Istio - Service mesh

Georgios Andrianakis

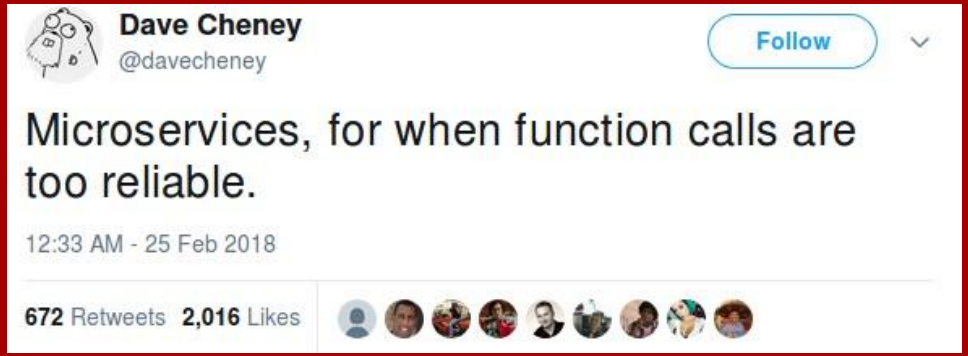
Senior Software Engineer - Red Hat

We of course need to start with...

Microservices

Microservices are Distributed Systems

They introduce non-trivial complexity into the space between our microservices - the ***network***



Microservices Architecture Challenges

- Resilience and fault tolerance
  - Retries
  - Timeouts
  - Circuit Breaker
- Discover, load balance services
- Logging
- Metrics
- Distribute traces

...challenges continued

- Rate Limiting
- Canary Deployments
- A/B Testing
- Fault injection

How do Microservices Architectures deal with these challenges?

The *library* approach

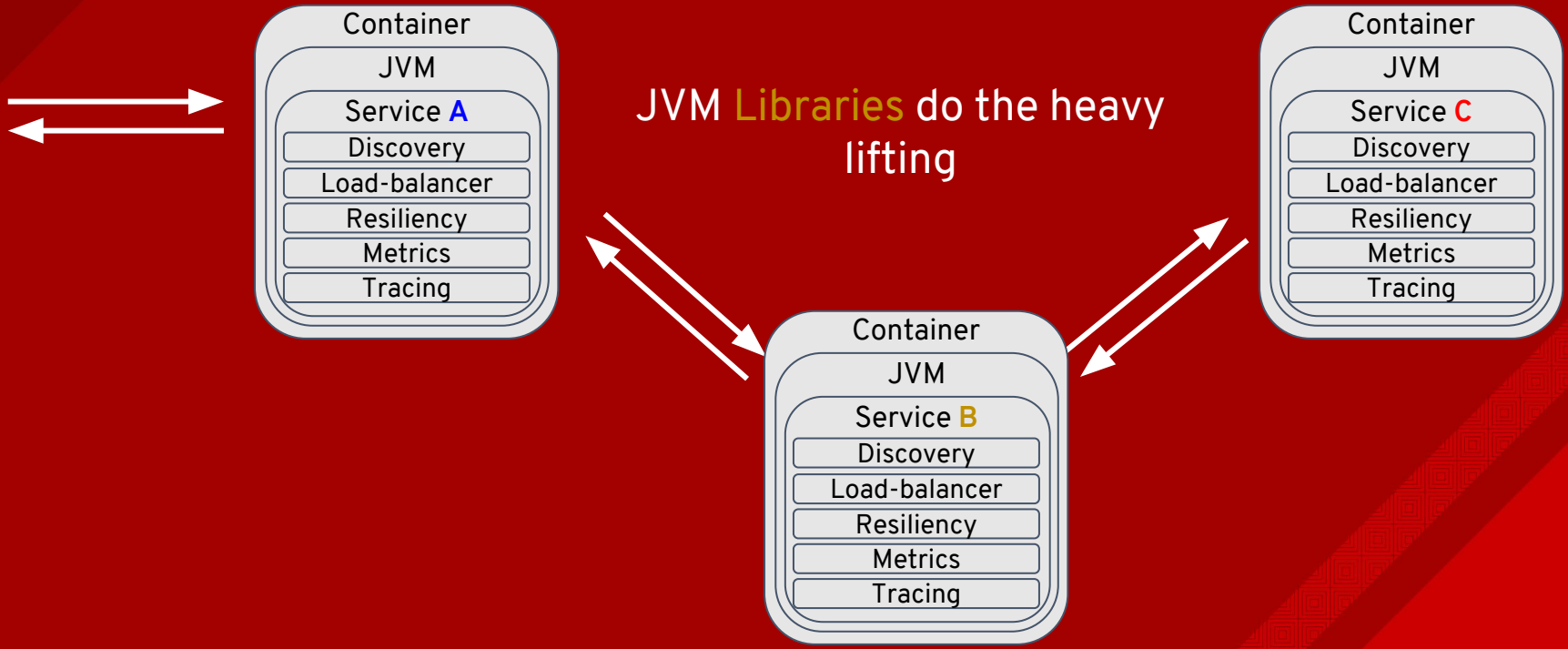
NETFLIX | OSS

- Netflix Hystrix (circuit breaking / bulk heading)
- Netflix Zuul (edge router)
- Netflix Ribbon (client-side service discovery / load balance)
- Netflix Eureka (service discovery registry)
- Brave / Zipkin (tracing)
- Netflix spectator / atlas (metrics)

But I’m using Spring!!!

- spring-cloud-netflix-hystrix
- spring-cloud-netflix-zuul
- spring-cloud-netflix-eureka-client
- spring-cloud-netflix-ribbon
- spring-cloud-netflix-atlas
- …..
- .....
- @Enable....150 different Things

Microservices embedding Capabilities



Drawbacks to the library approach

- Need an implementation for each combination of runtime/framework
  - Might end up forcing specific languages / frameworks on teams
- Need to maintain, upgrade, retire
- Classpath / namespace pollution
  - We end up with large deployables even for very small microservices

The platform approach

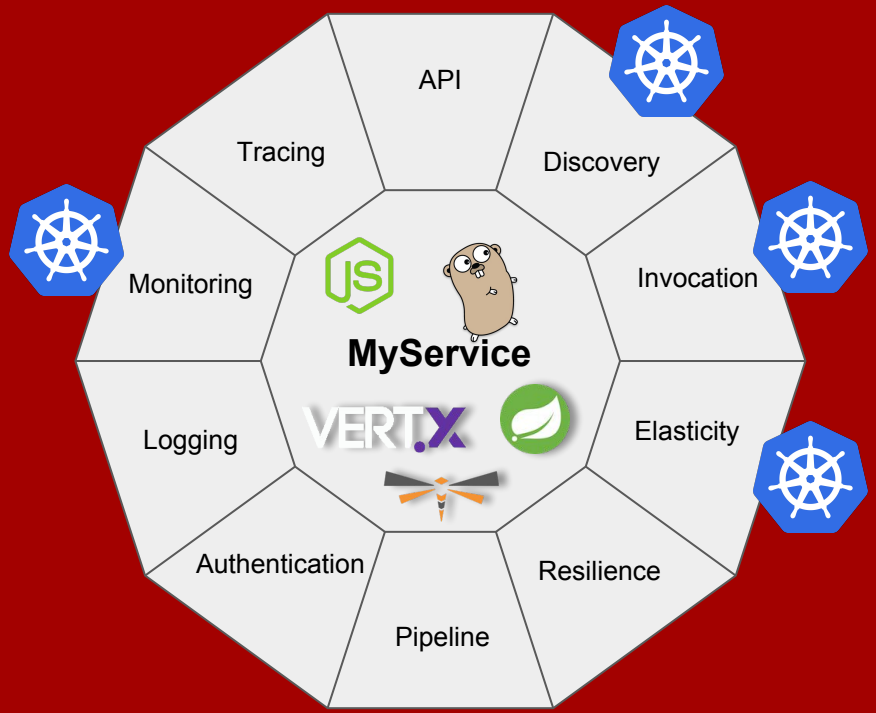
Move horizontal concerns into the platform and apply to all services regardless of implementation.

Benefits of the platform approach

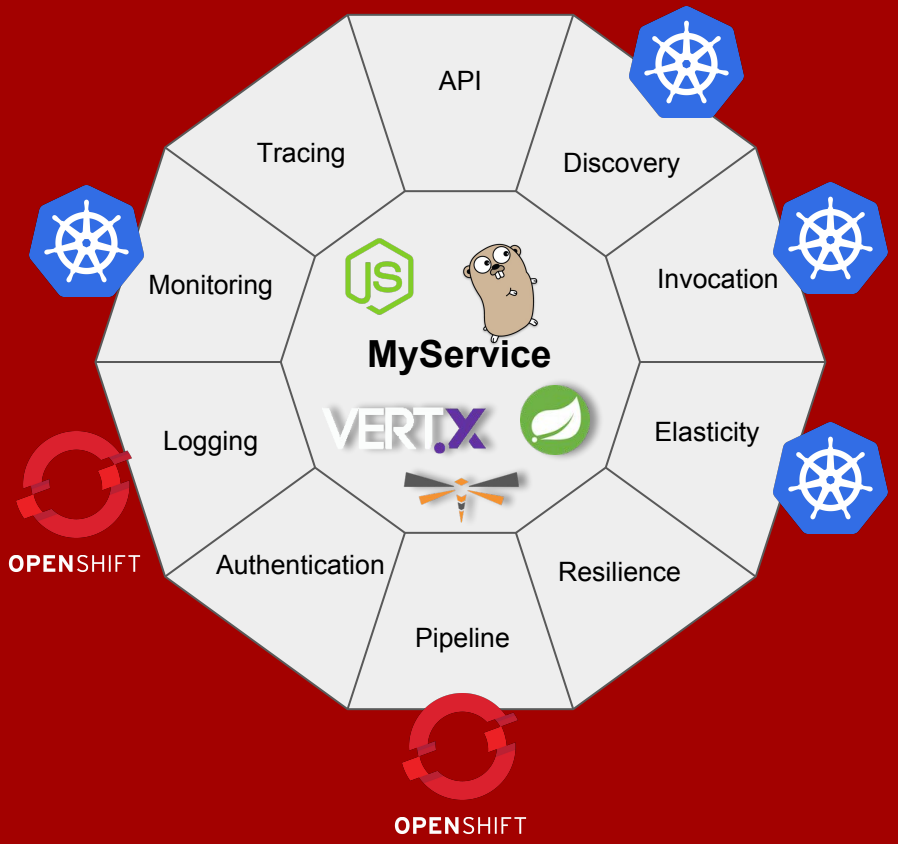
- Allow heterogeneous architectures
- Consistently and correctly enforce approaches to microservices challenges

What do the various platforms look like?

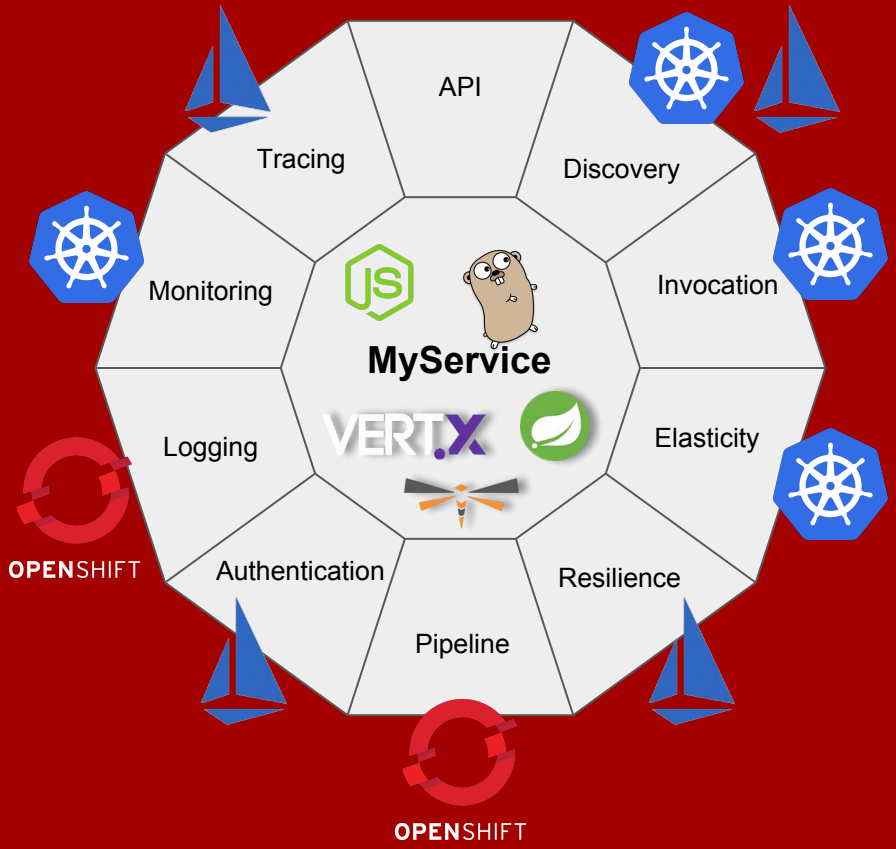
Microservices + Kubernetes



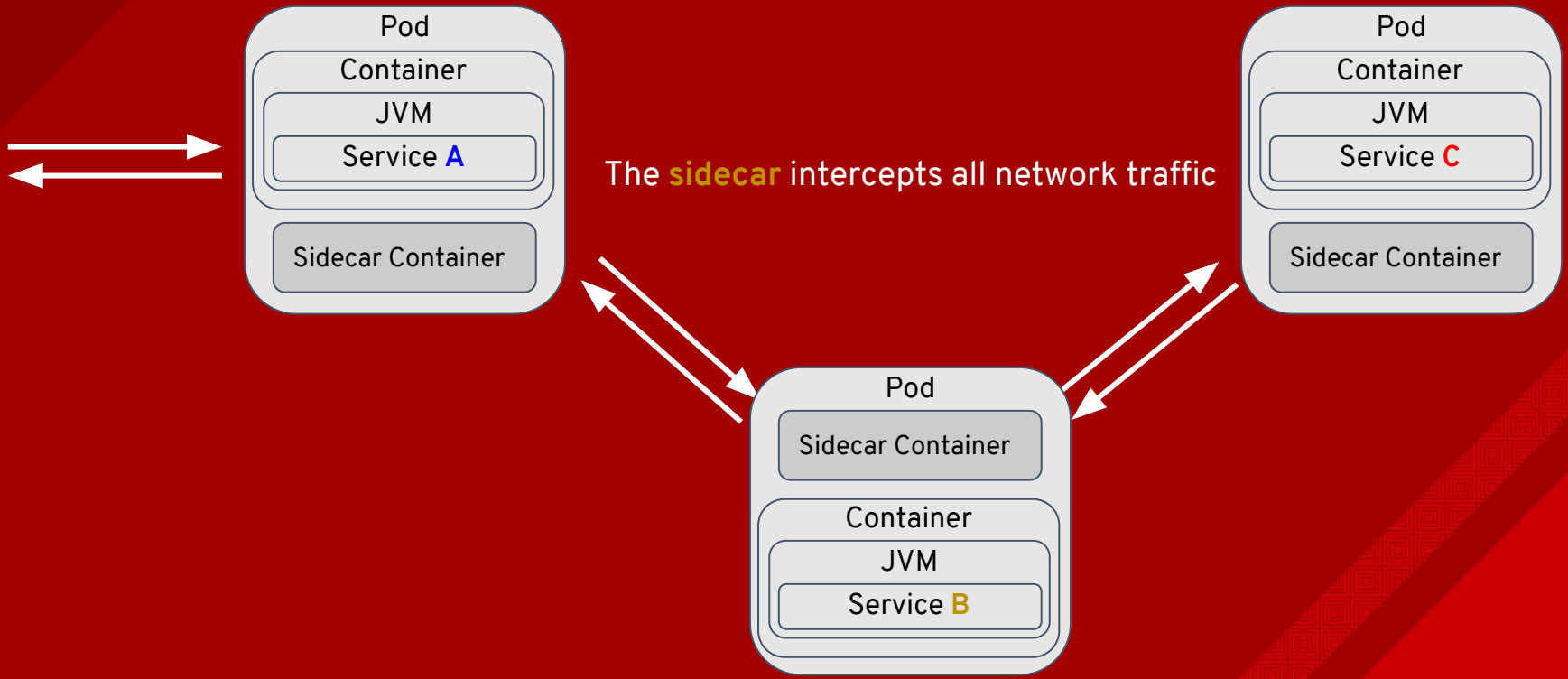
Microservices + OpenShift



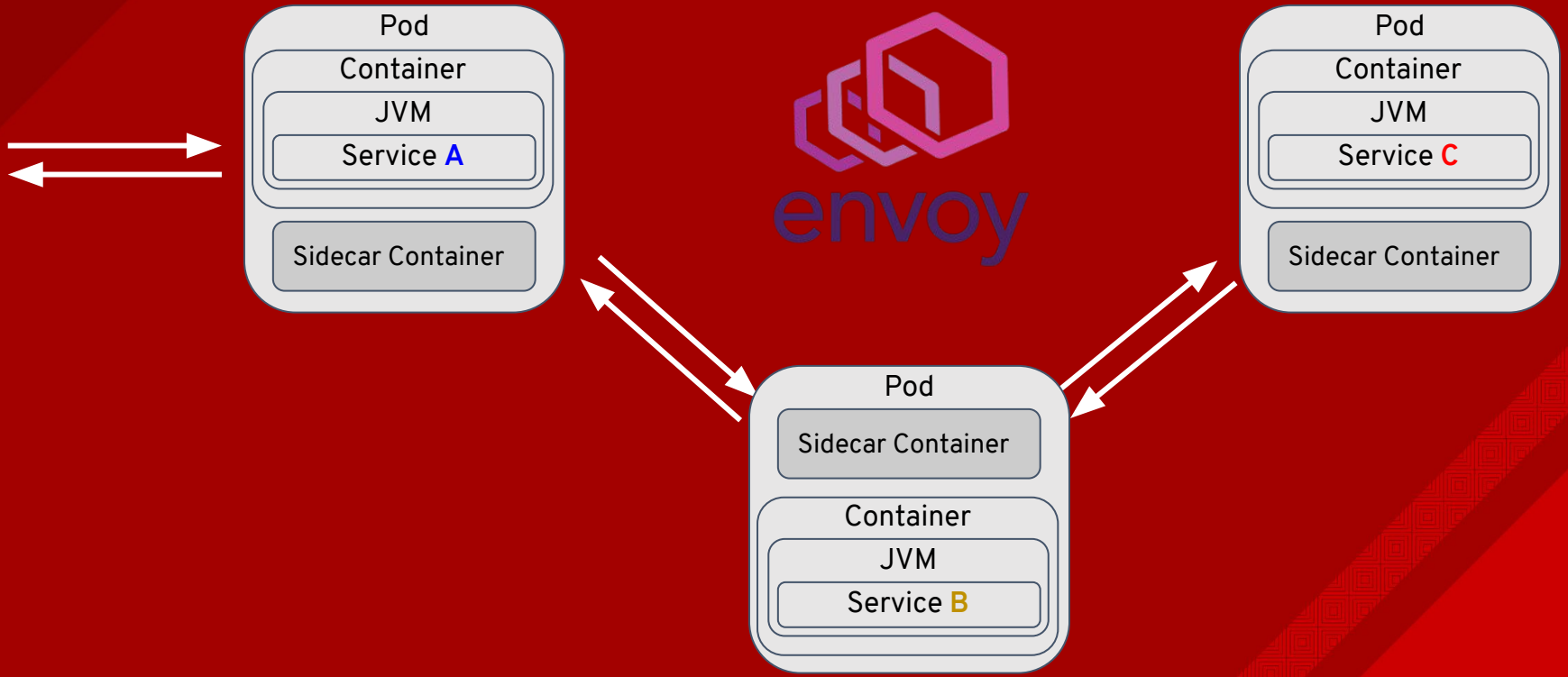
Microservices + Istio



Microservices embedding Capabilities



Envoy is the sidecar in Istio



Meet Envoy Proxy

http://envoyproxy.io

Envoy …

- Is written in C++, highly parallel, non-blocking
- L3/4 network filter
- out of the box L7 filters
- HTTP 2, including gRPC
- baked in service discovery / health checking
- advanced load balancing
- stats, metrics, tracing

How do we reason about a fleet of these service proxies in a large cluster?

Time for definitions

A **service mesh** is *decentralized* application-networking infrastructure *between your services* that provides *resilience*, *security*, *observability*, and routing *control*.

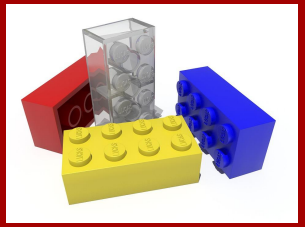
A service mesh is comprised of a *data plane* and a *control plane*.

Meet Istio.io

http://istio.io

A *control plane* for service proxies

Components

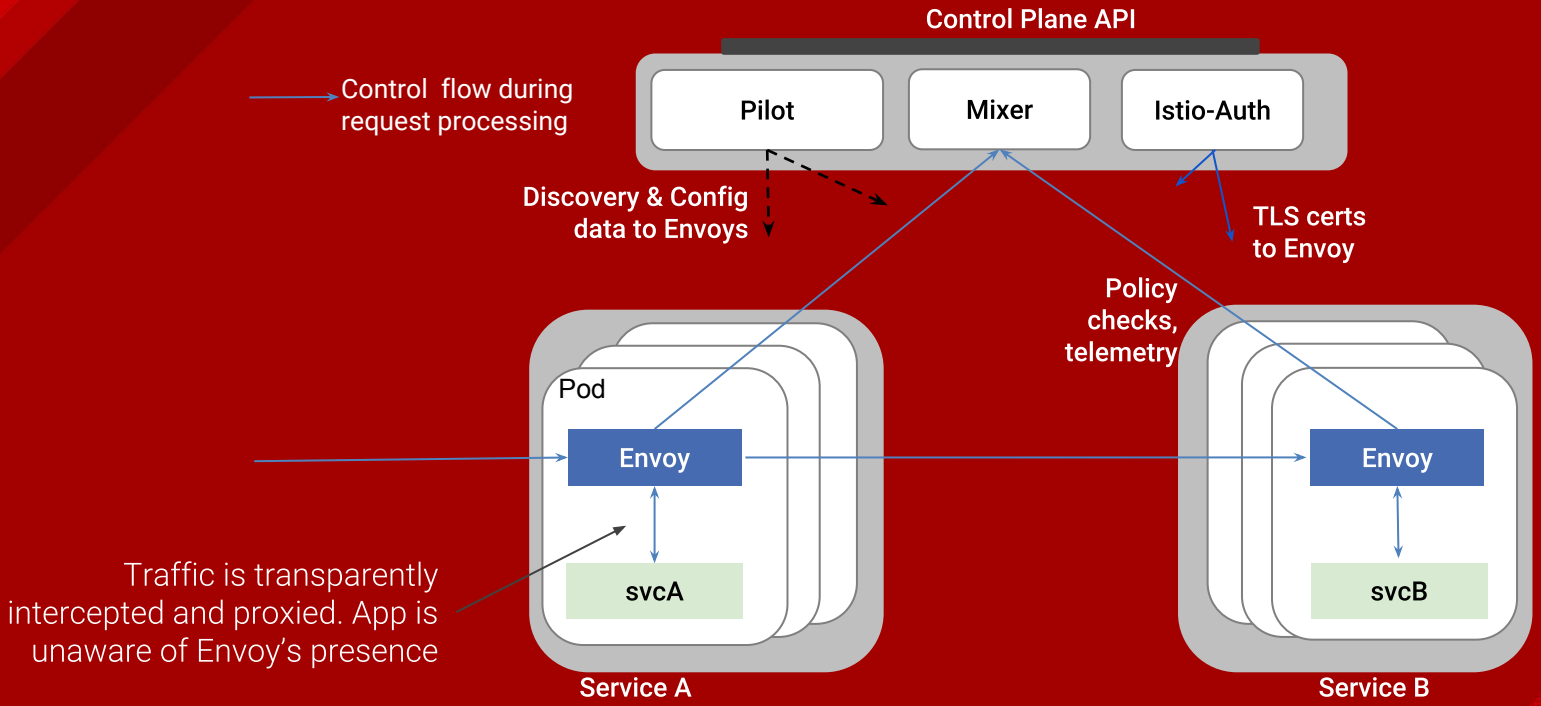


1. **Envoy:** *proxy*, to *mediate* all inbound and outbound traffic for all services in the service mesh

2. **Pilot**: *Push configuration to envoy fleet,* responsible for service discovery, registration and load balancing

3. **Istio-Auth** provides strong *service-to-service* and end-user authentication using *mutual TLS*

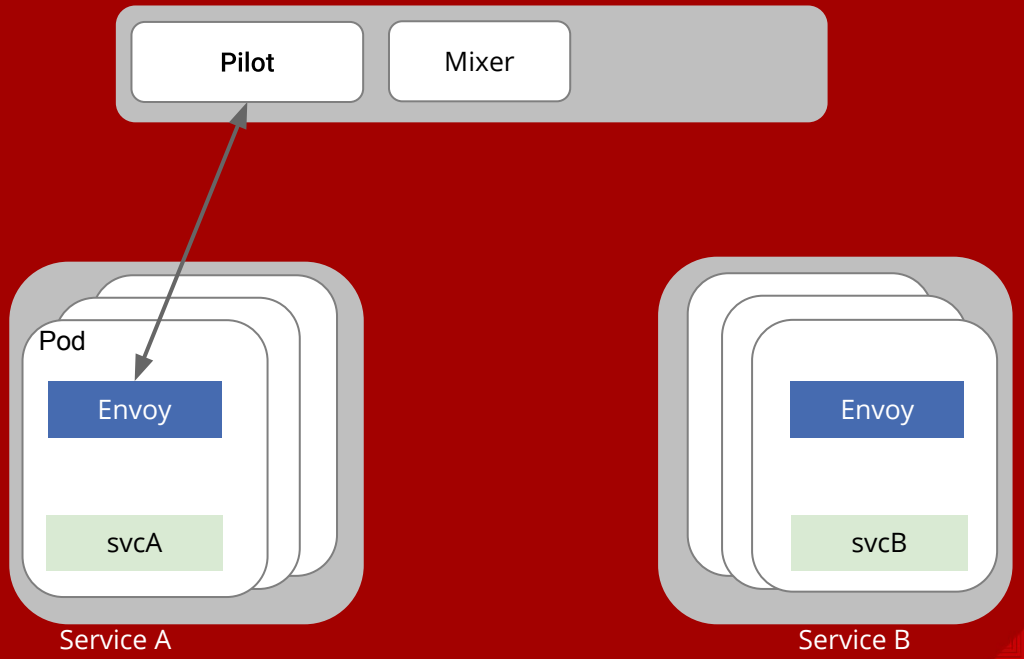
4. **Mixer** is responsible for enforcing *access control* across the service mesh & collecting *telemetry* data from the Envoy proxy and other services


Control Plane API
Control flow during request processing
Pilot
Mixer
Istio-Auth
Discovery & Config data to Envoys
TLS certs to Envoy
Policy checks, telemetry
Pod
Envoy
svcA
Envoy
svcB
Traffic is transparently intercepted and proxied. App is unaware of Envoy's presence
Service A
Service B

How it works

Life of a request in the mesh

Service A comes up. Envoy is deployed alongside it and fetches service information, routing and configuration policy from Pilot.

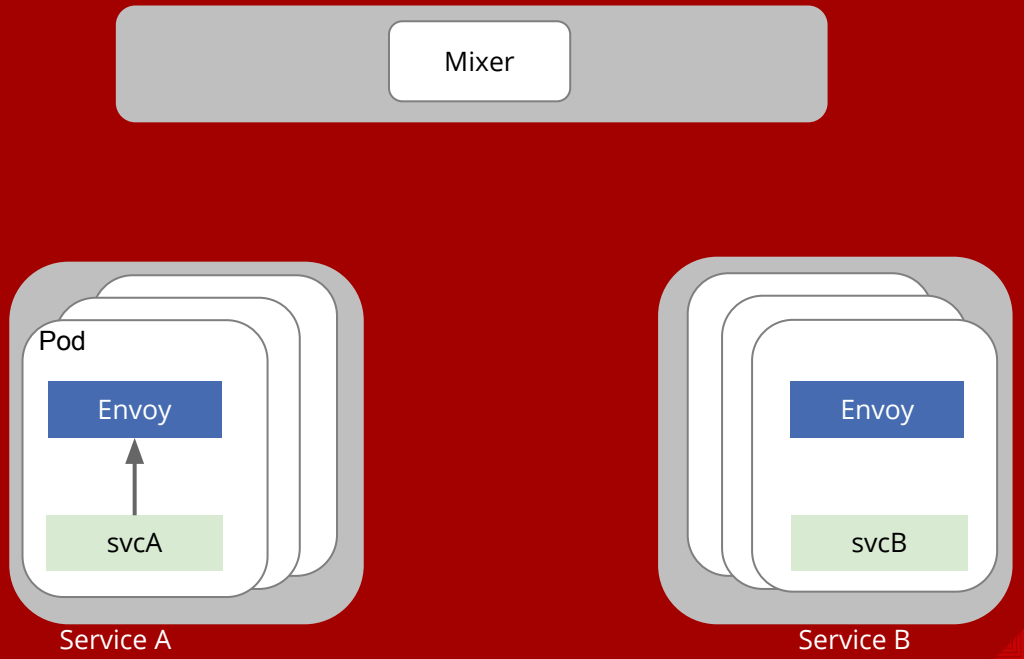


Life of a request in the mesh

Service A makes a call to service B

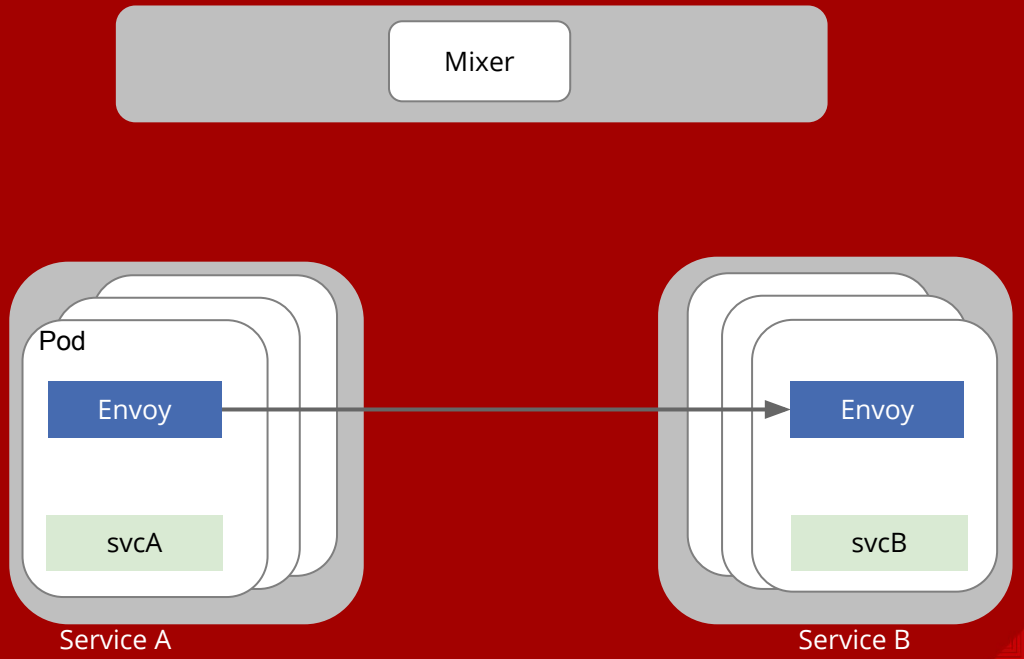
Client-side Envoy intercepts the call.

Envoy consults config from Pilot to know how/where to route call to service B



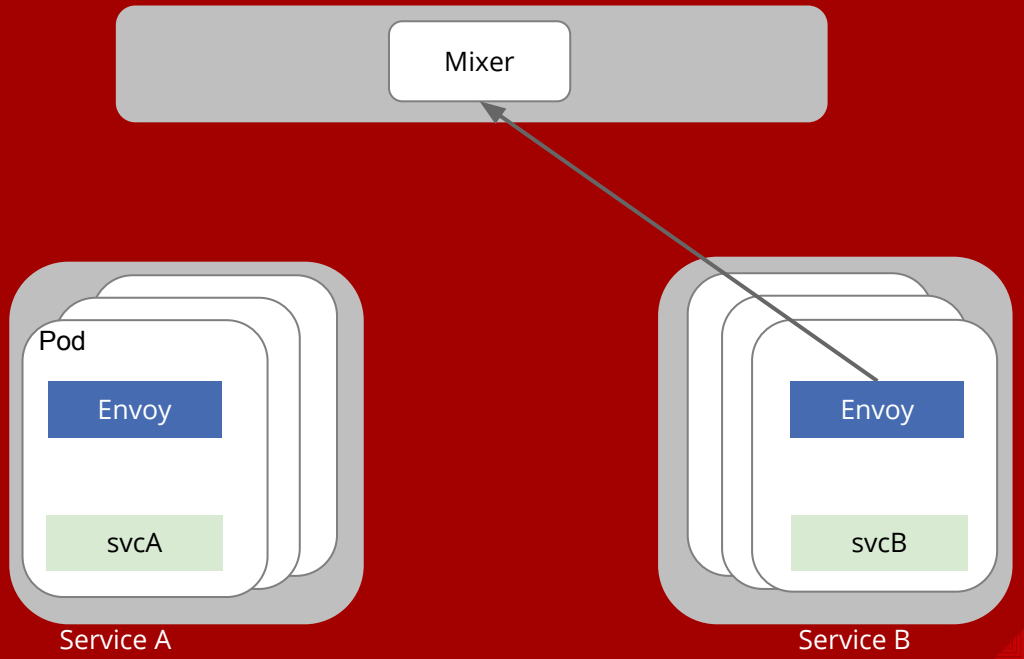
Life of a request in the mesh

Envoy forwards request to appropriate instance of service B



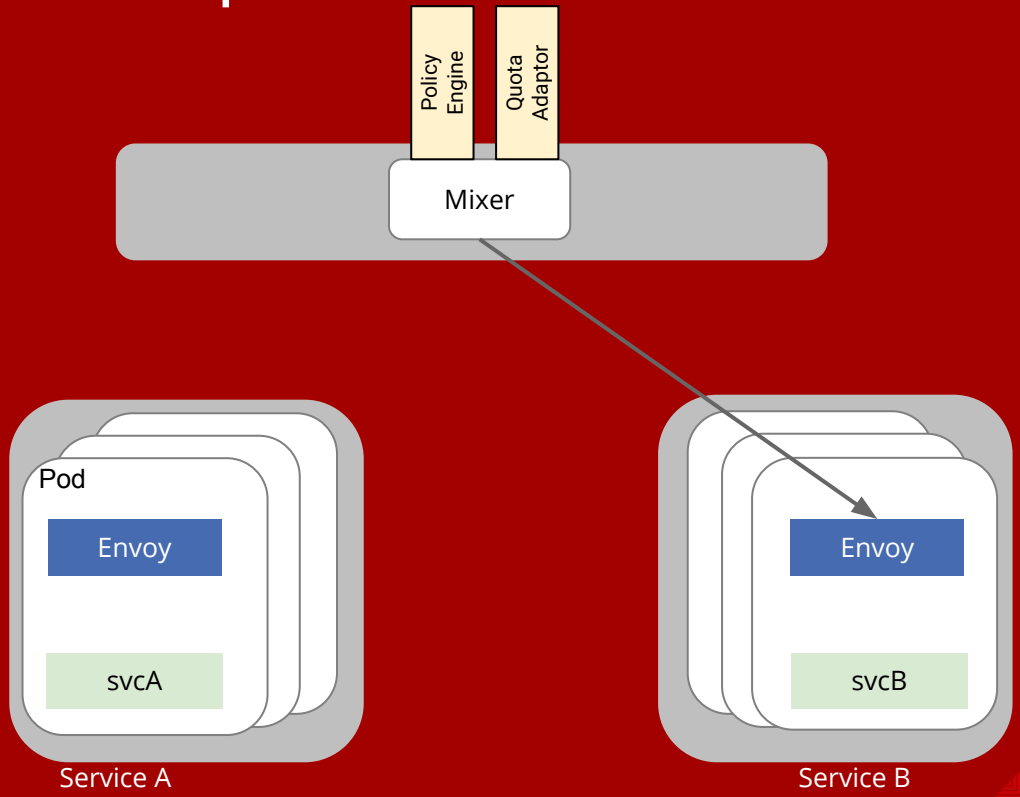
Life of a request in the mesh

Server-side Envoy checks with Mixer to validate that call should be allowed (ACL check, quota check, etc).



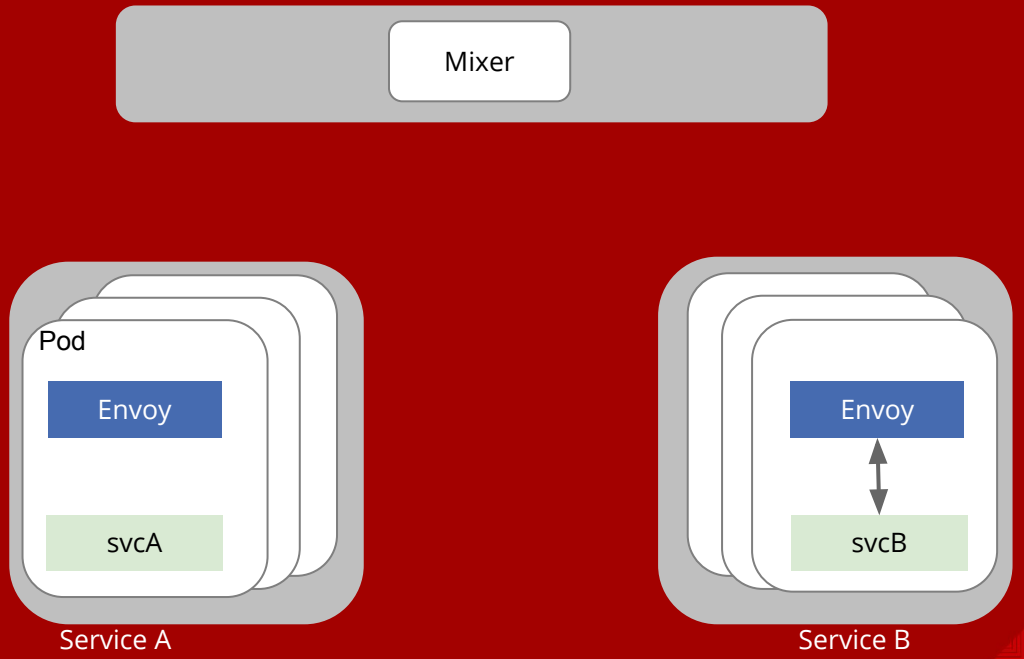
Life of a request in the mesh

Mixer checks with appropriate adaptors (policy engine, quota adaptor) to verify that the call can proceed



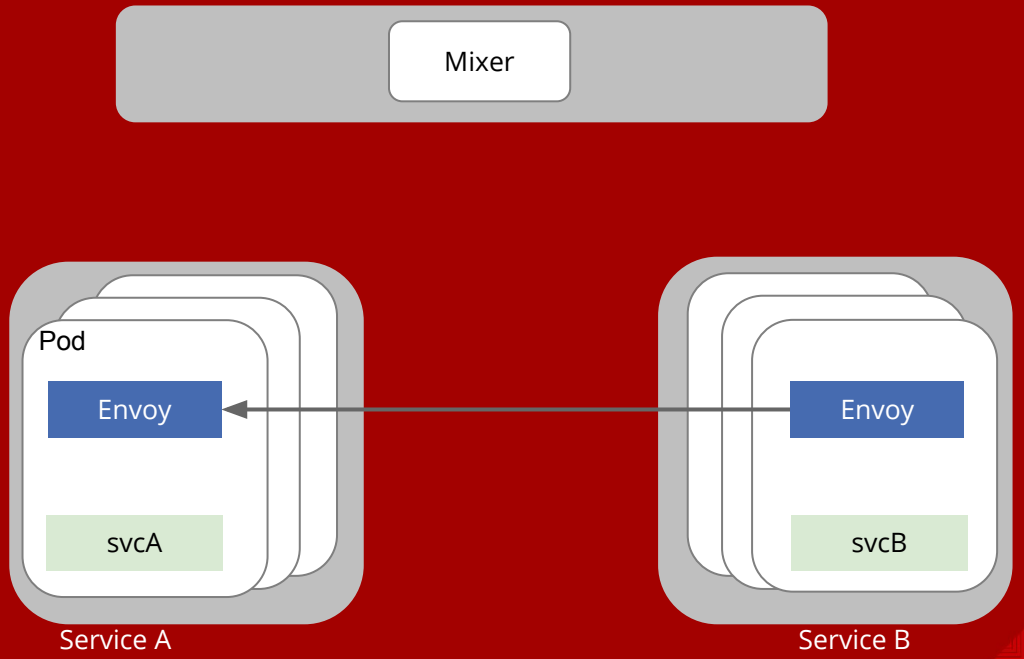
Life of a request in the mesh

Server-side Envoy forwards request to service B, which processes the request and returns response.



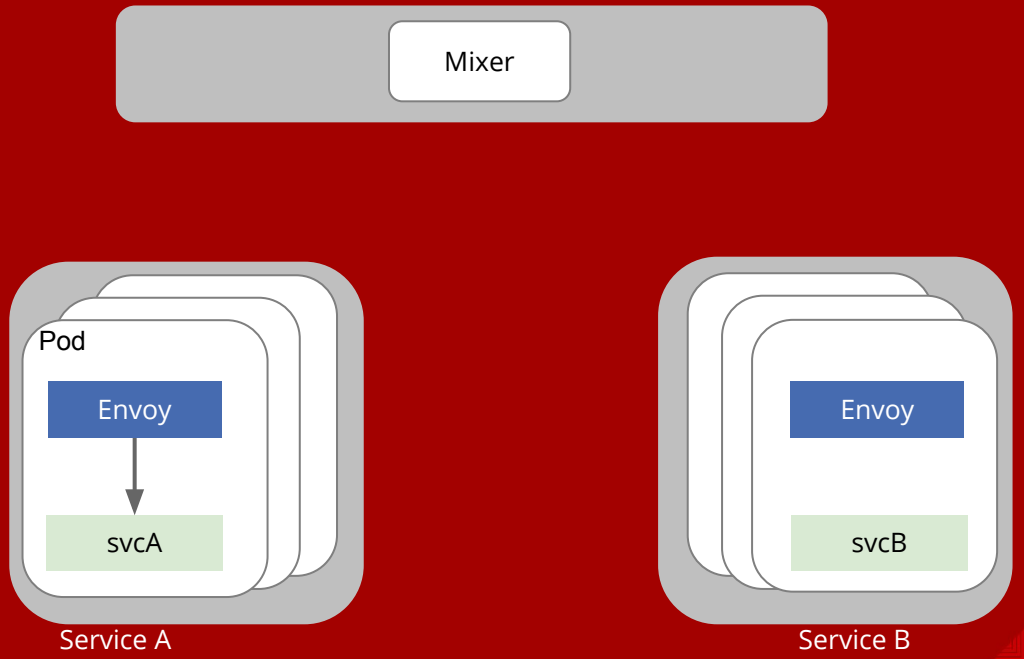
Life of a request in the mesh

Envoy forwards response to the caller



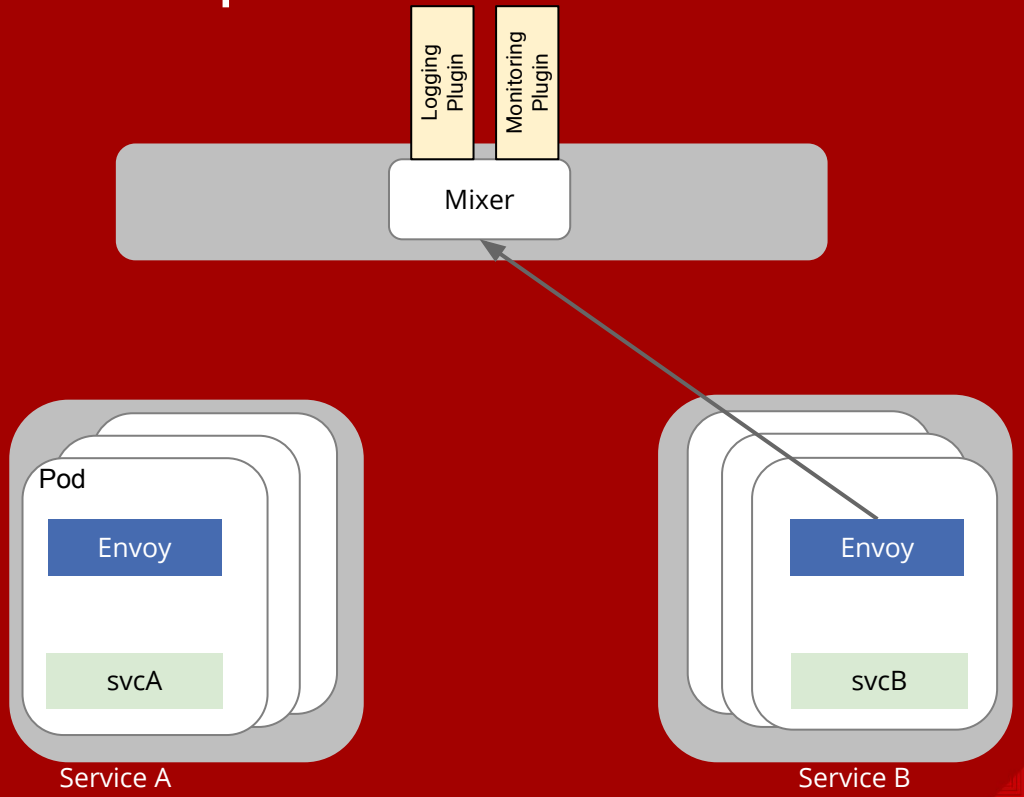
Life of a request in the mesh

Client-side Envoy forwards response to original caller.



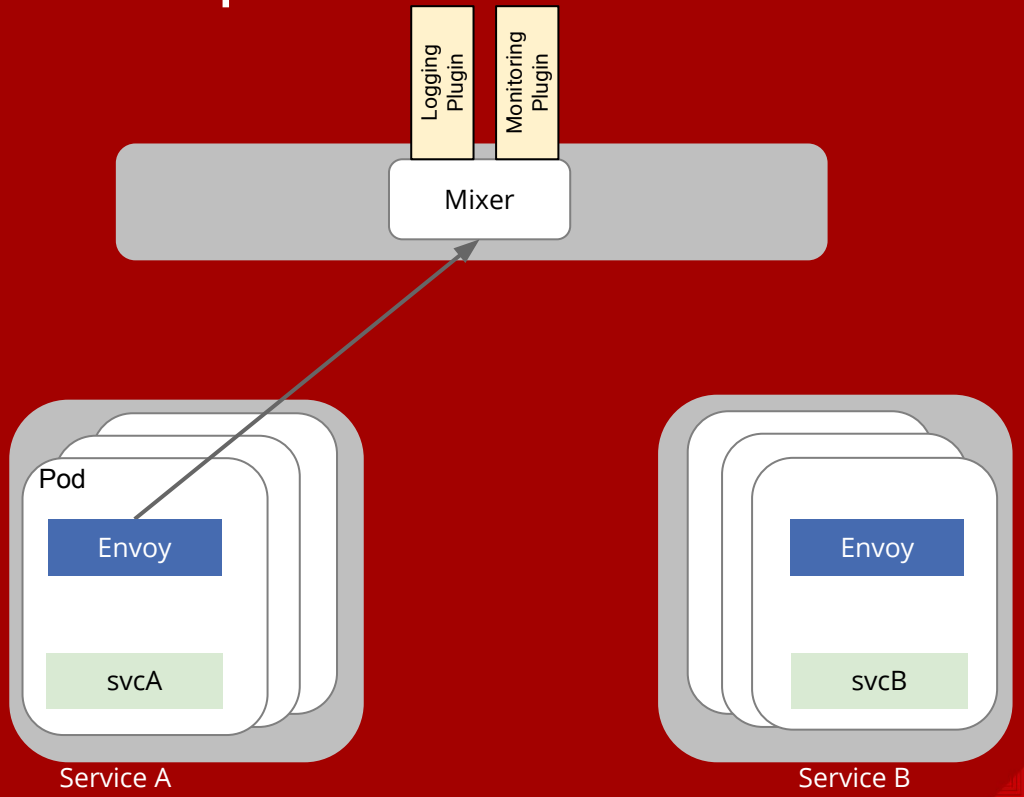
Life of a request in the mesh

Envoy reports telemetry to Mixer, which in turn notifies appropriate plugins



Life of a request in the mesh

Client-side Envoy reports telemetry to Mixer (including client-perceived latency), which in turn notifies appropriate plugins



How to “operate” Istio

- Dedicated command line tool
  - Istioctl
    - istioctl kube-inject

Example configuration

Deploy BookInfo sample on Kubernetes:

```
kubectl apply -f <(istioctl kube-inject -f samples/bookinfo/kube/bookinfo.yaml)
```

Example configuration

Create a custom RouteRule

```
cat <<EOF | kubectl create -f -

apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: reviews-test-v2
  namespace: default
  ...
spec:
  destination:
          name: reviews
  match:
          request:
             headers:
               cookie:
                  regex: ^(.*?;)?(user=jason)(;.*)?$
  precedence: 2
  route:
  - labels:
          version: v2

EOF
```

Easiest way to play with Istio

https://www.katacoda.com/rafabene/courses/workshop/

Thanks!

Twitter: @geoand86

Email: geoand@gmail.com

Slides: http://slideshare.net/GeorgiosAndrianakis

Further reading

- http://envoyproxy.io
- http://istio.io
- https://medium.com/@mattklein123/
- http://blog.christianposta.com/istio-workshop/slides/
- http://blog.christianposta.com/tags/#istio


Q&A