IBM Training
Red Hat OpenShift Service Mesh

—

Luca Floris
Technical Specialist, IBM Automation Expert Labs

Agenda

Day 1

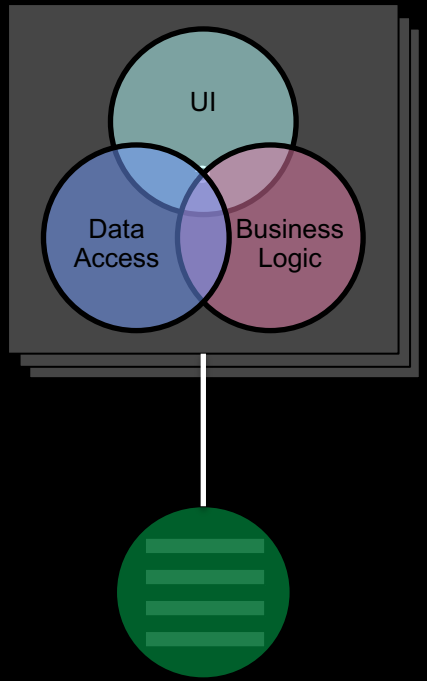
Service Mesh

- Microservices recap
- Introduction to Red Hat OpenShift Service Mesh
- Service Mesh vs Community Istio
- Jaeger
- Kiali
- Designing Service Mesh for Production
- Advanced Use Cases

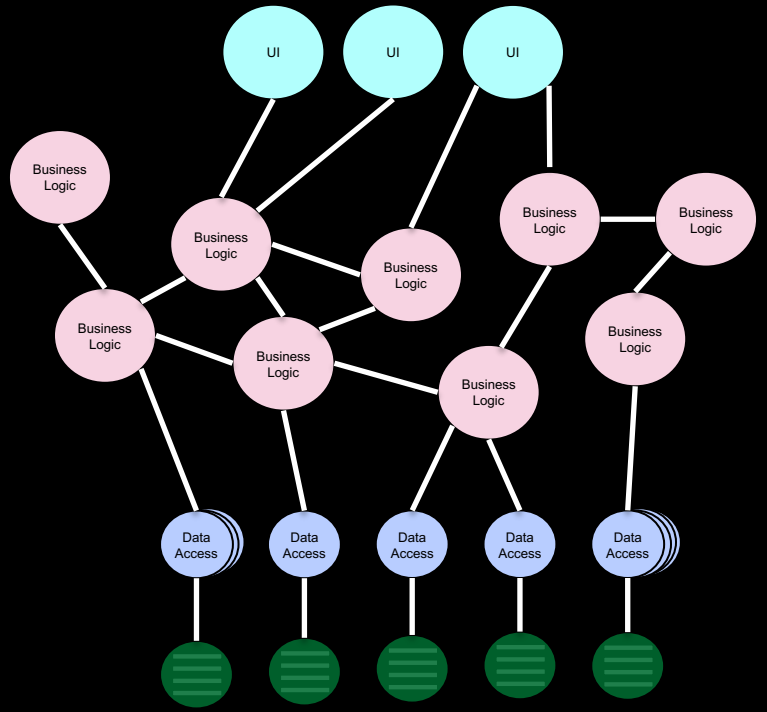
Microservices Recap

Microservice Application Architecture

An **engineering approach** focused on decomposing an application into **single function** modules with **well defined interfaces** which are **independently** deployed and operated by a **small team** who owns the **entire lifecycle** of the service


UI
Data Access
Business Logic

versus


UI
UI
UI
Business Logic
Business Logic
Business Logic
Business Logic
Business Logic
Business Logic
Business Logic
Business Logic
Business Logic
Data Access
Data Access
Data Access
Data Access
Data Access

Introduction to the Service Mesh

Service mesh describes the **network of microservices** that make up applications and the corresponding **interactions** between them.

Introduction to the Service Mesh

Provides enhanced security and traffic management for microservices applications.

Load balancing

Dynamic service discovery

Shadowing (duplicating)

Traffic splitting

Routing rules

Layer 7 routing

Service monitoring and logging

View metrics for services

Secure cross-service communications

Service identity

Service level isolation

Verify user-level tokens

Service Mesh Use Cases

Managing hundred of microservices in and across different Kubernetes cluster can be a nightmare.

Key use cases:

To Manage multiple Kubernetes clusters in multiple cloud vendor.

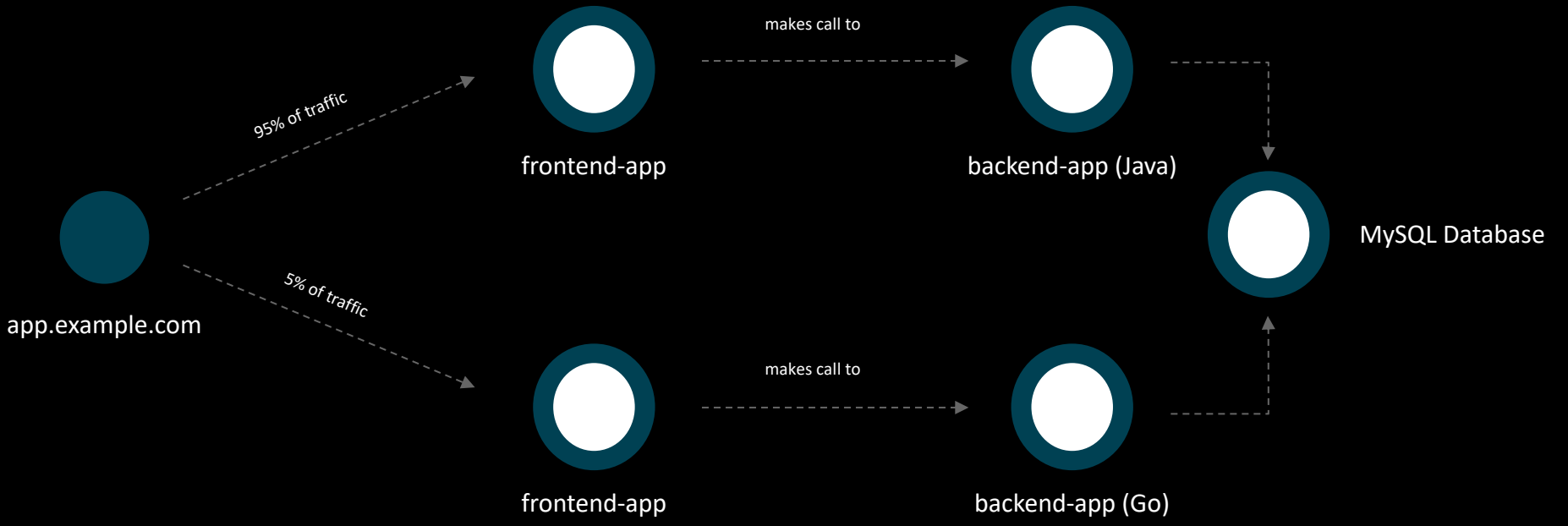
Configuring different networking features i.e. ingress, egress, load balancing, rate limiting, routing etc.

Manage and trace communication between services.

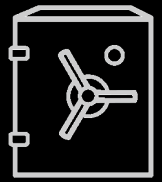
Securing all the service to service commination.

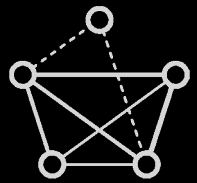
Gaining visibility of all the communication between micro services.

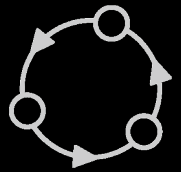
Why do we need a Service Mesh?

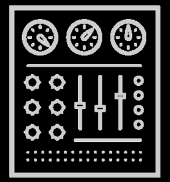


Why do we need a Service Mesh?









Secure

Connect

Observe

Control

Service Mesh Limitations

On its own, the Service Mesh is just the communication layer

Limited observation capabilities

Limited measurement functionality

Not a complete set of tools developers need to build and deploy microservices

Red Hat OpenShift Service Mesh

Provides a platform for behavioural insight and operational control over your networked microservices in a service mesh.

Red Hat OpenShift Service Mesh

Enhanced Application Flexibility through Istio, Kiali, and Jaeger

Enhanced gRPC support for efficient microservice development

Managed Updates to the Control Plane via Operator Driven Deployment

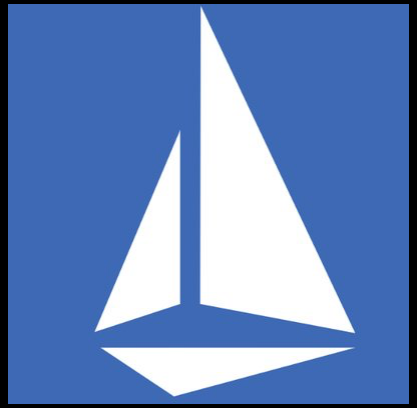
Enhanced Authorization Traffic Controls

Increased visibility with new Kiali and Jaeger functionality

Increased Traffic Management Capabilities

Istio

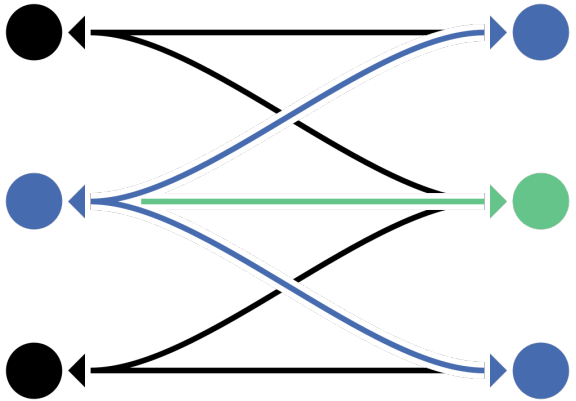
Why Istio?



- Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic
- Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection
- A pluggable policy layer and configuration API supporting access controls, rate limits and quotas
- Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress
- Secure service-to-service communication in a cluster with strong identity-based authentication and authorization

Connect

Intelligently control the flow of traffic and API calls between services, conduct a range of tests and upgrade gradually with red / black deployments

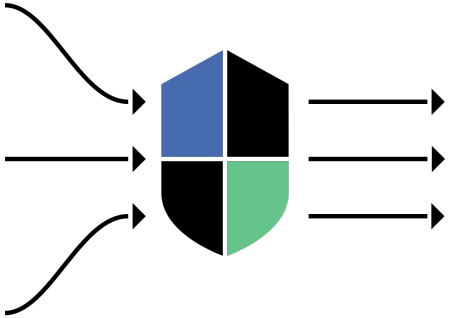


Secure

Automatically secure your services through managed authentication, authorization and encryption of communication between services

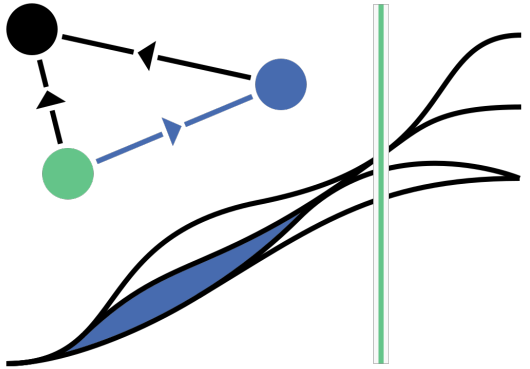
Control

Apply policies and ensure that they are enforced and that resources are fairly distributed among consumers



Observe

See what’s happening with rich automatic tracing, monitoring and logging of all your services

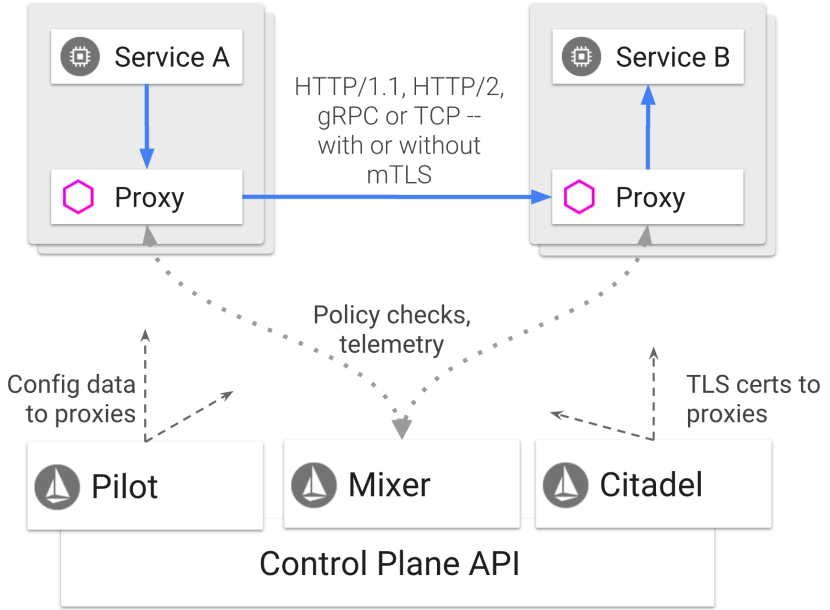


Istio Architecture
Data Plane & Control Plane

Istio is logically composed from a data plane and a control plane

Data Plane
- Intelligent proxies are deployed as sidecars within the service pods
- The proxies mediate and control communication between microservices
- Proxies interface with the Mixer to provide telemetry data and enforce policy

Control Plane
- Configures the proxies for traffic routing
- Configures Mixers for policy enforcement and telemetry collection



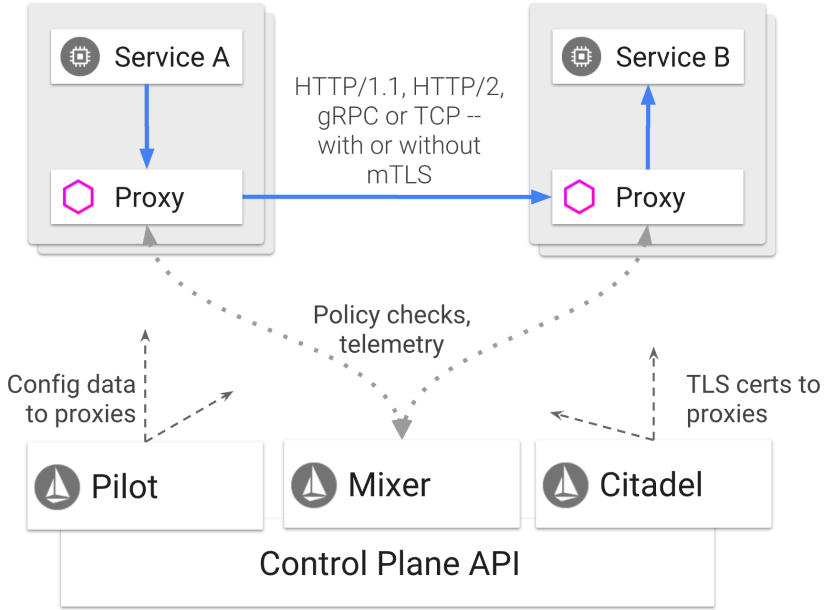
Istio Architecture
Envoy Proxy

A proxy that mediates all inbound and outbound network traffic for the service mesh services

The proxy provides dynamic service discovery, load balancing, TLS termination, HTTP/2 and gRPC proxies, circuit breakers, health checks, staged rollouts with including percentage based traffic split, fault injection and rich metrics

Deployed as a sidecar (container sharing the pod) of the service that is included as part of the mesh

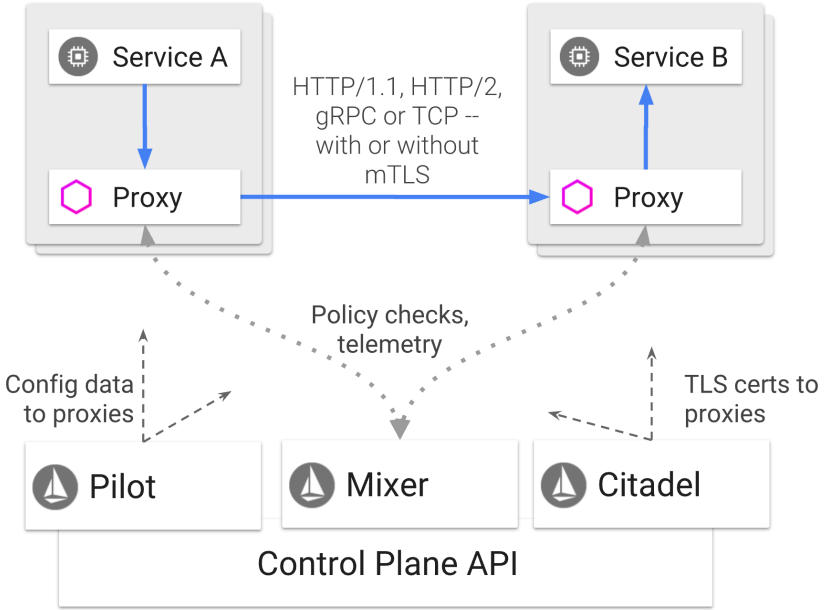
Enables Istio to add capabilities to a deployment without adding to the application code



Istio Architecture
Mixer

Mixer collects telemetry data and enforces usage policies and access control policies throughout the service mesh

The proxy (Envoy) extracts request level attributes and forwards them for evaluation by the Mixer



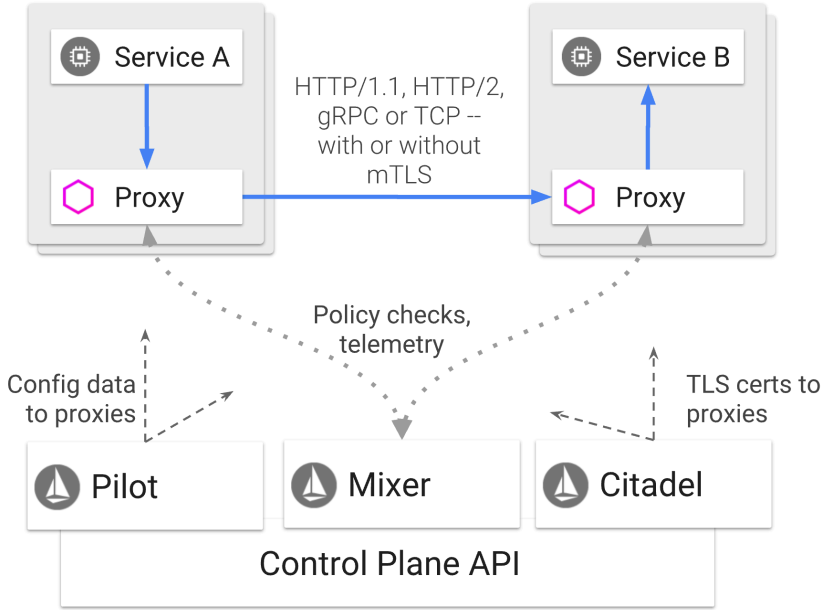
Istio Architecture
Pilot

Pilot performs service discovery for the proxy sidecars, traffic management for intelligent routing and network traffic resiliency

Intelligent routing and resiliency include A/B testing, canary deployments, timeouts, retries and circuit breaking

High-level routing rules are converted by pilot into Envoy configurations and used for traffic control

The framework used by Istio and the loose coupling allows Istio to extend beyond Kubernetes

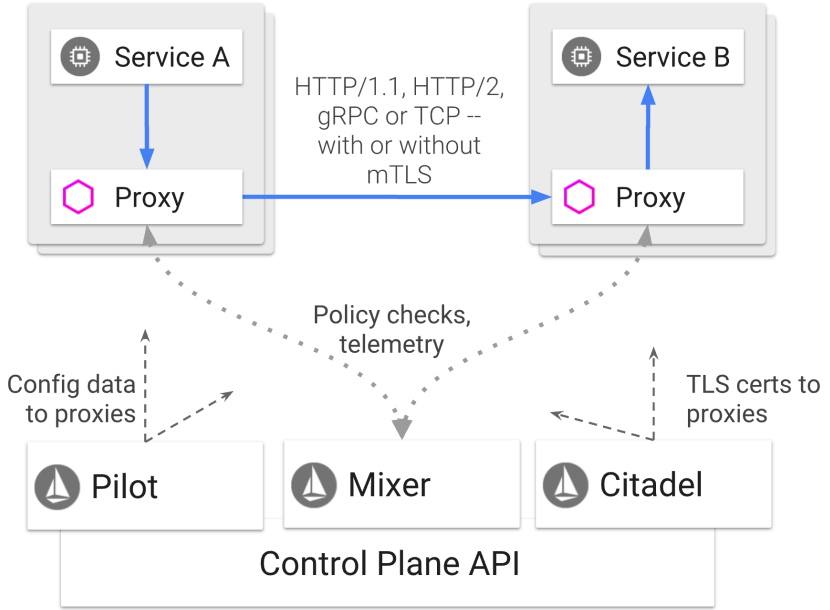


Istio Architecture
Citadel and Galley

Citadel uses its internal identity and credential management capacity provide strong service to service and end-user authentication

With the use of Citadel policy enforcement can be based upon the identity of a service rather than on network controls and the authorization feature controls who can access these services

Galley validates user authored Istio API configuration on behalf of the the control plane and will eventually become responsible as the top-level configuration ingestion, processing and distribution component



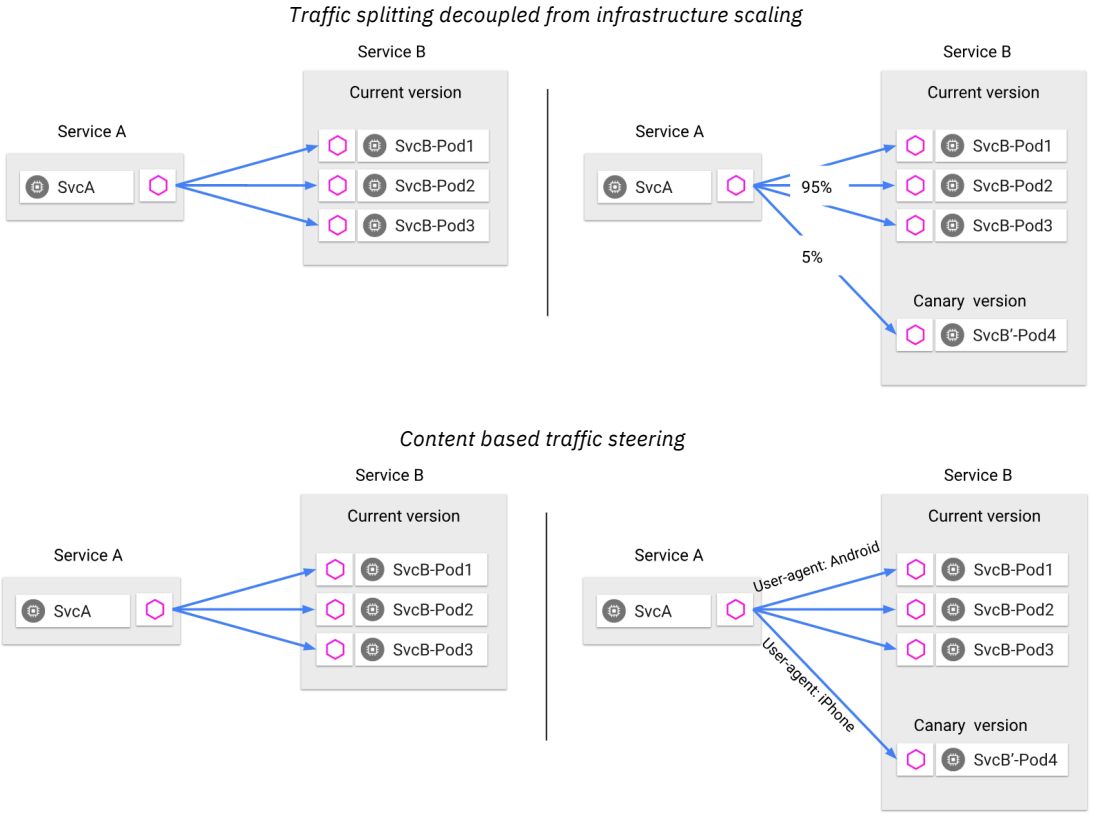
Istio – Managing Traffic

Istio Traffic Management Overview

The traffic management model decouples traffic flow and infrastructure scaling giving you the option of specifying via rules and Pilot how traffic should flow

For example, you can direct a percentage of traffic for a particular service to a canary service or only direct to the canary based upon the content of the request

Decoupling traffic flow from scaling of infrastructure allows for traffic management features outside of the application code including failure recovery via timeouts, retries, circuit breakers and fault injection to test failure recovery procedures



Pilot & Envoy

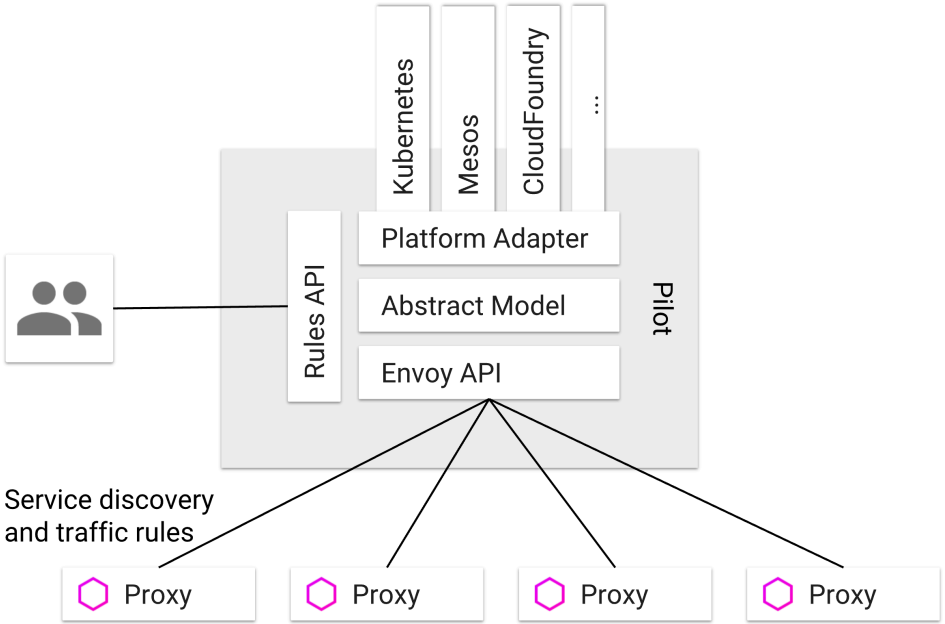
Pilot manages and configures the Envoy proxy instances

Pilot allows the operator to configure traffic routing rules and fault recovery features

Pilot maintains a canonical model of the services participating in the mesh and uses this information to inform each Envoy instance about the other Envoy instances

Each Envoy persists the load balancing information based upon periodic health-checks of the other pool members and information it gets from Pilot, allowing it to route traffic intelligently

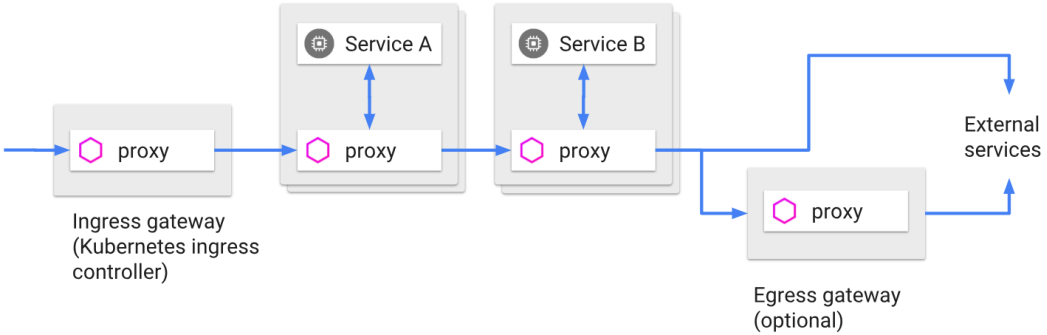
Pilot is manages the lifecycle of Envoy instances



Ingress & Egress

All traffic entering and leaving the mesh passes via Envoy proxies

By routing traffic to and from external web services via an Envoy you add the same failure recovery, timeouts, retries, circuit breakers etc. and can collect connection metrics for these external services (similar to internal to the mesh capabilities)



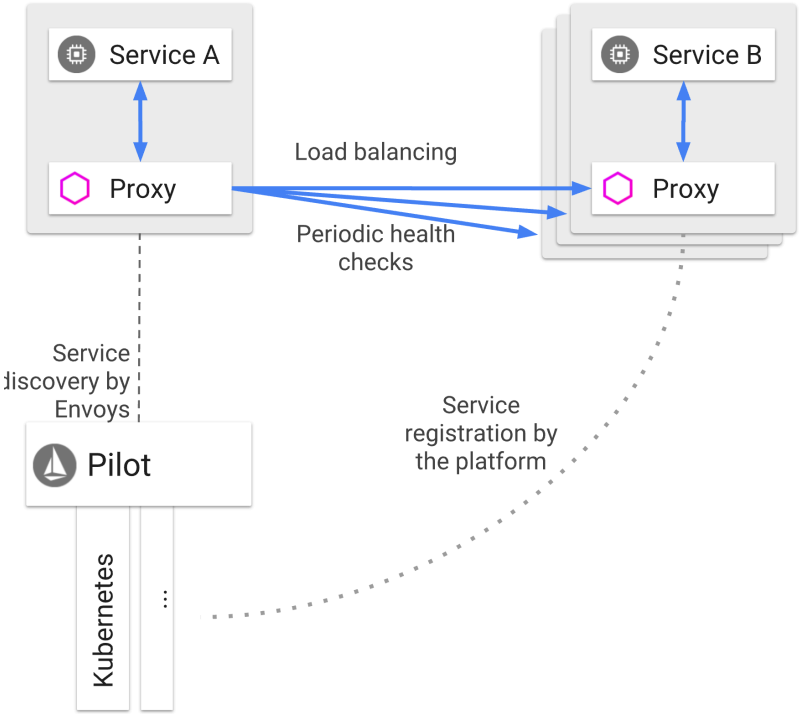
Discovery & Load Balancing

Istio load balancing across instances of a service

It uses the service registry to gain awareness of these application pools

It relies on Kubernetes to maintain the health of these pools

Envoy instances are also performing periodic health checking of the pools and update their load balancing accordingly



Security

Breaking down a monolithic application into atomic services offers various benefits, including better agility, better scalability and better ability to reuse services

However, microservices also have particular security needs:

- To defend against the man-in-the-middle attack, they need traffic encryption
- To provide flexible service access control, they need mutual TLS and fine-grained access policies
- To audit who did what at what time, they need auditing tools

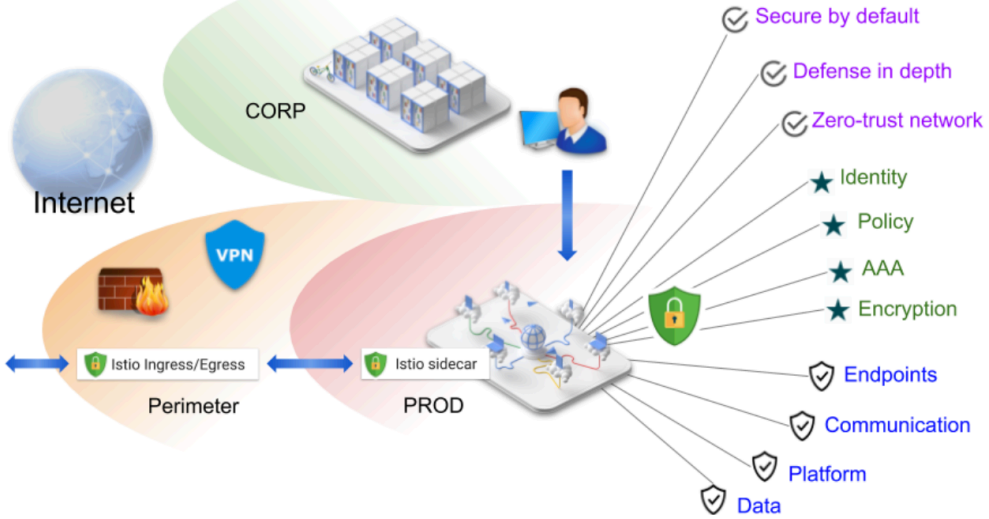
Istio Security tries to provide a comprehensive security solution to solve all these issues

Istio Security Features Overview

The Istio security features provide strong identity, powerful policy, transparent TLS encryption, authentication, authorization and audit (AAA) tools to protect your services and data.

The goals of Istio security are:

- **Security by default**: no changes needed for application code and infrastructure
- **Defense in depth**: integrate with existing security systems to provide multiple layers of defense
- **Zero-trust network**: build security solutions on untrusted networks



Rule Configuration

Istio provides a simple configuration model to control how API calls and layer-4 traffic flow across various services in an application deployment

The configuration model allows you to configure service-level properties such as circuit breakers, timeouts, and retries, as well as set up common continuous deployment tasks such as canary rollouts, A/B testing, staged rollouts with %-based traffic splits, etc.

There are four traffic management configuration resources in Istio
VirtualService, DestinationRule, ServiceEntry, and Gateway:

- A VirtualService defines the rules that control how requests for a service are routed within an Istio service mesh
- A DestinationRule configures the set of policies to be applied to a request after VirtualService routing has occurred
- A ServiceEntry is commonly used to enable requests to services outside of an Istio service mesh
- A Gateway configures a load balancer for HTTP/TCP traffic, most commonly operating at the edge of the mesh to enable ingress traffic for an application

Example: You can implement a simple rule to send 100% of incoming traffic for a *reviews* service to version “v1” by using a VirtualService configuration as follows:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
```

Red Hat OpenShift Service Mesh vs Istio Community

Red Hat OpenShift Service Mesh vs Istio Community

The modifications to Red Hat OpenShift Service Mesh are necessary to resolve issues, provide additional features, or to handle differences when deploying on OpenShift Container Platform.

Red Hat OpenShift Service Mesh vs Istio Community

Multi-tenancy

Supports multiple independent control planes within the cluster

Command line tools

The command line tool for Red Hat OpenShift Service Mesh is oc

Opt-in injection

Does not automatically inject the sidecar to any pods.

You to opt in to injection using an annotation without labeling projects

Extends RBAC

Extends the ability to match request headers by using a regular expression

Automated Routes for Gateways

Every time an Istio Gateway is created, updated or deleted inside the service mesh, an OpenShift route is created, updated or deleted.

OpenSSL

BoringSSL is replaced with OpenSSL

Jaeger

Distributed tracing platform that you can use for monitoring, network profiling, and troubleshooting

Kiali

Management console for observability

Service Mesh 2.x

New features as of Service Mesh 2.x

- New control plane - The Mixer component has been deprecated and will be removed in a future release. The other control plane components, Pilot, Galley, Citadel, have been combined into a single binary known as Istiod.
- Adds support for Envoy's Secret Discovery Service (SDS) - SDS is a more secure and efficient mechanism for delivering secrets to Envoy side car proxies.
- Updates the ServiceMeshControlPlane resource to v2 - Streamlined configuration to make it easier to manage the Control Plane.

Jaeger

Distributed tracing is a technique that is used to tie the information about different units of work together — usually executed in different processes or hosts - to understand a whole chain of events in a distributed transaction. Developers can visualize call flows in large microservice architectures with distributed tracing.

Jaeger

Discover service relationships and process times, transparent to the services

Visualize the service execution times across the application

Identify potential latency issues in each service

Jaeger Features

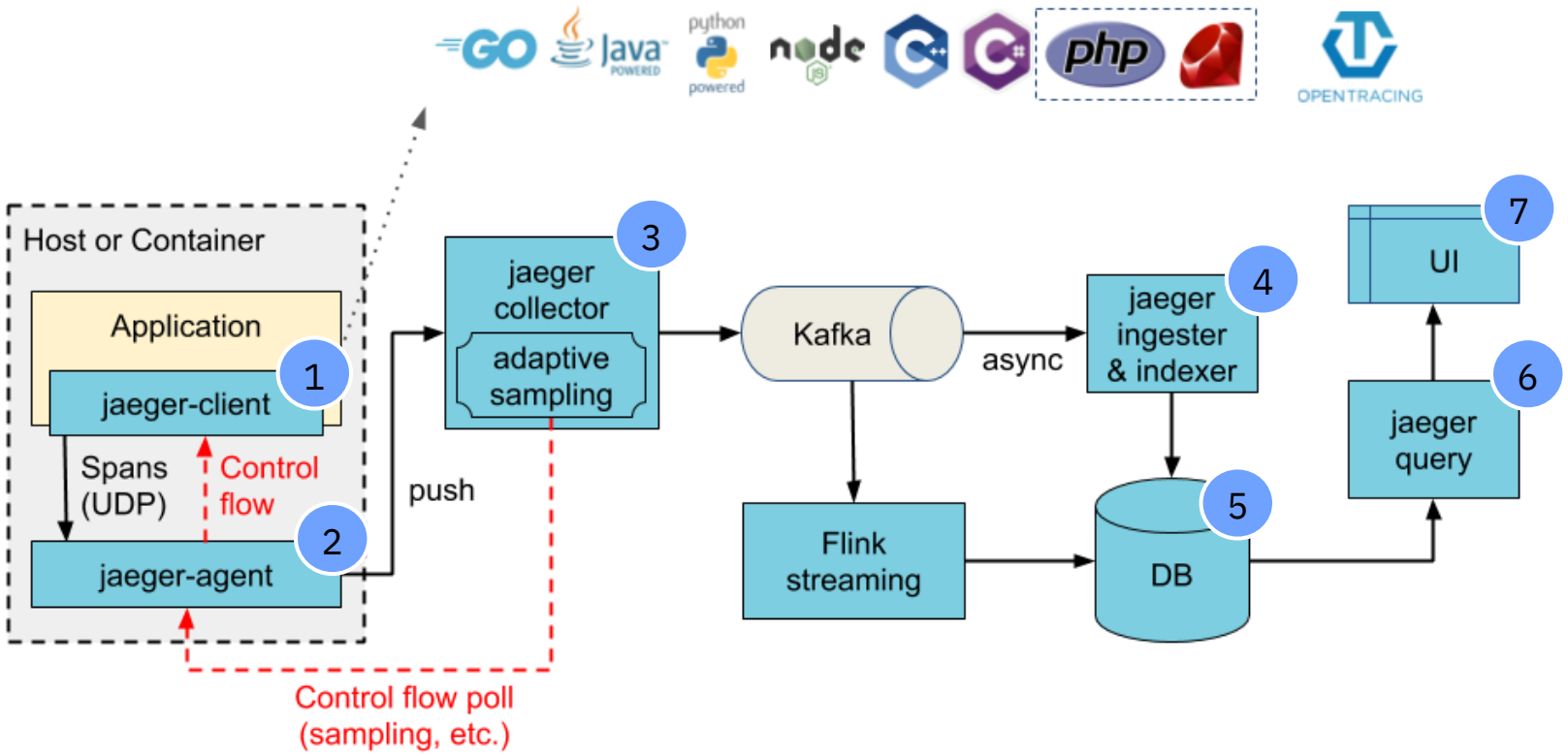
Integration with Kiali – When properly configured, you can view Jaeger data from the Kiali console.

High scalability – The Jaeger backend is designed to have no single points of failure and to scale with the business needs.

Distributed Context Propagation – Lets you connect data from different components together to create a complete end-to-end trace.

Backwards compatibility with Zipkin – Jaeger has APIs that enable it to be used as a drop-in replacement for Zipkin, but Red Hat is not supporting Zipkin compatibility in this release.

Jaeger Architecture



Kiali

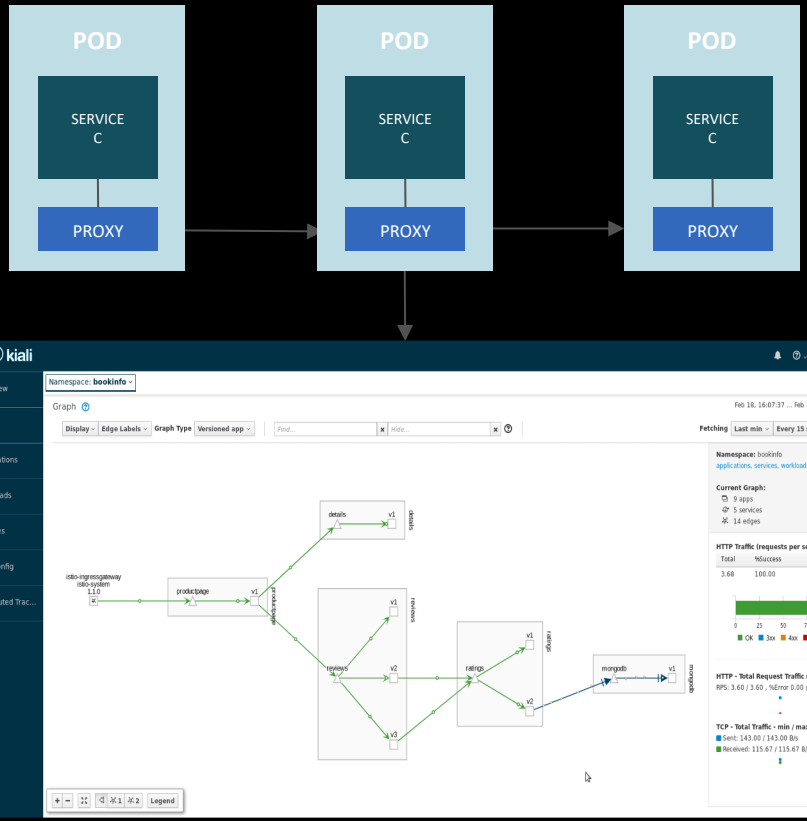
Kiali helps you define, validate, and observe the connections and microservices of your Istio service mesh.

Kiali

Kiali works to visualise the service mesh topology

Identify which services are part of the service mesh and how they are connected

Understand the topology and health of the service mesh



Kiali Architecture

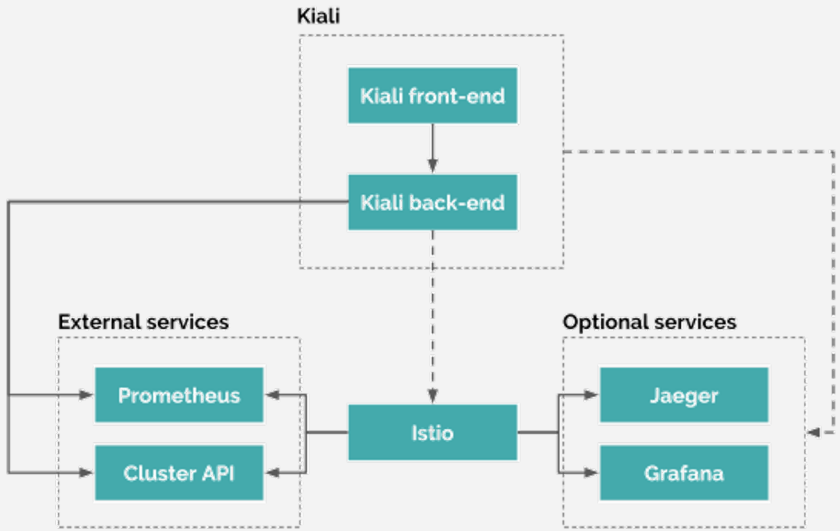
Back-end

Communicates with Istio

Retrieves and processes data

Front-end

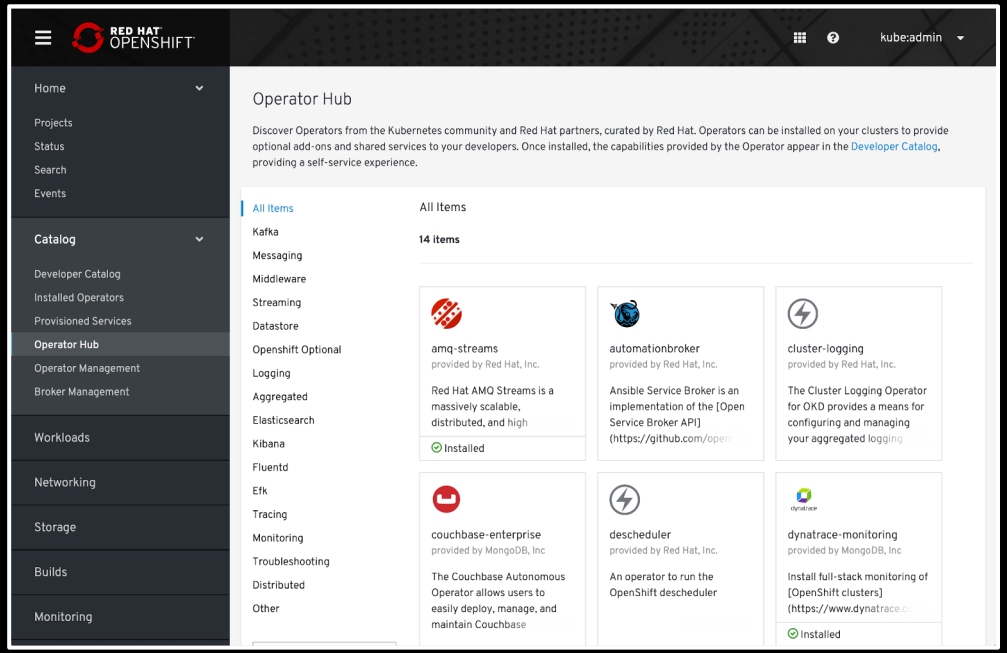
Web application served by the back-end



Red Hat OpenShift Service Mesh Availability

OpenShift Service Mesh is available at no additional cost for licensed OpenShift customers

OpenShift Service Mesh Operator will be found from the embedded OperatorHub interface through the OpenShift interface



Red Hat OpenShift Service Mesh Demo

Red Hat OpenShift Service Mesh Lab

Visit https://github.com/lfloris/openshift-service-mesh-workshop/tree/main/Labs for lab materials

Goals

Deploy an application using the Service Mesh and configure it for various use cases

Designing for Production

Questions to ask yourself…

Do I rely on Istio alone for traffic routing isolation and security, or do I need dedicated hardware?

How many gateways do I need? Does this number need to grow with the load?

Do I need certificates signed by my organisation or is self signed ok?

Do I need to persist all the tracing data?

Development

Single ingress/egress gateways, without autoscaling

Only one ingress gateway and one egress gateway is configured for all inbound/outbound Service Mesh traffic

In-memory trace data

Trace data is stored within the Jaeger pod volatile memory

Service Mesh pods are distributed across the cluster

All components of the Service Mesh run on any worker node in the cluster and are subject to evictions depending on the cluster activity (e.g. actively adding/removing compute nodes)

Production

Multiple ingress/egress gateways, with autoscaling

Dedicated Ingress Controllers are deployed for dedicated instances of ingres/egress gateways to isolate traffic between applications or environments

mTLS enabled

Proxies have mutual TLS enabled

Dedicated hardware for ingress/egress gateways + Ingress Controller

Additional compute nodes are deployed that exclusively run Ingress Controllers and gateways

Dedicated hardware for all other Service Mesh components

All other Service Mesh components are deployed to dedicated compute nodes

ElasticSearch for trace persistence

ElasticSearch is deployed for Jaeger to consume to store trace data

Example Production Topology

Advanced Use Cases

Use Case 1

Additional Ingress Gateways

"I have multiple stages within the same OpenShift cluster, but I need to keep the application traffic separate"

Use Case 2

Versioned Traffic Routing

"I wanted to deploy different versions of microservices but control how traffic is routed to them"

Use Case 3

Traffic Mirroring

"I need to replicate some production traffic to my development cluster to see what's going on"

User Case 4

Application Debugging

"With so many services deployed it's difficult to observe the whole topology and discover which services are causing issues"

Advanced Use Cases Labs

Visit https://github.com/lfloris/openshift-service-mesh-workshop/tree/main/Labs for lab materials

Goals

Modify the Service Mesh to deploy an additional ingress gateway

Modify the Bookinfo application to route different versions to different users

Set up a Pre-Production and Production Bookinfo application with traffic mirroring

Debug a failed Bookinfo service using Kiali and Istio

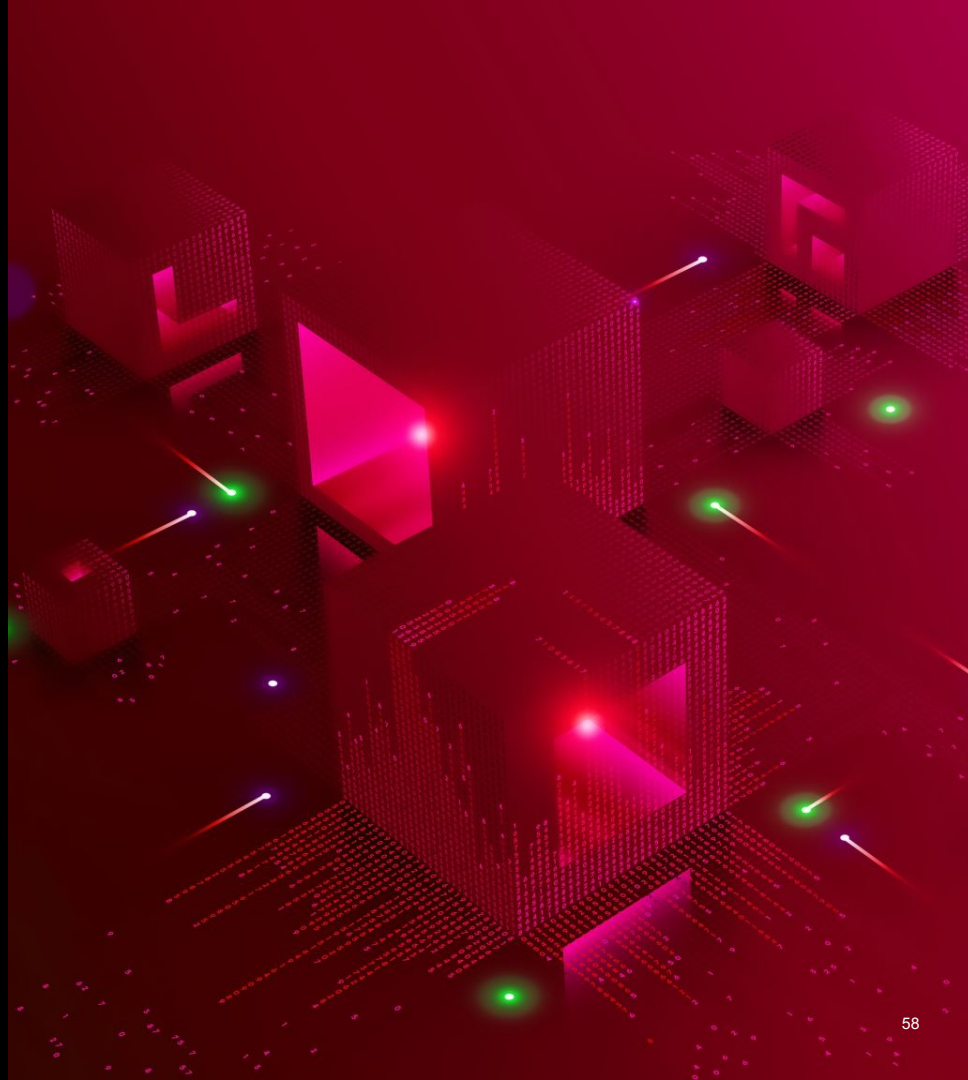
Other Best Practices

Isolation – Try to achieve isolation where possible as this makes controlling issues a lot easier

Control – Use VirtualServices for each service to allow you to fine tune traffic flow

JWT – JSON Web Tokens can restrict what can access your mesh. After authentication, a user or service can access routes, services that are associated with that token

Authorization Policies – Istio's approach to isolation



Questions/Discussions?