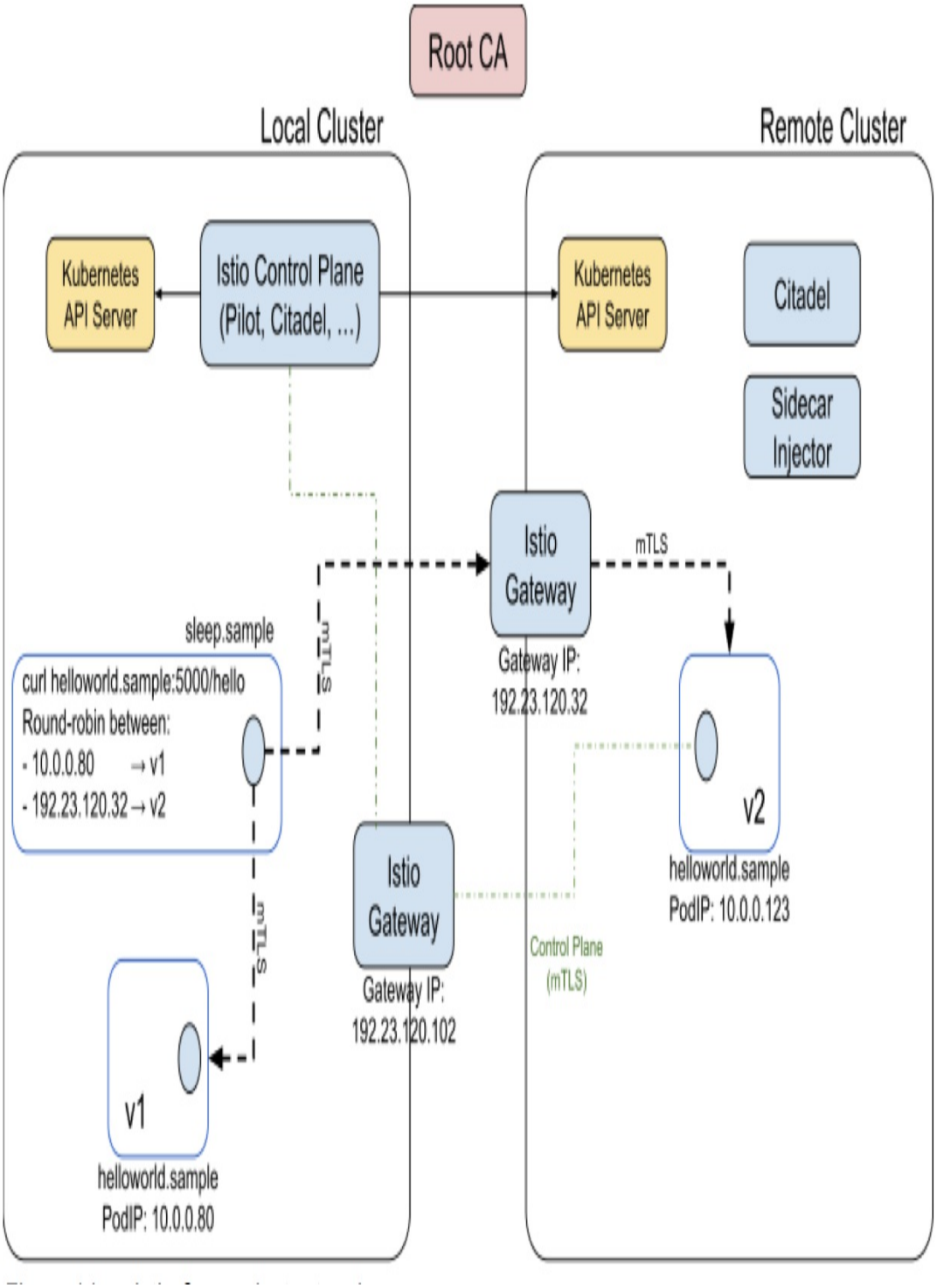


Figure 6-1. Figure 14.1 Istio Cross-cluster topology

In Figure 14.1, we see two clusters. The first cluster running much of the control plane, while cluster two running just admission controller to auto inject init containers. You can extend it to support non-flat networks, using NAT or VPN.

In an Istio Multi-cluster environment only one cluster would run all the control plane components with communication being routed to those components from the remote

installations. The remote installations would be setup with automatic sidecar injection and Citadel.

One use case for this type of deployment would be to bridge an on-premise Istio service mesh into the public cloud via a VPN. This allows developers to verify, through things like canaries or A/B or something custom, their service on a cloud provider, reducing the need to have that traffic reach the on-premise production environment. It would allow for a more easy migration -- step-by-step -- into the public cloud or a more hybrid approach if regulatory requirements bind you to locality. This is one of many stories that open up when you have hybrid connectivity into the public cloud from a private environment.

It also isn't a requirement that you co-locate workload where your control plane runs. For some use cases, this could become an interesting configuration. Centralized control plane dedicated to those components, remote data planes where workload runs.

Our second option is *Istio Cross-Cluster*. This is illustrated in figure 14.x below.

Cross-cluster deployments allow for a decentralized group of Istio service meshes to be federated using Istio route rules that we deploy within each service mesh. In this scenario, each Kubernetes cluster is running its own instances of the control plane. Both are being leveraged to run workloads.

From an end-user point of view, your requirements are simpler for cross-cluster than they are for multicluster. For one, you don't need to setup a direct connect, VPN, VPC peering, or something similar; however, you do have the requirement of some type of ingress endpoint the cluster can communicate with and the port open to communicate across that tunnel.

Each cluster must be able to communicate with each destination cluster on the port you've chosen, e.g. 80 or 443. On a public cloud provider this would translate to a public ingress on each side. For example, you would need to ensure your source cluster can communicate via ELB to the target cluster and vice versa. The ingress is used as a *ServiceEntry* within Istio.

Let's pause here and define a *ServiceEntry*. A *ServiceEntry* contains a variety of properties that define it -- hosts, addresses, port, protocol, and endpoints. You would define a *ServiceEntry* to tell Istio about services outside of the mesh or internal to the mesh. The purpose is to tell Istio about a service that hasn't been auto-discovered by Istio internally in the mesh.

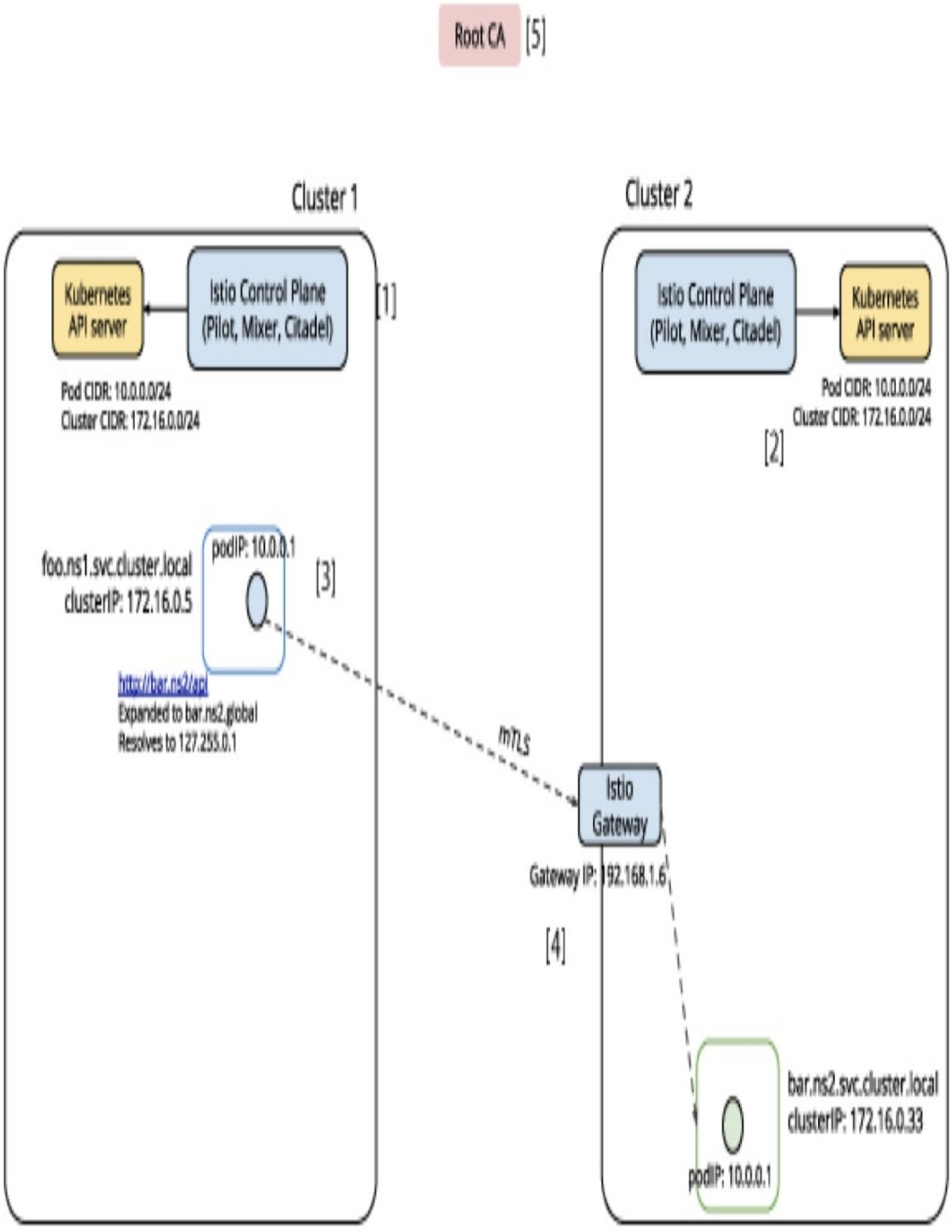


Figure 6-2. Figure 14.2: Istio 1.0 Multi-cluster topology

Cross-cluster can benefit you by removing the need for a VPN to connect each cluster together as you have to do with multicluster. It also protects you from having a single point of failure. The downside will be policies, for now, stay unique to each

environment and if you wish to apply duplicate policies to both you need to do that separately. Solutions such as Galley will hopefully provide configuration management services for Istio.

Additionally, since these are disparate Kubernetes clusters you still need to find a solution for replicating objects you need to be in both environments. In Kubernetes this can be solved through Kubernetes Federation which would allow you to ensure most objects get federated and created on each cluster participating as members in the federation.

Another problem is you still need to solve cloud load balancing if you've chosen to go down the path of running your services across multiple cloud providers and want true redundancy. We cover this further into the chapter.

Istio multicluster **does not** equal multicloud out of the box. Remember, the requirement is that all Kubernetes clusters must be able to route traffic to each other with no CIDR overlap. If not planned properly, Operators could run into headaches when they do choose to move towards unifying their infrastructure under a single service mesh like Istio. So, before you begin to look at adopting Istio multicluster, first inspect your current network setup and topology. Were you re-using the same CIDR space each time you built a new cluster? If so, you need to re-IP your clusters. Can your service traffic reach all services in both clusters? If not, then you need to ensure you can route traffic.

Multicluster

The first approach for multiple service meshes participating together as one is *Istio Multicluster*. In this approach, we rely on a centralized control plane that runs all of the Istio components. We then import remote remote clusters' data plane. Note, multicluster creates a single point of failure with the centralization of the control plane; however, you can also dedicate a cluster to the control plane while your services are all living on the remote clusters, too.

In this exercise we will need to ensure unique CIDRs exist across all clusters that will participate in the mesh, they must be able to route without issue, and the Kubernetes API itself must be routable across the mesh.

With an Istio Multicluster topology, Envoy running on the remote clusters would communicate with the centralized control plane. Setup is pretty simple. We assume you are familiar with how to install and build a Kubernetes cluster. If you don't wish to go through the hassle, we recommend using two clusters built on AWS in the same VPC for this exercise.

Handily, Helm Charts have been created that automate much of this setup; however, we'll walk you through what each step is doing so you understand what's happening under the hood.

First, clone the official GitHub repository for Istio found at https://github.com/istio/istio. Once that has been done pick whatever cluster you want to be your control plane and which will be your remote. You would then deploy the full Istio package to the control plane cluster. To do this, follow the standard installation process for deploying Istio.

You will only need to perform installation of the control plane on one cluster. At this point, pause and collect the following information from your central cluster:

- Pod IP for Pilot
- Pod IP for Mixer
- Pod IP for Statsd
- Pod IP for Mixer telemetry
- Pod IP for Zipkin, now Jaeger.

Next, follow these steps to bring in a remote data plan into your central control plane. These steps are borrowed from the official Istio documentation:

Generate your YAMLs based upon a template. This is done using Helm's templating feature. We assume you have Helm installed locally and the Istio repository cloned. You will issue the following command:

helm template install/kubernetes/helm/istio-remote --namespace istio-system \

--name istio-remote \

--set global.remotePilotAddress=*PILOT_POD_IP* \

--set global.remotePolicyAddress=*POLICY_POD_IP* \

--set global.remoteTelemetryAddress=*TELEMETRY_POD_IP* \

--set global.proxy.envoyStatsd.enabled=true \

--set global.proxy.envoyStatsd.host=*STATSD_POD_IP* \

--set global.remoteZipkinAddress=*ZIPKIN_POD_IP* > $HOME/istio-remote.yaml

You will need to run the following against your **remote** Kubernetes cluster.

Create the *istio-system* namespace manually.

kubectl create ns istio-system

Next, apply the Helm template output you generated above.

kubectl apply -f $HOME/istio-remote.yaml

You'll want to ensure the remote clusters all have automatic sidecar injection enabled. Note, if you try this using EKS it will not work as of this writing.

kubectl label namespace default istio-injection=enabled

Your next hurdle is you will need to generate a *kubeconfig* file for each remote cluster. This will allow Istio to discover services, endpoints, and pod attributes across all clusters, not just the central cluster. Once you've created each *kubeconfig* file, you will need to place those in the working directory of the ServiceAccount for the remote cluster.

On each remote cluster you would instantiate the secrets by doing the following:

kubectl create secret generic ${CLUSTER_NAME} --from-file ${KUBECFG_FILE} -

n ${NAMESPACE}

kubectl label secret ${CLUSTER_NAME} istio/multiCluster=true -n
${NAMESPACE}

<multicluster follow-up>

Cross-Cluster

In this approach, we setup and configure Istio rules for *Gateways*, *ServiceEntries*, and *VirtualServices*. We defined *ServiceEntry* earlier in the chapter, but we haven't defined what a *Gateway* or *VirtualService* is.

A Gateway opens up our cluster to be accessed, via TLS, by another Kubernetes cluster running Istio and meeting the same, base requirements. We do this by applying a group of Istio route rules using *mcc* as our tool of choice, an open-source tool for managing cross-cluster Istio setups.

You will need to ensure a few requirements before you give this a shot. First, pull down all the YAML from the book's GitHub repo found here. In our exercise we will show you how to stitch two disparate Istio service meshes together across two separate cloud providers.

Second, as of this writing, you need to have a Kubernetes cluster running version 1.11 or higher along with Istio installed running 1.0 or higher. We recommend using a product like NetApp Kubernetes Services (NKS), Google Kubernetes Engine (GKE), or Azure Kubernetes Service (AKS) to get these environments online with the correct requisites in place to save you some time. Of course, you don't need to use those services. Any CNCF conformant Kubernetes deployment should suffice.

In our demo, we leverage NKS to build a Kubernetes 1.12 cluster on AWS and Azure. With NKS we're able to deploy all pre-reqs in a single action; however, you're not bound to this.

The reason we go with public cloud providers for this example is the ease at which we can get a public ingress for both clusters. Cross-cluster requires an external IP to allow

service traffic to transit the public Internet. It also illustrates how Istio can help you elegantly solve multicloud cluster communication.

SETTING UP CROSS-CLUSTER

In our example, we'll be unifying the service mesh across two different clusters, each running at a cloud provider. This example *does not* cover requirements for clusters running in private datacenters; however, the requirements would be similar. To do this onpremise, you will need to punch a hole in your firewall to allow ingress traffic to reach the service mesh. If you want to do multicloud between private and public cloud, we would recommend Istio Multicluster described earlier.

You will need to ensure you have *kubectl* access to both clusters. You will not need shell access. Note, though, all steps need to be performed on each cluster. In spots where you are required to have information from the cluster such as its IP we will call that out and explain why it is needed and where to obtain it.

In our example we'll be deploying *bookinfo*, the default Istio demo app, to both clusters. This ensures we don't need to perform any hack in CoreDNS; however, if you want to have partial services you will need to make an edit to the CoreDNS ConfigMap.

This can be done with following template:

```
template IN A remote {

match (.*?)[.]remote[.]$

answer "{{ .Name }} 3600 IN A 10.1.1.1"

}
```

This would be added because Envoy Proxy will expect a response from CoreDNS so we have it respond with a dummy (bad) IP address. This IP can be anything as long as it isn't routable within the CIDR of your Kubernetes cluster.

First, you will need to create your *Gateway*. This is done on both **Cluster A** and **Cluster B**. In our example our clusters are named *spc44nvwmq* for Cluster A and *spc78rkksh* for Cluster B).

You would use *kubectl apply -f gateway.yaml* against both clusters.

apiVersion: networking.istio.io/v1alpha3

kind: Gateway

metadata:

generation: 1

name: nks-ingress-gateway

namespace: “”

resourceVersion: “”

selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default/gateways/nks-ingress-gateway

uid: “”

spec:

selector:

istio: ingressgateway

servers:

- hosts:

- '*'

```
port:

name: http

number: 80

protocol: HTTP

- hosts:

- '*'

port:

name: https

number: 443

protocol: HTTPS

tls:

caCertificates: /etc/istio/ingressgateway-ca-certs/ca-chain.cert.pem

mode: MUTUAL

privateKey: /etc/istio/ingressgateway-certs/tls.key

serverCertificate: /etc/istio/ingressgateway-certs/tls.crt
```

You'll notice that we provide paths to *tls* certificates.

[DESCRIBE TLS STUFF]

Next, you would switch your context to **Cluster A** and apply the following again with *kubectl*. You will need to populate the *endpoints* entry with **Cluster B**'s ingress and the *hosts* entry with the remote cluster service name.

apiVersion: networking.istio.io/v1alpha3

kind: ServiceEntry

metadata:

generation: 1

name: nks-egress-service-entry

namespace: “”

resourceVersion: “”

selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default/serviceentries/nks-egress-service-entry

uid: “”

spec:

endpoints:

- address: aa99e9d64b25011e8918e02ccbc3211d-4890909.us-east-2.elb.amazonaws.com

hosts:

- svc.spc78rkksh.remote

location: MESH_EXTERNAL

ports:

- name: https

number: 443

protocol: HTTPS

resolution: DNS

Now, switch your context to **Cluster B** and create the following *ServiceEntry* on the other cluster. Before you do this, ensure you've changed the *endpoints* and *hosts* entry to point to **Cluster A**. Note, as a reminder, you must be able to reach the endpoints from within the cluster.

apiVersion: networking.istio.io/v1alpha3

kind: ServiceEntry

metadata:

generation: 1

name: nks-egress-service-entry

namespace: ""

resourceVersion: ""

selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default/serviceentries/nks-egress-service-entry

uid: ""

spec:

endpoints:

- address: a6491f5c5b25011e8a5b702e39755299-1119496581.us-east-2.elb.amazonaws.com

hosts:

- svc.spc44nvwmq.remote

location: MESH_EXTERNAL

ports:

- name: https

number: 443

protocol: HTTPS

resolution: DNS

At this point, you're ready to split traffic up across both clusters. In this example, we're going to split traffic from **Cluster A** to **Cluster B**. This is done with a *DestinationRule*. We are leveraging *bookinfo*'s "reviews" service as our example. In this setup both services are running on both clusters. This makes the exercise a bit easier. As noted earlier, if we don't do this we would need to add a CoreDNS template to make it all work correctly.

A *DestinationRule* tells Istio where to send the traffic. These rules can specify various configuration options. In the following example we're creating a *DestinationRule* that allows for mTLS origination for egress traffic on port 443.

You will want to switch to the context of **Cluster A** and then apply the following rule. Our *host* attribute defines our remote **Cluster B**. Our traffic will route across 443 to the external cluster, too.

apiVersion: networking.istio.io/v1alpha3

kind: DestinationRule

metadata:

generation: 1

name: nks-reviews-tls-origination

namespace: “”

resourceVersion: “”

selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default/destinationrules/nks-reviews-tls-origination

uid: “”

spec:

host: svc.spc78rkksh.remote

trafficPolicy:

portLevelSettings:

- port:

number: 443

tls:

caCertificates: /etc/certs/cert-chain.pem

clientCertificate: /etc/certs/cert-chain.pem

mode: MUTUAL

privateKey: /etc/certs/key.pem

We then create a *VirtualService* on **Cluster A**, too. Remember, we’re looking to split traffic from **A** to **B**. In our example we’re route 50% of our traffic destined to our “reviews” service to **Cluster B** while leaving the remaining 50% going to the service running locally in **Cluster A**.

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

generation: 1

name: nks-reviews-egress-splitter-virtual-service

namespace: “”

resourceVersion: “”

selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default/virtualservices/nks-reviews-egress-splitter-virtual-service

uid: “”

spec:

hosts:

- reviews.default.svc.cluster.local

http:

- rewrite:

authority: reviews.default.svc.spc78rkksh.remote

route:

- destination:

host: svc.spc78rkksh.remote

port:

number: 443

weight: 50

- destination:

host: reviews

weight: 50

We then need to add some additional *VirtualServices* to allow ingress traffic to actually reach the “reviews” service, i.e. users browsing the app from the public Internet. The following will create this and should be done on **Cluster A**. We define the previous gateway we created earlier on in the chapter and the URI for the service.

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

generation: 1

name: bookinfo-vs

namespace: “”

resourceVersion: “”

selfLink:
/apis/networking.istio.io/v1alpha3/namespaces/default/virtualservices/bookinfo-vs

uid: “”

spec:

gateways:

- nks-ingress-gateway

hosts:

- '*'

http:

- match:

- uri:

prefix: /productpage

route:

- destination:

host: productpage

Lastly, on **Cluster B** we create another *VirtualService* to allow traffic to hit the service directly. Here we're defining the remote service in our *hosts* attribute and the route destination.

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

generation: 1

name: nks-reviews-ingress-virtual-service

namespace: ""

resourceVersion: “”

selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default/virtualservices/nks-reviews-ingress-virtual-service

uid: “”

spec:

gateways:

- nks-ingress-gateway

hosts:

- reviews.default.svc.spc78rkksh.remote

http:

- route:

- destination:

host: reviews

By this point, you should be able to see 50% of your traffic hitting the service on **A** and 50% hitting the service on **B**.

You can play around with various scenarios using the above setup outside of splitting traffic equally. Cross-cluster now opens up the door for a variety of stories -- circuit breaking across providers or regions, performing canaries on lower cost clusters elsewhere, and so on.

CERTIFICATES

Implicit in this is the need for certificates to share a common root of trust, even when signed by different Citadels. Citadel is leveraged to generate and manage our mTLS

certificates. It provides those certificates to Envoy to use for mTLS. These same certificates exist in both clusters and are used for securing our cross-cluster communications. Citadel is described in-depth in Chapter 10.

About the Authors

Lee Calcote, senior director of technology strategy at SolarWinds, is an innovative thought leader, passionate about developer platforms and management software for clouds, containers, infrastructure, and applications. Advanced and emerging technologies have been a consistent focus through Calcote's tenure at SolarWinds, Seagate, Cisco, and Pelco. An organizer of technology meetups and conferences, a writer, author, and speaker, Lee is active in the tech community.

Jack Butcher is a core contributor to Istio.