Service Mesh & Istio

Tejas Parikh (t.parikh@northeastern.edu)

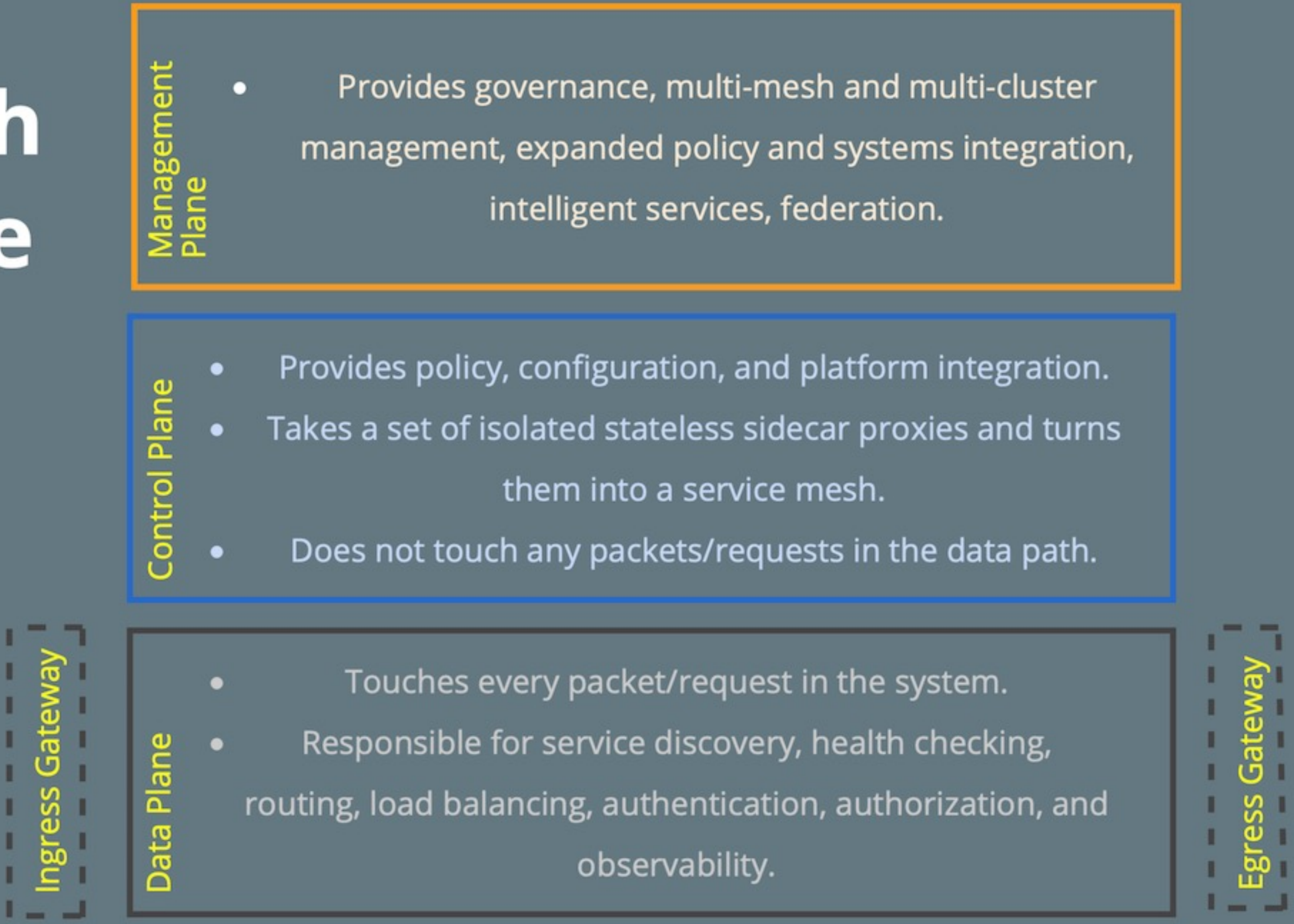
CSYE 7125 – Advanced Cloud Computing

Northeastern University

What is Service Mesh?

- A **service mesh** is the network of microservices that make up applications in a distributed microservice architecture and the interactions between those microservices.
- Service meshes provide policy-based, network services for network-connected workloads by enforcing desired behavior of the network in the face of constantly changing conditions and topology.
  - Conditions that change can be load, configuration, resources, and workloads being deployed.

Service Mesh Architecture



Management Plane

- Provides governance, multi-mesh and multi-cluster management, expanded policy and systems integration, intelligent services, federation.

Control Plane

- Provides policy, configuration, and platform integration.
- Takes a set of isolated stateless sidecar proxies and turns them into a service mesh.
- Does not touch any packets/requests in the data path.

Data Plane

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

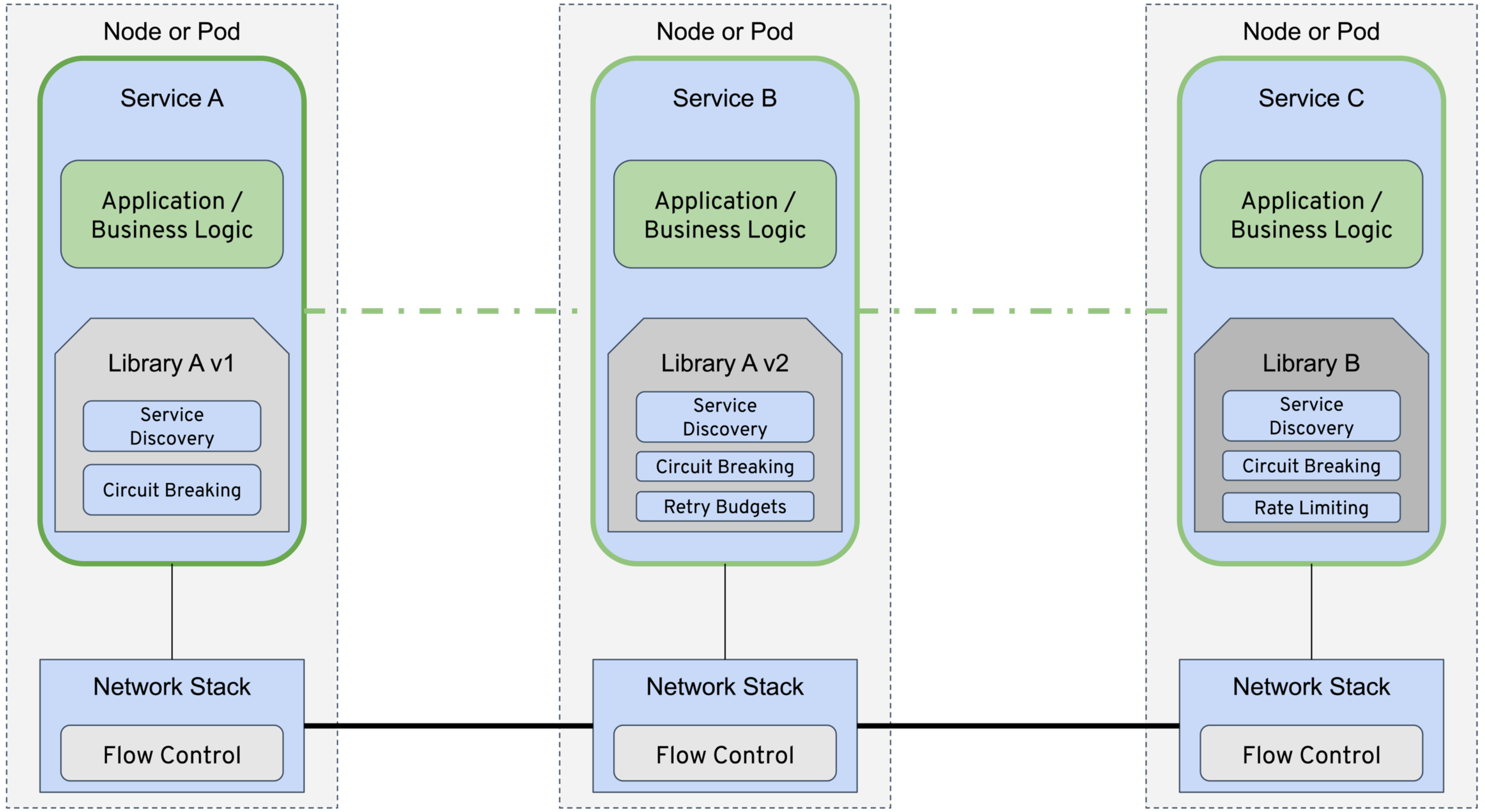
Management planes - a key component

@lcalcote

Service Mesh Fundamentals

- **Service meshes** are entirely software based.
- Service meshes are built using service proxies.
- Service proxies of the data plane carry traffic.
  - Traffic is transparently intercepted using iptable rules in the pod namespace.
- This uniform layer of infrastructure combined with service deployments is commonly referred to as a ***service mesh***.

Services architecture using client libraries coupled with application logic



Client Libraries: The First Service Meshes

- Complicated task of managing microservices, some organizations started using *client libraries* as frameworks to standardize implementations.
- The two **benefits** of creating client libraries are
  - Resources consumed are locally accounted for each and every service.
  - The developers are empowered to self-service their choice of an existing library or building a new language-specific library.
- The most significant **drawback** of client libraries is the tight coupling of infrastructure concerns with application code.
  - Client libraries' **nonuniform**, language-specific design makes their functionality and behavior inconsistent, which leads to poor observability characteristics, bespoke practices to augment services that are more or less controllable by one another, and possibly compromised security.

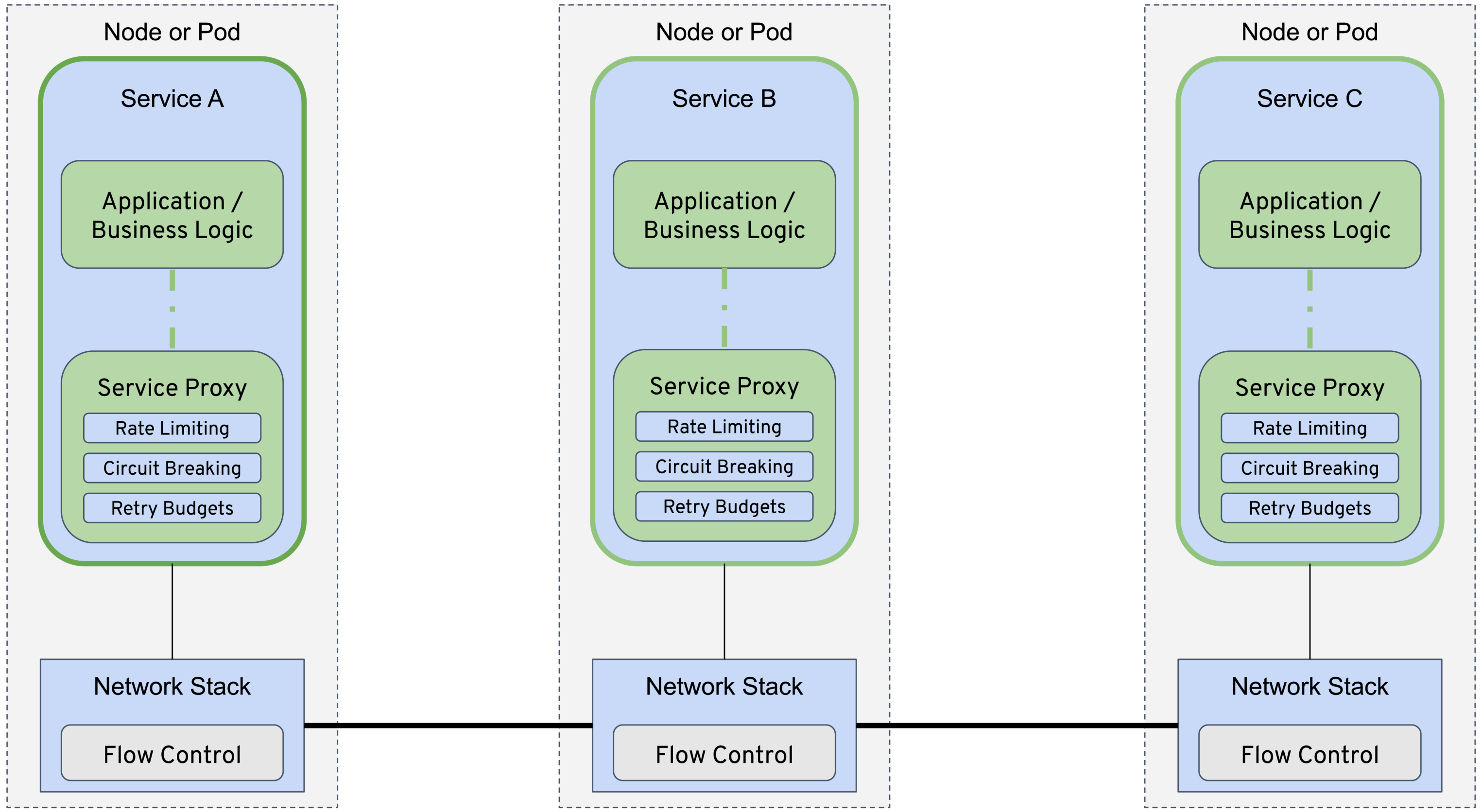
Client Libraries

- Service Discovery
  - Eureka - https://github.com/Netflix/eureka
- Distributed configuration
  - Archaius - https://github.com/Netflix/archaius
- Fault tolerance
  - Hystrix - https://github.com/Netflix/hystrix
- Inter-process and service communication
  - Ribbon - https://github.com/Netflix/ribbon
- Dynamic Proxy
  - Zuul - https://github.com/Netflix/zuul
- The Netflix OSS Stack (https://netflix.github.io/)

Client Libraries Drawbacks

- Networking is hard. Creating a client library that eliminates client contention by introducing jitter and an exponential back-off algorithm in the calculation of timing the next retry attempt isn’t necessarily easy, and neither is attempting to ensure the same behavior across different client libraries (with the varying languages and versions of those libraries).
- Coordinating upgrades of client libraries is difficult in large environments as upgrades require code changes, rolling a new release of the application and, potentially, application downtime.

Services architecture using service proxies decoupled from application logic



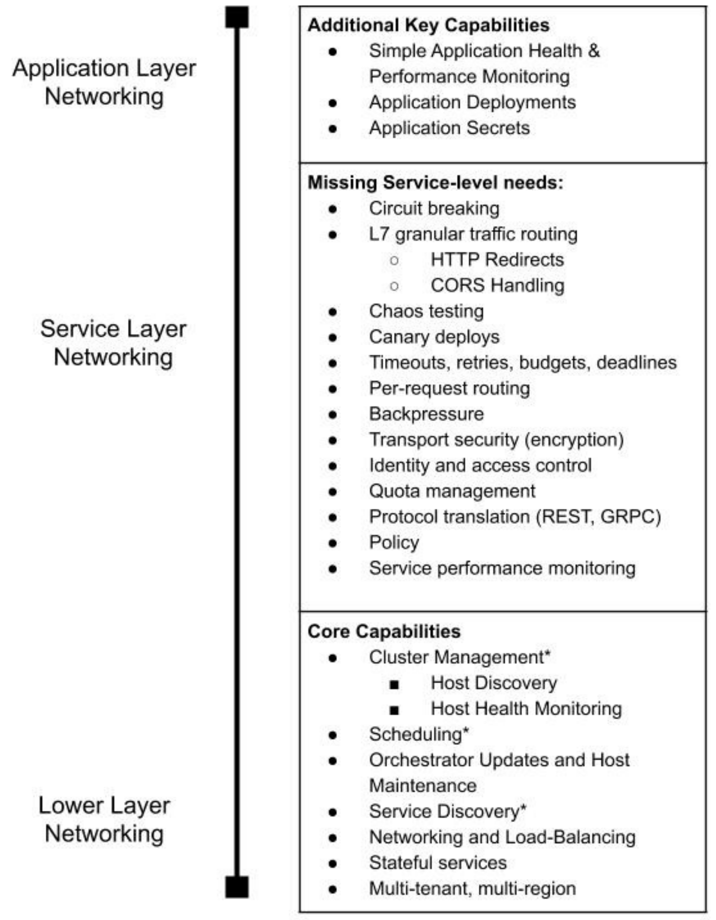
Service Proxy

- With a **service proxy next to each application instance**, applications no longer need to have language-specific resilience libraries for **circuit breaking, timeouts, retries, service discovery, load balancing**, and so on.
- Service meshes seem to deliver on the promise that organizations implementing microservices could finally realize the dream of using the best frameworks and language for their individual jobs without worrying about the availability of libraries and patterns for every single platform.

Why do we need a Service Mesh?

- A service mesh is a dedicated infrastructure layer for making service-to-service communication safe, fast, and reliable, at times relying on a container orchestrator or integration with another service discovery system.

Container Orchestration Platform vs. Service Mesh



Additional Key Capabilities
- Simple Application Health & Performance Monitoring
- Application Deployments
- Application Secrets

Missing Service-level needs:
- Circuit breaking
- L7 granular traffic routing
  - HTTP Redirects
  - CORS Handling
- Chaos testing
- Canary deploys
- Timeouts, retries, budgets, deadlines
- Per-request routing
- Backpressure
- Transport security (encryption)
- Identity and access control
- Quota management
- Protocol translation (REST, GRPC)
- Policy
- Service performance monitoring

Core Capabilities
- Cluster Management*
  - Host Discovery
  - Host Health Monitoring
- Scheduling*
- Orchestrator Updates and Host Maintenance
- Service Discovery*
- Networking and Load-Balancing
- Stateful services
- Multi-tenant, multi-region

The Value of a Service Mesh

- Service meshes provide a uniform way to connect, secure, manage, and monitor microservices.
- Observability
- Traffic Control
- Security

Observability

- Service meshes give you **visibility, resiliency, and traffic control**, as well as **security control** over distributed application services.
- Service meshes are transparently deployed and give visibility into and control over traffic without requiring any changes to application code.

Traffic Control

- Service meshes provide granular, declarative control over network traffic to determine, for example, where a request is routed to perform a canary release.
- Resiliency features typically include circuit-breaking, latency-aware load balancing, eventually consistent service discovery, retries, timeouts, and deadlines.

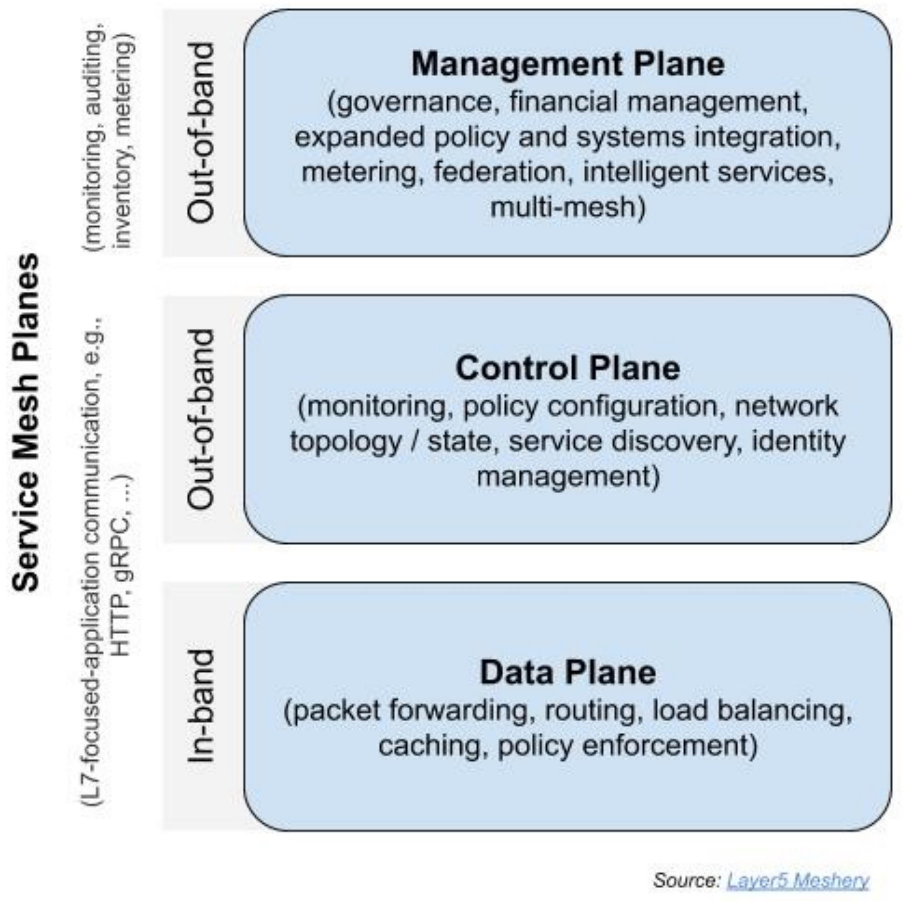
Security

- A service mesh is a powerful tool for enforcing security, policy, and compliance requirements across their enterprise.
- Most service meshes provide a certificate authority (CA) to manage keys and certificates for securing service-to-service communication.
- Assignment of verifiable identity to each service in the mesh is key in determining which clients are allowed to make requests of different services as well as in encrypting that request traffic.
- Certificates are generated per service and provide a unique identity for that service.

The Istio Service Mesh

- **Istio** (in the Greek alphabet, ιστίο) is Greek for "sail," and is pronounced "**iss-teeh-oh**."
- **Istio's** accompanying **command-line interface** (CLI), **istioctl**, is pronounced "**iss-teeh-oh-c-t-l**".
- Istio is an open source implementation of a service mesh first created by Google, IBM, and Lyft.
- Istio is composed of a number of Cloud Native Computing Foundation (CNCF)–governed projects like Prometheus, OpenTelemetry, Fluentd, Envoy, Jaeger, Kiali, and many contributor-written adapters.

Service Mesh Architecture


Service Mesh Planes
(monitoring, auditing, inventory, metering)
(L7-focused-application communication, e.g., HTTP, gRPC, ...)
Out-of-band
Management Plane
(governance, financial management, expanded policy and systems integration, metering, federation, intelligent services, multi-mesh)
Out-of-band
Control Plane
(monitoring, policy configuration, network topology / state, service discovery, identity management)
In-band
Data Plane
(packet forwarding, routing, load balancing, caching, policy enforcement)
Source: Layer5 Meshery

- At a high level, service mesh architectures including Istio commonly comprise two planes: a **control plane** and **data plane**, while a third **management plane** might reside in incumbent/infrastructure systems.

Istio’s Data Plane

- Istio’s **data plane intercepts every packet** in the request and is responsible for health checking, routing, load balancing, authentication, authorization, and generation of observable signals.
- **Operating in band**, **service proxies are transparently inserted**, and as applications make service-to-service calls**, applications are unaware of the data plane’s existence**.
- **Data planes are responsible** for **intra cluster communication** as well as **inbound** (*ingress*) and **outbound** (*egress*) cluster network traffic.
- Whether **traffic** is **entering** or **leaving** the mesh, application service **traffic** is **directed first to the service proxy** for handling.
    - With Istio, traffic is transparently **intercepted** using **iptables rules** and redirected to the service proxy.
    - The Istio data plane touches every packet/request in the system and is responsible for service discovery, health checking, routing, load balancing, authentication, authorization, and observability.

Istio's Control Plane

- Istio control plane provides policy and configuration for services in the mesh via APIs for operators to specify desired routing/resilience behavior
- Istio control plane combines a set of isolated stateless sidecar proxies into a service mesh:
  - APIs for the data plane to consume localized configuration
  - Service discovery abstraction for the data plane
- Istio control plane uses APIs for specifying usage policies via quota and usage restrictions
- Istio control plane provides security via certificate issuance and rotation
- Istio control plane assigns workload identity
- Istio control plane handles routing configuration:
  - Doesn't touch any packets/requests in the system
  - Specifies network boundaries and how to access them
  - Unifies telemetry collection

Istio Control-Plane Components

- Pilot
- Galley
- Mixer
- Citadel

Pilot

- Pilot is the head of the ship in an Istio mesh, so to speak.
- Pilot stays synchronized with the underlying platform (e.g., Kubernetes) by tracking and representing the state and location of running services to the data plane.
- Pilot interfaces with your environment’s service discovery system, and produces configuration for the data-plane service proxies.

Galley

- Galley is Istio’s configuration aggregation and distribution component.
- Galley uses the Mesh Configuration Protocol (MCP) as a mechanism to serve and distribute configuration.

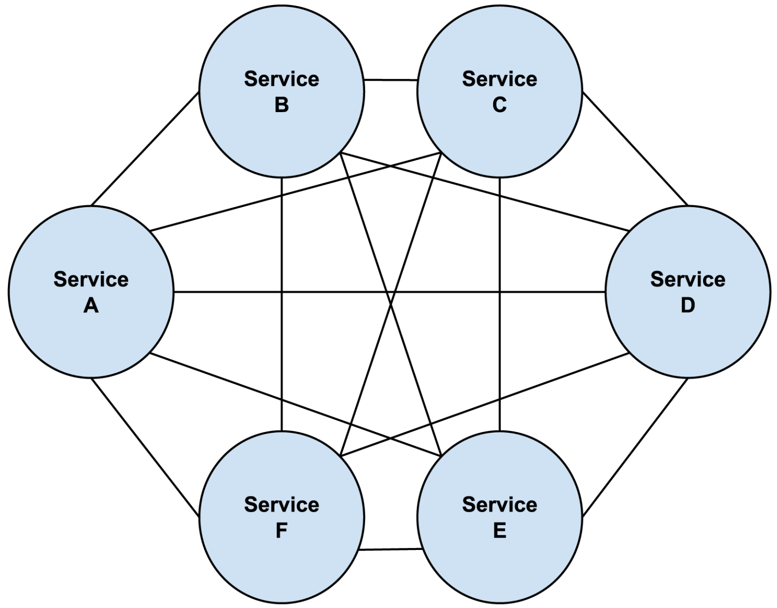
Mixer

- Mixer is a control-plane component designed to abstract infrastructure backends from the rest of Istio, where infrastructure backends are things like Stackdriver or New Relic.
- Mixer bears responsibility for precondition checking, quota management, and telemetry reporting. It does the following:
  - Enables platform and environment mobility
  - Provides granular control over operational policies and telemetry by taking responsibility for policy evaluation and telemetry reporting
  - Has a rich configuration model
  - Abstracts away most infrastructure concerns with intent-based configuration
- Service proxies and gateways invoke Mixer to do precondition checks to determine whether a request should be allowed to proceed (check), whether communication between the caller and the service is allowed or has exceeded quota, and to report telemetry after a request has completed (report).

Citadel

- Citadel empowers Istio to provide strong service-to-service and end-user authentication using **mutual Transport Layer Security (mTLS),** with built-in identity and credential management.
- Citadel's CA component approves and signs certificate-signing requests (CSRs) sent by Citadel agents, and it performs key and certificate generation, deployment, rotation, and revocation.
- Citadel has an optional ability to interact with an identity directory during the CA process.
- Citadel has a pluggable architecture in which different CAs can be used so that it's not using its self-generated, self-signed signing key and certificate to sign workload certificates.
- The CA pluggability of Istio enables and facilitates the following:
    - Integrates with your organization's public key infrastructure (PKI) system.
    - Secures communication between Istio and non-Istio legacy services (by sharing the same root of trust).
    - Secures the CA signing key by storing it in a well-protected environment (e.g., HashiCorp Vault, hardware security module, or HSM)

Fully interconnected service proxies form the mesh



Istio Data-Plane Components

- Envoy Proxy

Envoy Proxy

- Istio uses an extended version of the **Envoy proxy**, a high-performance proxy developed in C++, **to mediate all inbound and outbound traffic for all services in the service mesh**.
- Istio uses Envoy's features such as dynamic service discovery, load balancing, TLS termination, HTTP/2 and gRPC Remote Procedure Call (gRPC) proxying, circuit breakers, health checks, staged rollouts with percent-based traffic split, fault injection, and rich metrics.
- **Envoy is deployed as a sidecar to the relevant service in the same Kubernetes pod.** This allows Istio to extract a wealth of signals about traffic behavior as attributes, which in turn it can use in Mixer to enforce policy decisions and send to monitoring systems to provide information about the behavior of the entire mesh.

Injection

- The **sidecar proxy model** also **allows** you to **add Istio** capabilities to an existing deployment with **no need to redesign or rewrite code**. This is a significant attraction to using Istio.
- The promises of an immediate view of top-level service metrics, of detailed control over traffic, and of automated authentication and encryption between all services without having to change your application code *or* change your deployment manifests.
- In Kubernetes, automatic proxy injection is implemented as a webhook using a Kubernetes API Server with the mutating webhook admission controller.

Gateways

- Ingress
- Egress

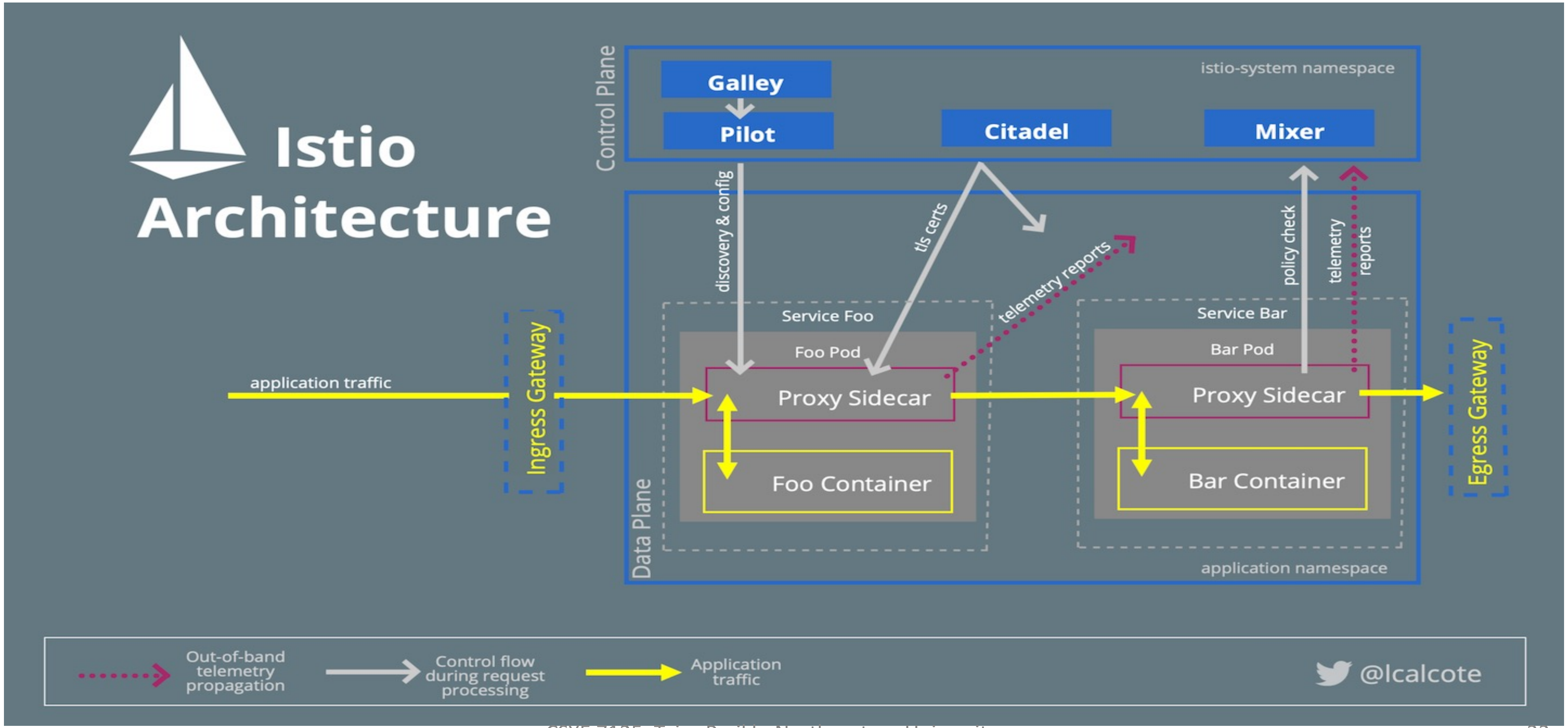
Ingress Gateways

- Configuring ingress gateways enables you to define traffic entryways into the service mesh for incoming traffic to flow through.
- Consider that ingressing traffic into the mesh is a reverse proxy situation—akin to traditional web server load balancing.

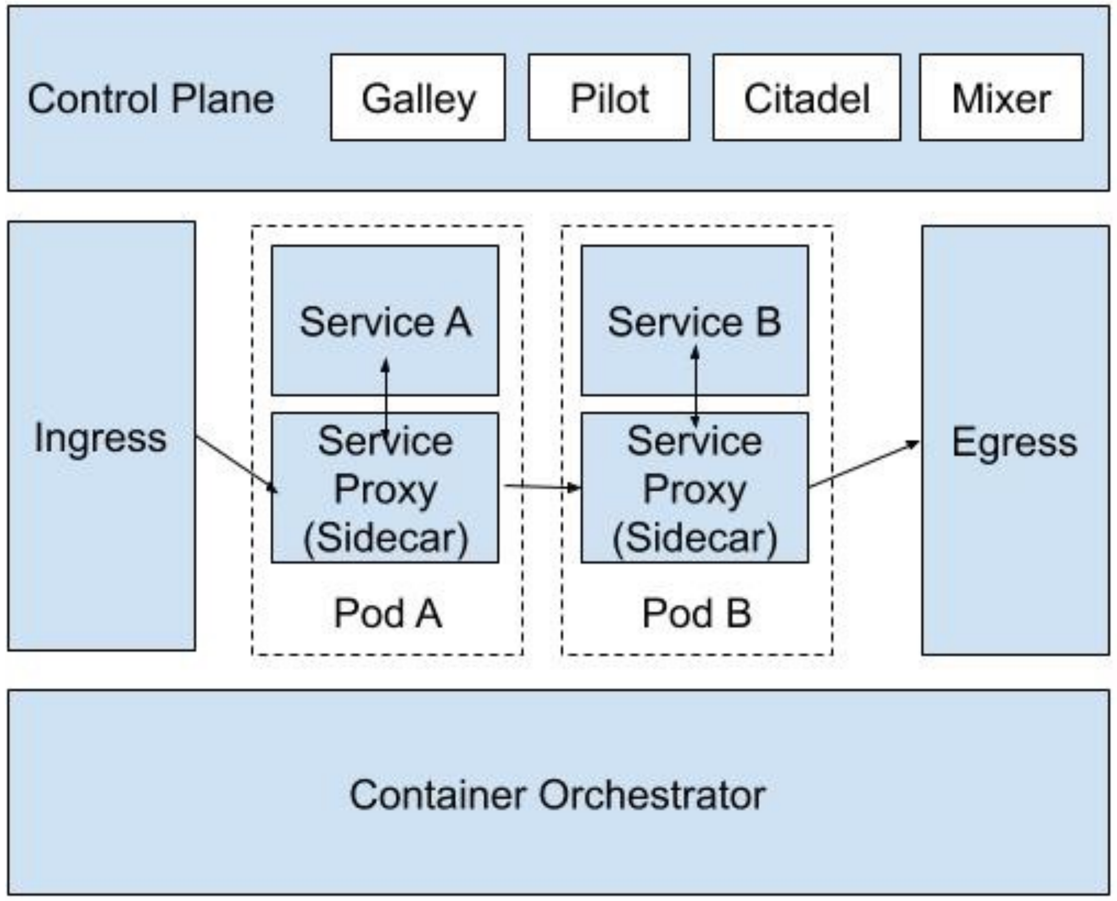
Egress Gateways

- Configuration for egressing traffic out of the mesh is a forward proxy situation in which you identify which traffic to allow out of the mesh and where it should be routed.
- Traffic can exit an Istio service mesh in two ways: directly from the sidecar or funneled through an egress gateway, where you can apply traffic policy.
- **By default, Istio-enabled applications are unable to access URLs external to the cluster.**

Istio's architecture and components

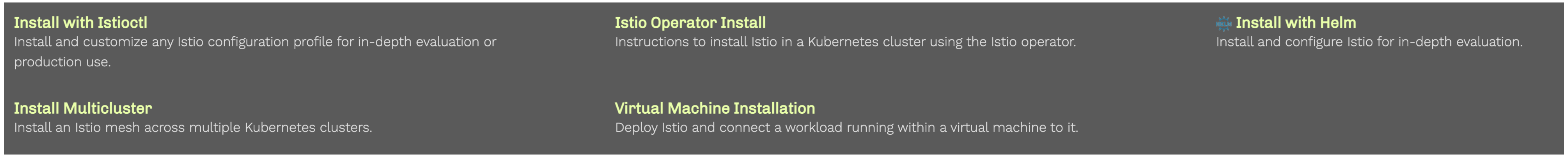


Istio Deployment on Kubernetes



Deploying Istio

Installation Guides



Install with Istioctl
Install and customize any Istio configuration profile for in-depth evaluation or production use.

Istio Operator Install
Instructions to install Istio in a Kubernetes cluster using the Istio operator.

Install with Helm
Install and configure Istio for in-depth evaluation.

Install Multicluster
Install an Istio mesh across multiple Kubernetes clusters.

Virtual Machine Installation
Deploy Istio and connect a workload running within a virtual machine to it.

Install with Istioctl

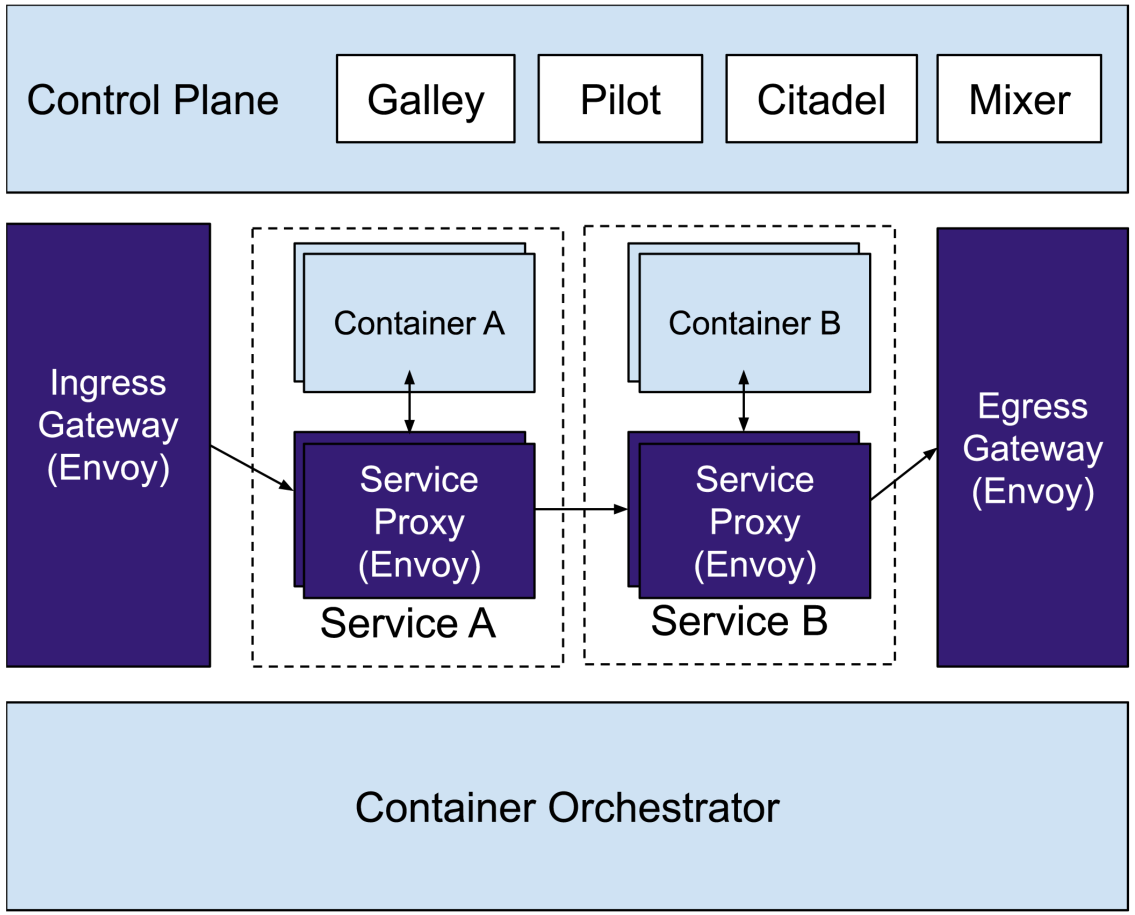
- istioctl install or istioctl install --set profile=demo
- kubectl -n istio-system get deploy
- Ref: https://istio.io/latest/docs/setup/install/istioctl/

Service Proxy

Service Proxy

- **Forward proxies** focusing on outbound traffic with the aim of improving performance and filtering requests are typically deployed as the interface between users on private networks and their internet requests.
  - Forward proxies commonly improve performance because they can cache static web content and provide a level of security by preventing users from accessing specific categories of websites.
- **Reverse proxies** focus on inbound traffic coming from the internet to private networks.
  - They are commonly used to secure and filter HTTP requests, providing load balancing across real (backend) servers.
  - To the extent that forward proxies typically represent user traffic to external servers, reverse proxies are commonly used to represent real servers to users (clients).
- A **service proxy** is the client-side intermediary transiting requests on behalf of a service.

Envoy as the Istio service proxy in Istio deployments



Sidecar Injection

- **Adding a service proxy** consists of two things: **sidecar injection** and **network capture**.
- **Sidecar injection**—or “sidecarring”—is the method of adding a proxy to a given application.
- **Network capture** is the method of directing inbound traffic to the proxy (instead of the application) and outbound traffic to the proxy (instead of directly back to the client or directly to subsequent upstream application services).

Istio Init Container

- In Istio’s case, init containers are used to set up network filters—iptables—to control the flow of traffic.

mTLS, Authentication, Authorization

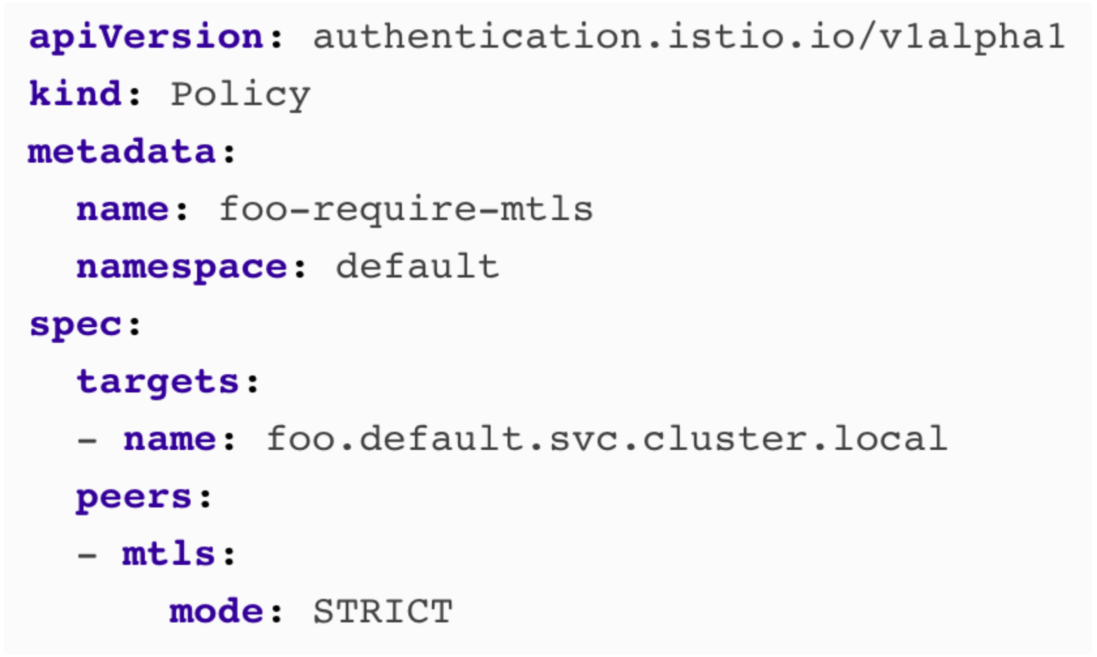
mTLS

- mTLS is TLS in which both parties, client and server, present certificates to each other.
- mTLS allows the client to verify the identity of the server, like normal TLS, but it also allows the server to verify the identity of the client attempting to establish the connection.

Authentication Policy: Configuring mTLS

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

Example



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

- To enable mTLS for a single service, we create a policy in that service's namespace with that service as the target, requiring mTLS.
- This policy applies to the default namespace and marks TLS as required for talking to service foo.

Authorization Policy: Configuring Who Can Talk to Whom

- With authentication policy in place, we want to use the identities across the system to control which services can communicate.
- In other words, we want to describe a service-to-service communication policy.
- Istio's authorization policy is described using an RBAC system. Like most RBAC systems, it defines two objects that are used together to write policy:
  - **ServiceRole** - Describes a set of actions that can be performed on a set of services by any principal with the role.
  - **ServiceRoleBinding -** Assigns roles to a set of principals. In this context, the principals are the service identities Istio issues.

ServiceRole

```yaml
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: bar-viewer
  namespace: default
spec:
  rules:
  - services:
    - bar.default.cluster.local
    methods:
    - GET
```

- Example of a ServiceRole that allows read access (HTTP GET requests) to the bar service.

ServiceRoleBinding

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

Pilot

Pilot

- Pilot is responsible for programming the data plane, ingress and egress gateways, and service proxies in an Istio deployment.
- Pilot has three main sources of configuration:
  - **Mesh configuration** - A set of configurations global to the service mesh
  - **Networking configuration** - The configuration for ServiceEntries, DestinationRules, VirtualServices VirtualServices, Gateways, and service proxies
  - **Service discovery** - The location and metadata information from registries about the catalog of services resident in one or more underlying platforms

Gateways & VirtualServices

- **Gateways** and **VirtualServices** results in Listeners for Envoy.
- **Gateways** result in physical listeners (listeners that bind to a port on the network), whereas **VirtualServices** result in virtual listeners (listeners that do *not* bind to a port, but instead receive traffic from physical listeners).

A VirtualService that binds to the Gateway

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
 name: foo-default
spec:
 hosts:
 - bar.foo.com
 gateways:
 - foo-com-gateway
 http:
 - route:
   - destination:
       host: bar.foo.svc.cluster.local
```

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

Path Routing with VirtualService

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
 name: foo-default
spec:
  hosts:
  - bar.foo.com
  gateways:
  - foo-com-gateway
  http:
  - match:
    - uri:
        prefix: /whiz
    route:
    - destination:
        host: whiz.foo.svc.cluster.local
  - route:
    - destination:
        host: bar.foo.svc.cluster.local
```

Traffic Management

Understanding How Traffic Flows in Istio

- Envoy intercepts all incoming and outgoing requests at runtime (as traffic flows through the service mesh).
- Interception is done transparently via iptables rules or a Berkeley Packet Filter (BPF) program that routes all network traffic, in and out through Envoy.
- Envoy inspects the request and uses the request's hostname, SNI, or service virtual IP address to determine the request's target.
- Envoy applies that target's routing rules to determine the request's destination.
- Envoy applies the destination's rules.
  - Destination rules include load-balancing strategy, which is used to pick an endpoint.
  - Services generally have more than one worker available to process requests. Requests can be balanced across those workers.
- Finally, Envoy forwards the intercepted request to the endpoint.

Additional Resources

See Lecture Page