Service meshes in a microservices architecture

This document introduces service meshes and how they function at a high level. It defines service meshes, explains their role and architecture, and describes their importance in distributed enterprise application systems. It also discusses some service mesh attributes. This guide is for system architects and platform developers that review, assess, and plan the use of service meshes in their architecture.

What are the new problems in microservices architecture?

Companies are increasingly adopting microservices, containers, and Kubernetes. The need to modernize, and the need to increase developer productivity, application agility, and scalability drives this increase. Many organizations are also venturing into cloud computing and adopting a distributed microservice architecture for both new and existing applications and services. Monolithic applications (a single application providing multiple functions) are complex to build and slow to release. While the architecture of microservices helps simplify creating individual services, it leads to additional or increased complexities, like those in the following sections.

Security

In a monolithic application, all function-to-function calls are secure inside the monolith. Consider how microservices authenticate, authorize, encrypt, and communicate. Also consider the additional auditing tools needed to trace service-to-service communication.

Network resiliency

In distributed architectures, consider the effect of latency and the overall response time when multiple services communicate to produce a response. Also consider fault tolerance. How does a distributed architecture ensure that a service in one downstream service does not cause cascading failure in other services?

Communication policy

In distributed architectures, some services can become bottlenecks or dependencies for other services. Network policies that manage quotas and rate limits for all services can ensure that a rogue service making too many calls does not overload the services it calls. To effectively control services, create policies that specify which services can and can't make calls.

Observability

When compared with monolithic applications, observability is more important in microservice-based architectures. In monolithic applications, log files are sufficient to identify the source of an issue. In a microservices architecture, multiple services can span a single request. Latency, errors, and failures can happen in any service within the architecture. Developers need logging, network metrics, and distributed tracing and topology to investigate problems and pinpoint their location.

Why are these problems more prominent for enterprises?

If one (or a few) applications are split into microservices, it makes security, network resiliency, policy, and observability easier to address. However, enterprises might have dozens, hundreds, or thousands of microservices. Therefore, any solution must scale. If not done correctly, the complexity of applications and the amount of microservices creates a greater dependence between those services.

What are service meshes?

A service mesh is a platform layer on top of the infrastructure layer that enables managed, observable, and secure communication between individual services. This platform layer enables companies or individuals to create robust enterprise applications, made up of many microservices on a chosen infrastructure. Service meshes use consistent tools to factor out all the common concerns of running a service, like monitoring, networking, and security. That means service developers and operators can focus on creating and managing applications for their users instead of worrying about implementing measures to address challenges for every service.

Service meshes are transparent to the application. The service mesh monitors all traffic through a proxy. The proxy is deployed by a sidecar pattern

(https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/#example-1-sidecar-containers)
to the microservices. This pattern decouples application or business logic from network functions, and enables developers to focus on the features that the business needs. Service meshes also let operations teams and development teams to decouple their work from one another.

Features and functionality

Service meshes provide specific functionality to manage and control communication relationships between services. Some of this functionality is discussed in the following subsections.

Multi-tenancy

The multi-tenancy deployment pattern isolates groups of microservices from each other when a tenant exclusively uses those groups. A typical use case for multi-tenancy is to isolate the services between two different departments within an organization or to isolate entire organizations altogether. The goal is that each tenant has its own dedicated services. Those services can't access the services of other tenants at all, or can only access other tenants' services when authorized.

The simplest form of multi-tenancy is to have infrastructure dedicated to a single tenant. Each tenant has its own network, compute, storage, and additional components like Kubernetes and microservices, without sharing infrastructure. While this form of multi-tenancy is possible, its infrastructure use is inefficient in many situations. It's more efficient to share infrastructure among tenants and rely on service mesh configuration and policies to separate them.

Service mesh multi-tenancy is based on one of two tenancy forms (https://istio.io/docs/ops/deployment/deployment-models/#tenancy-models): namespace tenancy and cluster tenancy.

Namespace tenancy

The namespace tenancy form provides each tenant with a dedicated namespace within a cluster. Because each cluster can support multiple tenants, Namespace tenancy maximizes infrastructure sharing.

To restrict communication between the services of different tenants, expose only a subset of the services outside the namespace (using a sidecar configuration) and use service-mesh-authorization policies to control the exposed services. Configure each namespace individually for the set of available services. Because access to each service is authorized, only allowed tenants can access each other's services. While multi-mesh federation (https://istio.io/docs/ops/deployment/deployment-models/#multiple-meshes) supports this use case, it's not necessary to create a multi-mesh federation.

A namespace can span one or more clusters. The tenancy is defined solely by the namespace and is independent of the clusters that support the namespace. In fact, two different service meshes can have the same namespace. An example of this concept is one service mesh representing a staging tenant, and one service mesh representing a production tenant. Both can have a *customer* namespace. Because this naming scheme is confusing, it's not ideal.

Cluster tenancy

The cluster tenancy form exclusively dedicates the entire cluster, including all namespaces, to a tenant. A tenant can also have more than one cluster. Each cluster has its own mesh.

Cluster tenancy means separation on a cluster level; it's not truly multi-tenant in terms of sharing resources between tenants. However, because it's mentioned in Istio tenancy model documentation (https://istio.io/docs/ops/deployment/deployment-models/#tenancy-models), it's included here.

Security

Security is important regardless of architecture type. Microservices have additional security needs compared to other architectures. For example, authentication, authorization, and traffic-flow control between microservices. The security features within service meshes address these needs.

Traditional network security is based on a strong perimeter to prevent unauthorized access. After users are inside the network perimeter, they are considered trusted actors and are allowed to communicate without verifying their identity.

In 2010, Forrester popularized the concept of *zero trust*. In a zero trust environment, it's no longer assumed that anything within a specific security perimeter is trusted. Instead, it's assumed that the network is compromised and unaware of it. Everything is verified. In this case, the only trusted perimeter is within the service itself. Anything else, even if within the same network, is implicitly untrusted.

Before service mesh, achieving zero trust was difficult. Trust required tooling to manage certificates for services and workloads, as well as service authentication and authorization. After implementing a service mesh, achieving zero trust is less complex. Service meshes provide these authentication and authorization identities through a central certificate authority that provides certificates for each service.

Service meshes use these identities to authenticate and authorize services inside and outside of the mesh. The certificate authorities and the availability of certificates let developers implement authorization policies that provide fine-grained control over which services can communicate with each other. it's also possible to granularly specify the paths and HTTP verbs that are allowed for certain services.

Service meshes give platform developers the ability to enforce policies, like mutual TLS (https://istio.io/latest/docs/tasks/security/authentication/mtls-migration/), to ensure encrypted traffic between services and to help prevent person-in-the-middle (https://wikipedia.org/wiki/Man-in-the-middle_attack) attacks. After the service mesh is deployed, it's responsible for encryption and decryption of all requests and responses.

Observability and analysis

Observability is a set of activities that include measuring, gathering, and analyzing multiple signals from a system. Observing systems was less complex before microservice architectures. Requests came to a single service and it collected the data.

In a distributed microservices architecture, the response data must be gathered from multiple services to get the full response. As previously discussed, all traffic to and from a service in the mesh passes through a proxy. This proxy enables operators to gain greater visibility into service interactions. Each proxy reports on its portion of the request to produce the same comprehensive view that existed in monolithic applications. A mesh typically generates the following types of telemetry to provide observability: metrics, distributed traces, and access logs.

Metrics

A mesh produces metrics for all traffic coming into the mesh, within the mesh, or leaving the mesh. Examples of these metrics include error rates, number of requests per second, and request response times. These metrics help developers understand service behavior in aggregate.

The mesh can produce the following metrics, as described in the Istio documentation (https://istio.io/docs/concepts/observability/):

- **Proxy-level metrics**: Sidecar proxies generate a large set of metrics about all inbound and outbound proxy traffic. These metrics include detailed statistics about the proxy's administrative functions, like configuration information and health information.
- **Service-level metrics**: Service-level metrics cover the four golden signals of monitoring: latency, traffic, errors, and saturation.
- **Control plane metrics**: Control plane metrics monitor the service mesh control plane rather than the services within the mesh.

Distributed traces

A service mesh can generate distributed trace spans for each service within it. Use these traces to follow a single request through the mesh across multiple services and proxies.

Access logs

A service mesh can generate a full access log that includes all the service calls, including the source of the call and its destination which allows for auditability at the service level.

Compliance

Compliance means enforcing the policies and rules used to govern a system. These policies and rules are either self-imposed or imposed by industry or government regulation.

One form of enforcement is monitoring and auditing workloads to determine whether there are any policy or rule violations that compromise compliance. Another form of enforcement is using formal implementations of policies and rules that ensure that they are in effect. In practice, both are put into place: formal implementations for policies and rules, in addition to real-time monitoring and auditing.

The following list provides a general compliance overview. Additional policies and rules might apply depending on the industry or the workload.

- **Monitoring and auditing**: Monitoring and auditing workloads helps to determine whether there are any policy or rule violations within the system.
- **Security**: All systems within a microservice architecture must be secure, authenticated, and able to provide authorized access to all of their endpoints.
- **Redundancy**: To avoid a single point of failure, redundancy requires deploying each microservice in more than one location. With Istio, it's possible to check whether a microservice is deployed more than once. Redundancy only requires a microservice to be deployed more than once, it might be in the same zone. If deployed in the same zone, the architecture does not provide high availability.
- **High availability**: A high availability deployment continues to function during a zone outage. That means the zone does not become a single point of failure. Each service and each component must be deployed in at least two zones to ensure continuity. Aside from the microservice functionality itself being able to react to a zone outage, the service mesh configuration can be used to automatically analyze complete redundancy in at least two zones.
- **Disaster recovery**: Disaster recovery is similar to high availability. The difference is that a system deployed for disaster recovery continues to function during a single region outage. Like with highly available systems, service mesh configurations can be automatically analyzed to ensure they are appropriately deployed.
- **Partitioning (multi-tenancy)**: Microservices can be used to implement multi-tenant systems that support several tenants at the same time. In this document, see Multi-tenancy (#multi-tenancy) for a detailed discussion of namespace partitioning and cluster partitioning. Analyzing the service mesh configuration can help ensure it's properly partitioned.
- **Runtime properties**: Policies and rules focus on a static deployment or configuration or they can be runtime policies. For example, a runtime policy might enforce an upper latency limit. In this case, the system interrupts invocations that take longer than the defined latency period. Whatever the defined policy, it must be enforced at runtime. For more information, see the resilience (#resilience) section later in this document.

The preceding list is a subset of the compliance policies and rules that an application or a company must comply with and enforce.

There are some policies and rules that can't be enforced with a service mesh. For example, data residency requirements are outside the scope of a service mesh. If user data must reside in the same country as a user's specified home location, a service mesh is insufficient to configure or to enforce this requirement. Another example is storing all data and their history for a specific number of years. A service mesh can't implement and supervise that policy.

Traffic control

A service mesh controls the flow of traffic between services, into the mesh, and to outside services. Custom resources allow users to manage this traffic. These resources vary depending on the service mesh chosen. They enable users to create canary rollouts, create blue/green rollouts, and create fine-grained control over specific routes for services.

The service mesh maintains a service registry of all services in the mesh by name and by their respective endpoints. It maintains the registry to manage the flow of traffic (for example, Kubernetes Pod IP addresses). By using this service registry, and by running the proxies side-by-side with the services, the mesh can direct traffic to the appropriate endpoint.

Load balancing

In most microservices architectures, there are multiple instances of each service running (for example, Pods in Kubernetes). Traffic is load balanced across the instances. A service mesh can control the load-balancing behavior of those services. Usually, the default behavior is round-robin (https://wikipedia.org/wiki/Round-robin_DNS) across the instances of the service. However, it can be random, weighted according to a specific percentage of traffic, or directed to the service with the least traffic.

Source (caller) restriction

In general, much focus is placed on the microservices receiving invocations from clients. For example, load balancing, abstraction into virtual services, or retry invocations. To ensure that the only microservices that communicate are those that require the invocation, restrict the callers of microservices. Microservices must avoid any accidental communication or erroneous communication. When it can't be avoided, it must be identified.

For example, in a banking application there is a `debitCredit` service that can add or subtract amounts from a checking account. That service is only available to services in the area of funds transfer. The goal is to prevent the invocation of `debitCredit` by non-funds-transfer services.

There are several ways to enable correct service mesh communications. One way is to identify callers by their identifying service name and list them individually to specify the permitted callers for a service. Another way is to use labels as identifiers instead of service names. In the previous banking example, all services that belong to the group of funds transfer services could be labeled *fundsTransfer*. The service mesh uses the label to specify which callers are permitted to call a service.

Resilience

A service mesh can increase the invocation resilience of microservices deployed on Kubernetes. There are two classes of resilience measures:

- Increasing the reliability of microservice invocations
- Intentionally creating invocation failures

Increasing the reliability of microservice invocations

The reliability of a microservice invocation increases if failures are abstracted from the caller. If a failure occurs, the service mesh can use the following strategies to try to address it transparently without returning a failure to the caller:

- Timeout (https://istio.io/docs/concepts/traffic-management/#timeouts)
- Retry (https://istio.io/docs/concepts/traffic-management/#retries)
- Circuit breaking (https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/)

Intentionally creating invocation failures

A service mesh provides resilience by specifying timeouts or retries. Applications can also implement resiliency. For example, it's possible that an application must invoke three microservices sequentially to process input and obtain a result. If one of these invocations fails, the application can retry the sequence again to see if the invocations work on the second attempt.

To test that an application functions properly, it's possible to inject intentional invocation faults (https://istio.io/docs/concepts/traffic-management/#fault-injection) through a service mesh at runtime. One type of fault is a delay. The invocation is intentionally delayed and that delay tests the ability of the application to deal with a variation in latencies. Another type of fault is an abort. An abort interrupts the invocation. The application observes an invocation failure and then decides how to deal with that failure.

These measures are applied at runtime to actual invocations in a production system.

Architecture

At a basic level, a service mesh consists of services and proxies running as sidecars to the services. It also includes some authority that configures those proxies to combine the proxies and services into a proper distributed system, including a data plane and a control plane. All requests to or from a service pass through two proxies within the mesh: the proxy for the calling service and the proxy for the receiving service.

This architecture abstracts all functions that are not related to the business logic away from services and service developers. The data plane manages the proxies and services. The control plane is the authority that provides policy and configuration to the data plane.

The service mesh control plane enables the proxies to perform the following functions:

- Service discovery
- Service routing
- Load balancing
- Authentication and authorization
- Observability

The control plane is responsible for the following:

- **Service registry**: The control plane must have a list of available services and endpoints to provide them to the proxies. The control plane compiles this registry by querying the underlying infrastructure scheduling system, like Kubernetes, to get a list of all available services.
- **Sidecar proxy configuration**: The configuration for sidecar proxies includes policies and mesh-wide configurations that the proxies need to be aware of to appropriately perform their functions.

For a diagram of the services that interact with the control plane, see proxies running as sidecars (https://hackernoon.com/service-mesh-with-envoy-101-e6b2131ee30b).

Design considerations

Although a service mesh can look like a perfect solution for many aspects of microservice system design and system implementation, there are caveats. Some of these caveats are described in the following sections.

Processing overhead

Invocations from one microservice to another are routed through a proxy and possibly through a load balancer. In addition, the invocations are tracked, and possibly modified, through encryption. While encryption does not cause significant overhead on an individual level, in aggregate it adds to latency and to resource requirements. To determine whether the overhead for a given use case is significant, analyze it with performance and scalability measurements (https://istio.io/docs/ops/deployment/performance-and-scalability/).

Configuration design complexity

Creating a service mesh configuration is a design activity that must ensure that requirements are properly implemented. It requires knowledge about the configuration capabilities of service meshes in general. Knowledge of how to create the correct configurations for specific applications is also required. When service meshes are configured, that configuration must reflect the system requirements.

Test configuration validity

After a service mesh configuration is in place, use tools like Istioctl Analyze (https://istio.io/docs/ops/diagnostic-tools/istioctl-analyze/) to validate the configuration. Because the configuration could change as the application implementation progresses, it's important to repeat the validation constantly as part of the CI/CD process. After you validate the configuration, test the service mesh configuration to confirm that the intent of the behavior expressed in the configuration matches the expected microservice invocation behavior. For more information, see the Testing section (#testing).

Verify the service mesh configuration

The existence of a service mesh control plane does not automatically ensure system security and reliability. A service mesh must be configured, and that configuration must be properly tested and verified. Doing so makes it possible to avoid problems like undetected insecure invocations.

Changes to microservices (like additions or updates) might change the communication behavior of the mesh, but not so much that the configuration considers it a change. To ensure that all changes are properly covered by the service mesh configuration, perform a service mesh configuration review for every change.

A service mesh does not cover all security aspects that require implementation in an enterprise setting. A service mesh addresses all aspects around service communication; any infrastructure security requirements, like firewalls and network security, need to be addressed separately.

Service mesh control plane updates

A service mesh is implemented as a system that might change over time. For example, changes to address performance improvements, scalability improvements, additional features, or bug fixes might be implemented. When updating the control plane of a service mesh, it's important to regression test the existing configuration against the updated system. Regression tests should ensure that the new system version of the service mesh does not negatively alter the service mesh behavior.

Testing

Testing ensures the proper setup and functioning of a service mesh. To perform comprehensive testing, include the following checks:

- General service mesh configuration check
- Service mesh configuration check against microservice requirements
- Service mesh control plan deployment version check

General service mesh configuration check

Use the Istioctl Analyze tool to perform a general service mesh configuration check. Use its configuration parameters (https://istio.io/docs/reference/commands/istioctl/#istioctl-experimental-analyze) to fine-tune behavior. Developers can find the current set of analyzers in the Istio GitHub repository (https://github.com/istio/istio/tree/release-1.5/galley/pkg/config/analysis/analyzers). When Istioctl Analyzer detects an issue, it returns an error message (https://istio.io/docs/reference/config/analysis/) like `IST0114: PolicySpecifiesPortNameThatDoesntExist`.

Service mesh configuration check against microservice requirements

A service mesh manages and controls microservice communication. Testing helps to ensure that the formal service mesh configuration implements the requirements of the microservice. Some example service mesh features include the following:

- Communication: Which microservices can communicate, which can't?
- Security: Is the communication configured as expected? For example, does it use HTTP or HTTPS?
- Dynamic behavior: Does the service mesh throttle communication enough to avoid overwhelming one or more microservices?

A service mesh is configured through declarative configuration files. These files are part of the code repository. They contain specifications that address specific microservice use-case requirements that are inherent to the microservice's implementation and logic. For example, the specifications state which microservices can communicate and which can't. The specifications also state how much throughput is possible and when it has to be throttled.

From a software engineering perspective, treat a service mesh like regular microservice implementation code. Perform both positive and negative tests on it. A positive test asserts that functionality or behavior is present. A negative test asserts that a specific functionality or feature is absent. In the context of a service mesh configuration, both tests are possible. Both tests can prove whether a service mesh configuration corresponds to the microservice requirements. Tests can be unit tests and integration tests, depending on the particular use cases.

To test communication, also use both positive and negative test cases. For example, microservice MS1 calls microservice MS2, but not the other way around. Establish one test to assert that MS1 can call MS2, and establish another test to assert that MS2 can't reach MS1.

One way to build the tests is to implement two mock microservices. Create one for MS1 and one for MS2 such that both can invoke each other. Optionally, establish a test where both MS1 and MS2 can invoke the other without the service mesh being present. When the service mesh is present, the invocation from MS1 to MS2 must work, while the invocation from MS2 to MS1 must fail.

It is a good practice to define additional tests that account for resilience (#resilience). For example, if fault injection is creating invocation delays, it's important to confirm that the clients can deal with those delayed invocations.

Many aspects of a microservice are covered by service meshes, like security, reliability, or scalability. Because the overall system relies on their proper configuration, it's important to establish test cases with a high percentage of overall coverage. In some areas, like security, the testing coverage might reach 100%.

Check the control plane deployment version

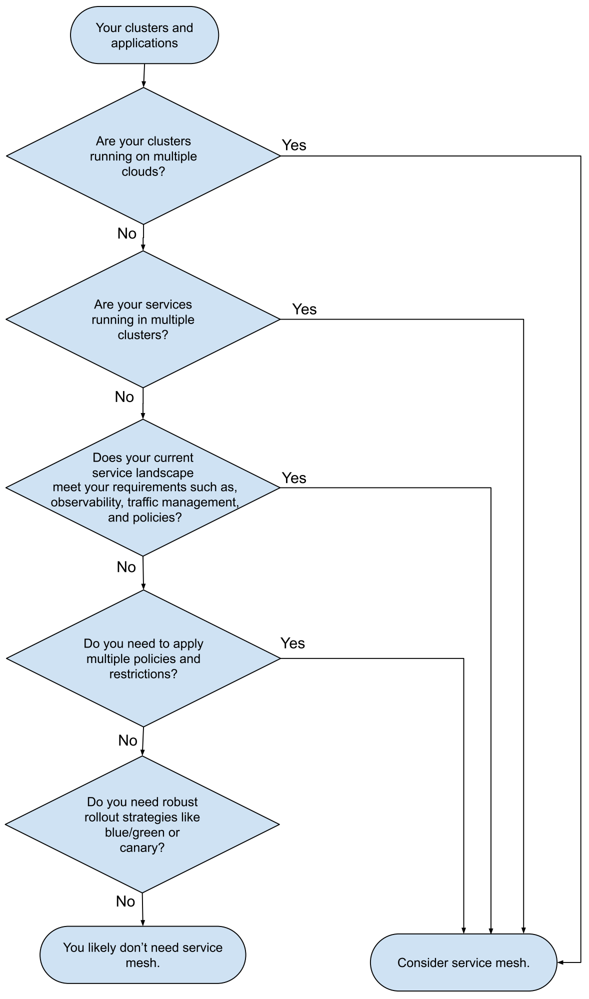
It's possible to install and to run two different versions of the service mesh control plane (https://istio.io/latest/docs/setup/upgrade/) at the same time. Testing must establish that the current version of the control plane and the newer version of the control plane behave in the same way. Alternatively, testing must establish that the behavior of the newer version is an improvement. Defining automated tests that check the control flow plane behavior removes the need for manual tests. For example, observing the behavior with monitoring tools and log file examination. Manually observing control flow plane behavior means that one or more people compare what is observable. They then assess whether the behavior of the current version and the new version are the same.

In addition to observing the overall service mesh behavior, the specific unit and integration tests (#service_mesh_configuration_check_against_microservice_requirements) must behave in the same way. If they don't, it means there is a service degradation to evaluate. This evaluation is part of a regular CI/CD and testing practice. However, the process is important to application configuration changes and platform changes.

Only test a new control plane version when it's deployed in all clusters. Ideally, the unit and integration tests run continuously as part of the CI/CD pipeline.

Use cases

There are certain use cases and architectures that lend themselves to service meshes. The following flowchart represents a decision tree to determine whether deploying a service mesh is a feasible solution.


Your clusters and applications
Are your clusters running on multiple clouds?
Yes
No
Are your services running in multiple clusters?
Yes
No
Does your current service landscape meet your requirements such as, observability, traffic management, and policies?
Yes
No
Do you need to apply multiple policies and restrictions?
Yes
No
Do you need robust rollout strategies like blue/green or canary?
No
You likely don't need service mesh.
Consider service mesh.

As shown in the preceding decision tree, if services are running in multiple clouds—whether it's on multiple cloud providers or an on-premises data center—then consider using a service mesh. If services are running on a single cloud, consider the following non-exhaustive factors when deciding whether to use a service mesh:

- **Running in multiple clusters**: Service meshes enable application developers to abstract away all the communication overhead from services and offload it to the mesh. If your services are running in multiple clusters, consider a service mesh.
- **Service landscape and requirements**: There is no magic number for the minimum number of services that warrant a service mesh. Consider organizational requirements for network functions. For example, observability, service level policy management, and

traffic control. Also consider a team's ability and ease to instrument and add libraries to services to meet those requirements. If there are five services that are hard to instrument, that would be a reason to consider a service mesh. Conversely, if there are 100 services that are already instrumented to match organizational requirements, that would be a reason to not implement a service mesh.

- **Policies and restrictions**: Service meshes improve the ability to apply policies to an entire mesh or to granular services. Therefore, if there are many policies or if there is a need to have fine-grained control with minimal additional work, consider a service mesh.
- **Robust rollout strategies**: Service meshes can improve the implementation of robust rollout strategies, such as blue/green and canary. If you need robust rollout strategies, consider a service mesh.

Example service mesh: Istio

Istio is a service mesh implementation (https://istio.io/). Istio's architecture contains a data plane and a control plane. The data plane consists of Envoy (https://www.envoyproxy.io/docs/envoy/latest/) proxies that control the communication between microservices and also collect metrics. Incoming traffic (called ingress), outgoing traffic (called egress), and traffic between services (mesh traffic). Each microservice instance (container or VM) has a dedicated Envoy proxy.

The control plane is the management layer for the Envoy proxies. It manages the proxies so that the correct invocation routing occurs. The istiod binary is the core of the control plane and provides service discovery, configuration, and certificate management.

For more information about the various control plane components, refer to Istio's architecture page (https://istio.io/docs/ops/deployment/architecture/).

Istio supports the following deployment models (https://istio.io/docs/ops/deployment/deployment-models/):

- single or multiple Kubernetes clusters
- single or multiple networks
- single or multiple control planes
- single or multiple meshes

The preceding deployment options support various use cases. Istio users can select the best option for their scenarios.

Istio is not the only commercially available service mesh. Linkerd (https://linkerd.io/2/overview/) is also available. Anthos Service Mesh (/anthos/service-mesh) is Google Cloud's fully managed service mesh. It gives developers an Anthos tested and supported distribution of Istio, letting them create and deploy a service mesh on Google Cloud or on Anthos clusters on VMware with full Google support.

What's next

- Try Google Kubernetes Engine if you are not already familiar with it by creating a cluster (/kubernetes-engine/docs/how-to/creating-a-cluster).
- Learn more about Anthos Service Mesh (/anthos/service-mesh), Google's fully managed service mesh
- Follow these tutorials for different deployment options for Istio on GKE:
  - Building a multi-cluster service mesh on GKE with shared control-plane, single-VPC architecture (/solutions/building-multi-cluster-service-mesh-across-gke-clusters-using-istio-single-control-plane-architecture-single-vpc)
  - Building a multi-cluster service mesh on GKE using replicated control-plane architecture (/solutions/building-a-multi-cluster-service-mesh-on-gke-using-replicated-control-plane-architecture)
  - Building a GKE multi-cluster service mesh with Istio: Shared control plane across disparate networks (/solutions/building-gke-multi-cluster-service-mesh-with-istio-shared-control-plane-disparate-networks)
- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).

Last updated 2021-02-17 UTC.