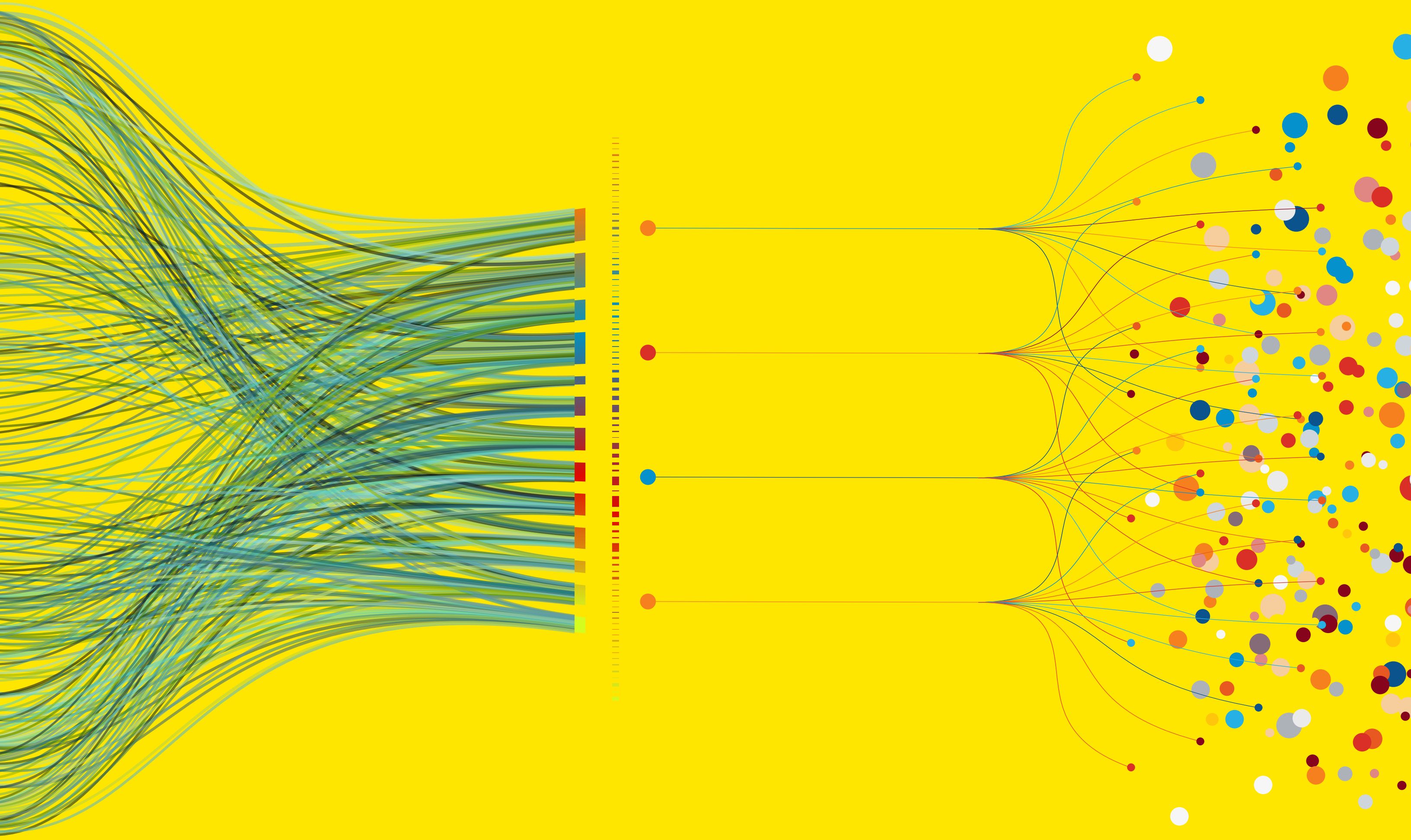
The InfoQ eMag / Issue #75 / July 2019

Service Mesh

PAST, PRESENT, AND FUTURE



InfoQ

API Gateways and Service Meshes: Opening the Door to Application Modernisation

Inter-Cluster Communication Using a Service Mesh

The Potential for Using a Service Mesh for Event-Driven Messaging

FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

InfoQ @ InfoQ InfoQ InfoQ

Service Mesh

IN THIS ISSUE

GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

CONTRIBUTORS

William Morgan

is the co-founder and CEO of Buoyant, a startup focused on building open-source service-mesh technology for cloud-native environments. Prior to Buoyant, he was an infrastructure engineer at Twitter, where he helped move Twitter from a failing monolithic Ruby on Rails app to a highly distributed, fault-tolerant microservice architecture. He was a software engineer at Powerset, Microsoft, and Adap.tv, a research scientist at MITRE, and holds an MS in computer science from Stanford University.

Andrew Jenkins

is the CTO at Aspen Mesh, where he's building an enterprise service mesh to help organizations take the burden out of managing microservices. His expertise includes software development in C++, JavaScript (Node.js), Python, C, Go, and Java. Jenkins also has experience in software and hardware testing, FPGAs, and board design for space scientific instruments.

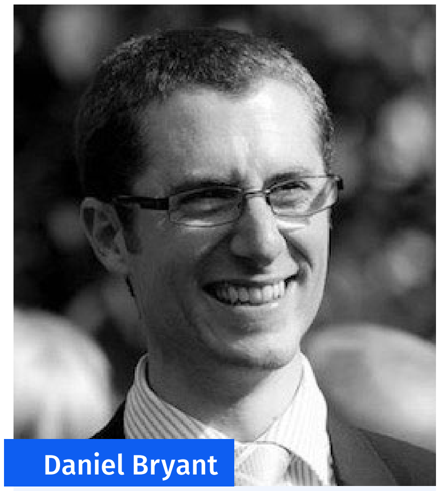
Kasun Indrasiri

is the director of Integration Architecture at WSO2 and is an author/evangelist on microservices architecture and enterprise-integration architecture. He wrote the books Microservices for Enterprise (Apress) and Beginning WSO2 ESB (Apress). He is an Apache committer and has worked as the product manager and an architect of WSO2 Enterprise Integrator.

Christian Posta

(@christianposta) is Global Field CTO at Solo.io, former Chief Architect at Red Hat, and well known in the community for being an author (Istio in Action, Manning, Microservices for Java Developers, O'Reilly 2016), frequent blogger, speaker, open-source enthusiast and committer on various open-source projects including Istio, Kubernetes, and many others.

A LETTER FROM THE EDITOR



Daniel Bryant

works as an Independent Technical Consultant and Product Architect at Datawire. His technical expertise focuses on 'DevOps' tooling, cloud/container platforms, and microservice implementations. Daniel is a Java Champion, and contributes to several open source projects. He also writes for InfoQ, O'Reilly, and TheNewStack, and regularly presents at international conferences such as OSCON, QCon and JavaOne. In his copious amounts of free time he enjoys running, reading and traveling.

The concept of using a "service mesh" to manage service-to-service communications within a cluster burst onto the scene back in 2016, driven partly by the hype-driven adoption of microservices-based architectures, and also the acceptance of Kubernetes as the de facto way to run containers. Microservices brought many benefits, such as the ability to construct a system of "polyglot" services that used languages suitable for each task. However, this architectural trend also introduced challenges inherent within distributed systems that transfer data over an unreliable and heterogeneous network, such as the difficulty of implementing cross-cutting communication concerns that were previously included within the monolithic codebase, like security, reliability, and observability. The initial microservices maxim of creating "smart endpoints and dumb pipes" has now evolved, and service meshes are vying to put just enough smarts back into the pipes.

History Doesn't Repeat Itself, But it Often Rhymes

As William Morgan highlights in the first article of this emag, service meshes didn't spontaneously appear one day; they evolved from communication frameworks and libraries that large (often "unicorn") organisations were developing alongside the growth of their internal distributed systems. Twitter created the Scala-powered Finagle from which the Linkerd service mesh emerged, Netflix created an entire suite of JVM-based utilities (including the Prana sidecar for non-JVM processes), and Google developed their Stubby RPC framework that evolved into gRPC, and the Google Frontend (GFE) and Global Software Load Balancer (GSLB), traits of which can be seen in Istio. Even service meshes that has emerged outside of the unicorns, such as HashiCorp's Consul, took inspiration from the aforementioned technology, often aiming to implement the CoreOS coined concept of "GIFEE"; Google infrastructure for everyone else.

As I discuss in the second article of this emag, service meshes have been primarily designed to handle what has traditionally been referred to as "east-west" remote procedure call (RPC)-based traffic: request/

response type communication that originates internally within a datacenter and travels service-to-service. This is in contrast to an API gateway or edge proxy, which are designed to handle "north-south" traffic: communication that originates externally and ingresses to an endpoint or service within the datacenter. My article introduces the importance of effectively managing Layer 7 (OSI model) networking concerns, as many modern protocols like HTTP/2 and gRPC function at this layer, but traditional gateways and load balancers were built to work primarily with Layers 3 and 4. I also explore how the combination of a modern API gateway and service mesh can be used to drive application modernisation programs by homogenising ingress routing and service communication.

A service mesh consists of two high-level components: a control plane, and a data plane. Matt Klein, creator of the Envoy Proxy, has written an excellent deep-dive into the topic of "service mesh data plane versus control plan". Broadly speaking the data plane is responsible for "conditionally translating, forwarding, and observing every network packet that flows to and from a [network endpoint]". In modern systems the data plane is typically implemented as a proxy, such as Envoy, that is run out-of-process alongside each service as a "sidecar". Klein states that within a service mesh, the data plane "touches every packet/request in the system, and is responsible for service discovery, health checking, routing, load balancing, authentication/authorization, and observability".

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

For readers keen to understand the topic of building a control plane in more depth, I have previously published "Ambassador: Building a Control Plane for an Envoy-Powered API Gateway on Kubernetes" which explores the implementation of the Ambassador API gateway, and Christian Posta has written "Guidance for Building a Control Plane to Manage Envoy Proxy at the Edge, as a Gateway, or in a Mesh"

The implementation of current service meshes is typically focused around managing a single mesh, but it has been recognised that a future requirement will include the ability to deploy and operate meshes within multiple data centers and compute clusters. Many innovators are exploring this space, and in the third article of this emag Andrew Jenkins explores the topic of "To Multicluster, or Not to Multicluster: Inter-Cluster Communication Using a Service Mesh". Although this article primarily focuses on Istio, other work is being undertaken by the Rancher Labs team with Submariner, within HashiCorp with their Consul Gateways, and via Solo's SuperGloo.

Current Service Mesh Benefits, and Challenges

Service meshes currently provide a place to implement cross-cutting communication concerns, but in the fourth article of this emag, Kasun Indrasiri argues that lessons from the past should be acknowledged in "Application Integration for Microservices Architectures: A Service Mesh Is Not an ESB". Echoing Twitter comments made by microservices thought leader Sam Newman, this article explores how to avoid the temptation to push business or application integration logic into a service mesh, which could make the "pipes overly smart" and hence provide accidental complexity in the form of high coupling between components.

Continuing on the theme of caution, there is of course no such thing as a free lunch, and introducing a service mesh into a technology stack will incur operational overhead, both in terms of human usage and understanding, and also technology-related side effects that affects throughput and latency. Lee Calcote and the Layer5 team have created Meshery, "a multi-service mesh performance benchmark and playground", to help engineers experiment with the operation of a mesh, and also explore the performance impact of funneling all communications through a service mesh. There has been some discussion recently around benchmarking Istio, and the 1.1 release of Istio deliberately addressed some of the issues within the Mixer component.

It is also worth noting that service meshes are a rapidly evolving space. For example, Linkerd has gone through a major rewrite from Scala to Go and Rust, and even though Istio was announced as GA back in April 2018, it has undergone major architectural and operational redesigns. This does not mean that the technology is not production-ready -- in fact many organisations are openly talking about using Linkerd in production and Istio in production -- but it does demand a certain level of risk tolerance and commitment to working alongside the vendors and community.

What Does the Future Hold?

As service mesh technology is still within the early adoption phase, there is a lot of scope for future work. Broadly speaking, I believe there are three areas of particular interest: adding support for use cases beyond RPC; standardising the interface and operations; and pushing the service mesh further into the platform fabric.

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

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

The final future-looking topic of integrating service meshes with the platform fabric can be further divided into two sub-topics. First, there is work being conducted to reduce the networking overhead introduced by a service mesh data plane, such as the data plane development kit (DPDK), which is a userspace application that "bypasses the heavy layers of the Linux kernel networking stack and talks directly to the network hardware", and work by the Cilium team that utilizes the extended Berkley Packet Filter (eBPF) functionality in the Linux kernel for "very efficient networking, policy enforcement, and load balancing functionality". Another team is mapping the concept of a service mesh to L2/L3 payloads with Network Service Mesh, as an attempt to "re-imagine [network function virtualisation] NFV in a cloud-native way". Second, there are the multiple initiatives to integrate service meshes more tightly with public cloud platforms, as seen in the introduction of AWS App Mesh, GCP Traffic Director, and Azure Service Fabric Mesh.

To co-opt William Gibson's catchphrase, the future of service meshes is already here -- it's just not evenly distributed. The aim of this guide is to remove some of the confusion, and to help choose if, when, and how to deploy a service mesh. We welcome any feedback, and please do follow future developments in the service mesh space by following the topic on InfoQ.

Linkerd v2: How Lessons from Production Adoption Resulted in a Rewrite of the Service Mesh

by **William Morgan**, CEO - Buoyant

The service mesh is rapidly becoming a critical part of the modern cloud-native stack. Moving the mechanics of interservice communication (in datacenter parlance, "east-west traffic") from application code to the platform layer and providing tooling around the measuring and manipulation of this communication provides operators and platform owners with a much-needed layer of visibility and control that is largely independent of application code.

While the term "service mesh" has only recently entered the industry lexicon, the concepts behind it are not new. These concepts have been in production for a decade or more at companies like Twitter, Netflix, and Google, typically in the form of "fat client" libraries like Finagle, Hystrix, and Stubby. From the technical perspective, the co-deployed proxy approach of the modern service mesh is more like a repackaging of these ideas from library to proxy form, enabled by the rapid adoption of containers and orchestrators like Docker and Kubernetes.

The rise of the service mesh started with Linkerd, the earliest service mesh and the project to coin the term. First released in 2016, Linkerd currently features two parallel lines of development: the original 1.x branch, built on the "Twitter stack" of Scala, Finagle, Netty, and the JVM; and the 2.x branch, rebuilt from the ground up in Rust and Go.

The launch of Linkerd 2.0 represented not just a change in underlying implementation but also a significant change in approach, informed

by a wealth of lessons learned from years of production experience. This article discusses these lessons, and how they became the basis of the philosophy, design, and implementation of Linkerd 2.0.

What is Linkerd and why should I care?

Linkerd is an open-source service mesh and Cloud Native Computing Foundation member project. First launched in 2016, it currently powers the production architecture of companies around the globe, from startups like Strava and Planet Labs to large enterprises like Comcast, Expedia, Ask, and Chase Bank.

Linkerd provides observability, reliability, and security features for microservice applications. Crucially, it provides this functionality at the platform layer. This means that Linkerd's features are uniformly available across all services, regardless of implementation or deployment, and are provided to platform owners in a way that's largely independent of the roadmaps or technical choices of the developer teams. For example, Linkerd can add TLS to connections between services and allow the platform owner to configure the way that certificates are generated, shared, and validated without needing to insert TLS-related work into the roadmaps of the developer teams for each service.

Linkerd works by inserting transparent, layer-5/layer-7 TCP proxies around the services that the operator chooses to mesh. These proxies form Linkerd's data plane, and handle all incoming and outgoing traffic to their services. The data plane is managed by Linkerd's control plane, a set of processes that provide the operator with a single point for monitoring and manipulating traffic flowing through the services.

Linkerd is based on a fundamental realization: that the request traffic flowing through a microservice application is as much a part of its operational surface area as the code of the application itself. Thus, while Linkerd can't introspect the internals of a given microservice, it can report top-line health metrics by observing the success rate, throughput, and latency of its responses. Similarly, while Linkerd can't modify the application's error-handling logic, it can improve service health by automatically retrying requests to failing or lagging instances. Linkerd can also encrypt connections, provide cryptographically secured service identity, perform canaries and blue/green deploys by shifting traffic percentages, and so on.

Linkerd 1.x

Linkerd was born from our experience operating one of the world's earliest and largest microservice applications at Twitter. As Twitter migrated from a three-tiered Ruby on Rails application to a proto-cloud-native architecture built on Mesos and the JVM, it created a library, Finagle, that provided instrumentation, retries, service discovery, and more to every service. The introduction of Finagle was a critical part of allowing Twitter to adopt microservices at scale.

Linkerd 1.x was launched in 2016 and built directly on the production-tested Twitter stack of Finagle, Scala, Netty, and the JVM. Our initial goal was simply to provide Finagle's powerful semantics as widely as possible. Recognizing that the audience for a Scala library for asynchronous RPC calls was limited at best, we bundled Finagle in proxy form, allowing it to be used with application code written in any language. Happily, the contemporaneous rise of containers and orchestrators greatly reduced the operational cost of deploying proxies alongside each service instance. Linkerd thus gained momentum, especially in the cloud-native community, which was rapidly adopting technology like Docker and Kubernetes.

From these humble beginnings grew Linkerd and the service-mesh model itself. Today, the 1.x branch of Linkerd is in active use in companies around the globe and continues to be actively developed.

Linkerd lessons learned

Despite Linkerd's success, many organizations were unwilling to deploy Linkerd into production or were willing but had to make major investments in order to do so.

This friction was caused by several factors. Some organizations were simply reluctant to introduce the JVM into their operational environment. The JVM has a particularly complex operational surface area, and some operations teams, rightly or wrongly, shied away from introducing any JVM-based software into their stack — especially one playing a mission-critical role like Linkerd.

Other organizations were reluctant to allocate the system resources that Linkerd required. Generally speaking, Linkerd 1.x was very good at scaling up — a single instance could process many tens of thousands of requests per second, given sufficient memory and CPU — but it was not good at scaling down: it was difficult to get the memory footprint of a single instance below a 150-MB RSS. Scala, Netty, and Finagle worsened the problem as they were all designed to maximize throughput in resource-rich environments, i.e. at the expense of memory.

Since an organization might deploy hundreds or thousands of Linkerd proxies, this footprint was important. As an alternative, we recommended that users deploy the data plane per host rather than per process, allowing users to better amortize resource consumption. However, this added operational complexity, and limited Linkerd's ability to provide certain features such as per-service TLS certificates.

(More recent JVMs have improved these numbers significantly. Linkerd 1.x's resource footprint and tail latency are greatly reduced under IBM's OpenJ9, and Oracle's GraalVM promises to reduce it even further.)

Finally, there was the issue of complexity. Finagle was a rich library with a large feature set, and we exposed many of these features more or less directly to the user via a configuration file. As a result, Linkerd 1.x was customizable and flexible but had a steep learning curve. One design mistake in particular was the use of delegate tables (dtabs) — a backtracking, hierarchical, suffix-preserving routing language used by Finagle — as a fundamental configuration primitive. Any user who attempted to customize Linkerd's behavior would quickly run into dtabs and have to make a significant mental investment before being able to proceed.

Fresh start

Despite Linkerd's rising level of adoption, we were convinced by late 2017 that we needed to re-examine our approach. It was clear that Linkerd's value propositions were right, but the requirements it imposed on operational teams were unnecessary. As we reflected on our experience helping organizations adopt Linkerd, we settled on some key principles of what the future of Linkerd should look like:

1. **Minimal resource requirements**. Linkerd should impose as minimal a performance and resource cost as possible, especially at the proxy layer.
2. **Just works**. Linkerd should not break existing applications, nor should it require complex configuration simply to get started.
3. **Simple**. Linkerd should be operationally simple with low cognitive overhead. Users should find its components clear and its behavior understandable.

Each of these requirements posed its own set of challenges. To minimize system resource requirements, it was clear we would need to move off of the JVM. To "just work", we would need to invest in complex techniques such as network protocol detection. Finally, to be simple — the most difficult requirement — we would need to explicitly prioritize minimalism, incrementality, and introspection at every point.

Facing a rewrite, we realized we would have focused on a concrete initial use case. As a starting point, we decided to focus purely on Kubernetes environments and on the common protocols of HTTP, HTTP/2, and gRPC — while understanding that we would later need to expand all of these constraints.

Goal 1: Minimal resource requirements

In Linkerd 1.x, both the control plane and the data plane were written for the same platform (the JVM). However, the product requirements for these two pieces are actually quite different. The data plane, deployed alongside every instance of every service and handling all traffic to and from that service, must be as fast and as small as possible. More than that, it must be secure: Linkerd's users are trusting it with incredibly sensitive information, including data subject to PCI and HIPAA compliance regulations.

On the other hand, the control plane, deployed to the side and not in the critical path for requests, has more relaxed speed and resource requirements. Here, it was more important to focus on extensibility and ease of iteration.

From early on, it was clear that Go was the right language for the control plane. While Go had a managed runtime and garbage collector like the JVM, these were tuned for modern network services and did not impose even a fraction of the cost we saw from the JVM. Go was also orders of magnitude less operationally complex than the JVM and its static binaries, memory footprint, and startup times were a welcome improvement. While our benchmarks showed that Go was still slower than natively compiled languages, it was fast enough for the control plane. Finally, Go's excellent library ecosystem gave us access to a wealth of existing functionality around Kubernetes and we felt that the language's low barrier to entry and relative popularity would encourage open-source contribution.

While we considered Go and C++ for the data plane, it was clear from the outset that Rust was the only language that met our requirements. Rust's focus on safety, especially its powerful borrow checker, which enforced safe memory practices at compile time, allowed it to sidestep a whole class of memory-related security vulnerabilities, making it far more appealing than C++. Its ability to compile to native code and its fine-grained control of memory management gave it a significant performance advantage over Go and better control of memory footprint. Rust's rich and expressive language appealed to us Scala programmers, and its model of zero-cost abstractions suggested that (unlike with Scala) we could make use of that expressivity without sacrificing safety or performance.

Rust did suffer from one major downside, circa 2017: its library ecosystem significantly lagged behind those of other languages. We knew that the choice of Rust would also require a heavy investment in networking libraries.

Goal 2: Just works

With the underlying technology choices set, we moved on to satisfying the next design goal: Linkerd should just work. For Kubernetes applications, this meant that adding Linkerd to a pre-existing functioning application couldn't cause it to break nor could it require configuration beyond the bare minimum.

We made several design choices to satisfy this goal. We designed Linkerd's proxies so that they were capable of protocol detection: they would proxy TCP traffic, but could automatically detect the layer-7 protocol used. Combined with iptables rewiring at pod creation time, this meant that application code making any TCP connection would transparently have that connection proxied through its local Linkerd instance, and if that connection used HTTP, HTTP/2, or gRPC, Linkerd would automatically alter its behavior to layer-7 semantics — e.g., by reporting success rates, retrying idempotent requests, load balancing at the request level, etc. This was all done without requiring configuration from the user.

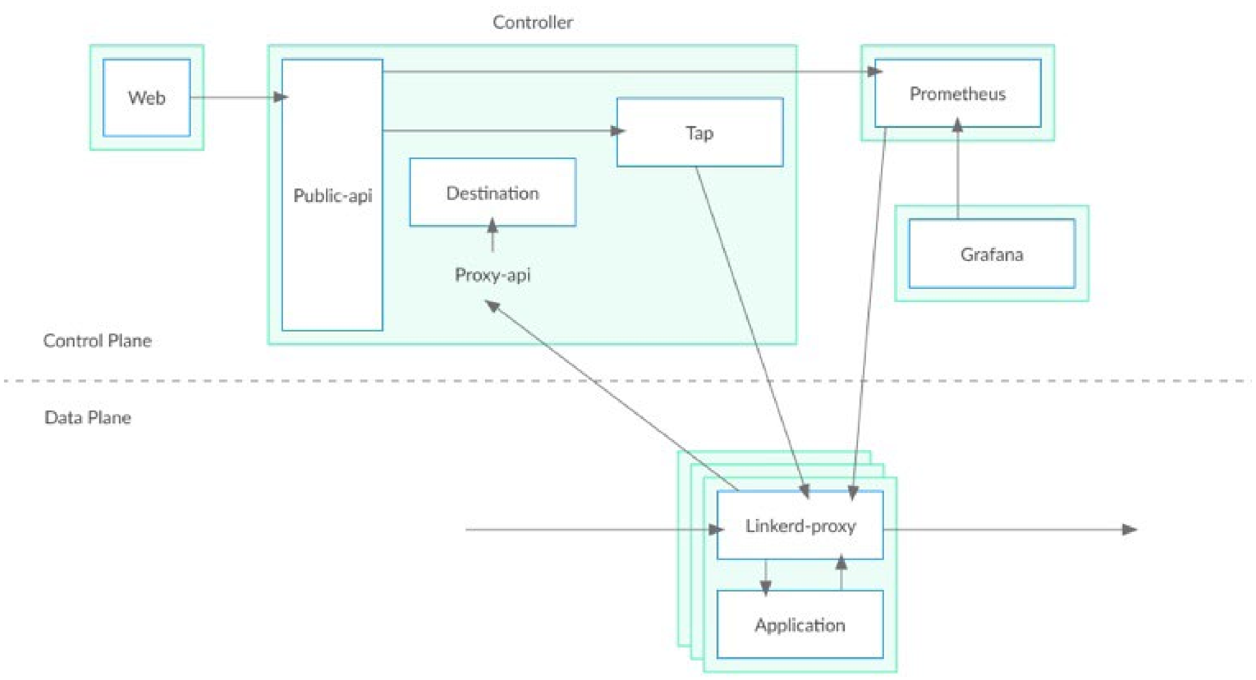


Figure 1 / Inbound and outbound TCP connections to/from application instances are automatically routed through the Linkerd data plane ("Linkerd-proxy"), which in turn is monitored and managed by the Linkerd control pane.

We also invested in providing as much functionality as possible out of the box. While Linkerd 1.x provided rich metrics on a per-proxy basis, it left the aggregation and reporting of these metrics to the user. In Linkerd 2.0, we bundled a small, time-bounded instance of Prometheus as part of the control plane so that we could provide aggregated metrics in the form of Grafana dashboards out of the box. We used these same metrics to power a set of UNIX-style commands that allow operators to observe live service behavior from the command line. Combined with the protocol detection, this meant that platform operators could get rich service-level metrics from of Linkerd immediately, without configuration or complex setup.

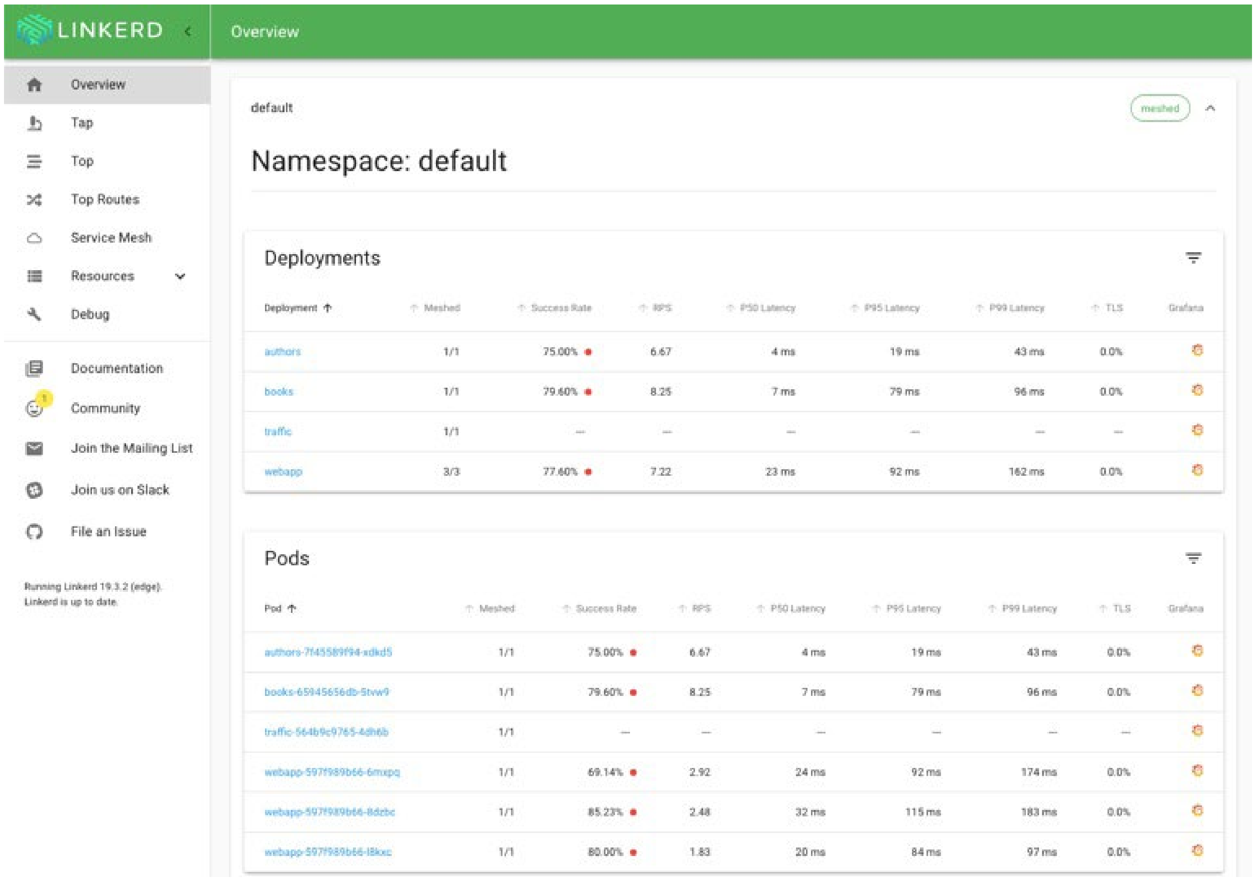
Goal 3: Simple

This was the most important goal, even though it was in tension with the goal of ease of use. (We're indebted to Rich Hickey's talk on simplicity versus easiness for clarifying our thinking on this matter.) We knew that Linkerd was an operator-facing product — i.e., as opposed to a service mesh that a cloud provider operates for you, we expect you to operate Linkerd yourself. This meant that minimizing Linkerd's operational surface area was of paramount concern.

Fortunately, our years of helping companies adopt Linkerd 1.x gave us concrete ideas about what this would entail:

- Linkerd couldn't hide what it was doing or feel overly magical.
- Linkerd's internal state should be inspectable.
- Linkerd's components should be well-defined, discrete, and clearly demarcated.

We made several design choices in service of this goal. Rather than unifying the control plane into a single monolithic process, we split it at its natural bound-



Running Linkerd 19.3.2 (edge).
Linkerd is up to date.

default

Namespace: default

Deployments

Deployment ↑	↑ Meshed	↑ Success Rate	↑ RPS	↑ P50 Latency	↑ P95 Latency	↑ P99 Latency	↑ TLS	Grafana
authors	1/1	75.00% ●	6.67	4 ms	19 ms	43 ms	0.0%	
books	1/1	79.60% ●	8.25	7 ms	79 ms	96 ms	0.0%	
traffic	1/1	—	—	—	—	—	—	
webapp	3/3	77.60% ●	7.22	23 ms	92 ms	162 ms	0.0%	

Pods

Pod ↑	↑ Meshed	↑ Success Rate	↑ RPS	↑ P50 Latency	↑ P95 Latency	↑ P99 Latency	↑ TLS	Grafana
authors-7f45589f94-xdkd5	1/1	75.00% ●	6.67	4 ms	19 ms	43 ms	0.0%	
books-65945656db-5tvw9	1/1	79.60% ●	8.25	7 ms	79 ms	96 ms	0.0%	
traffic-564b9c9765-4dh6b	1/1	—	—	—	—	—	—	
webapp-597f989b66-6mxpq	1/1	69.14% ●	2.92	24 ms	92 ms	174 ms	0.0%	
webapp-597f989b66-8dzbc	1/1	85.23% ●	2.48	32 ms	115 ms	183 ms	0.0%	
webapp-597f989b66-l8kxc	1/1	80.00% ●	1.83	20 ms	84 ms	97 ms	0.0%	

Figure 2 / The dashboard in Linkerd 2.0 mimics the look and feel of the Kubernetes dashboard, from the visual treatment to the navigation.

aries: a web service powers the web UI, a proxy-API service handles communication with the data plane, and so on. We exposed these components directly to the user on Linkerd's dashboard, and we designed the dashboard and CLI UX to fit into the idioms and expectations of the larger Kubernetes ecosystem: the `linkerdinstall` command emits a Kubernetes manifest in YAML form to be applied to the cluster via kubectl apply, the Linkerd dashboard looks and feels like the Kubernetes dashboard, and so on.

We also avoided complexity by exercising restraint. We operated on core Kubernetes nouns like deployments and pods, rather than introducing our own definitions of what a "service" was. We built on existing Kubernetes features like secrets and admission controllers whenever possible. We minimized our use of custom resource definitions because we knew that these added significant complexity to the cluster. And so on.

Finally, we added extensive diagnostics, allowing operators to inspect Linkerd's internal state and validate its expectations. We meshed the control plane (i.e., each control-plane pod has a data-plane sidecar proxying all traffic to and from it), allowing operators to use Linkerd's rich telemetry to understand and monitor the state of Linkerd, just as they do their own applications. We added commands like `linkerdendpoints`, which dumps Linkerd's internal service-discovery information, and `linkerdcheck`, which verifies that every aspect of a Kubernetes cluster and a Linkerd installation is operating as expected.

In short, we did our best to make Linkerd explicit and observable rather than easy and magical.

Linkerd 2.0 today

We launched Linkerd 2.0 in September 2018, approximately a year after beginning the efforts internally. While the value propositions were fundamentally the same, our focus on ease of use, operational simplicity, and minimal resource requirements resulted in a substantially different product shape from 1.x. Six months in, this approach has paid dividends, with many users who were previously unable to adopt the 1.x branch now happily adopting 2.x.

Our choice of Rust has garnered significant interest; while this was originally something of a gamble (in fact, we released an early version under the name "Conduit", afraid to tarnish the Linkerd brand), it is clear by now that the gamble has paid off. Since 2017, we've made significant investments in core Rust networking libraries such as Tokio,Tower, and Hyper. We've tuned Linkerd 2.0's proxy (called, simply enough, "linkerd2-proxy") to efficiently free the memory allocated to a request when the request terminates, allowing for incredibly sharp latency distributions as memory allocation and de-allocation is amortized across request flow. Linkerd's proxies now feature a p99 latency of less than a millisecond and a memory footprint of well under 10 MB, an order of magnitude smaller than Linkerd 1.x.

Today, Linkerd has a thriving community of adopters and contributors, and the future of the project is bright. With 50+ contributors to the 2.x branch, weekly edge releases, and an active and friendly community Slack channel, we are proud of our efforts and look forward to continuing to solve real-life challenges for our users while staying true to our design philosophy of simplicity, ease of use, and minimal resource requirements.

TL;DR

- Linkerd 2.0 introduced a substantial rewrite of the widely adopted service mesh, which was previously written in Scala, and was inspired by work undertaken at Twitter on the Finagle RPC system.

- This new version moved off the JVM to a split Go (for the control plane) and Rust (data plane) implementation.

- The Buoyant team made deep technical investments in the underlying Rust network stack, and refocused the UX on simplicity, ease of use, and low cognitive overhead. The result was dramatically faster, lighter, and simpler to operate.

- It has been over six months from the launch of Linkerd 2.0, and the team believe that the rewrite has paid dividends, with many users who were previously unable to adopt the 1.x branch now happily adopting 2.x.

API Gateways and Service Meshes: Opening the Door to Application Modernisation

by **Daniel Bryant**, Independent Tech Consultant | Consulting CTO | InfoQ Editor

One of the core goals when modernising software systems is to decouple applications from the underlying infrastructure on which they are running. This can provide many benefits, including: promoting innovation, for example, moving workloads to take advantage of cloud ML and "big data" services; reducing costs through the more efficient allocation of resources or colocation of applications; and improving security through the use of more appropriate abstractions or the more efficient upgrading and patching of components. The use of containers and orchestration frameworks like Kubernetes can decouple the deployment and execution of applications from the underlying hardware. However, not every application can be migrated to this type of platform, and even if they can, organisations will typically want this to be an incremental process. Therefore, a further layer of abstraction is required that decouples traffic routing from the networking infrastructure: both at the edge, via an ingress or API gateway, and within a datacenter, via a service mesh.

At Datawire, we have worked closely with the HashiCorp team over the past few months, and have recently released an integration between the Ambassador API gateway and the Consul service mesh. The combination of these two technologies allows traffic routing for applications to be fully decoupled from the underlying deployment and network infrastructure. Using the power of the Envoy Proxy, both technologies provide dynamic routing from the edge and service-to-service across bare metal, VMs and Kubernetes. The integration also enables end-to-end TLS, and adds support for other cross-functional requirements.

Application Modernisation: Decoupling Infrastructure and Applications

Many organisations are undertaking "application modernisation" programs as part of a larger digital transformation initiative. The goals of this are manyfold, but typically focus around increasing the ability to innovate via modularisation of functionality and integration with cloud ML and big data services, improving security, reducing costs, and implementing additional observability and resilience features at the infrastructure level. An application modernisation effort is often accompanied with a move towards high cardinality modern architecture patterns that strive to be loosely coupled, like microservices and function-as-a-service (FaaS), and an adoption of a "DevOps" or shared responsibility approach to working.

One of the key technical objectives with application modernisation is decoupling applications, services, and functions from the underlying infrastructure. Several approaches to this are being promoted by cloud vendors.

- AWS Outposts bring AWS services and operating models to a user's existing data center, via the installation of custom hardware that is fully managed by AWS.
- Azure Stack is an extension of Azure services that enables users to consistently build and run hybrid applications across the Azure cloud and their own on-premises hardware.
- Google Anthos extends core GCP services onto a user's infrastructure or another cloud via a software-based layer of abstraction and associated control plane.

Other projects like Ambassador, Consul, Istio and Linkerd are aiming to build on the existing cloud-agnostic, container-based abstractions for deployment, and provide a further layer of abstraction at the network to enable the decoupling of applications and infrastructure. Docker popularised the use of containers as a deployment unit, and Google recognised that the majority of applications were deployed as a collection of containers, which they named as a "pod" within Kubernetes. Here containers share a network and filesystem namespace, and utility containers that provide logging or metric collection can be composed with applications. The business functionality deployed within pods are exposed via a "service" abstraction, which provides a name and network endpoint. The use of this abstraction allows the deployment and releasing to be separated. Multiple versions of a service can be deployed at any given time, and functionality can be tested or released by selectively routing traffic to backend pods (for example, "shadowing" traffic, or "canary releasing"). This dynamic routing is typically achieved via proxies, both at the edge -- an "edge proxy", or "API gateway" -- and between services -- the inter-service proxies, which collectively are referred to as a "service mesh".

One of the biggest challenges for many organisations is implementing this application and infrastructure decoupling without disrupting end-users and internal development and operations teams. Due to the diversity of infrastructure and applications within a typical enterprise IT estate -- think mainframe, bare metal, VMs, containers, COTS, third-party applications, SaaS, in-house microservices, etc. -- a key goal is to establish a clear path that allows incremental modernisation and migration of legacy applications to newer infrastructure like Kubernetes and cloud services.

Modernisation and Migration, Without Disruption: The Role of an API Gateway and Service Mesh

The open source Envoy Proxy has taken the modern infrastructure world by storm, and with good reason: this proxy was born in the "cloud native" and Layer-7 (application) protocol-focused era, and it, therefore, handles all the characteristics of modern infrastructure and the associated developer/operator use cases very effectively and efficiently. End-user organisations like Lyft, eBay, Pinterest, Yelp, and Groupon, in combination with all of the major cloud vendors, are using Envoy at the edge and service-to-service to implement service discovery, routing, and observability. Crucially, they are often using Envoy to bridge communication between the old world of mainframes and VM-based applications to the more modern container-based services.

Although the data plane (the network proxy implementation itself) is extremely powerful, the control plane, from which the proxy is configured and observed, does have a steep learning curve. Accordingly, additional open source projects have emerged to simplify the developer-experience of using Envoy. Datawire's Ambassador API gatewayis an Envoy-powered edge proxy that provides a simplified control plane for configuring Envoy when used for managing ingress, or "north-south" traffic. HashiCorp's Consul service mesh is a control plane for service-to-service communication or "east-west" traffic, and this supports Envoy within its range of pluggable proxy configurations.

The key promise of using these two technologies is that they enable applications to run anywhere while remaining available and connected to both external and internal users:

- An API Gateway decouples application composition and location from external consumers. An API gateway dynamically routes external requests from end-users, mobile apps, and third-parties to various internal applications, regardless of where they are deployed.
- A service mesh decouples applications from internal consumers by providing location transparency. A service mesh dynamically routes internal service-to-service requests to various applications, regardless of where they are deployed.

Ambassador and Consul: Route to VMs, Containers, and More

A typical deployment of Consul consists of multiple Consul servers (providing high-availability), and a Consul agent on each node. Consul acts as the configuration "source of truth" for the entire data center, tracking available services and configuration, endpoints, and storing secrets for TLS encryption. Using Consul for service discovery, Ambassador is able to route from a user-facing endpoint or REST-like API to any Consul service in the data center, whether this is running on bare metal, VMs, or Kubernetes. Consul can also transparently route service-to-service traffic via Envoy proxies (using the service "sidecar" pattern), which ensures end-to-end traffic is fully secured with TLS.

Ambassador serves as a common point of ingress to applications and services, providing cross-cutting functionality for north-south traffic, such as user authentication, rate limiting, API management, and TLS termination. Consul acts as the service mesh and enables the definition of service names to provide location transparency, and policy-as-code to be declaratively specified to defined cross-cutting security concerns, such as "segmenting" the network. Securing service-to-service communication with firewall rules or IP tables does not scale in dynamic settings, and therefore service segmentation is a new approach to securing services via their identity, rather than relying on network-specific properties; complicated host-based security groups and network access control lists are eschewed in favor of defining access policies using service names.

Getting Started

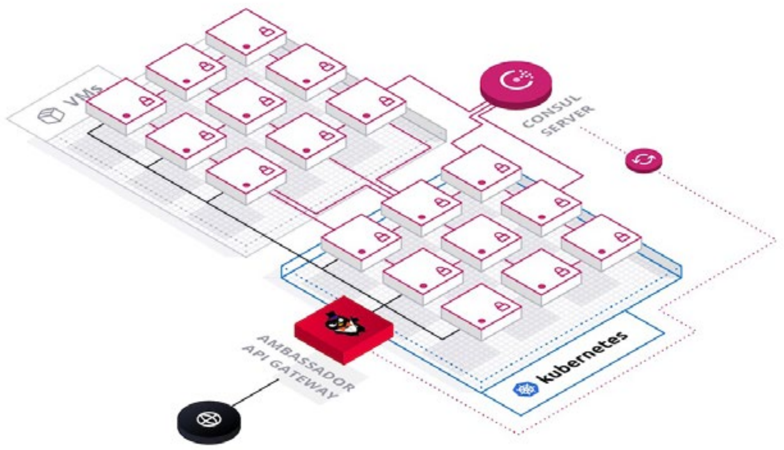
Ambassador uses a declarative configuration format built on Kubernetes annotations. So to use Consul for service discovery, you first register Consul as a resolver via an annotation placed within a Kubernetes service:

```
apiVersion: ambassador/v1
kind: ConsulResolver
name: consul
address: consul-server
datacenter: dc3
```

Ambassador can now be configured to route to any service using the standard annotation-based configuration format. All Ambassador features such as gRPC, timeouts, and configurable load balancing are fully supported. The example below demonstrates a mapping between `/foo/` and the proxy of the foo service (“`foo-proxy`”) registered in Consul:

```
apiVersion: ambassador/v1
kind: Mapping
prefix: /foo/
service: foo-proxy
timeout_ms: 5000
resolver: consul-server
tls: consul-tls-cert
load_balancer:
  policy: round_robin
```

Although optional, the tls property defines the TLS context that



Ambassador uses to communicate with the Consul service's proxy. Ambassador synchronizes TLS certificates via the Consul API automatically. To guarantee all of the traffic is secure, the service itself should be configured to only receive traffic from the Consul service proxy, for example, by configuring the service within a Kubernetes pod to bind to the local network address, or by configuring the underlying VM to only accept inbound communication via the port the proxy is listening to.

There are several additional benefits from rolling out Ambassador and Consul across your datacenter. The use of a Layer-7 aware proxy such as Envoy at the edge means that modern protocols such as HTTP/2 and gRPC will be load balanced correctly (a great discussion of why the use of Layer-4 load balancers is not appropriate in this use case, see the great post “We rolled out Envoy at Geckoboard”). Consul also provides additional primitives that are useful when building distributed systems, such as a key-value store that also includes the ability to watch entries, distributed locks, and health checks, and it also supports multiple data centers out-of-the-box.

Related Technologies

IBM, Google and several other organisations and individuals founded the Istio project to provide a simplified control plane for Envoy that focused on inter-service communication. The project later added the concept of a “gateway” for managing ingress, which is still evolving. Currently Istio only supports deployment on Kubernetes, but additional platform support is on the roadmap. Buoyant have created the Linkerd service mesh, which although primarily focused on managing east-west traffic, does also provide integrations with popular north-south proxies. The Kong API gateway team also have an early-stage service mesh solution that is powered by NGINX.

Beware of the “Big Bang” Service Mesh Rollout

Through my work at Datawire, I've talked with several organisations that have attempted an organisation-wide rollout of a service mesh. Networking operations is arguably one of the last bastions within software delivery that has remained relatively centralised -- even with the adoption of cloud and software-defined networking (SDN) -- and sometimes this leads to the thinking that any networking technology must be centrally managed. In general, this type of approach when deploying a service mesh has not gone well. It appears unreasonable to orchestrate all engineers within a large enterprise to move all applications en masse to a mesh. Instead, an incremental approach to adoption appears more practical, and I believe this should start at the edge. Once an organisation can decentralize the configuration and release of applications and products that are exposed externally, and also learn how to take advantage of the functionality offered by modern proxies, this is an ideal starting point to continue rolling out a service mesh incrementally for internal services.

The first iteration of rolling out a service mesh is typically focused on routing. As I have demonstrated in the Ambassador and Consul configurations above, once you have a modern edge proxy in place, you can selectively migrate traffic routing to your existing Consul-registered services, regardless of where or how they are deployed. Once the routing is complete you can then incrementally add a Consul proxy alongside each of your services, and route traffic securely (using TLS) from the edge to each service endpoint. It is completely acceptable to have a mix of services implementing TLS and some plaintext within a data center. The goal is typically to secure all traffic, and using the combination of Ambassador and Consul, it is possible to roll out the end-to-end encryption of traffic, from end-user to service, incrementally.

Summary

In this article, I have discussed the motivations for decoupling applications from infrastructure as part of an application modernization program. I have explored how deploying an integrated API gateway and service mesh can provide an incremental path to routing traffic from end users to both new and existing services, regardless of where these applications are deployed. If and when applications are migrated to newer platforms, their identity that is used for routing traffic from the edge to the service, or service-to-service, remains the same. In addition, there are several other benefits from implementing this gateway-to-mesh solution, including end-to-end traffic encryption and improved observability of application-level networking metrics, both globally and service-to-service.

Further details on the Ambassador and Consul integration can be found in the Ambassador docs, and a tutorial can be found within the Consul docs.

TL;DR

- Modernising applications by decoupling them from the underlying infrastructure on which they are running can enable innovation (shifting workloads to the cloud or external ML-based services), reduce costs, and improve security.
- Container technology and orchestration frameworks like Kubernetes decouple applications from the compute fabric, but they also need to be decoupled at the networking infrastructure.
- An API Gateway can decouple applications from external consumers, by dynamically routing requests from end-users and third-parties to various internal applications, regardless of where they are deployed within the data center.
- A service mesh decouples applications from internal consumers by providing location transparency, by dynamically routing internal service-to-service requests, regardless of where they are exposed on the network.

To Multicluster, or Not to Multicluster: Inter-Cluster Communication Using a Service Mesh

by **Andrew Jenkins**, CTO at Aspen Mesh

Kubernetes has become the de facto standard for container orchestration in the enterprise. And there's good reason for it — it provides a slew of capabilities that makes managing containerized applications much easier. Kubernetes also creates some new challenges, a primary one being the need to deploy and manage multiple Kubernetes clusters in order to effectively manage large-scale distributed systems.

Imagine you've already got an app designed and coded, and you've built containers — you just need to run them. Getting from code to running app is exhilarating, but as anyone who has built a containerized application knows, it's not nearly as simple as it might appear at first glance. Before deploying to production, there are various dev/test/stage cycles. Additionally, there's a scaling aspect — your production application may need to run in many different places, for reasons like horizontal scalability, resiliency, or close proximity to end users.

More environments, more (cluster) problems

You typically don't get far before even a simple greenfield app concept ends up requiring multiple deployment environments. If you're migrating an existing application, you're bound to run into even more challenges like different security domains, different organizations/billing, and affinities for one cloud vendor's machine-learning toolkit.

The most common approach to solving this problem is to create multiple Kubernetes clusters, each one dedicated to running your app components in its particular environment. In a high-security domain, you will make extensive use of Kubernetes role-based access control (RBAC) and have auditing capabilities. The test environment should reproduce a lot of production behaviors but be tailored for easy debugging and inspection. For your dev environment... — well, maybe you're like me and you'll just open up the Docker preferences and check the Kubernetes box. Ease of use and ephemerality are the name of the game here.

You'll likely end up with at least a few Kubernetes clusters, each cluster hosting microservices. The communication between these microservices in a cluster can be enhanced by a service mesh. Inside of a cluster, Istio provides common communication patterns to improve resiliency, security and observability. What about between and across clusters?

Operating multiple Kubernetes clusters doesn't have to be scary, but running multiple clusters does require you to consider how they will communicate and interact in order to easily deliver the great apps that run on top. A service mesh like Istio can make multicluster communication painless. Istio has multicluster-support, added new functionality in 1.1, and plans to add even more in the future. Teams should think about employing a service mesh to simplify communication across multiple clusters as well.

Common use cases

At Aspen Mesh, we get asked about running a multicluster service mesh most commonly for these user needs:

1. I have multiple clusters because of my organization size, and I want one place to see and manage them. My clusters don't generally do inter-cluster traffic, or when they do, it's through well-defined APIs.
2. I have multiple clusters for high availability; they are clones of each other and it's very important that if one cluster fails, the other cluster can take over.
3. I have multiple clusters that combine to form a higher-level app. Microservices in one of those clusters need to

communicate with microservices in another to provide the proper end-to-end app experience.

It's the third category where multicluster requires inter-cluster traffic. If you want inter-cluster traffic support, your implementation will depend on the networking between the clusters, and on your requirements for fault tolerance.

What you can get out of multicluster

When you're thinking about multicluster and service mesh, you should start by identifying what you want, then shift to how you get it.

Single pane of glass

Your multiple service meshes (as many as one for each cluster) operate from one place. You can view the config, metrics, and traces for all clusters in a single pane of glass.

Unified trust domain

You use a service mesh to provide workload identification, protected by strong mutual-TLS encryption. This zero-trust model is better than trusting workloads based on topological information like source IP. you're relying on cryptographic proof of what they are, not a fragile perimeter stack to control where they came from.

A unified trust domain means that all the workloads can authenticate each other (what they are) by tying back to a common root of trust. The service-mesh control planes are each configured for that common root of trust, whether there are one or several of these planes.

Independent fault domains

A cluster that does not rely on another cluster for proper operation of the cluster and associated infrastructure itself is an independent fault domain. I am including service mesh as associated infrastructure — if you're installing a service mesh, you are doing it to move communication resiliency to an infrastructure layer underneath the application. If a failure of the service mesh in one cluster can break the service mesh in another cluster, it doesn't qualify as an independent fault domain.

Intercluster traffic

You need traffic between clusters to remain part of the service mesh if you want services in one cluster to communicate directly with services in another, and you want that communication to have the benefits of a service mesh, like advanced routing, observability, or transparent encryption. In other words, you want your east/west traffic to leave one cluster, transit some intermediate network like the Internet, and then enter another cluster.

This is probably the first thought for most people when they think about a multicluster service mesh, but I'm calling it out separately here because it has implications for fault tolerance.

Heterogeneous/non-flat network

A non-flat network supports services across multiple clusters, without flat networking requirements. This means you can do things like allocate IPs in one mesh without consideration for another, and you don't need VPNs or network tunnels for communication across meshes.

If your organization has already created a bunch of different clusters without de-conflicting pod IP-address ranges or you just don't ever want to enter that morass again, this will be an attractive property to you.

Multicluster service-mesh approaches

Having laid out the different properties you might be looking for out of multicluster, I can walk through what various approaches deliver.

Independent clusters

This is the un-multicluster. Just because you have multiple clusters and each uses a service mesh doesn't mean you must adopt a unifying multicluster service mesh. Ask yourself why you ended up with multiple clusters in the first place. If you want each cluster to be its own independent fault domain, it makes sense to isolate and remove cross-cluster dependencies. If that meets your needs, there's no harm in treating a service mesh as another per-cluster service like pod scheduling or persistent disk management.

Common management

A step above the independent-clusters approach is a common management system for multiple clusters. In this model, each cluster operates independently but you manage the set of meshes via a common management interface. It's good design to have the thing you use to monitor and debug your system (or, in this case, systems) reside outside of the system itself so when the system is broken, you can still inspect and fix it.

If you choose to use a common root of trust (certificate authority or signing certificate) across these clusters then you can also have a unified trust domain.

This is a good choice if independent fault domains is a top priority. This option is a good fit for consuming software as a service because you can get one external pane of glass to unify everything, backed by a service-level agreement.

We made sure Aspen Mesh supports this use case even if you don't enable any of the inter-cluster approaches because we see value in this resiliency. This approach supports the others, too, but we're careful not to rush our users into inter-cluster traffic when it's not required.

Cluster-aware service routing through gateways

This approach in Istio involves multiple independent service meshes, one in each cluster, and some configuration trickery to allow services in one cluster to communicate with services in another cluster. You start by creating one unified trust domain for all your meshes. Next, you configure an ingress gateway to accept trusted traffic that comes from a service in another peer cluster. Finally, you configure service entries to allow traffic for certain services to be routed out of one cluster and sent to another.

This is the first method that allows services in different clusters to communicate directly with each other. Also, each cluster is still an independent mesh control plane and fault domain. This means that if the service mesh in cluster B fails, cluster A will still work; it'll just look like the services in cluster B are unavailable. The burden of configuring this cross-cluster traffic is placed on the user.

Flat network

This model dictates a service mesh across all your clusters. You arrange it so that the pods in each cluster have non-overlapping IP addresses, so any pod can route traffic to any other pod in any cluster. You might have a bunch of clusters behind a common firewall or you might build VPN tunnels between clusters. You configure your service mesh to combine the discovered pods, services, and configs from each cluster into one overall view.

A flat network makes it appear as if you have one super service mesh that spans all your clusters. There are some downsides. This super service mesh is managed by one control plane, so if it has problems, the service mesh in all clusters will have problems. If you originally partitioned into multiple Kubernetes clusters for fault tolerance, this approach negates that. Another consideration is that the control plane has to scale to manage all clusters. And you have to make this flat network perform well enough to handle the control plane and cross-cluster traffic.

Split-horizon Endpoints Discovery Service (EDS)

This approach also creates one service mesh across clusters but does not require flat networking. You still have one control plane that discovers pods, services, and configs from each cluster but Istio's EDS, which has functionality akin to split-horizon DNS, replaces the requirement for a flat network.

The sidecar for a pod in one cluster is configured with a list of endpoints for each service it wants to talk to. If a pod is in the same cluster, it shows up directly in the EDS list. If a pod is in another cluster, an ingress gateway for the other cluster appears instead. The pod chooses an endpoint to talk to and sends traffic — if the endpoint is local, the communication is direct pod to pod. If the pod chooses a remote endpoint, it sends traffic to the address of the relevant ingress gateway, marked for the service the pod wants to talk to. The ingress gateway takes the traffic and sends it to one of the pods in its cluster that implements the service. The ingress gateway uses Server Name Indication (SNI) to know where the traffic is destined.

Like the flat-network approach, this creates a unified service-mesh control plane and adds a single fault domain and single trust domain. It does not require a flat network, it only requires that one cluster can send traffic to the public address of ingress gateways for the other clusters.

To multicluster or not to multicluster

If you are going to be running multiple clusters for dev and organizational reasons, it's important to understand your requirements and decide whether you want to connect these in a multicluster environment and, if so, to understand various approaches and associated tradeoffs with each option.

If you have read this far, you probably have decided to multicluster. The real question is what's the best method to implement. Hopefully, the table below will help you to decide on the right approach for you.

	Unified Management	Unified Trust	Heterogeneous Network	Independent Fault Domain	Inter-cluster traffic
Independent			✓	✓	
Common Management	✓		✓	✓	
Flat Network	✓	✓			✓
Split Horizon	✓	✓	✓		✓
Cluster-aware Service Routing		✓	✓	✓	✓

A service mesh-like Istio can help, and when used properly can make multicluster communication painless. If you'd like to talk more about my take on why and how teams should think about employing a service mesh to simplify communication across multiple clusters. multicluster communication with a service mesh, or a service mesh at all, feel free to reach out to me at andrew@aspenmesh.io.

TL;DR

- Kubernetes has become the de facto standard for container orchestration, and many organisations run multiples clusters. Communication within clusters is a solved issue, but communication across clusters requires more design and operational overhead.
- Before deciding on whether to implement multicluster support, you should understand your communication use case.
- You should also identify what you want from a solution (single pane of glass observability, unified trust domain etc), and then create a plan on how you implement this.
- There are several multicluster service mesh approaches, such as common management, cluster-aware service Routing through gateways, flat network and split-horizon endpoints discovery service (EDS).
- Istio has existing multicluster support, additional new functionality in 1.1, and even more appearing in the future.

Application Integration for Microservices Architectures: A Service Mesh Is Not an ESB

by **Kasun Indrasiri**, Director of Integration Architecture at WSO2

Integration of APIs, services, data, and systems has long been one of the most challenging yet most essential requirements in the context of enterprise software application development.

We used to integrate all of these disparate applications in point-to-point style, which was later replaced by enterprise service buses (ESBs) alongside service-oriented architecture (SOA).

However, in modern microservices and cloud-native architecture, we barely talk about application integration anymore. That doesn't mean that all these modern architectures have solved all the challenges of enterprise application integration.

Application integration challenges have stayed pretty much the same, but the way we solve them has changed.

From ESB to smart endpoints and dumb pipes

Most enterprises that adopted SOA used an ESB as the central bus to connect and integrate all the disparate APIs, services, data, and systems.

If a given business use case required talking to different entities in the organization, it was the job of the ESB to plumb all these entities and create composite functionality.

Hence, ESB solutions are usually powerhouses of all the built-in integration capabilities, such as connectors to disparate systems and APIs, message routing, transformation, resilient communication, persistence, and transactions.

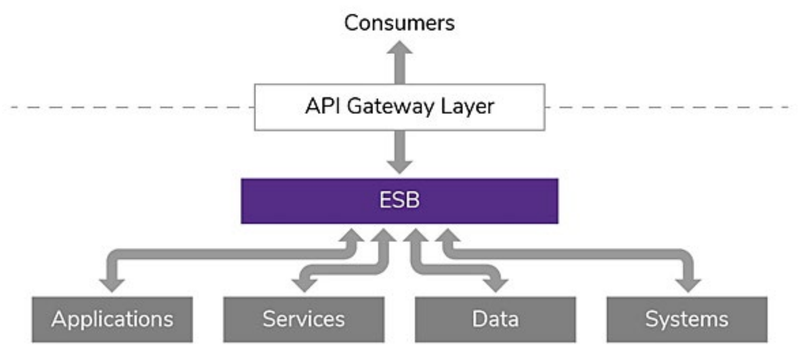


Figure 1 / Using ESB for integration

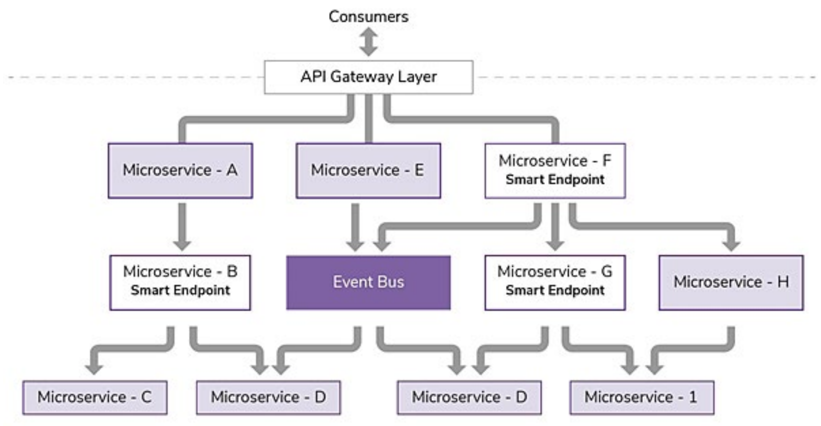


Figure 2 / Microservices inter-service communication and composition

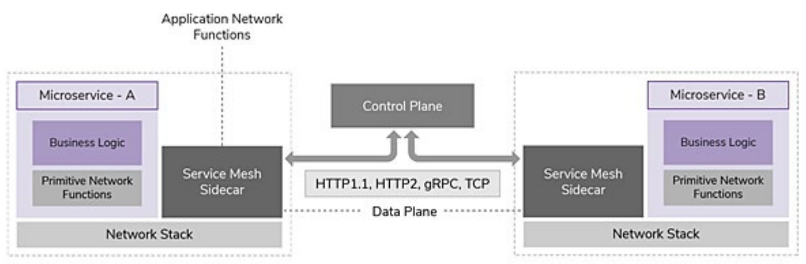


Figure 3 / Service composition logic versus service mesh

However, microservices architecture opts to replace the ESB with the building of smart endpoints and dumb pipes, meaning that your microservices have to take care of all the application integration.

The obvious tradeoff from the decentralization of the smart ESB is that the code complexity of your microservices will dramatically increase, as they have to cater to these application integrations in addition to the service's business logic. For example, figure 2 shows several microservices (B, F, and G) that act as smart endpoints, which contain both the logic of communication structure between multiple other services and business logic.

One of the other challenges with microservices architecture is how to build the commodity features that are not part of the service's business logic such as resilient communication, transport-level security, publishing statistics, tracing data to an observability tool, etc. Those services themselves have to support such commodity features as part of the service logic. It is overwhelmingly complex to implement all of these in each microservice, and the effort required could greatly increase if our microservices are written in multiple (polyglot) languages. A service mesh can solve this problem.

The key idea of a service mesh is to keep all the business-logic code as part of the service while offloading the network-communication logic to the inter-service communication infrastructure. When using a service mesh, a given microservice won't directly communicate with the other microservices. Rather, all service-to-service communications will take place via an additional software component, running out of process, called the service-mesh proxy or sidecar proxy. A sidecar process is colocated with the service in the same virtual machine (VM) or pod (Kubernetes). The sidecar-proxy layer is known as the data plane. All these sidecar proxies are controlled via the control plane. This is where all the configuration related to inter-service communications is applied.

A service mesh is not for application integration

Since a service mesh offers some of the capabilities that are part of ESBs, there is a misconception that it is a distributed ESB that also takes care of application integration. That is not correct. A service mesh is only meant to be used as infrastructure for communicating between services, and we shouldn't be building any business logic inside it. Suppose that you have three microservices called X, Y, and Z, which communicate in request/response style with X talking to both Y and Z in order to implement its business functionality (see figure 4). The composition

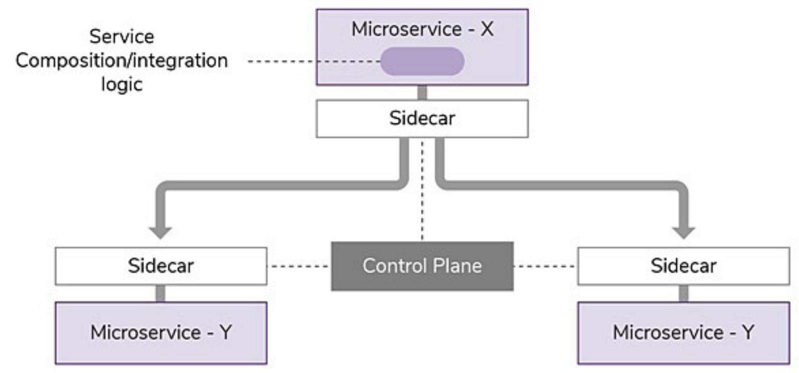


Figure 4 / Service composition logic versus service mesh

business logic should be part of microservice X's code, and the service mesh sidecar shouldn't contain anything related to that composition logic.

Similarly, for any service that uses event-driven communication, the service code should handle all the business-logic details (and it's also worth mentioning that service-mesh implementations are yet to fully support event-driven architecture). So, even if we run our microservices or cloud-native applications on top of a service mesh, the integration of those services or applications is still essential. Application integration is one of the most critical yet largely concealed requirements in the modern microservices and cloud-native architecture era.

Integration in microservices and cloud-native apps

In the context of microservices and cloud-native apps, application integration or building smart endpoints is all about integrating microservices, APIs, data, and systems. These integration requirements range from the integration of several microservices to integrating with monolithic subsystems to create anti-corruption layers. A closer look at application-integration requirements in microservices and the cloud-native applications reveals the following key capabilities that we need to have in an application-integration framework:

- The integration runtime must be cloud native, able to run smoothly within Docker/Kubernetes and provide seamless integration with the cloud-native ecosystem.
- It needs service orchestrations/active compositions so that a given service contains the logic that invokes multiple other services to compose a business functionality.
- It needs service choreography/reactive compositions

so that inter-service communication takes place via synchronous event-driven communication and no central service contains the service-interaction logic.

- I must have built-in abstractions for a wide range of messaging protocols (HTTP, gRPC, GraphQL, Kafka, NATS, AMQP, FTP, SFTP, WebSockets, TCP).
- It must support forking, joining, splitting, looping, and aggregation of messages or service calls.
- It needs to store and forward, persistent delivery, and idempotent messaging techniques.
- It must have message-type mapping and transformations.
- It must integrate with SaaS (e.g., Salesforce), proprietary (e.g., SAP), and legacy systems.
- There should be business-logic-oriented routing of messages.
- It must support distributed transactions with compensations.
- It must have long-running workflows.
- The anti-corruption layers must bridge microservices and monolithic subsystems.

All these capabilities are common in any microservices or cloud-native application, but building them from scratch can be a daunting task. This is why it's really important to carefully analyze these integration capabilities when we build microservices or cloud-native applications and to pick the right technology or framework based on the integration requirements. For example, if we need to build a service that has a complex orchestration logic then we should select the integration framework or technology that makes it easy to write those kinds of compositions. If we want to build a service that is long-running and has compensation capabilities then we need to select a framework that has built-in support for workflows and compensations (in the InfoQ article "Events, Flows and Long-Running Services: A Modern Approach to Workflow Automation", Martin Schimak and Bernd Rücker provide great in-depth analysis of the current state of workflow technologies for cloud-native architectures).

Although application integration has largely been neglected by most of the microservices experts, authors such as Christian Posta (former chief architect at Red Hat and field CTO at Solo.io) have emphasized the importance of application integration, such as in Posta's blog post "Application Safety and Correctness Cannot Be Offloaded to Istio or Any Service Mesh". Bilgin Ibryam has written about how application-integration architecture has evolved from SOA to cloud-native architecture in his InfoQ article on "Microservices in a Post-Kubernetes Era", in which he emphasizes the decentralization of the application integration with cloud-native architecture and how application integration is being built on top of the service mesh.

Development and integration in the CNCF landscape

The Cloud Native Computing Foundation (CNCF) is at the forefront of building microservices and cloud-native applications, and it aims to build sustainable ecosystems and foster a community around a constellation of high-quality projects that orchestrate containers as part of a microservices architecture. The CNCF hosts projects composed of open-source technologies and frameworks that can implement different aspects of microservice or cloud-native architecture. It is interesting to see where these application-integration technologies fit into their technology stack.

The CNCF's recommended path through the cloud-native landscape has an App Definition and Development section, but no category dedicated to application development or integration.Given the importance of application integration, however, we could see it enter the CNCF landscape in the future. Figure 5 includes Application Integration technologies under App Definition and Development.

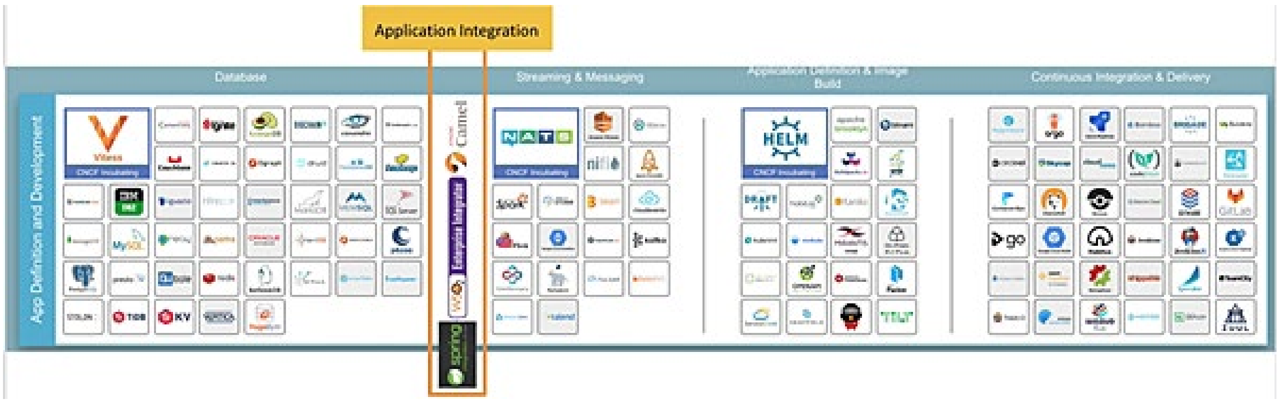


Figure 4 / Application integration in a future CNCF landscape

Technologies for application integration

Although there are quite a lot of monolithic application-integration technologies available, most are not suitable for cloud-native or microservices architectures. Only a handful of existing integration providers have implemented cloud-native variants of their products and tools.

There are dedicated integration frameworks that facilitate all of the common integration patterns in the application-integration space. Usually, most of these technologies are inherited from the conventional ESB-based integration but they have been modified and natively integrated into cloud-native architectures:

- Apache Camel/Camel-K is one of the popular open-source integration frameworks and the Camel-K project offers seamless support for the Kubernetes ecosystem on top of the Camel runtime.
- WSO2 Micro Integrator is a cloud-native variant of the open-source WSO2 Enterprise Integrator platform. Micro Integrator offers a lightweight integration runtime that natively works in the Kubernetes ecosystem.
- Although Spring Integration doesn't have a dedicated runtime to work on Kubernetes, it works well for building application integrations in cloud-native architecture.

Some of the application development frameworks also cater to the application-integration requirements:

- Spring Boot is not an integration framework per se but it has many substantial capabilities required for application integration.
- Vert.x is a toolkit for building reactive cloud-native applications, which can also be used for application integration.
- Micronaut is a modern, JVM-based, full-stack framework for building modular and easily testable microservice and serverless applications. There are quite a few integration abstractions built into the framework, and it avoids the complexities of conventional frameworks such as Spring.
- Programming languages such as Go, JavaScript/Node.js, etc., have certain application-integration features built in or available as libraries. There are emerging new languages such as Ballerina that offer integration abstractions as part of the language.
- Quarkus is a new Kubernetes-native Java stack that has been tailored for GraalVM and OpenJDK HotSpot, assembled from the best-of-breed Java libraries and standards. It's a combination of multiple application development libraries such as RESTeasy, Camel, Netty, etc.

Conclusion

With the segregation of monolithic applications into microservice and cloud-native applications, the requirement to connect these apps is becoming increasingly challenging. The services and applications are dispersed across the network and connected via disparate communication structures. Realizing any business use case requires the integration of the microservices, which needs to be done as part of the service implementation logic. As a result, cloud-native application integration is one of the most critical yet largely concealed requirements in the modern era of microservices and cloud-native architecture.

The service-mesh pattern overcome some of the challenges in integrating microservices but only offers the commodity features of inter-service communication, which are independent from the business logic of the service, and therefore any application-integration logic related to the business use case should still be implemented at each service level. Accordingly, it is important to select the most appropriate development technology for building integration-savvy services and minimize the development time required to weave together services. Several frameworks and technologies are emerging to fulfill these application-integration needs in the cloud-native landscape, which we need to evaluate against each specific use case.

TL;DR

- Integration of APIs, services, data, and systems has long been one of the most challenging yet most essential requirements in the context of enterprise software application development.

- We used to integrate all of these disparate applications in point-to-point style, which was later replaced by the ESB (Enterprise Service Bus) style, alongside the Service Oriented Architecture (SOA).

- As the popularity of microservices and “cloud native” architectures grows, the concept of a “service mesh” has emerged. The key idea with a service mesh is to keep all the business logic code as part of the service, while offloading the network communication logic to the inter-service communication infrastructure.

- Since a service mesh offers some of the capabilities that are part of ESBs, there is a misconception that this is a distributed ESB, which also takes care of application integration. That is not correct.

- A service mesh is only meant to be used as infrastructure for communicating between services, and developers should notbe building any business logic inside the service mesh. Other frameworks and libraries can be used to implement cloud native enterprise application integration patterns.

The Potential for Using a Service Mesh for Event-Driven Messaging

by **Kasun Indrasiri**, Director of Integration Architecture at WSO2

Service meshes are increasingly becoming popular as an essential technology and an architectural pattern on which to base microservices and cloud-native architecture. A service mesh is primarily a networking infrastructure component that allows you to offload the network communication logic from your microservices-based applications so that you can fully focus on the business logic of your service.

A service mesh is built around the concept of a proxy, which is colocated with the service as a sidecar. Although a service mesh is often advertised as a platform for any cloud-native application, popular implementations of service meshes (Istio/Envoy, Linkerd, etc.) currently only cater to the request/response style of synchronous communication between microservices. However, interservice communication takes place over a diverse set of patterns, such as request/response (HTTP, gRPC, GraphQL) and event-driven messaging (NATS, Kafka, AMQP) in most pragmatic microservices use cases. Since service-mesh implementations do not support event-driven communication, most of the commodity features that service meshes offer are only available for synchronous request/response service - event-driven microservices must support those features as part of the service code itself, which contradicts the very objective of service-mesh architecture.

It is critical that a service mesh supports event-driven communication. This article looks at the key aspects of supporting event-driven architecture in a service mesh and how existing service-mesh technologies are trying to address these concerns.

Implementing event-driven messaging

In a typical request/response synchronous messaging scenario, you will find a service (server) and a consumer (client) that invokes the service. The service-mesh data plane acts as the intermediary between the client and the service. In event-driven communication, the communication pattern is drastically different. An event producer asynchronously sends the events to an event broker, with no direct communication channel between the producer and consumer. The communication style can either be pub-sub (multiple consumers) or queue-based (single consumer), and depending on the style, the producer can send messages to either a topic or queue respectively.

The consumer decides to subscribe to a topic or a queue that resides in the event broker, which is fully decoupled from the producer. When there are new messages available for that topic or queue, the broker pushes those messages to the consumer.

There are a couple of ways to use service-mesh abstraction for event-driven messaging.

Protocol-proxy sidecar

The protocol-proxy pattern is built around the concept that all the event-driven communication channels should go through the service-mesh data plane (i.e., the sidecar proxy). To support event-driven messaging protocols such as NATS, Kafka, or AMQP, you need to build a protocol handler/filter specific to the communication protocol and add that to the sidecar proxy. Figure 1 shows the typical communication pattern for event-driven messaging with a service mesh.

As most event-driven communication protocols are implemented on top of TCP, the sidecar proxy can have protocol handlers/filters built on top of TCP to specifically handle the abstractions required to support each of the various messaging protocols.

The producer microservice (Microservice A) has to send messages to the sidecar via the

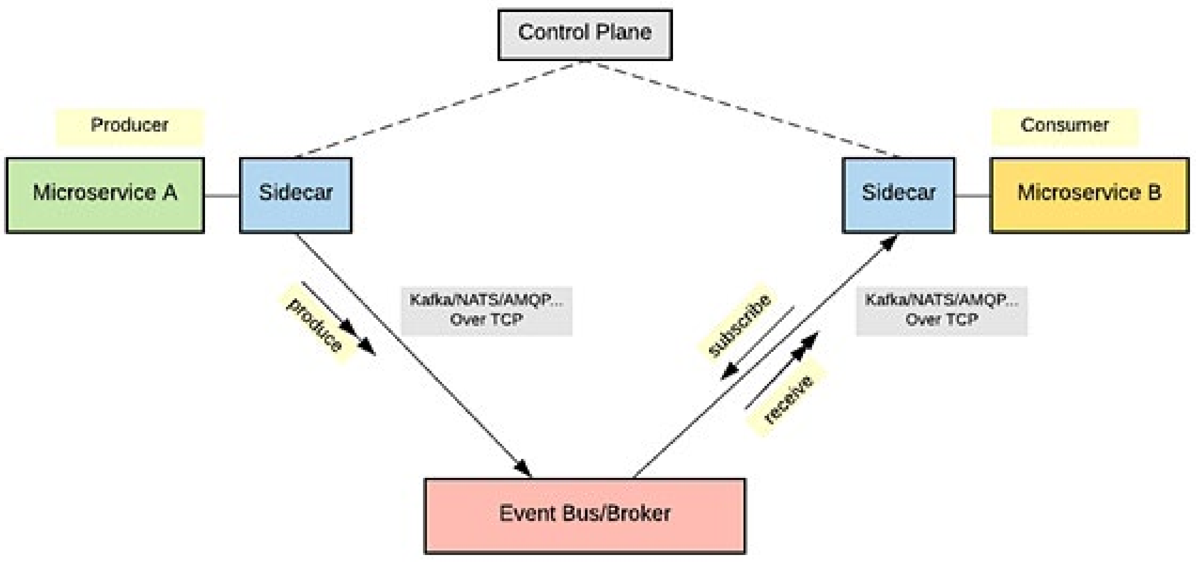


Figure 1 / Event-driven messaging with a service mesh

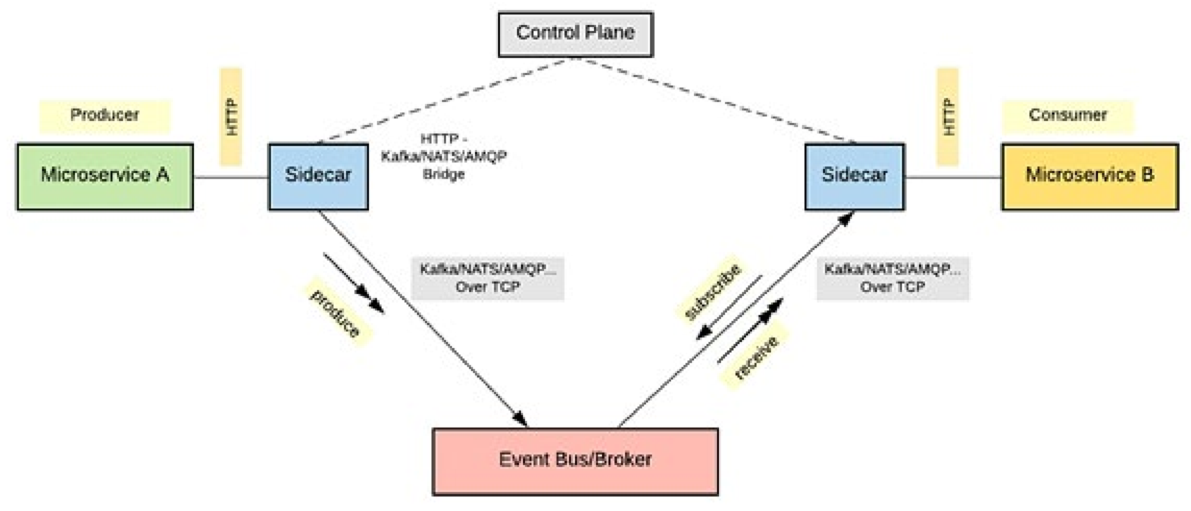


Figure 2 / The HTTP bridge allows the service to communicate with the event broker via HTTP

underlying messaging protocol (Kafka, NATS, AMQP, etc.), using the most simple code for the producer client while the sidecar handles most of the complexities related to the protocol. Similarly, the logic of the consumer service (Microservice B) is also quite simple while the complexity resides at the sidecar. The abstractions provided from the service mesh may change from protocol to protocol.

The Envoy team is currently working on implementing Kafka support for the Envoy proxy based on the above pattern. It is still work in progress, but you can track the progress at GitHub.

HTTP-bridge sidecar

Rather than using a proxy for the event-driven messaging protocol, we can build an HTTP bridge that can translate messages to/from the required messaging protocol. One of the key motivations for building this bridging pattern is that most of the event brokers offer REST APIs (e.g., the Kafka REST API) to consume and produce messages. As shown in figure 2, the existing microservices can transparently consume the underlying event broker's messaging system simply by controlling the sidecar that bridges the two protocols. The sidecar proxy is primarily responsible for receiving HTTP requests and translating them into Kafka/NATS/AMQP/etc. messages and vice versa.

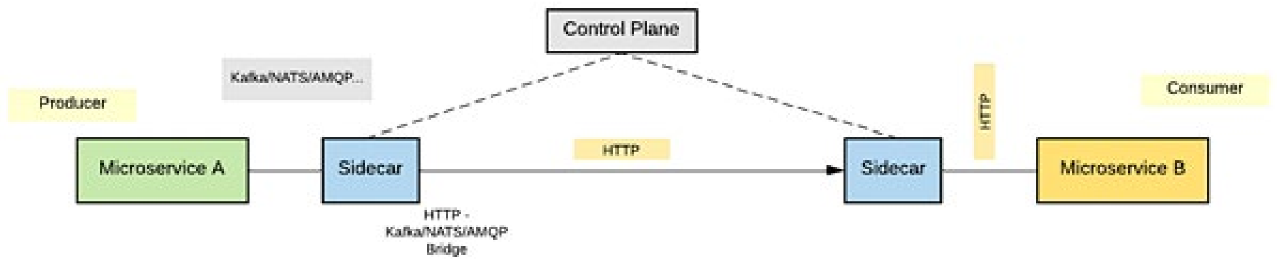


Figure 3 / The HTTP Bridge allows services based on event-driven messaging protocols to consume HTTP services

Similarly, you can use the HTTP bridge to allow microservices based on Kafka/NATS/AMQP to communicate directly with HTTP (or other request/response messaging protocols) microservices as in figure 3. In this case, the sidecar receives Kafka/NATS/AMQP requests, forwards them as HTTP, and translates HTTP responses back to Kafka/NATS/AMQP. There are some ongoing efforts to add support for this pattern on Envoy and NATS (e.g., AMQP/HTTP Bridge and a NATS/HTTP bridge, both for Envoy).

Although the HTTP-bridge pattern works for certain use cases, it is not strong enough to serve as the standard way of handling event-driven messaging in the service-mesh architecture because bridging event-driven messaging protocol with a request/response messaging protocol always has limits. It is more or less a workaround that might work for certain use cases.

Key capabilities of an event-driven service mesh

The capabilities of a conventional service mesh based on request/response-style messaging are somewhat different from the capabilities of a service mesh that supports messaging paradigms. Here are some of the unique capabilities a service mesh that supports event-driven messaging will offer:

- **Consumer and producer abstractions** - With most messaging systems, such as Kafka, the broker itself is quite abstract and simple (a dumb pipe in microservices context) and your services are smart endpoints (most of the smarts live in the producer or consumer code). This means that the producers or consumers must have a lot of messaging-protocol code alongside the business logic. With the introduction of a service mesh, you can offload such commodity features (e.g., partition rebalancing in Kafka) related to the messaging protocol to the sidecar and fully focus on the business logic in your microservice code.
- **Message-delivery semantics** - There are many message-delivery semantics such as “at most once”, “at least once”, “exactly once”, etc. Depending on what the underlying messaging system supports, you can offload those tasks to the service mesh (this is analogous to supporting circuit breakers, timeouts, etc. in the request/response paradigm).
- **Subscription semantics** - You can also use the service-mesh layer to handle the subscription semantics, such as durable subscription of the consumer-side logic.
- **Throttling** - You can control and govern the message consumption limits (rate limiting) based on various parameters

such as the number of messages, message size, etc.

- **Service discovery** (broker, topics, and queue discovery) - The service-mesh sidecar allows you to discover the broker location, topic, or queue name during message production and consumption. This involves handling different topic hierarchies and wildcards.
- **Message validation** - Validating messages that are used for event-driven messaging is becoming important because most of the messaging protocols such as Kafka, NATS, etc. are protocol agnostic. Hence message validation is a part of consumer or producer implementation. The service mesh can provide this abstraction so that a consumer or producer can offload the message validation. For example, if you use Kafka along with Avro for schema validation, you can use the sidecar to do the validation (i.e., fetch the schema from an external scheme registry such as Confluent and validate the message against that scheme). You can also used this to check messages for malicious content.
- **Message compression** - Certain event-based messaging protocols, such as Kafka, allow the data to be compressed by the producer, written in the compressed format to the server, and decompressed by the consumer. You can easily implement such capabilities at the sidecar-proxy level and control them at the service-mesh control plane.
- **Security** - You can secure the communication between the broker and consumers/producers by enabling TLS at the service-mesh sidecar level so that your producer and consumer implementations do not need to worry about secured communication and can communicate with the sidecar in plain text.
- **Observability** - As all communications take place over the service-mesh data plane, you can deploy metrics, tracing, and logging out of the box for all event-driven messaging systems.

TL;DR

- The current popular implementations of service meshes (Istio, Linkerd, Consul Connect, etc.) only cater to the request-response style synchronous communication between microservices
- For the advancement and adoption of service meshes, we believe that it is critical that they support event-driven or messaging-based communication
- There are two main architectural patterns for implementing messaging support within a service mesh; the protocol proxy sidecar, which is a proxy for all the inbound and outbound events from the consumer and producer; and the HTTP bridge sidecar which translates or transforms event-driven communication protocol to HTTP or similar protocol
- Regardless of the bridging pattern that is used, the sidecar can facilitate the implementation (and correction abstraction) of cross-functional features such as observability, throttling, tracing etc.

Towards a Unified, Standard API for Consolidating Service Meshes

by **Christian Posta**, Global Field CTO at Solo.io

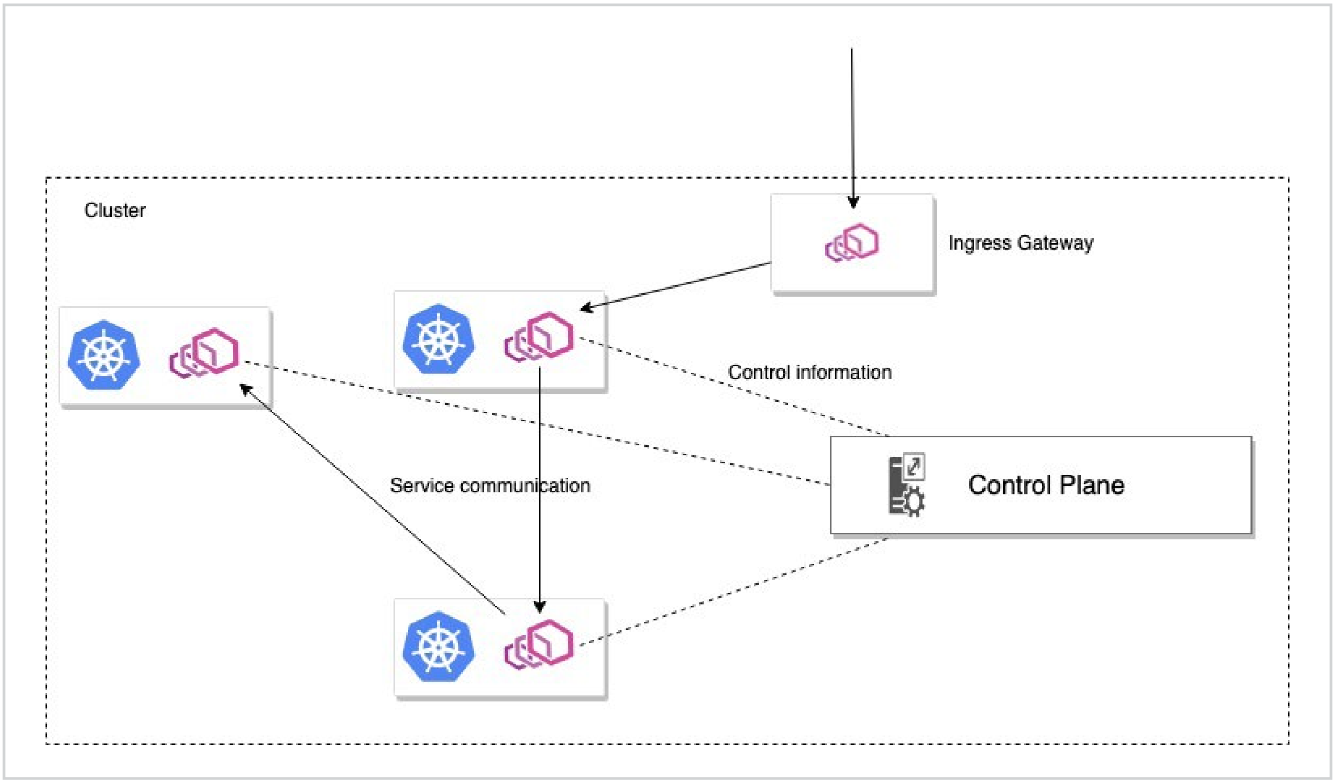
Organizations are trying to modernize their software stack with cloud-native technologies and infrastructure that can greatly improve their ability to quickly and safely deliver software. As our applications become more decentralized, and our infrastructure more "cloudy", maintaining availability, correctness and safety when doing dozens (or more) releases per day becomes very difficult. Kubernetes and containers have helped us standardize on a packaging and deployment model based on "immutable delivery", but once those applications are deployed, they need to communicate with each other over the network. This situation presents numerous challenges for developers and operators. Service mesh gives us a foundational framework from which to build and operate services that go a long way to solving some of the challenges of application networking and safe, progressive, delivery.

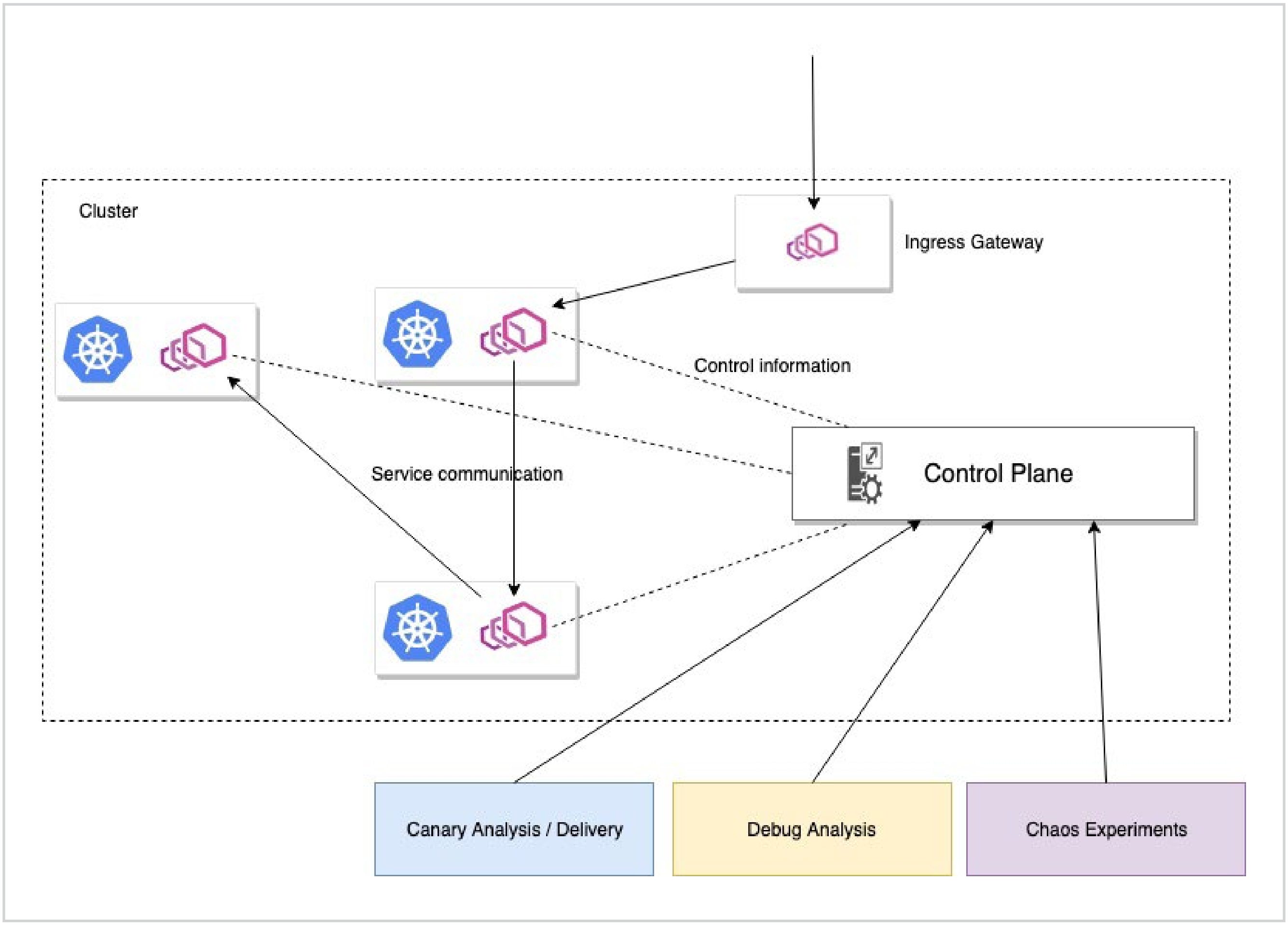
Services operators and application developers need to solve for service-to-service communication challenges like service discovery, client-side load balancing, configuring transport security, identity provisioning, traffic routing/shaping, communication resilience with circuit breaking, network request observability, and many others. Across languages and frameworks, this can be a very error-prone and tedious exercise. If you build it for one language, how do you end up supporting other languages? If you've built it for one cloud-platform, is it extendable to others? Service mesh solves these problems by injecting a service proxy next to each workload, through which that workload interacts with any other services over the network, that is able to transparently and consistently solve these issues.

A service proxy is typically dedicated to a particular workload instance and they are not shared with other instances within the cluster. This collection of service proxies is known as the “data plane”. All requests to services in a service mesh flow through these proxies and are intercepted through the proxy and enriched with powerful behavior like retries, timeouts, circuit breaking, service discovery, load balancing and much more. Envoy Proxy is an open-source service proxy that is a popular choice for implementing a service mesh. Since all requests to and from a workload instance traverse its service proxy, the proxies can capture metrics about the request and affect a change to the communication over the network. The operator of a service mesh interacts with the control plane which is a set of components that lives outside of the request path that supports the data plane.

Service mesh follows the 80/20 rule. Service mesh can get you 80 percent of the way to solving these application-networking problems in a cloud, language, and framework-agnostic way. The last 20 percent involves writing your own “glue code” to drive and operate the service mesh to “orchestrate it” to deliver a valuable higher-order service. The service mesh exposes a set of powerful individual capabilities, but for your organization to take full advantage of it, you'll need to integrate and combine them for your organization. Examples of higher-order services enabled by service mesh include

- smart auto-scaling/ back-pressure
- progressive delivery





- traffic-pattern detection
- chaos injection and analysis
- request provenance and chain-of-custody
- zero-trust networking practices
- A/B experiments (which is a more mature version of progressive delivery).

A service mesh has the ability to get you 80% of the way there, but to really achieve these capabilities, you need to extend the mesh with 20% your code.

Building on top of the service mesh to reduce risk

Let's look at a single example that combines multiple features of a service mesh to help us get to our end goal of reducing risk when making changes to our system.

Progressive delivery is a term coined by Sam Guckenheimer product owner of Azure Devops and Redmock analyst James Governor in 2018 to describe the approach to progressively exposing new application functionality in such a way that reduces the blast radius of bad behaviors. Using a combination of traffic control, feature flagging, request routing, and metrics collection, one could build a powerful CI/CD pipeline following progressive delivery techniques that can reduce the risk of delivering new code and code changes.

Without changing any application/service code, we could leverage the service mesh as a framework to build a progressive-delivery capability orchestrating capabilities like traffic control, request routing, and metric collection. If we want to make a change to a recommendation service for example, we could deploy a new version and finely control which users get routed to this new deployment. Following a progressive delivery approach, we could deploy v2 of our `recommendation` service into production and only expose it to internal employees. We would then observe the metrics and logs for the new service and set certain thresholds for what it means to continue our delivery. If we find that this new version behaves undesirably, we can roll it back (or reduce traffic) thus limiting its blast radius. For example, if we notice more failures or higher latency in request processing, we can halt the progressive delivery of this canary and maybe even roll it back. If things look good, we can open up the traffic to more users (say, 1% of live users). We can follow this approach, slowly unveiling the new service to other groups of users, watching its indicators through metrics and logs, and either continue rollout or roll back.

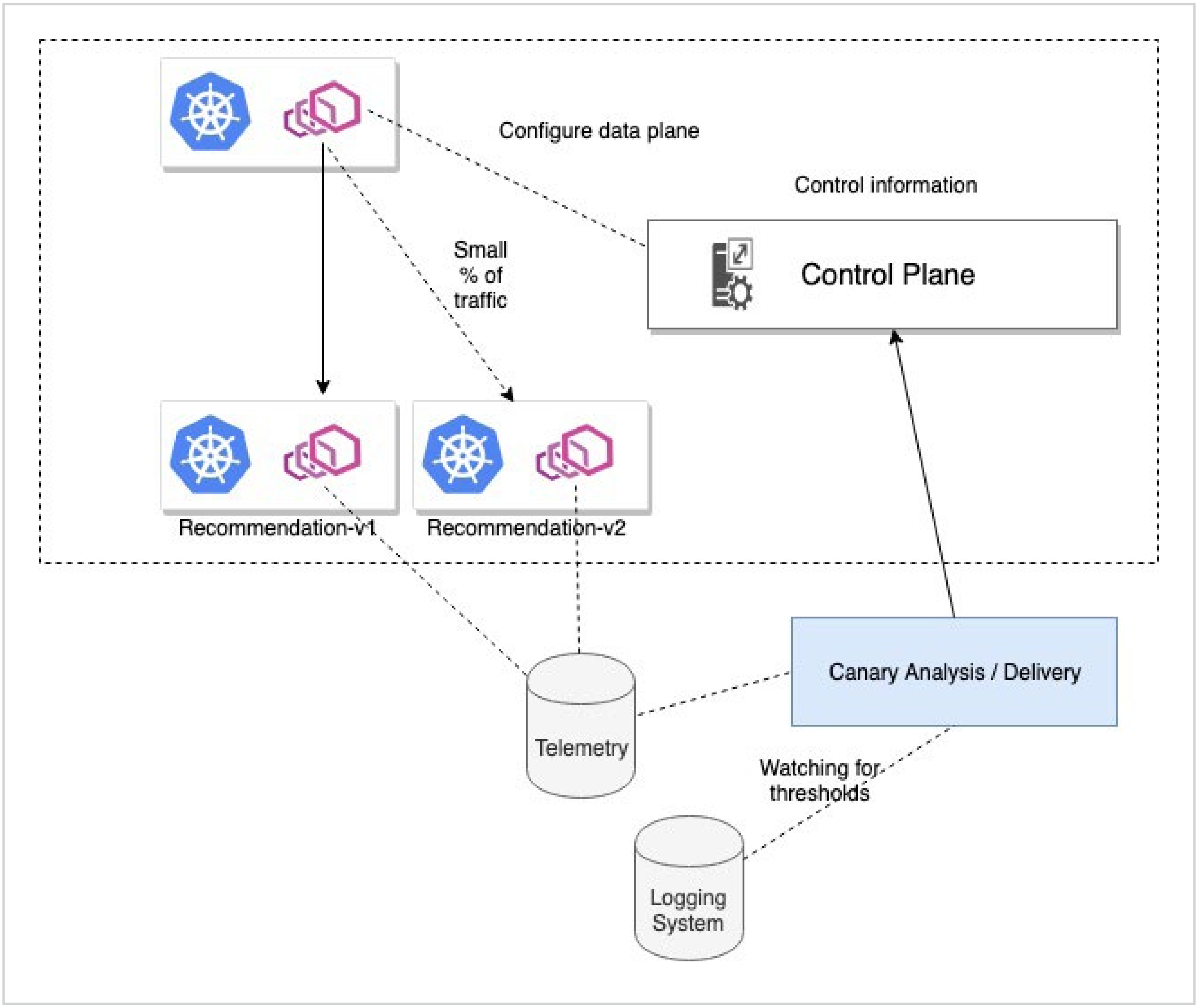
A service mesh is implemented with a control-plane/data-plane architecture where the data plane is the individual service proxies through which request metrics are collected and request routing control can be enforced. The control plane is the set of components, outside of the request path, that operators and users can interact with to collect metrics, establish policy, and affect configuration changes to the data plane.

A control plane has an API surface or some opinionated approach to driving configuration for the mesh. In most open-source service mesh implementations, we typically see a declarative configuration/API surface with a "declarative intent" with the service mesh responsible for "reconciling that intent". For example, with Istio -- an open-source service mesh -- configuration is declared as YAML resources that describe the intent of service-network behavior and the Istio control plane synthesizes this intent into configuration for the service proxies. For a `recommendation` service, we may want to control traffic to a canary release by splitting the percentage of traffic that gets routed to each version. In Istio we can use a `VirtualService` to do that:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation-vs
spec:
  hosts:
  - recommendation.prod.svc.cluster.local
  http:
  - route:
    - destination:
        host: recommendation.prod.svc.cluster.local
        subset: v1
      weight: 90
    - destination:
        host: recommendation.prod.svc.cluster.local
        subset: v2
      weight: 10
```

Istio also collects metrics from the service proxies using statsD, Prometheus, and/or Jaeger for distributed tracing. These metrics can then be syndicated to your backend analytics engines. They can also be used to make important decisions about whether to continue with the delivery/release process. Using these signals, we may instruct the service mesh to route 20% of live traffic to the canary if things are going well, or we may scope back the amount of traffic (maybe even all of it) if our signals tell us this new version is behaving adversely.

Another implementation of a service mesh, AWS App Mesh, uses a similar declarative API for describing traffic routing rules. We create JSON resources and feed that to the App Mesh control plane which then synthesizes the intent into Envoy configuration (Envoy is used as the App Mesh data plane as well):



AWS App Mesh can deliver metrics and request telemetry to CloudWatch, X-Ray, et. al. from which your progressive-deliver engine can evaluate whether to continue opening the traffic flow to your new versions.

Other service mesh implementations like Linkerd and Consul Connect are working on their traffic-control APIs but will likely follow a similar pattern.

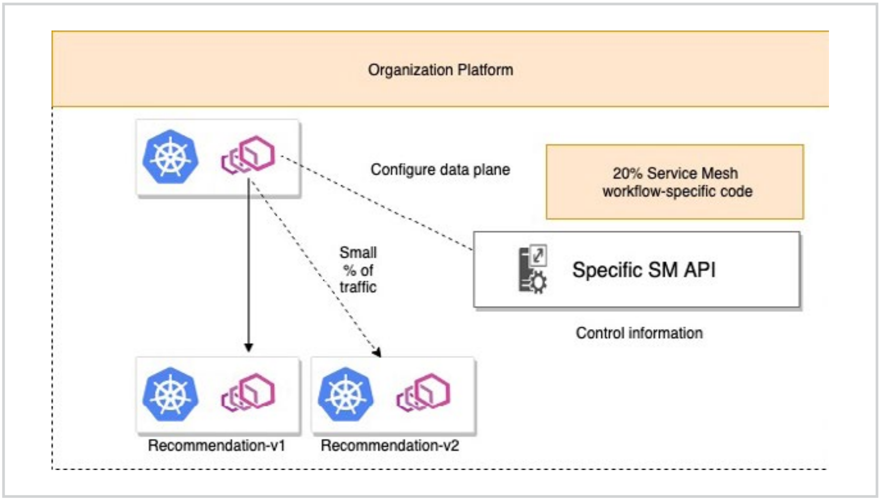
As we consider our progressive delivery implementation on top of a service mesh, we can start to see what parts are 80% and what parts are 20%. Although a service mesh can both collect metrics about what is happening at the service/request level and it can help shift/restrict/route traffic to services, it cannot know the process for service promotion or rollback including thresholds/high watermarks that should be used as triggers to open the service up to more users vs rollback to fewer. That control logic, which can be very opinionated and organization dependent, will be the 20% of glue code an organization will need to write on top of the service mesh. To do this, the organization will need to use the specific service-mesh control plane APIs.

```
{
    "routeName": "recommendation-route",
    "spec": {
        "httpRoute": {
            "action": {
                "weightedTargets": [
                    {
                        "virtualNode": "recommendation-v1-vn",
                        "weight": 9
                    },
                    {
                        "virtualNode": "recommendation-v2-vn",
                        "weight": 1
                    }
                ]
            },
            "match": {
                "prefix": "/"
            }
        }
    },
    "virtualRouterName": "recommnedation-vr"
}
```

The need for an open, service-mesh API

As an organization begins to build out its progressive delivery capabilities, or any capabilities on top of a service mesh as described previously, it will necessarily build its glue code around the specific service-mesh API it has decided to adopt. At the moment, for all service mesh implementations, these APIs are changing and evolving. Smart organizations have begun building out their own specific configuration APIs that more cleanly express their intents, fits in better with the value-added services they build on top of service mesh, and abstracts them from constant API churn. They then use a translation layer to convert their configurations into the specific service-mesh implementation APIs. By abstracting away the specific service-mesh APIs, an organization can limit the impact of any one specific service-mesh API implementation details especially when upgrading and finding non-backward compatible changes.

Another advantage to abstracting away service-mesh-specific APIs is the ability to swap out service mesh technologies at a later point without disrupting the valuable 20% code and functionality that's been written on top of a particular service mesh. In fact, as service mesh implementations



continue to evolve, we'll see those that are very focused or specialized for a certain area of mesh capabilities (let's say, security or observability), while others find their strengths in other areas (let's say traffic control). Having a separate, mesh-agnostic, API allows an organization to federate and plug/play certain mesh capabilities.

Lastly, as organizations attempt to unify their processes/deployments on premises with those they run in a public cloud, a single API for abstracting away the network becomes invaluable. The tooling you may have written to extend Istio for on-premises workloads can still be used when deploying into AWS. For example, AWS's App Mesh is an AWS-native service mesh and if you deploy to AWS, it makes sense to leverage their native service mesh. With a single unified API for your service mesh and capabilities built upon it, you don't have to recreate the wheel when adopting new platforms and can safely build on top of it.

On one hand, leading organizations have been cobbling together this "informal, mesh-agnostic API" for the reasons listed above, on the other hand, it's a bit of a waste for every organization to do this. What if we could collaborate in an open-source project to provide a stable API on service mesh implementations that give us the flexibility to adopt the solutions we need/want, plug and play where needed, and also confidently own the 20% integration glue code on top of our mesh infrastructure?

Some approaches to building an open, service mesh API

SuperGloo is an open-source project originally started to build an open, stable, mesh-agnostic API. With a simplified service-mesh API that can be used to abstract any service mesh, we can achieve the following goals:

- Create a consistent, simplified experience for installing any service mesh
- Build extensions to a service mesh (using 20% glue code) and hedge any upheaval in the service-mesh landscape (adoption choices, version changes, best-of-breed, etc)
- Discover service mesh implementations and resources and manage them
- Manage multiple implementations of service mesh installations under a single pane of glass and API
- Manage multiple clusters of one or more implementations of service mesh

At the most recent KubeCon (KubeCon EU 2019), the creator of SuperGloo, Solo.io, and other partners Microsoft, HashiCorp, and Buoyant announced the Service Mesh Interface (SMI) spec to play the role of this mesh abstraction API for things like traffic routing, metrics collection and policy enforcement. SuperGloo plays the role of both a reference implementation for SMI with the ability to convert SMI objects to any service mesh, but also as a way to manage service mesh installations, group them, and network them to provide federation and allow organizations to build upon and extend them. Let's take a look at SuperGloo and SMI.

SuperGloo has APIs for installing and discovering the capabilities of a service mesh with the `Install` and `Mesh` configuration objects, respectively. For example, the installation of Linkerd using SuperGloo's `Install` API looks like this:

```
- apiVersion: supergloo.solo.io/v1
  kind: Install
  metadata:
    creationTimestamp: "2019-05-01T14:12:58Z"
    generation: 1
    name: linkerd
    namespace: supergloo-system
    resourceVersion: "5571565"
    selfLink: /apis/supergloo.solo.io/v1/namespaces/supergloo-system/installs/linkerd
    uid: 389b5e09-6c1b-11e9-92c9-42010a8000c0
  spec:
    installationNamespace: linkerd
    mesh:
      linkerd:
        enableAutoInject: true
        enableMtls: true
        version: stable-2.3.0
```

The mesh features itself are surfaced in the `Mesh` config:

```
- apiVersion: supergloo.solo.io/v1
  kind: Mesh

  metadata:
    creationTimestamp: "2019-05-01T14:13:09Z"
    generation: 1
    labels:
      created_by: mesh-discovery
      discovered_by: linkerd-mesh-discovery
    name: linkerd
    namespace: supergloo-system
    resourceVersion: "5571484"
    selfLink: /apis/supergloo.solo.io/v1/namespaces/supergloo-system/meshes/linkerd
    uid: 3f70f63e-6c1b-11e9-92c9-42010a8000c0
  spec:
    discoveryMetadata:
      enableAutoInject: true
      injectedNamespaceLabel: linkerd.io/inject
      installationNamespace: linkerd
      meshVersion: stable-2.3.0
      mtlsConfig:
        mtlsEnabled: true
    linkerd:
      installationNamespace: linkerd
      version: 2.3.0
    mtlsConfig:
      mtlsEnabled: true
```

The equivalent usage for both Istio and AWS App Mesh would be similar.

Things become interesting when we define traffic rules for a mesh. For example, using the SMI spec, we can assign traffic rules to a mesh with the `TrafficSplit` API. To specify a weighted-routing rule to shift traffic between v1 and v2 of a `reviews` service, we could do the following:

```
apiVersion: split.smi-spec.
io/v1alpha1
kind: TrafficSplit
metadata:
  name: example-routing
spec:
application.
  service: reviews.default
  backends:
  - service: reviews-v1.
default
    weight: 1
  - service: reviews-v3.
default
    weight: 100m
```

Note that SMI routes to services using their FQDN. These services are defined by using their associated labels and grouping metadata which can select specific versions of a deployment.

SMI also supports APIs around collecting telemetry from a service mesh with TrafficMetrics, policy enforcement using the TrafficTarget resources. These objects apply to any service-mesh implementation. To see the full API, check the specification document. At the moment, SuperGloo supports Istio, Linkerd, and AWS App Mesh via SMI with support for more meshes coming soon.

The unification of the APIs around specific service mesh implementations allows us to use SuperGloo to manage any service mesh implementation consistently including in more complicated scenarios where you may have multiple meshes and multiple clusters (even if just multiple clusters of a single mesh). Any of the 20% code we write can stay mesh agnostic and provide valuable extensions on top of a mesh. A great example of this is building progressive-deliver capabilities. In fact, the Flagger project from Weaveworks did just this and can leverage SuperGloo and SMI to stay mesh agnostic.

Final thoughts

As organizations continue to adopt service-mesh technology and build upon it, they will be weary to tie their valuable 20% code to anyone implementation until a clear winner or standard emerges. Even with a single implementation, things move fast, APIs change, and this turbulence can wreak havoc at an organization that cannot wait to implement the kinds of solutions that a service mesh brings. With SuperGloo, you can hedge against this volatility, own your 20% code, and even tackle the more advanced use cases of federating multiple meshes or multiple clusters.

TL;DR

- Service mesh exposes knobs and levers for controlling service-to-service (east-west) traffic
- The “levers” exposed by a service mesh can be used by specialized workflows to reduce the risk of doing releases and improve the mean time to recovery (MTTR)
- To get the real value out of service mesh, extensions need to be written on top of the mesh to drive its capabilities
- At the moment, service mesh implementations vary in regard to API and technology, and this shows no signs of slowing down
- Building on top of volatile APIs can be hazardous
- There is a need for a simplified, workflow-friendly service-mesh API to shield organization platform code from specific service-mesh implementation details