Using Service Meshes in AWS

Cloud native networking use cases and good practices

November 2020

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Introduction 1

From Monoliths to Microservices 1

Service Mesh Concept 2

Use Cases 5

Use Case 1: Strengthening Security Posture 6

Use Case 2: Automating Service Discovery 6

Use Case 3: Enabling Progressive Deployments 7

Use Case 4: Monitoring and Troubleshooting 8

Good Practices 9

Selection Criteria 9

Operational Considerations 11

Interoperability 15

Basics: The TCP/IP Stack 15

On Specifications and De-Facto Standards 16

Conclusion 18

Contributors 18

Further Reading 18

Document Revisions 19

Abstract

Organizations are in the process of adopting cloud native applications or migrating existing on-premises workloads into the cloud. Many are looking into service meshes, trying to assess if and when to adopt this relatively new technology. In this whitepaper we review the concept of a service mesh, discuss features and use cases, and provide good practices about introducing and managing service meshes.

Introduction

From Monoliths to Microservices

During your journey, from on-premises monoliths to containerized microservices in the cloud you will discover the term *service mesh*, and if/when you should adopt a service mesh. This decision primarily depends on where you are in your journey. Did you just recently start moving into the cloud? Have been using Docker and related technologies for a year or two in production?

To better visualize service mesh adoption, we will look at a concrete exemplary path in the context of a cloud native adoption process shown in Figure 1.

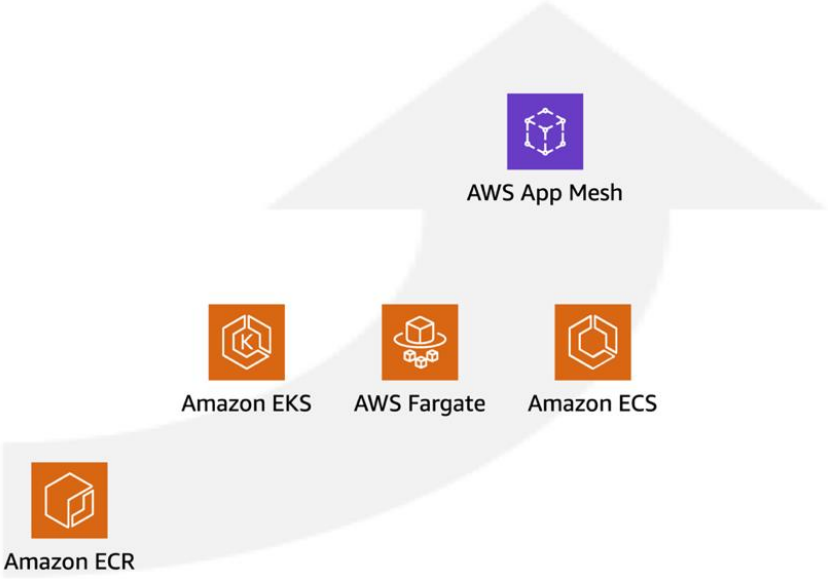

Figure 1—Exemplary path of a cloud native adoption process.

Early in the process an organization introduces a process to generate immutable artifacts including container images, stored in a container registry such as Amazon Elastic Container Registry (Amazon ECR). When developers are enabled to produce artifacts, the organization also needs to decide on a container orchestrator and compute engine. This whitepaper doesn't provide guidance for selecting a orchestrator. The following examples used throughout this paper assume you selected a combination of Amazon Elastic Container Service (Amazon ECS) on AWS Fargate, or Amazon Elastic Kubernetes Service (Amazon EKS) on Amazon Elastic Compute Cloud (Amazon EC2) with managed node groups, depending on the workload. Rather than re-inventing the wheel for a number of service communication related challenges such as traffic control,

security, and observability, you decide to utilize a service mesh such as, a fully managed offering such as AWS App Mesh.

The above figure and example path are only one potential path out of multiple available options. For example, one piece of feedback we often hear from customers around hybrid (on-prem/cloud) setups is the ask to abstract that details a way to make the decision of compute something that's not necessarily visible to developers.

> **Note:** Service mesh is still in the early adoption stages. Don't feel pressured to adopt hastily and monitor where the market and community are heading. For example, The New Stack reported in February 2020 some 16% production usage with the majority (around 46%) are either currently evaluating it or plan to evaluate.

Before we go further down the selection route let's first step back and review what makes a service mesh and what kind of features you would typically expect to find when adopting a service mesh.

Service Mesh Concept

When you face an increase of (containerized) microservices or are in the process of migrating a monolith into this direction, one aspect suddenly plays a central role: network communication between the parts (microservices). In the case of a monolith, the communication between different modules or components is, trivially, in-process. In case of a microservices setup, usually the processing of a request spans a handful of services and that involves the network.

Service meshes strive to address this issue. In this context, we refer to the idea of externalizing certain network traffic related functionality. So, rather than implementing, for example, encryption on the wire between two services directly in the application code, one would delegate this security feature to the mesh. The same is true for many other service network communication tasks such as troubleshooting, fine-grained traffic control, progressive deployment, and etc.

But how does it work? Figure 2, shows a conceptual architecture of a service mesh.

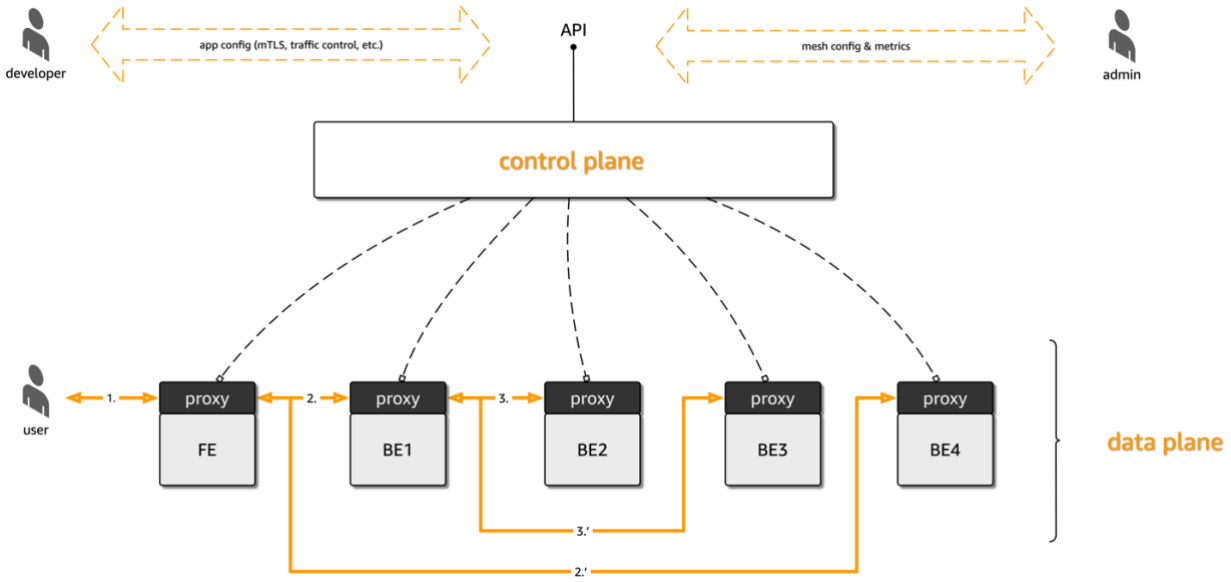

Figure 2—The service mesh concept: control and data plane.

As shown in Figure 2, a service mesh is comprised of a control plane and a data plane. In current implementations of service meshes, the data plane is made up of proxies sitting next to your applications or services, intercepting any network traffic that is under the management of the proxies. The configuration of the proxies—pass this packet, report metrics on flow X, encrypt messages between service A and B—comes from the mesh control plane. The control plane usually has an API that is accessible to infrastructure admins, to manage the mesh and derive metrics to application developers, enabling them to configure their services and get additional metrics.

In the data plane of Figure 2 there are a number of boxes in the lower half. These represent services (labelled `FE` for frontend, and `BEx` for backend X). While from a physical perspective, all network packets flow via the proxies, this is transparent for the service, hence should not impact the application developer. From a logical point of view, the request path in the example data plane is shown in Figure 3.

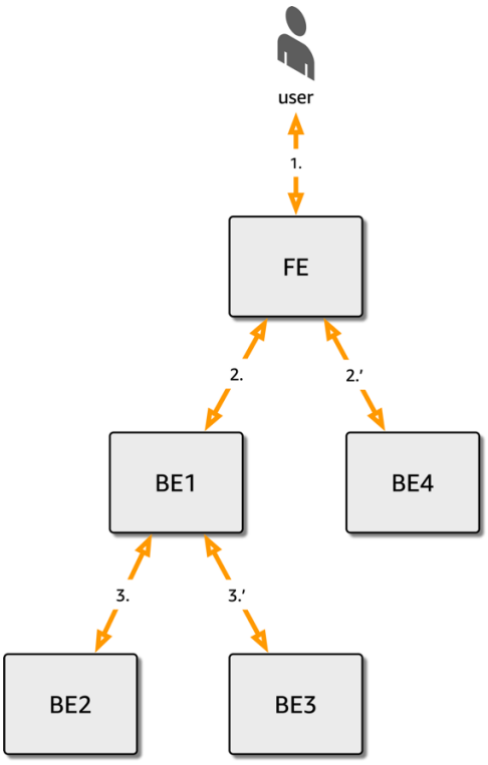

Figure 3—The logical request path of our example setup.

Figure 3 shows a frontend service `FE`, such as powering a web page or a mobile app user interface (UI), or receiving traffic from a user. In order to carry out its job, the frontend needs to call two other services (`BE1` and `BE4` in our case) which in turn may depend on other backend services (`BE1` depends on `BE2` and `BE3` here).

For the application developers, this logical call graph is all that matters. The service mesh enables them to control the physical flow. For example, `BE3` could be an external system mandating TLS.

Or, as a developer, you might be asking for how many HTTP response codes `500` happened when calling `BE4` over the past 30 minutes. These metrics and many other related ones are something service meshes can provide you (in either developer or ops role) automatically, that is, with little to no instrumentation efforts.

If you're considering using a service mesh, a great way to start is to approach the topic from a use case point of view. So, let's review some popular use cases, next.

Use Cases

In the following, we tackle the question as to when to use a service mesh. There are certain scenarios where service meshes represent a low-hanging fruit and/or it would be less beneficial to re-invent the wheel. Typical high-level indicators are:

- You're already further along in your cloud native journey (service meshes should not be the first project you're getting into).
- Polyglot and/or microservices setup: if you have a monolith without a path to a microservices setup, you don't need a service mesh. With the numbers of microservices you deploy as well as with the diversity in the programming languages and data stores different teams might employ, introducing a mesh becomes more and more attractive.

For more details on use cases, adoption strategies and requirements, we recommend reading Lee Calcote's book: The Enterprise Path to Service Mesh Architectures.

If service meshes do not seem the right fit, what's the alternative?

The following outlines potential options if a service mesh isn't the correct fit for your organization:

- An in-process solution, a library such as Netflix Hystrix or Twitter Finagle might be a more appropriate and effective way to address your service communication pain points.
- If you have a small number of (micro)services all written by the same team and/or in the same language there's little need to abstract the details away, hence a service mesh might be an overkill.
- You might want to avoid the added complexity of sidecar proxies (costs, debugging, etc.) making introducing a service mesh too early in the process might be counterproductive.
- Immaturity of the technology in general, lack of hands-on experience, or size of the community behind solutions.

The following section, dives deep into a few common use cases:

Use Case 1: Strengthening Security Posture

When running an application in the financial domain, in order to be PCI DSS compliant, certain rules need to be followed. For example, how to handle PII data or who has access to what kind of data. One common requirement is that the communication between endpoints is secured in order to avoid eavesdropping and/or person-in-the-middle attacks. This encryption on the wire is typically achieved using Transport Layer Security (TLS) protocol (as per RFC 8446 or comparable).

In addition, services can and should have unique identities for the duration of its existence. Based on this kind of micro-identity, service meshes can enable a range of authentication and authorization scenarios. For example, one can deploy Mutual TLS (mTLS), putting both sides of the communication into a position to cryptographically verify the identity of their peer. To acheive this, a number of options are available, from using open source components such as the `cert-manager` project to managed offerings like AWS Certificate Manager.

We also see combinations of open source projects customers write that work in tandem with a managed service such as AWS Key Management Service, and Skyscanner's KMS Issuer.

Finally, emerging specifications such as Cloud Native Computing Foundation (CNCF) Security Protection Identity Framework for Everyone (SPIFFE) can help building portable and interoperable systems while relying on vendor-neutral technologies.

Use Case 2: Automating Service Discovery

Your services should be able to automatically find each other, regardless of the type or size of the workload, containers, AWS Lambda functions, or a monolith running on a virtual machine in your data center.

Services exposing an HTTP(S) API can leverage DNS to implement service discovery. For example, using an DNS-based service discovery method, you can encode details of the service or endpoint—for example, its service ports—via the DNS `SRV` record while the fully qualified domain name (FQDN) of the API URL gets resolved to the IP addresses of the respective service. Some runtime environments such as Kubernetes have built-in mechanisms, whereas others outsource this functionality or provide a plug-able setup.

In the latter case an external system, typically a distributed datastore like ZooKeeper, `etcd`, or Consul stores discovery-relevant records in the registration process as well as provides look-up mechanisms for consumption by clients.

Important aspects in the context of service discovery are:

- Comprehensive support for both **HTTP-based** and **non-HTTP** services (for example, RDS), supported by AWS Cloud Map or HashiCorp Consul.
- Enabling **discovery across** compute and **environments**, for example the case of a hybrid setup where part of the application resides on-premises and part of it in the cloud.

Additionally, for applications operating across geos, federation aspects come into play, for example, Solo.io has demonstrated with Gloo Federation a possible direction, the multi-cluster communication capabilities of Linkerd are notable in this context, as well as HashiCorp's work around Consul in a multi-environment setup.

Use Case 3: Enabling Progressive Deployments

We can consider concepts such as **continuous delivery** —the capability of supplying artifacts such as binaries, container images, secrets, or runtime configurations in a safely automated, hands-off approach—as well as **continuous deployment** — pulling said artifacts and launching them in a runtime environment such as Kubernetes, Amazon ECS, or HashiCorp Nomad) —are already somewhat established. At the time of publication, certain good practices such as immutable artifacts and multi-staged caching have been shared and adopted widely. We refer to this collection of good practices and tooling collectively as **continuous delivery and deployment**, encompassing both delivery and deployment phases.

Progressive delivery and deployment are the latest trend, taking CD to the next level. It enables you to have fine-grained control over what client receives which version of your app. For example, you could say: "expose v3.42 to premium users in the EMEA geo, but only after business hours".

How does this work?

A service mesh provides the foundation to make various deployment capabilities possible and safe. It provides metrics and assess the state of the services, and implements traffic routing or flow primitives such as "direct 10% of the traffic to this endpoint". This enables developers to carry out patterns such as canary deploys or A/B testing, verifying how effective a certain change is.

> For more information about different CD patterns, see our Builder's Library article Automating safe, hands-off deployments and Christian Posta's post on Blue-green Deployments, A/B Testing, and Canary Releases.

Open source tooling to implement progressive deployments exists. For example, you can use Flagger with AWS App Mesh and a range of other meshes to make progressive delivery a reality in your organization.

A related theme is *production is the new test/staging/QA*. A number of practitioners have argued for this new way about using automation to accelerate the deployment cycle in a confident way, enabling you to roll back and assess impact quickly and in a scalable manner. For more information, see the following articles: Don't test in production; Why and How Testing in Production; and Testing in Production.

Somewhat overlapping with the previous use cases, there's a number of testing and troubleshooting techniques that service meshes enable or at least support. For example, traffic shadowing can help reducing deployment risks. Also, the traffic routing or shaping feature most meshes support can be used for regression tests.

Use Case 4: Monitoring and Troubleshooting

Once your microservices are deployed and running in production, you can start operations. Consider the case of being on-call for an application comprising of 489 microservices. It's 1am and the last episode of your favorite show is just wrapping up as you get paged. You have five minutes to identify and/or mitigate an issue before the ticket is escalated to your grumpy boss. Which service along a request path is the problem?

The good news is, service meshes can be a life saver here as they usually provide deep observability integrations, typically based off of de-facto standard such as CNCF OpenTelemetry or the up and coming IETF RFC on OpenMetrics. Service meshes deliver a torrent of signals out- of-the-box such as, logs, metrics or tracing, can be used for troubleshooting. For example, you can use AWS X-Ray with AWS App Mesh to pinpoint and offending service and zoom in on its inner workings.

The basic idea is telemetry automation. Since all traffic goes via the proxy, you have a central choke point to gather low-level metrics, automatically, hence eliminating the need to instrument the application. For higher-level/business metrics not derivable from lower-level ones this is often not the case, we will return to this topic further down.

Critical voices point out that not every observability type is a necessity. You should also be aware of the fact that while meshes can relieve developers considerably concerning instrumentation, they can't automatically glean and deliver different business metrics pertaining to a certain query. Some hand-holding and hinting from developers is still necessary, even with a mesh.

> For more details and recommendation around how to use service meshes in the context of microservices, we recommending reading the NIST Special Publication 800-204A on Building Secure Microservices-based Applications Using Service-Mesh Architecture.

Good Practices

In the following section, we discuss good practices for selecting and using a service.

> **Note:** We call this section good and not best practices. This acknowledges the fact that the service mesh topic is a very recent one and, while one can find organizations running service meshes in production, the community as a whole has not yet established practices that deserve to be called best. You may want to revisit your decisions and take the suggested selection criteria and usage recommendations with a grain of salt.

Selection Criteria

If you're considering using a service mesh, a great way to start is to approach the topic from a use case point of view. You can use the examples we discussed in the

Use Cases section as an initial starting point. This is a good idea as the use cases we discussed there the most common ones we encounter from customers, directly or via surveys by vendors and press. Also, consider focusing on low-hanging fruits such as observability and security, ideally in a green-field environment that comes with low risk.

Once you have identified a use case you can move on to the selection of the mesh. Given the maturity of the field, the exact mesh matters less than building on standards (See the Specifications and De-Facto Standards section for more details) as well as being able to externalize support. Ideally, you end up with a shortlist of two or three meshes you would consider fitting the bill. Next, you will want to evaluate and indeed test the shortlisted meshes for certain features and most importantly if they can address the selected use case.

There are three main categories you want to consider when selecting a service meash:

1. Runtime environment
2. Cross compute
3. Availability

The following sections discuss these three categories in greater detail.

Runtime environment

Your workload might run in any of the following environments: bare metal on-premises, virtual machines on-premises (such as with VMware or OpenStack), Kubernetes such as Amazon EKS, Amazon ECS, or AWS Lambda. Depending on the environment, different networking capabilities are available. Different runtime environments may have different levels of support concerning a certain mesh. Also, convenience functions, such as auto-ingestion of the proxy, may depend on the target runtime environment.

Consider asking the following questions:

- Is my targeted environment supported? If so, are there any conditions? For example, the mesh's model may draw from a certain underlying environment resource model (as the case in Istio, Linkerd and SMI-based meshes that implicitly assume a Kubernetes-based model and often define their primitives using the Kubernetes resource model).
- Does the mesh offer installation and lifecycle management of both the data plane components and the services using the mesh?

- Are there dependencies on the networking layer (requirements such as must support a certain SDN or can only be used with a certain orchestrator)?

Support for cross-compute

In most cases you want to connect workloads running in different environments. For example, one part might—due to legal or regulatory requirements—run in your data center as a monolithic app on a virtual machine and the other part might be a set of containerized microservices running in Amazon ECS.

Consider asking the following questions:

- Does the mesh support cross-compute workloads and are there any restrictions?
- Is the discovery of services across different runtime environments supported? Is it automated? Does the mesh discovery support HTTP-based and non-HTTP based APIs alike?
- For TLS across environments, is there a shared root of trust amongst different environments available? What about federation capabilities?
- How is authentication and authorization performed across environments?
- Is a global load-balancer available and/or integration for an existing cloud offering, such as App Mesh Ingress, done?

Availability and Costs

In this section, we explore the following: how and under which conditions can I use the mesh? The considerations here are of a non-technical nature. You should spend time on this topic since the outcome here may influence your decision as much, if not more, than feature shortcomings amongst different offerings.

Consider asking the following questions:

- Is the mesh a managed service? Is it a proprietary offering? Also, is the mesh available as open source (in total or parts of it)?
- What is the TCO (Total Cost of Ownership) of the service mesh? This can include extra runtime costs of running the proxy, license costs (per environment, per service, etc.), support, training, and also own engineering cycles that may be necessary for projects that are not actively maintained anymore.

- What's the impact on infra/ops teams and how does the mesh impact developers?

Operational Considerations

Once you have selected your mesh and evaluated it for a use case, you're about to deploy it into an environment and put workloads on it. In the following section, we share some considerations around managing and operating service meshes. Given that we're talking in general terms here, we're not aiming to provide a comprehensive and all-encompassing checklist. We will, however, where appropriate use specific examples to illustrate a point.

Automation

Make sure to automate any and all processes. This may sound like a trivial and obvious point; however, details are important.

The following is a non-exhaustive list of automation-related topics to check:

1. Installation and configuration of both control plane (in case of non-managed offering) and data plane (proxy and their interaction with a service).
2. Upgrades of the control plane (in case of non-managed offering) as well as the data plane (proxies).
3. Deployments/auto-ingestion of proxy into a service.
4. Health checks (on infra and app level).
5. Control plane outages impacting the data plane, and in this context, are automated rollbacks to last previously known good state available.
6. Security patches (for example, for the proxies) available and automatically applied.

For example, AWS App Mesh, as a fully managed service, takes care of most of the above points out-of-the-box. The install and configuration UX depends on the environment: for example, using App Mesh with EKS is straightforward given the combination of a Helm chart (via aws/eks-charts) and a custom controller, taking care of ingesting the Envoy proxies as well as configuring them.

Observability

One of the table stakes in service mesh land is built-in observability (o11y). Many vendors claim this and practitioners report on their experiences with it. You want to make sure your mesh indeed does provide you with all three types of o11y signals (logs, metrics, traces) out-of-the-box.

Verify your mesh meets the following criteria:

1. Check what metrics/logs/traces are delivered by the mesh. Are these signals configurable and if so, how?
2. Does the mesh support routing of logs and/or metrics to different down-stream sinks? For example, you might want to use Prometheus to scrape the traffic metrics from your proxies and view overall graphs in Grafana. For that, the mesh should support these common setups out-of-the-box as well as provide exporters and/or integrations with popular monitoring and dashboarding solutions such as Splunk and DataDog.
3. Build on open standards: support for OpenTelemetry, SMI, or OpenMetrics should be a high priority. At the very least verify plans by the service mesh provider to support such standards, going forward (road mapping conversations, GitHub projects, etc.).

In this context, one trailblazing example is CNCF Linkerd with its rich and automated dashboards for metrics. Same should hold for tracing, such as found in App Mesh with its rich tracing support for AWS X-Ray, CNCF Jaeger, and DataDog. Another up-and-coming player in this field is the Red Hat-originated Kiali project that positions itself as the “management console for Istio”.

In addition to open source based offerings, there’s a vibrant and extensive ecosystem of microservices o11y vendors catering directly to service meshes audiences, including but not limited to Honeycomb, Dynatrace, Lightstep, and HashiCorp. The CNCF End User Technology Radar: Observability from September 2020 might be useful to you as well to gather some further details in this space.

Integrations

In the general case, practitioners prefer the service mesh to be invisible. You can envision the service mesh features being part of the underlying networking fabric, such as an SDN or in the case of the cloud we would be thinking of the Amazon Virtual Private Cloud (Amazon VPC).

The service mesh is then a capability of the runtime environment: consider that just as today you can choose to use Amazon ECS on Amazon EC2 or on AWS Fargate, representing a compute capability, one day you may want to view service meshes in the same way, as a networking capability.

No matter on what layer service meshes happen and how visible they are to you as a user, there are a number of requirements for integration with others in this environment.

Validate how your selected mesh provides for the following:

1. Security features including certificate management (for example, SPIFFE support for workload identities), authentication, and authorization. In the case of AWS, you naturally are looking for deep IAM integration.
2. How policies such as access control or trust are supported. For example, does your mesh build on open formats like CNCF Open Policy Agent (OPA) Rego?
3. Integrations with observability tooling such as dashboarding and alerting. For more information, see the Observability section.
4. Can the selected mesh support North-South traffic and if so, what options (HTTP, gRPC, Layer 3/4) are available? For example, App Mesh natively supports Ingress Gateways.
5. Support for Continuous Deployment (CD) such as based on the GitOps paradigm as well as progressive delivery ala Flagger, which can also be used for other use cases such as auto-scaling.

There are many more integration points you could consider in addition to the points above as a useful baseline.

Interoperability

In this section, we look at interoperability aspects in the context of service meshes and discuss the technical foundations and emerging and de-facto standards.

Basics: The TCP/IP Stack

The TCP/IP stack, also sometimes called the Internet protocol suite as shown in the Figure 4 forms the basis for all further considerations. All security and observability considerations apply. Most if not all of the existing networking-related tooling will be of continued use. For example, how do you verify if the automatic mTLS of Linkerd in your East-West traffic actually takes place? You might as well find yourself ending up using `tshark`, the terminal version of the venerable, 20+ year old Wireshark.

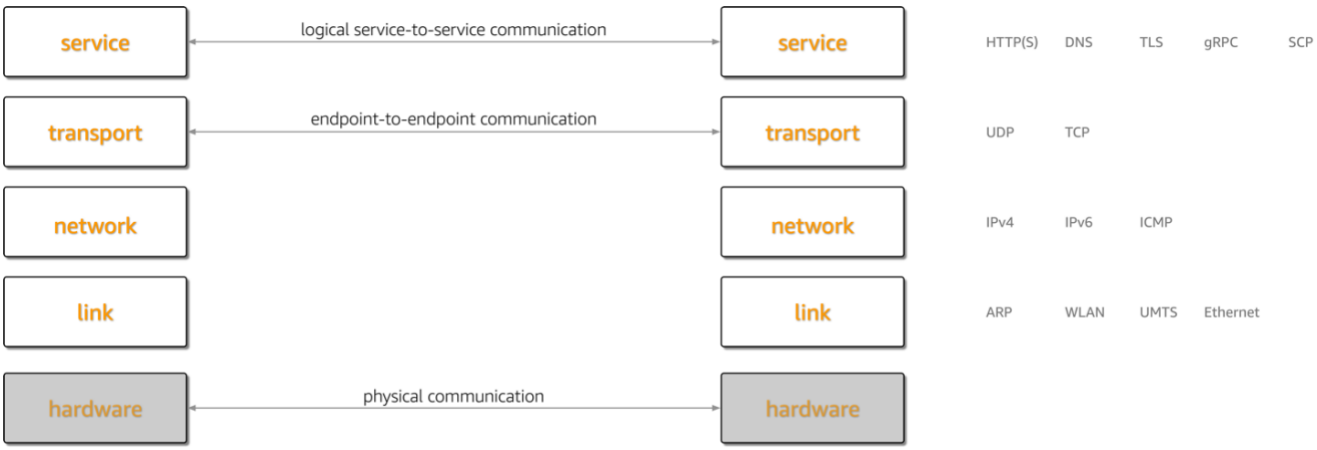

Figure 4—The TCP/IP stack as the basis of service meshes.

Do not underestimate the core message here: under the hood service meshes use, to a great extent, protocols and tools that have been around for a long time. This is good news since it minimizes risk on a principled level and fosters amortization of training and investments in existing systems, be that monitoring or hardware.

However, there are some important areas you should pay attention to, including the reliance on Software-Defined Networking (SDN) technologies and emerging standards—nowadays often developed in Marketing-orientated foundations such as CNCF rather than more traditional bodies such as W3C or IETF. In order to better understand that space, let's have a closer look at the current ecosystem.

On Specifications and De-Facto Standards

Envoy APIs (UDPA)

As discussed in the Service Mesh Concept section, the data plane of a service mesh is made up of proxies that sit next to your service, intercepting the traffic. The Envoy proxy powers the majority of open source and commercial service meshes.

Originally developed by Lyft and open sourced in late 2016, Envoy is an CNCF project and its APIs are considered the industry standard. This dominance of Envoy and its APIs has led to the creation of the Universal Data Plane API Working Group in CNCF which might yield a standardized data plane in the fullness of time.

For now, consider the Envoy APIs themselves as the de-facto standard specification for the service mesh data plane.

The Service Mesh Interface

Service Mesh Interface (SMI) is a CNCF project, created in 2019 and with active participating across cloud providers including AWS as well as many of our partners such as HashiCorp, Solo.io, and Cisco.

SMI is focused on Kubernetes and defines the following primitives, along with configuration and defaults via custom resources:

1. **Policy**: identity and transport encryption across services within a cluster:
    - Access control: enables configuration access to pods or routes, based on the identity of a client.
    - Protocol specs: enables defining how traffic looks on a per-protocol basis.
2. **Telemetry**: expose key/common metrics such as error rate and latency between services within a Kubernetes cluster for use by downstream tooling, for example a dashboard such as Grafana or autoscaling on the cluster or app level.
3. **Shift/split traffic**: incrementally direct a certain percentage of traffic between services within a Kubernetes cluster, for canary rollouts and A/B deploys.

Among the service mesh control plane specifications, SMI has the widest support.

BPF and Cilium

BPF, originally Berkeley Packet Filter however, nowadays a term sui generi, is a Linux feature that enables you to safely and efficiently extend the kernel functionality. BPF is exposed to user space via the `bpf(2)` system call and is implemented as a virtual machine (VM) in the Linux kernel. This VM uses a custom 64-bit RISC instruction set.

Figure 5 is a high-level BPF workflow overview, taken from Brendan Gregg's book Linux Extended BPF (eBPF) Tracing Tools.

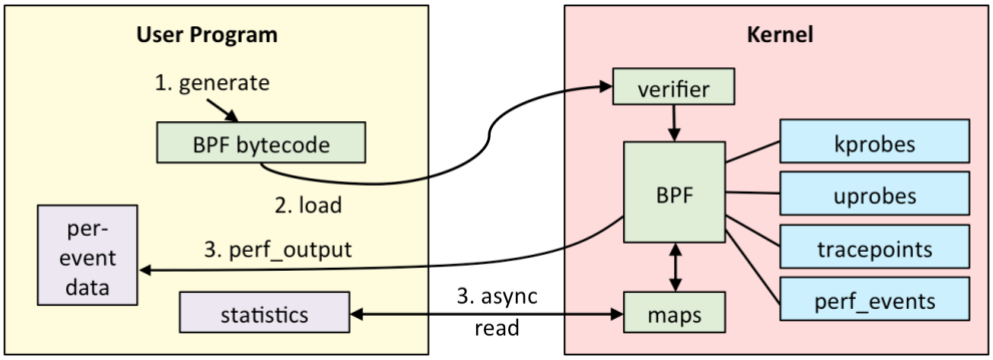

Figure 5—BPF workflow with separations of concerns between user land and Linux kernel.

The most important open source player in the BPF space is the Cilium project.

Additional Offerings

Hamlet

A VMware initiated project that describes itself as a set of API standards for enabling service mesh federation. For details, see the GitHub repo vmware/hamlet.

Service Mesh Hub

Service Mesh Hub is a Solo.io offering that aims to provide a multi-cluster service mesh management plane, helping organizations to streamline the installation, configuration, management, and extensibility of any service mesh on any cloud. For more information, see the documentation at solo.io/products/service-mesh-hub/.

Conclusion

Service meshes are a very useful addition in the context of (containerized) microservices, enabling you to externalize many service-to-service and North-South communication challenges including security and traffic shaping. It's still early days, so focus on standards and evaluating for use cases that show clear value.

In this whitepaper we reviewed what service meshes are, when they should be used, and good practices for selection. We also outlined the most common use cases and provided guidance around selecting and using service meshes.

Contributors

Contributors to this document are:

- Alexandr Moroz, Senior Product Manager Tech
- Brian Celenza, Senior Software Engineer
- Michael Hausenblas, Senior Open Source Product Developer Advocate

Further Reading

For additional information, see:

- The Service Mesh: What Every Software Engineer Needs to Know about the World's Most Over-Hyped Technology
- Service mesh use cases
- Service Mesh Comparison
- The Service Mesh Manifesto
- Review: AWS App Mesh – A service mesh for EC2, ECS, and EKS

Document Revisions

Date	Description
November 2020	First publication