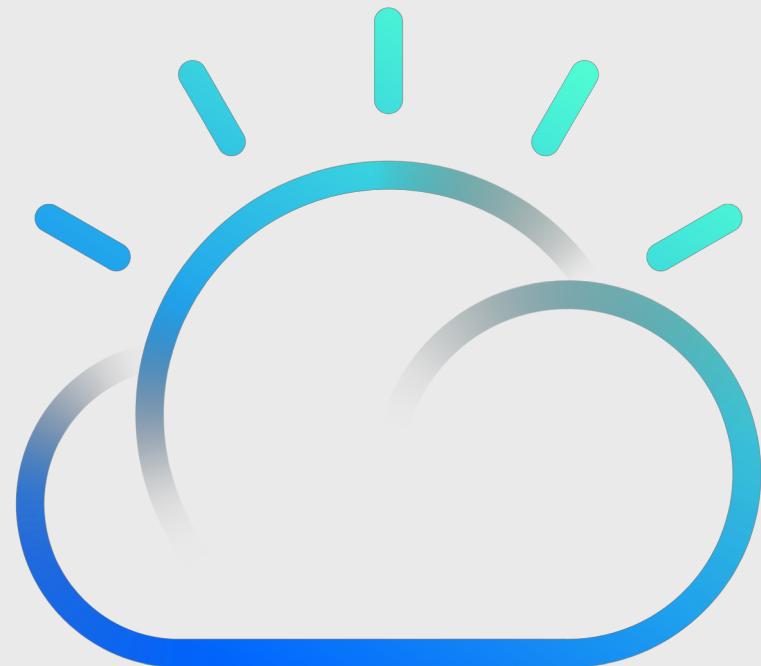


IBM

## IBM Cloud Private Istio – Microservice Mesh



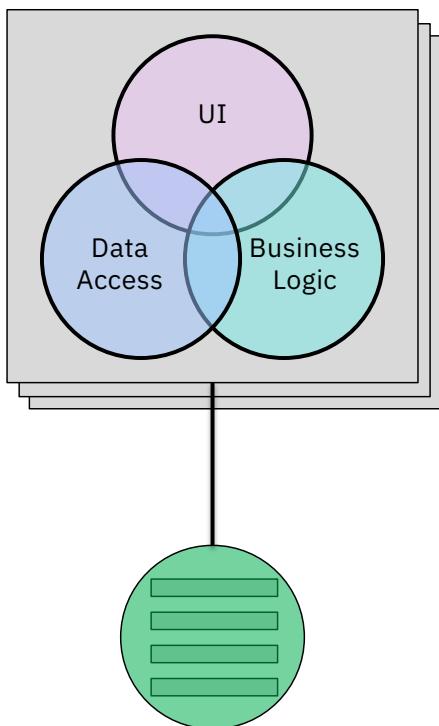
IBM Cloud

# Service Mesh Architecture Managing Traffic Operations

## Microservice Application Architecture

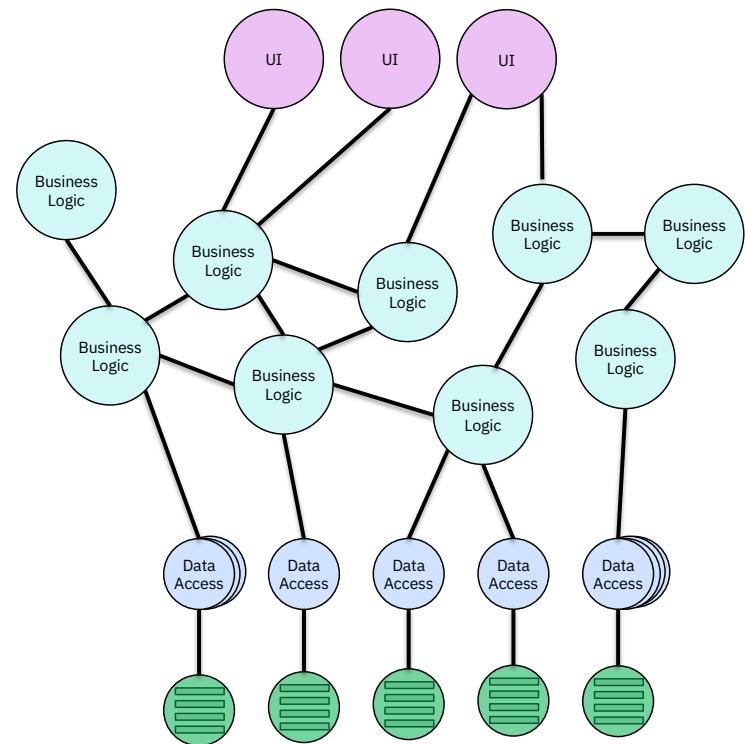
An **engineering approach** focused on decomposing an application into **single function** modules with **well defined interfaces** which are **independently** deployed and operated by a **small team** who owns the **entire lifecycle** of the service

# Monolithic



# versus

# Microservices



# Weighing the Microservice Investment

Improved delivery velocity and agility



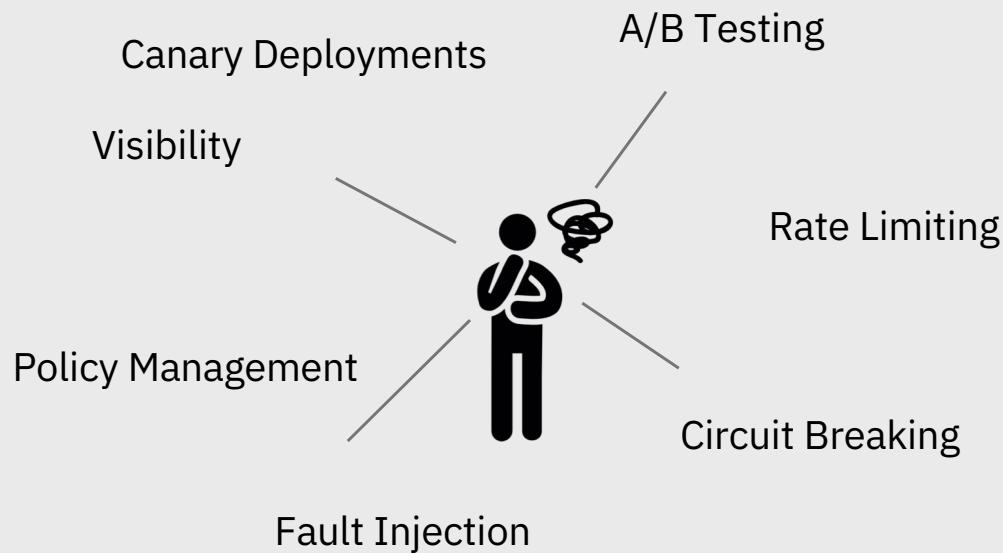
Increased operational complexity

IBM Cloud Private / Kubernetes enables the microservice design goals of clean packaging, consistency, scalability and rapid deployment

Kubernetes alone does not address all of the complexities of the challenge

# Microservice Adoption Considerations

Deploying microservice applications is not necessarily easy, the network layer is challenging and tooling is essential



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

# Istio

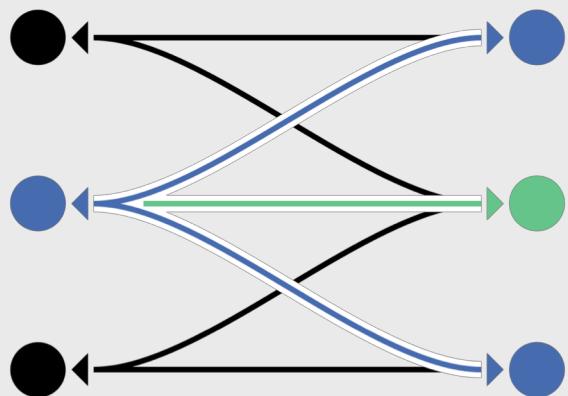
Connect, secure, control and observe services

# 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



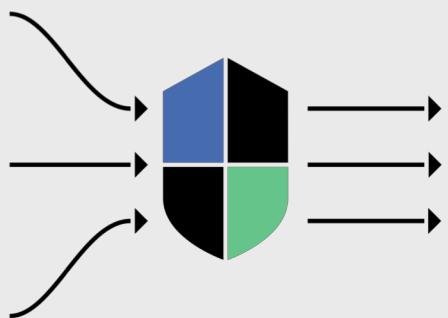
Intelligently control the flow of traffic and API calls between services, conduct a range of tests and upgrade gradually with red / black deployments

# Secure



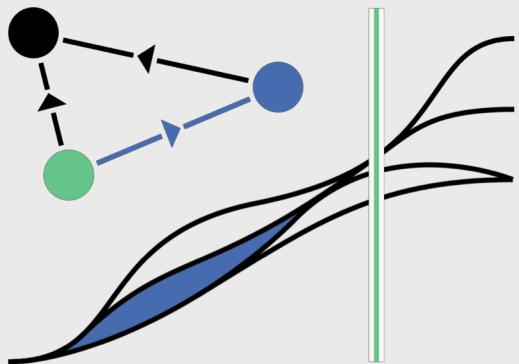
Automatically secure your services through managed authentication, authorization and encryption of communication between services

## 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 Core Features and Value

## Traffic management

- Easy-to-Configure routing and traffic control
- Simplified configuration of circuit breakers, timeouts, and retries supporting A/B testing, canary and staged rollouts
- High visibility into your traffic

## Security

- Free developers to focus on security at the application level
- Istio manages authentication, authorization, and encryption of service communication at scale
- Service communications are secured by default with little or no changes to the application
- Via integration with the platform secure pod-to-pod or service-to-service communication at the network AND application layers

## Observability

- Rich tracing, monitoring, and logging provide deep insights into the service mesh
- Understand upstream and downstream performance effects
- Out of the box dashboards provide deep visibility into service usage and performance
- Enables fine-grained control over all interactions between the mesh and infrastructure backends
- Detect, diagnose and fix issues with greater speed and agility

## Platform support

- Platform independence
- Deploy across services running in IBM Cloud Private (Kubernetes) and hosted on Virtual Machines

# Istio

## Component Architecture

# Istio Architecture

## Data Plane & Control Plane

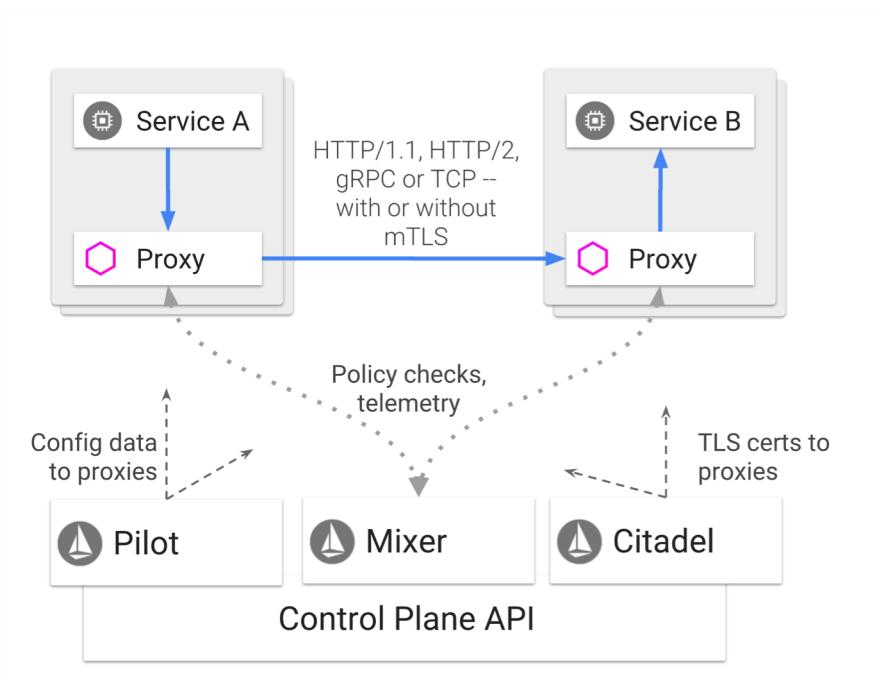
Istio is logically composed from a data plane and a control plane

### 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

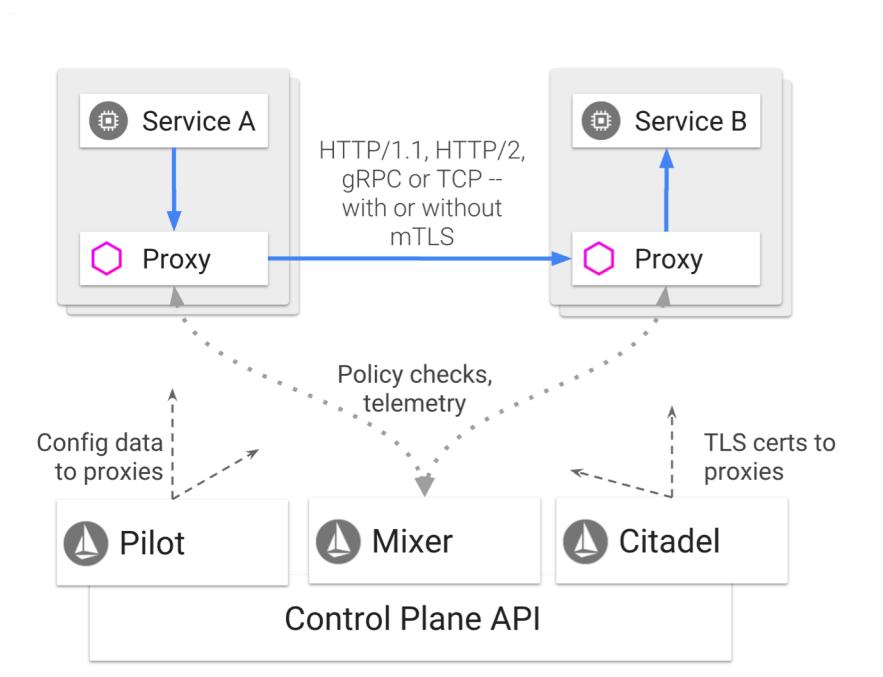
## Envoy Proxy

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

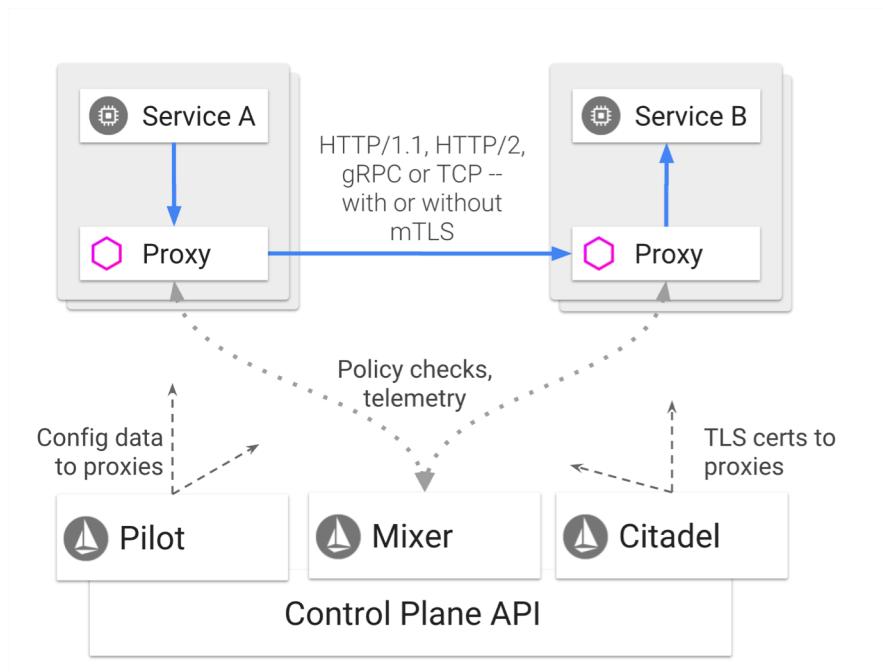


# Istio Architecture

## 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

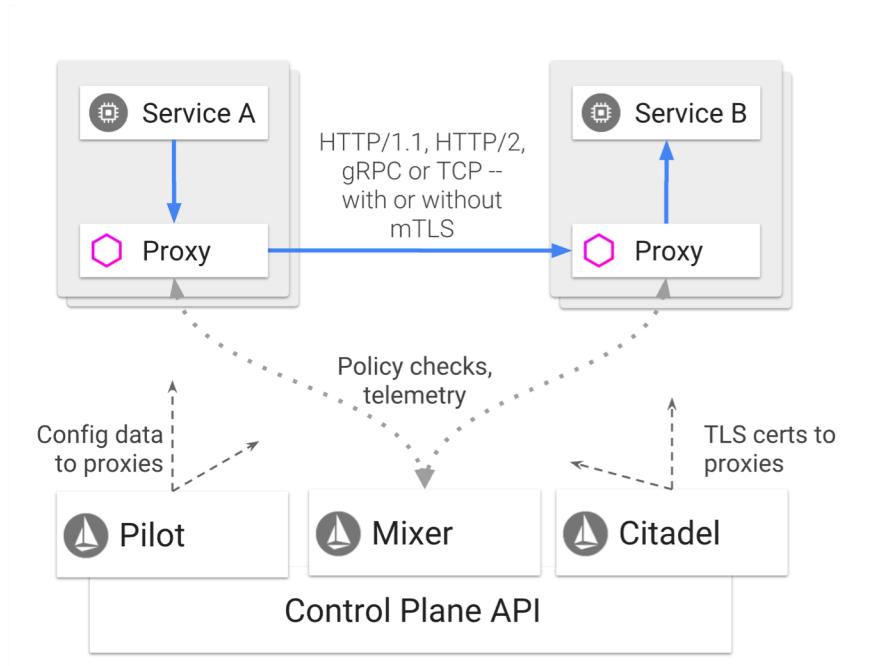
## Pilot

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



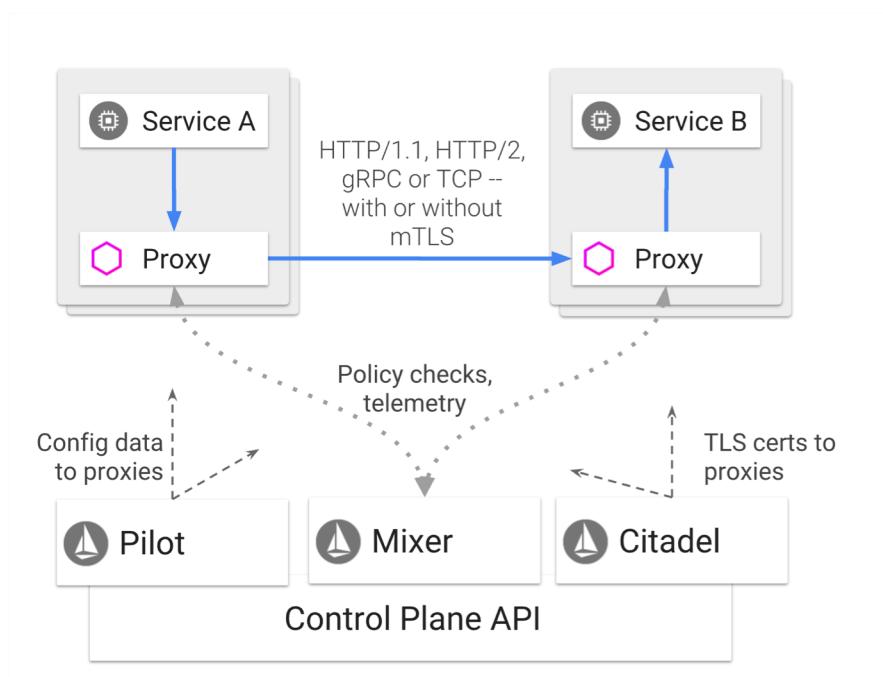
# Istio Architecture

## 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

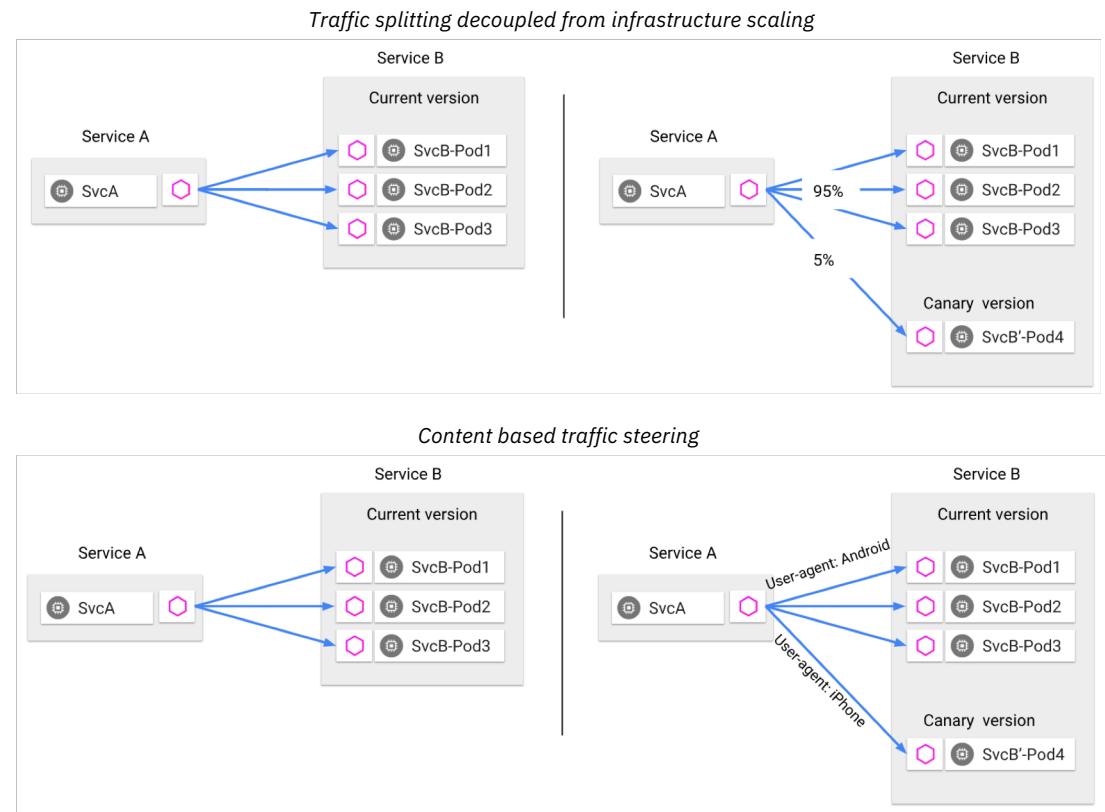
## Managing Traffic

# 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

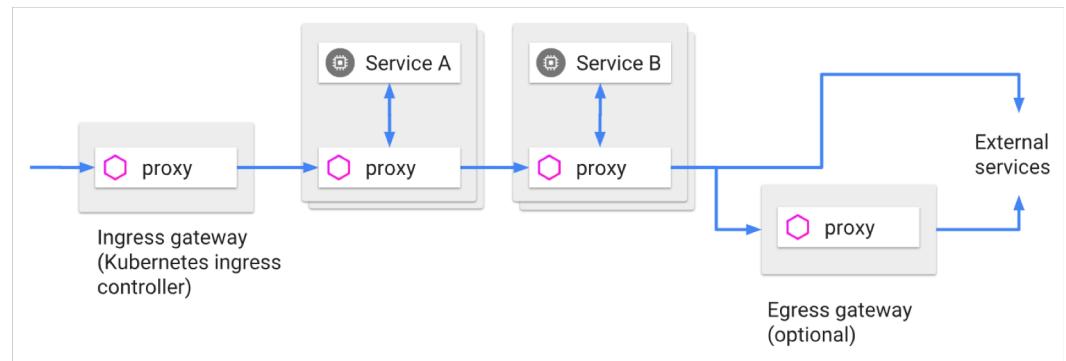
Decoupling traffic flow from scaling of infrastructure allows for traffic management features outside of the application code including failure recovery via timeouts, retries, circuit breakers and fault injection to test failure recovery procedures



# 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)



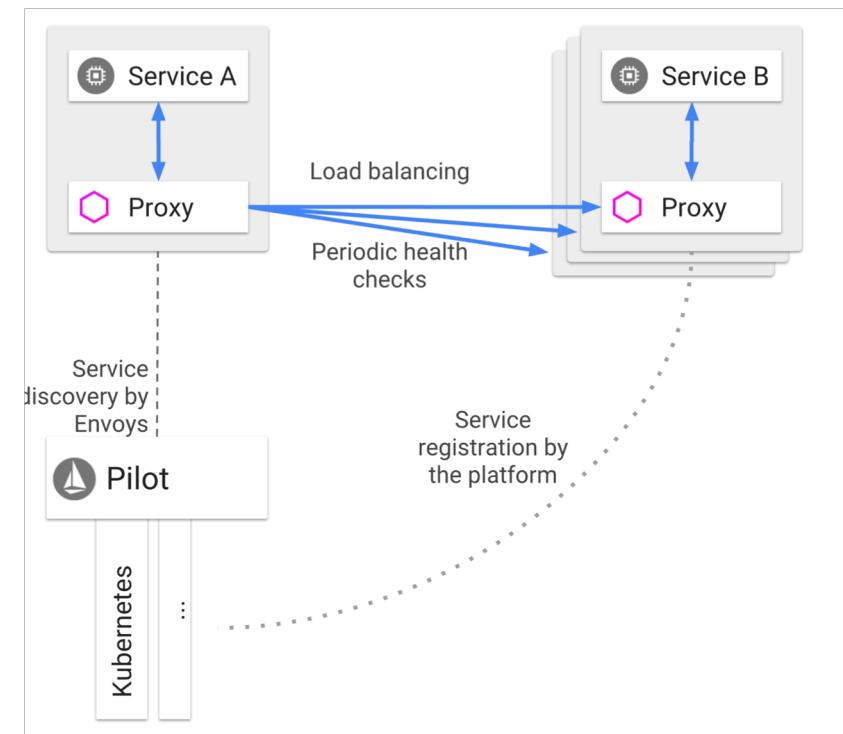
# Discovery & Load Balancing

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

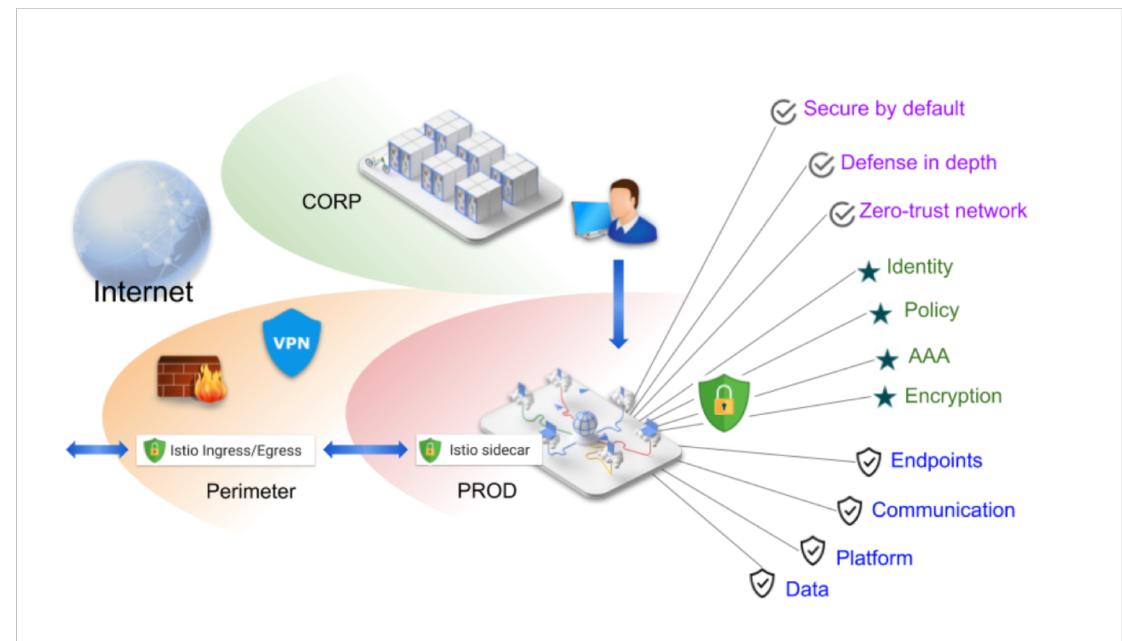
Istio Security tries to provide a comprehensive security solution to solve all these issues

# 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 (also applies to egress)

**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
```

# Istio

## Operations

# Istio “Up and Running”

Install Istio in ICP by using the latest Helm chart and configure your namespaces for automatic sidecar injection

**Note:** *kube-apiserver process has the admission-control flag set with the MutatingAdmissionWebhook and ValidatingAdmissionWebhook admission controllers added and listed in the correct order and the admissionregistration API is enabled*

```
$ kubectl api-versions | grep admissionregistration  
admissionregistration.k8s.io/v1alpha1 admissionregistration.k8s.io/v1beta1
```

Enable the injection by namespace label:

```
$ kubectl label namespace default istio-injection=enabled  
$ kubectl get namespace -L istio-injection  
NAME STATUS AGE ISTIO-INJECTION  
default Active 1h enabled  
istio-system Active 1h  
kube-system Active 1h
```

Alternatively inject at deployment time:

```
$ istioctl kube-inject -f your-deployment.yaml | kubectl apply -f -
```

Check your deployed workload to verify successful injection:

```
$ kubectl get deployment your-dep -o wide  
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE CONTAINERS IMAGES SELECTOR  
your-dep 1 1 1 1 2h sleep,istio-proxy tutum/curl,unknown/proxy:unknown app=sleep
```

# Adding an Istio Gateway for Ingress

Assume you have an application running with the following services defined:

```
$ kubectl get services
NAME           CLUSTER-IP EXTERNAL-IP PORT(S) AGE
docs           10.0.0.31   <none>      9080/TCP  6m
homepage       10.0.0.1    <none>      443/TCP   7d
my-app-base   10.0.0.120  <none>      9080/TCP  6m
blog           10.0.0.15   <none>      9080/TCP  6m
```

Create an ingress gateway for the service directing three URIs to the “my-app-base” service:

```
$ kubectl apply -f your-gateway.yaml
```

Retrieve the ingress IP and port:

```
$ kubectl get svc istio-ingressgateway -n istio-system
NAME           TYPE    CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
istio-ingressgateway   LB     172.21.109.129  130.211.10.121  80:31380/TCP,443:31390/TCP,31400:31400/TCP  17h
```

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: my-app-gateway
spec:
  selector:
    istio: ingressgateway # use istio default controller
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - "*"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: my-app
spec:
  hosts:
    - "*"
  gateways:
    - my-app-gateway
  http:
    - match:
        - uri:
            exact: /intropage
      - uri:
            exact: /login
      - uri:
            exact: /logout
    route:
      - destination:
          host: my-app-base
          port:
            number: 9080
```

## Example: Injecting an HTTP Delay Fault

To test the an application's microservices for resiliency you could inject a delay between services for a specific user

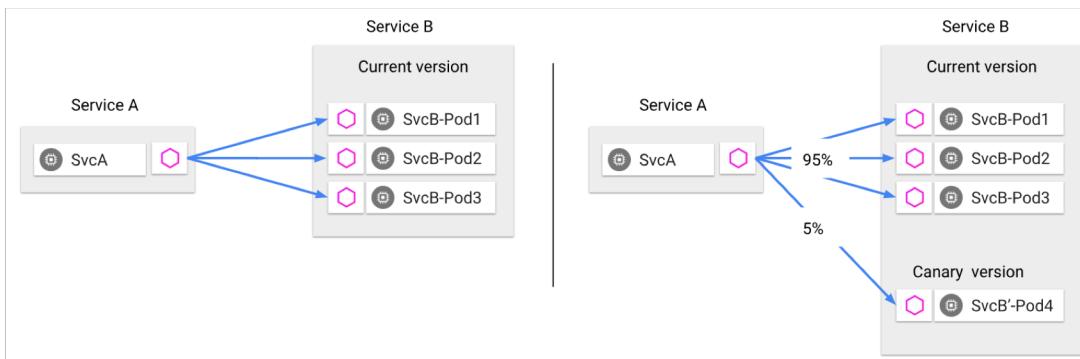
In the case shown here, traffic initiated by Jason to the ratings version 1 service will receive a 7 second delay

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
    - ratings
  http:
    - match:
        - headers:
            end-user:
              exact: jason
      fault:
        delay:
          percent: 100
          fixedDelay: 7s
      route:
        - destination:
            host: ratings
            subset: v1
        - route:
            - destination:
                host: ratings
                subset: v1
```

# Example: Traffic Splitting

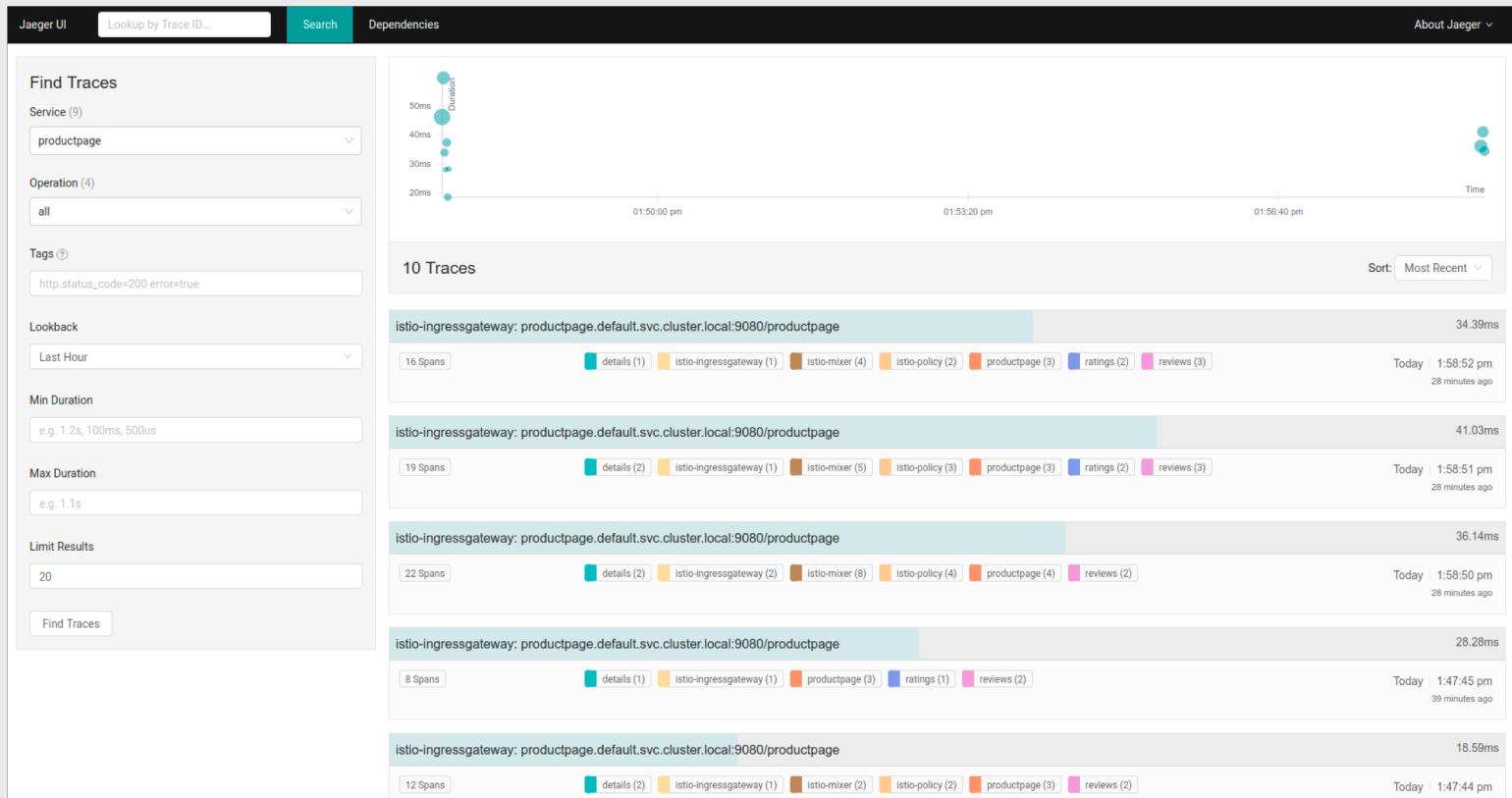
Traffic control is decoupled from infrastructure scaling

In this example we split the traffic between two application versions sending 5% to the new version



```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
    - route:
        - destination:
            host: reviews
            subset: v1
            weight: 95
        - destination:
            host: reviews
            subset: v2
            weight: 5
```

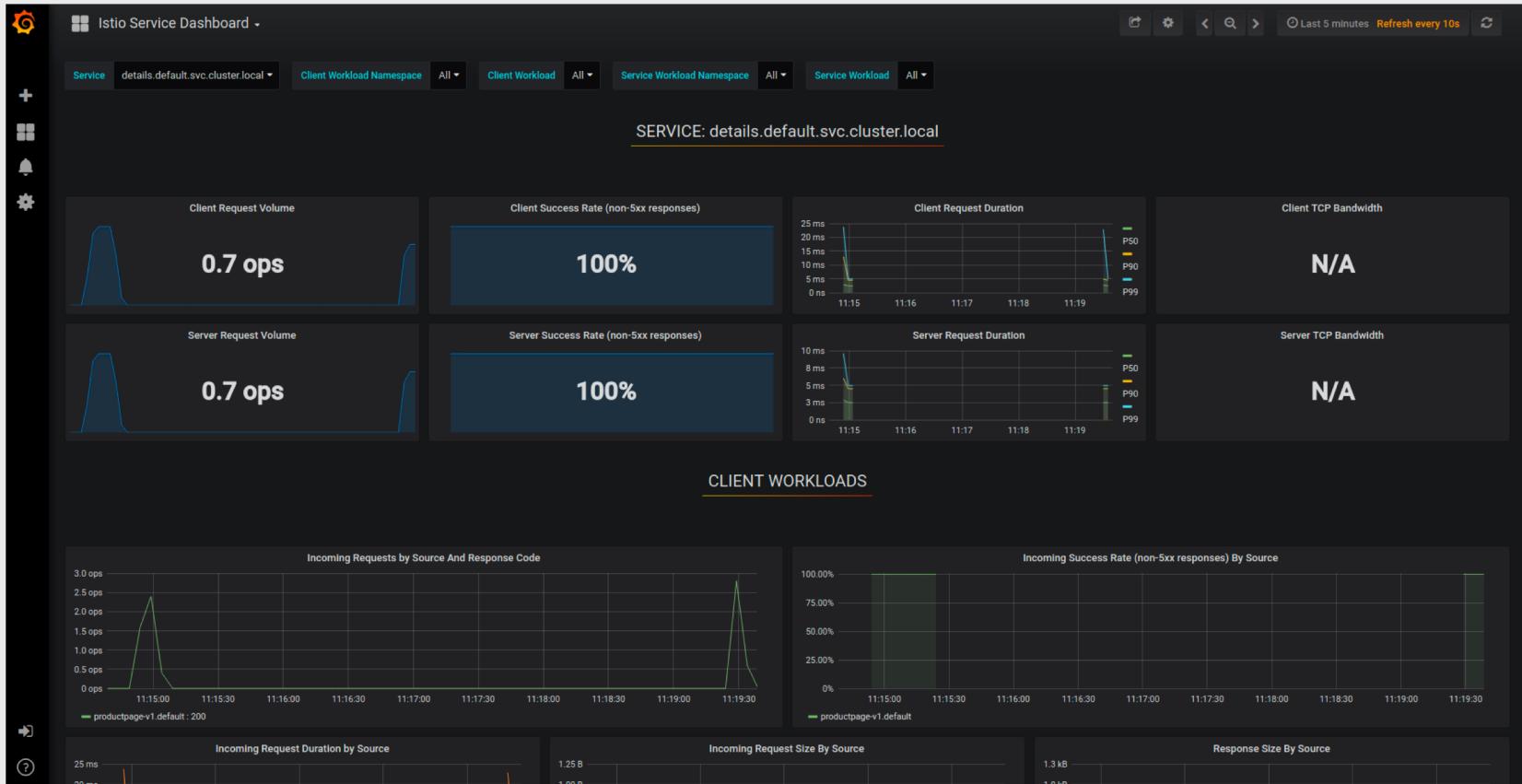
# Telemetry: Tracing Dashboard



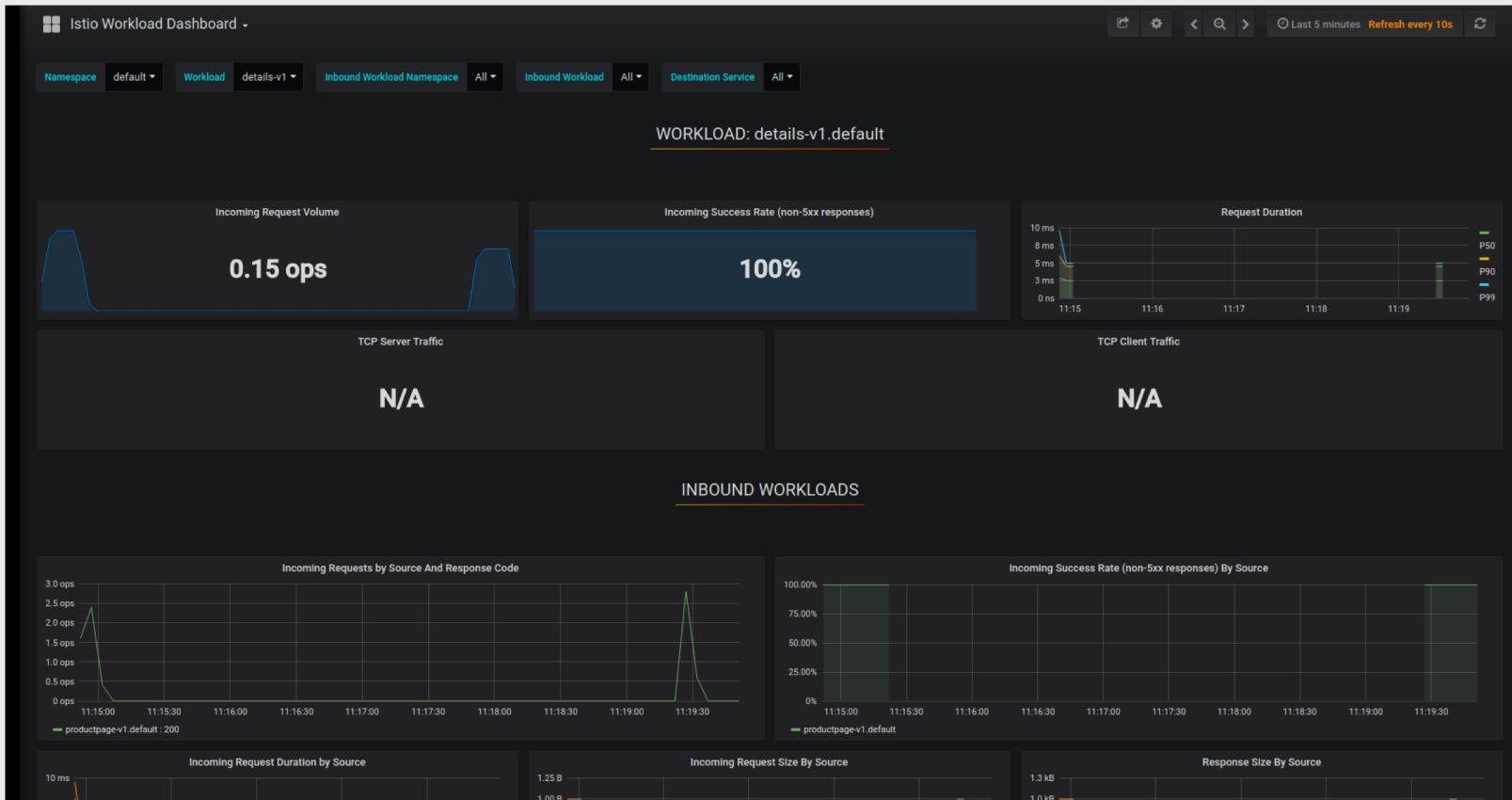
# Telemetry: Tracing Detail



# Telemetry: Visualize Service Dashboard



# Telemetry: Tracing Detail

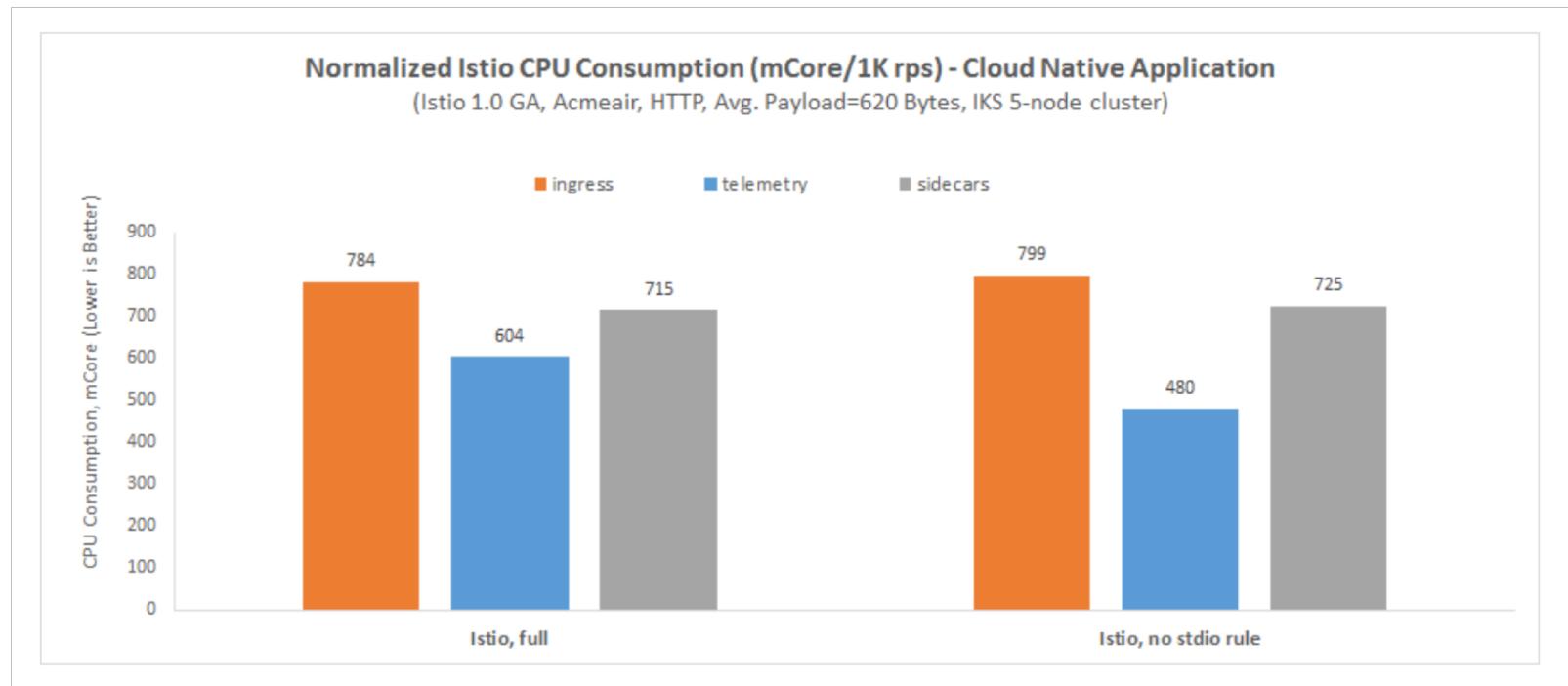


# Istio

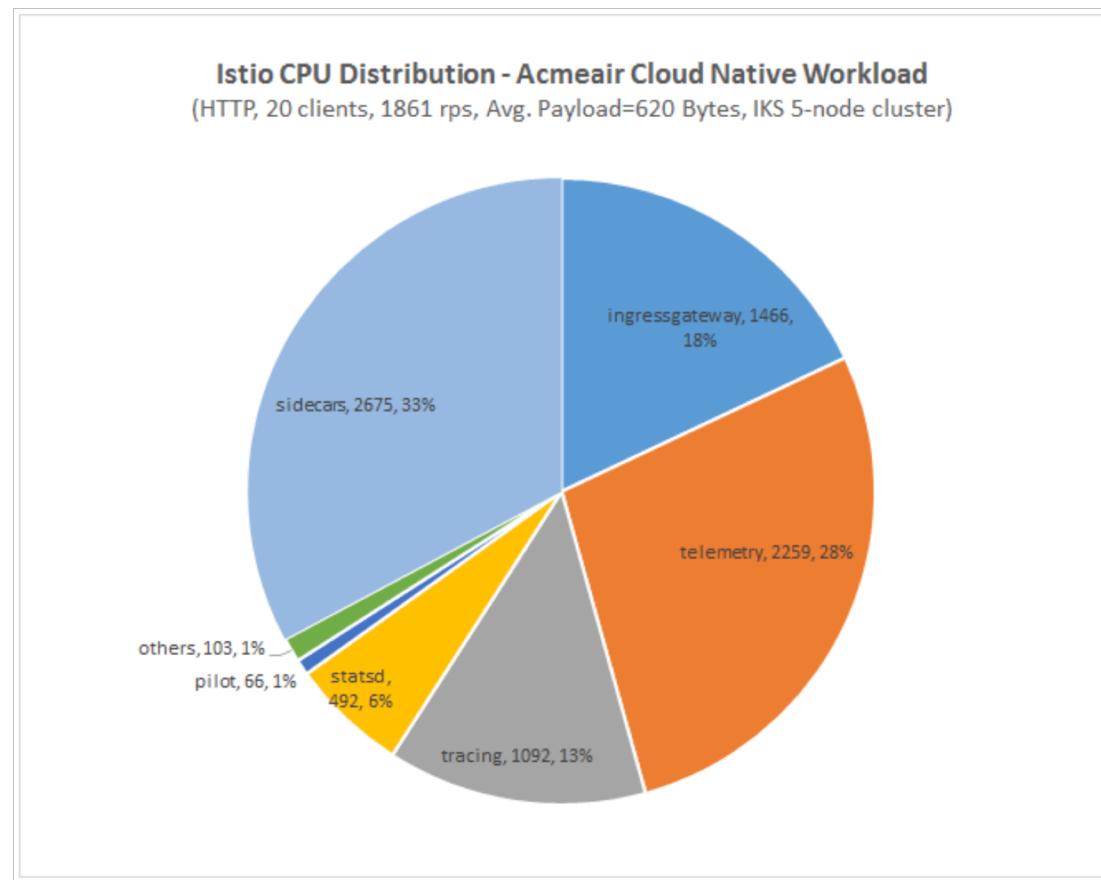
## Performance

# Istio Additional CPU Requirements

**How many mCore CPU resources are required to run 1000 requests per sec with Istio (ACME Air Polyglot)**



# What is consuming?



## Summary

- Istio provides us with a few micro-second extra transaction latency
- There is an incremental increase in required CPU (10%?)
- Istio (the project views this as excessive)
- The project is currently looking to cut these in half



