Istio – An introduction for developers

IBM **Developer**

Containers and Container Orchestration are emerging as key components in Cloud Native architecture

Executable package of software that includes everything needed to run it

Containers

Automate deployment, scaling, and management of containerized applications

Orchestration

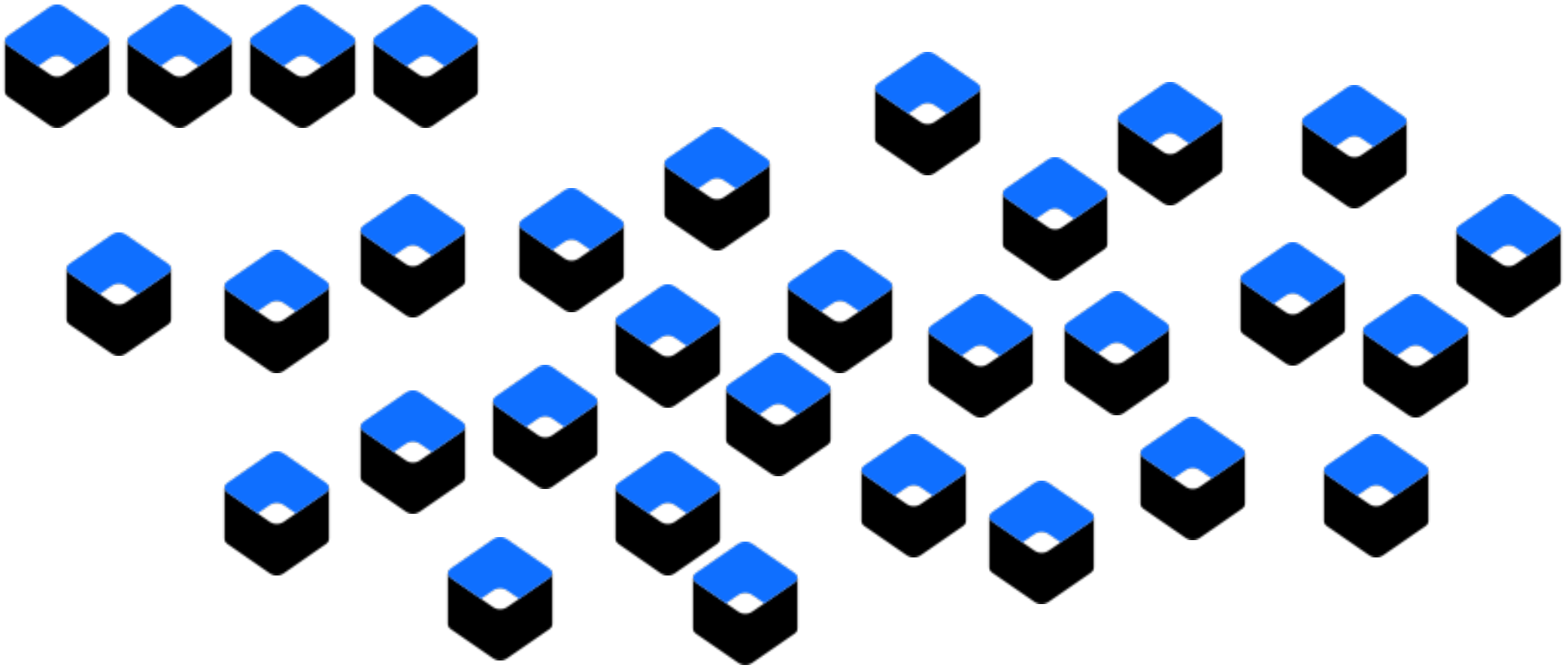
Define, install, and upgrade Kubernetes applications

Management

Manage deployed services with a transparent layer to provide security, routing and monitoring

Service Mesh

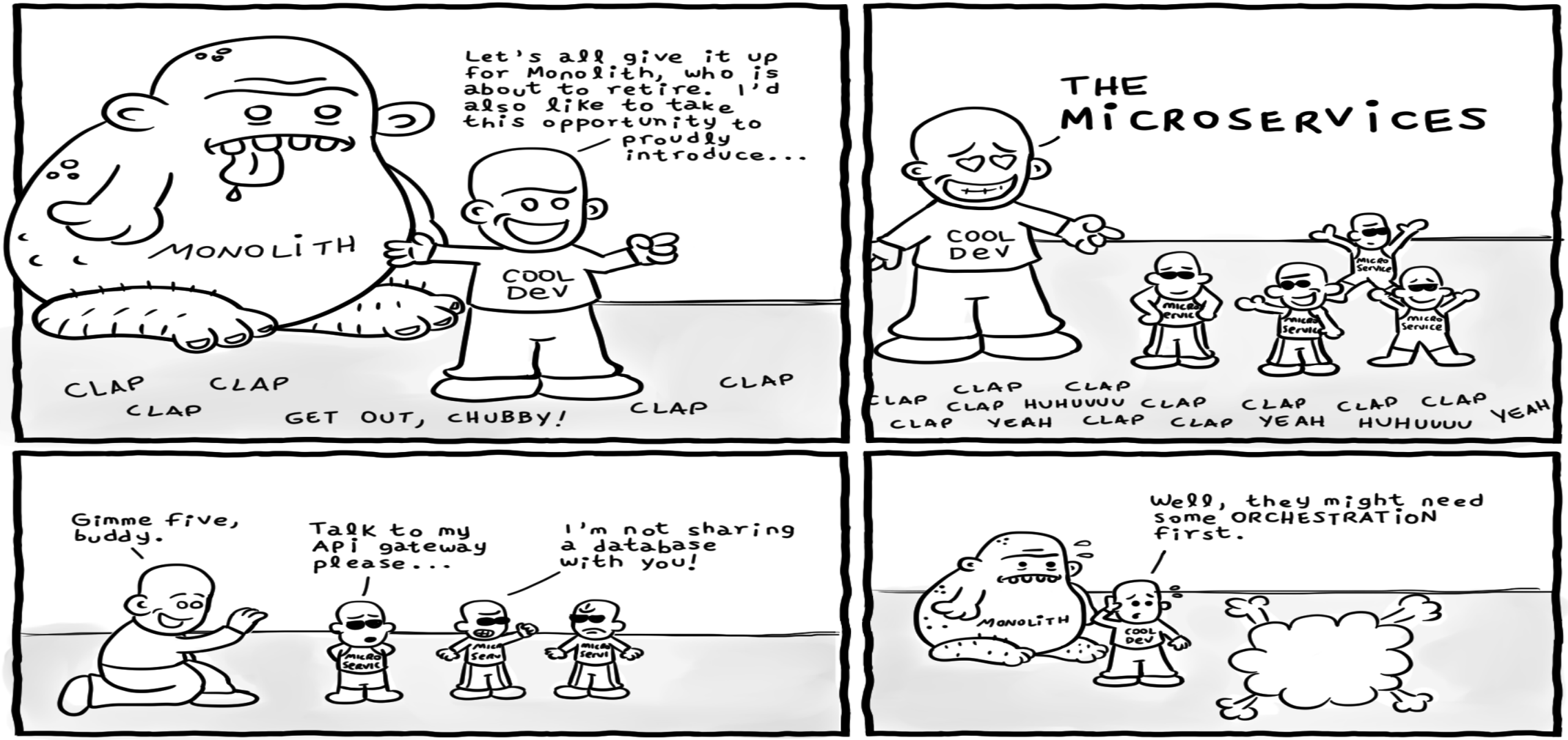
Managing few containers is easy...



But that doesn’t reflect a real workload

Container orchestration with Kubernetes unlocks value of containers as application components

Refactoring to Microservices



Daniel Stori {turnoff.us}

Microservice Challenges

Controlling Traffic
1. How do I do canary testing?
2. How do I A/B testing?

Resilient Services
1. How do I implement circuit breakers?
2. How do I test faults in the system?
3. How do I limit set rate limits for each service?

Telemetry
1. How can I trace a request through my system of multiple services
2. How can I perform monitoring in a distributed deployment?

Security
1. How can I apply policies across services?

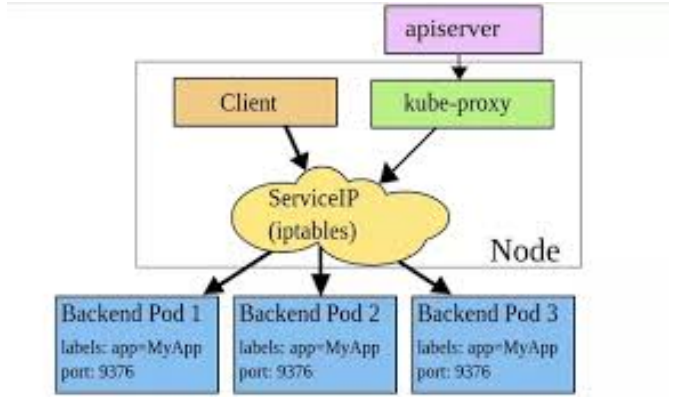
← All of these problems are common across services no matter the runtime.

← It would be cool if we could solve these problems in a standard way **without changing application code**

Kubernetes is great for Microservices…

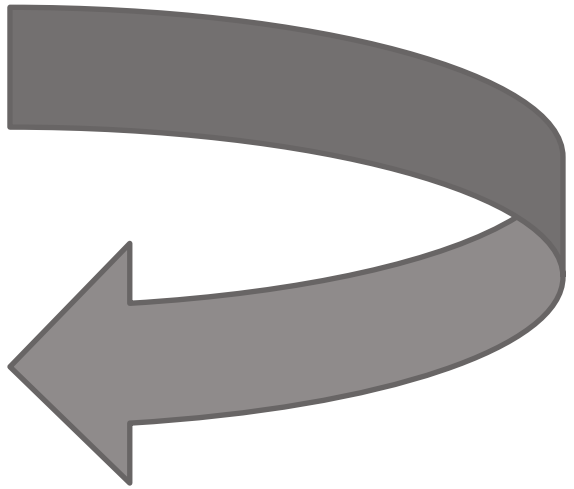
Why do we need a Service mesh and what is it?

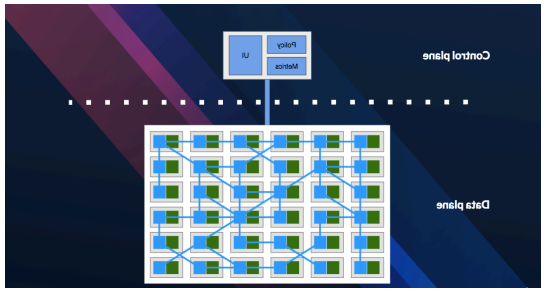
Kubernetes Service Vs Service Mesh



What a k8s Service component cannot do

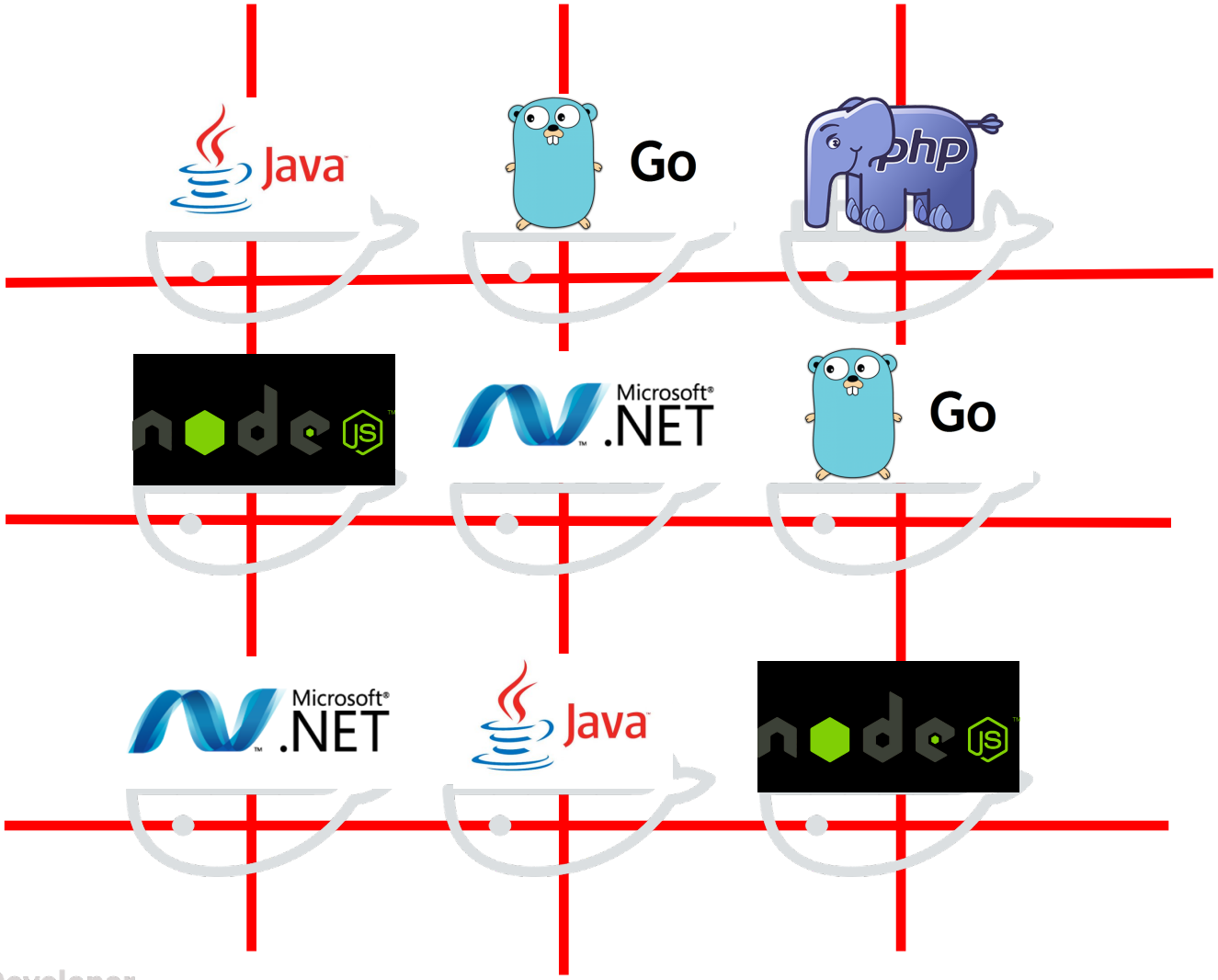
- To get more control of the traffic that goes to this API
- To support many API versions
- Do canary deployments
- Watch and keep track of each request that comes in





- L7 metrics
- Traffic Control - Splitting
- Rate limiting
- Resiliency & Efficiency - Circuit breaking
- Visibility
- Security
- Policy Enforcement .....

Need a no code option (developers will not code this !)



A mesh

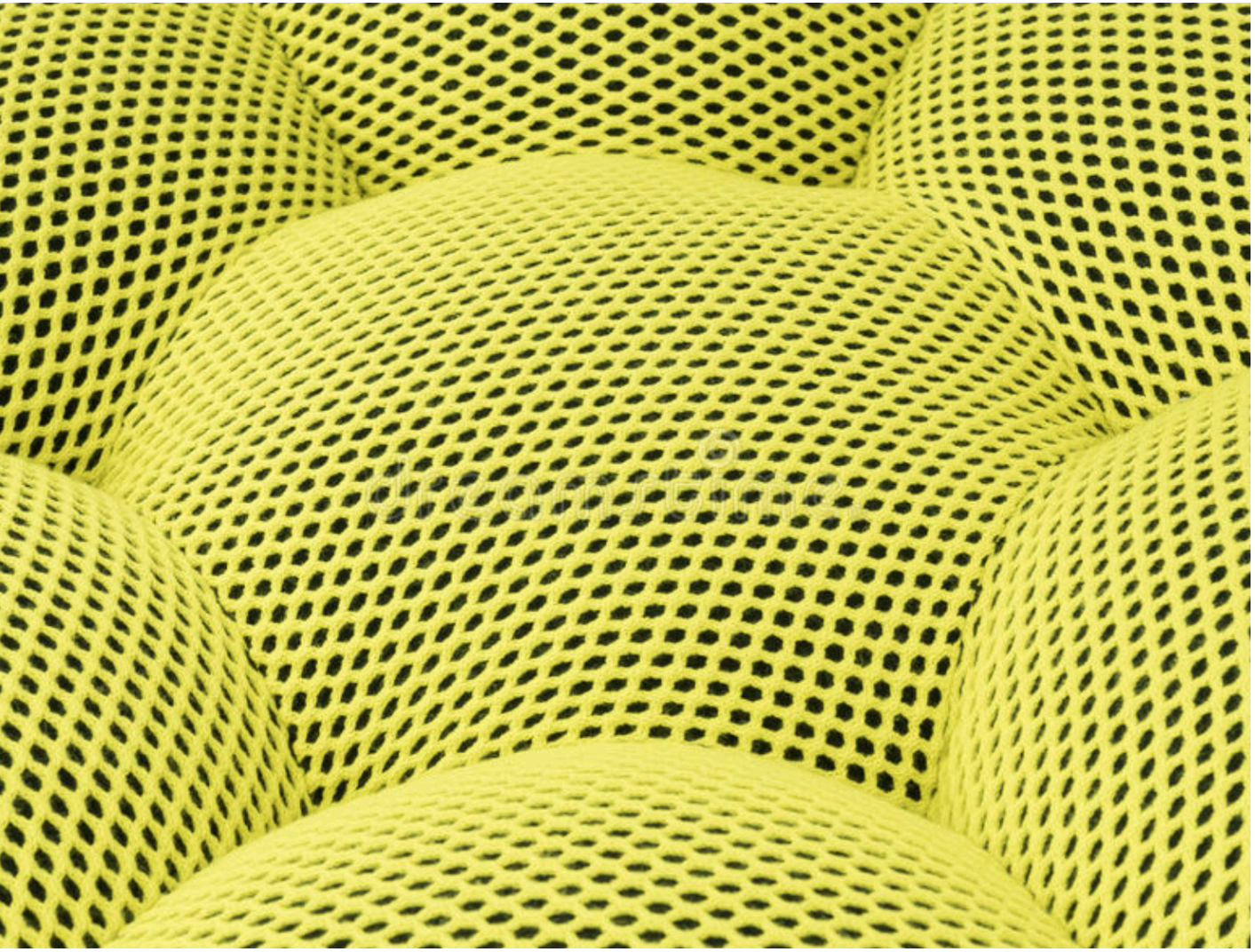
- Touches every microservice
- At infrastructure level

What is a ‘Service Mesh’ ?

A network for services, not bytes

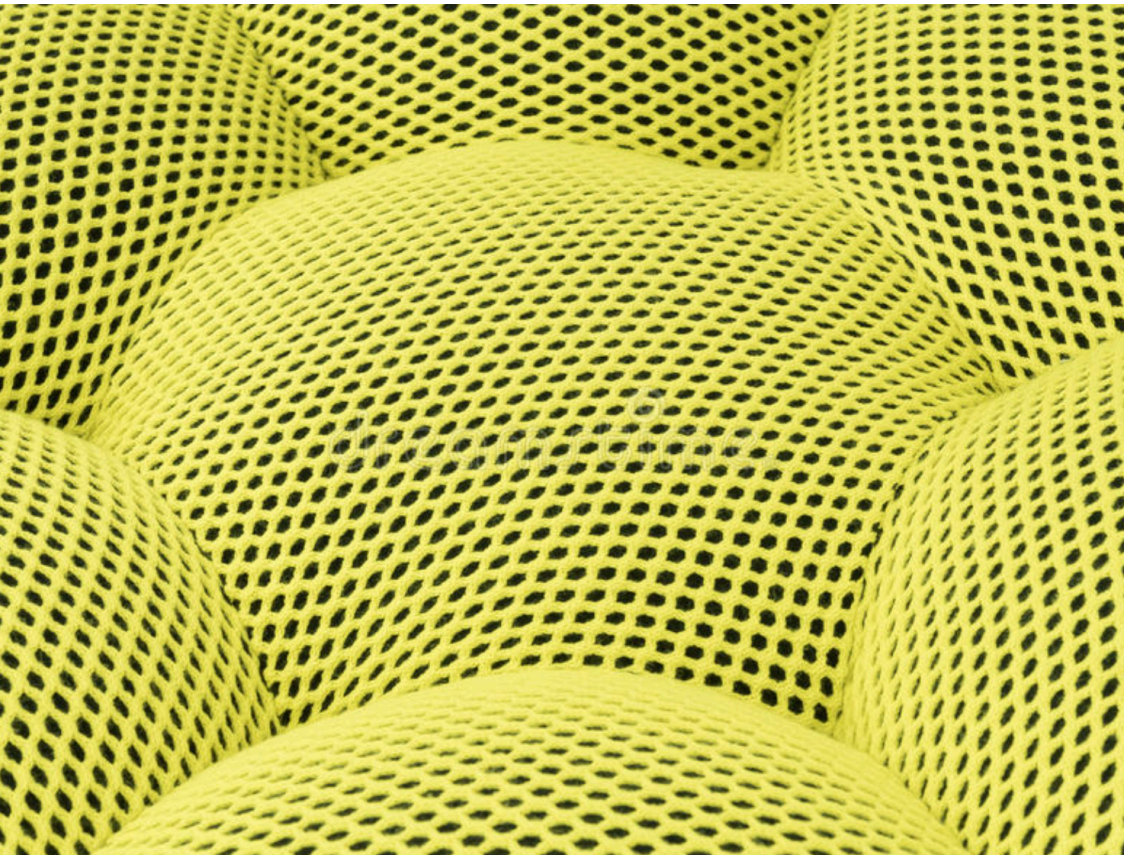
- Visibility
- Resiliency & Efficiency
- Traffic Control
- Security
- Policy Enforcement

What is a service mesh?



A service mesh provides a **transparent** and **language-independent** way to flexibly and easily manage the **communication** between microservices**.**

What is a service mesh?

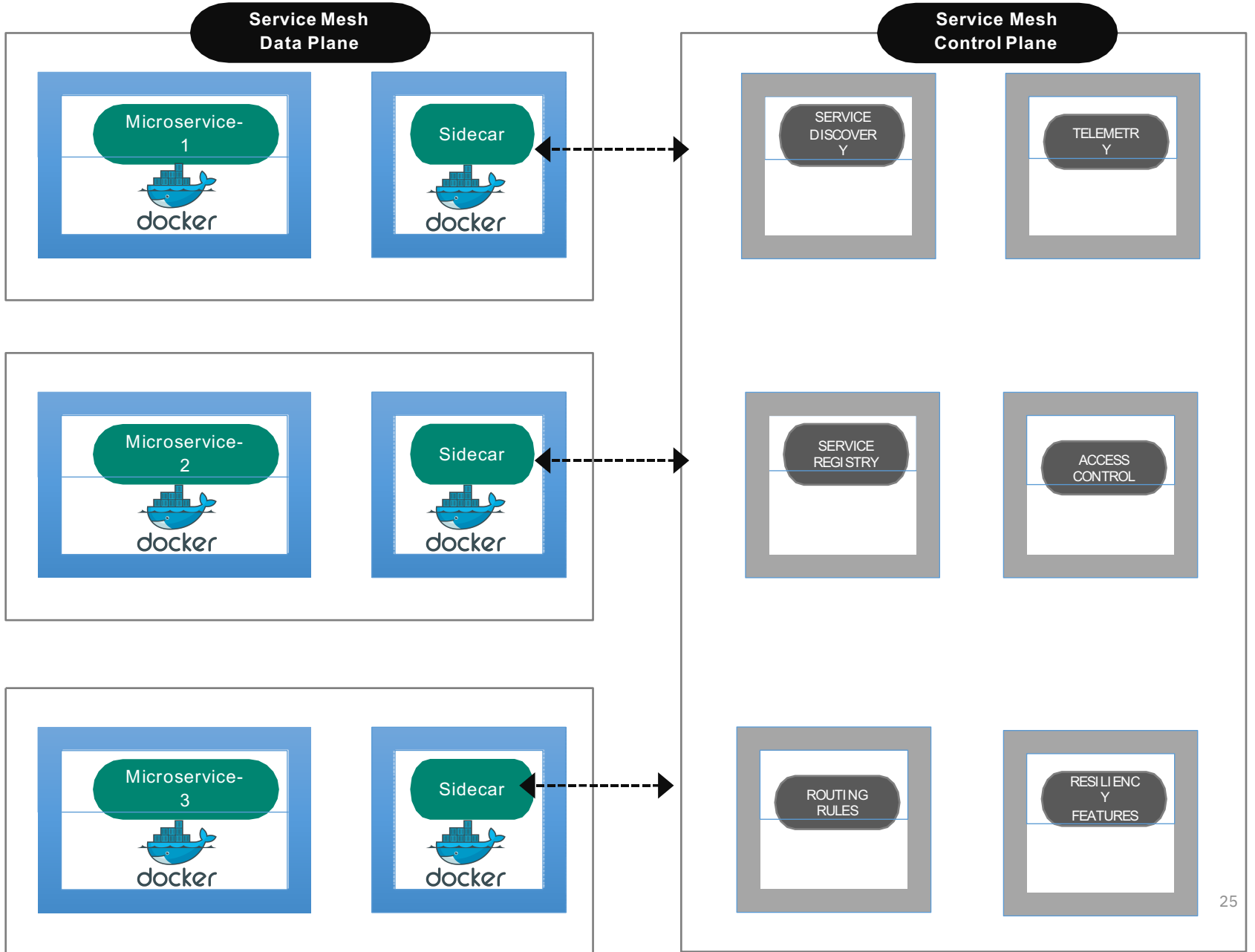


Service meshes can assist you with microservice implementations:

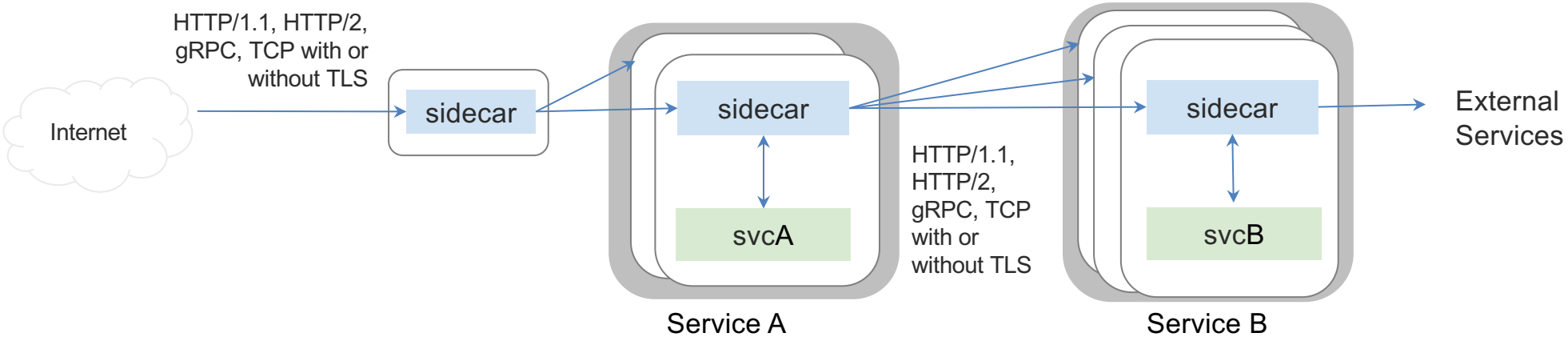
- How can I **find** the service I need?
- How can I **scale** my services?
- How can I test new **versions** of services without impacting new users?
- How can I test against **failures**?
- How can I **secure** service-to-service communication?
- How can I **route** traffic in a specific way?
- How can I test **circuit breaking** or fault injection?
- How can I **monitor** my microservices and collect **metrics**?
- How can I do **tracing**?
- Most importantly, how can I do all these things **without changing individual microservices application code**?

How to build a ‘Service Mesh’ ?

- Lightweight sidecars to manage traffic between services
- Sidecars can do ***much more*** than just load balancing!



Weaving the mesh



Outbound features:

- Service authentication
- Load balancing
- Retry and circuit breaker
- Fine-grained routing
- Telemetry
- Request Tracing
- Fault Injection

Inbound features:

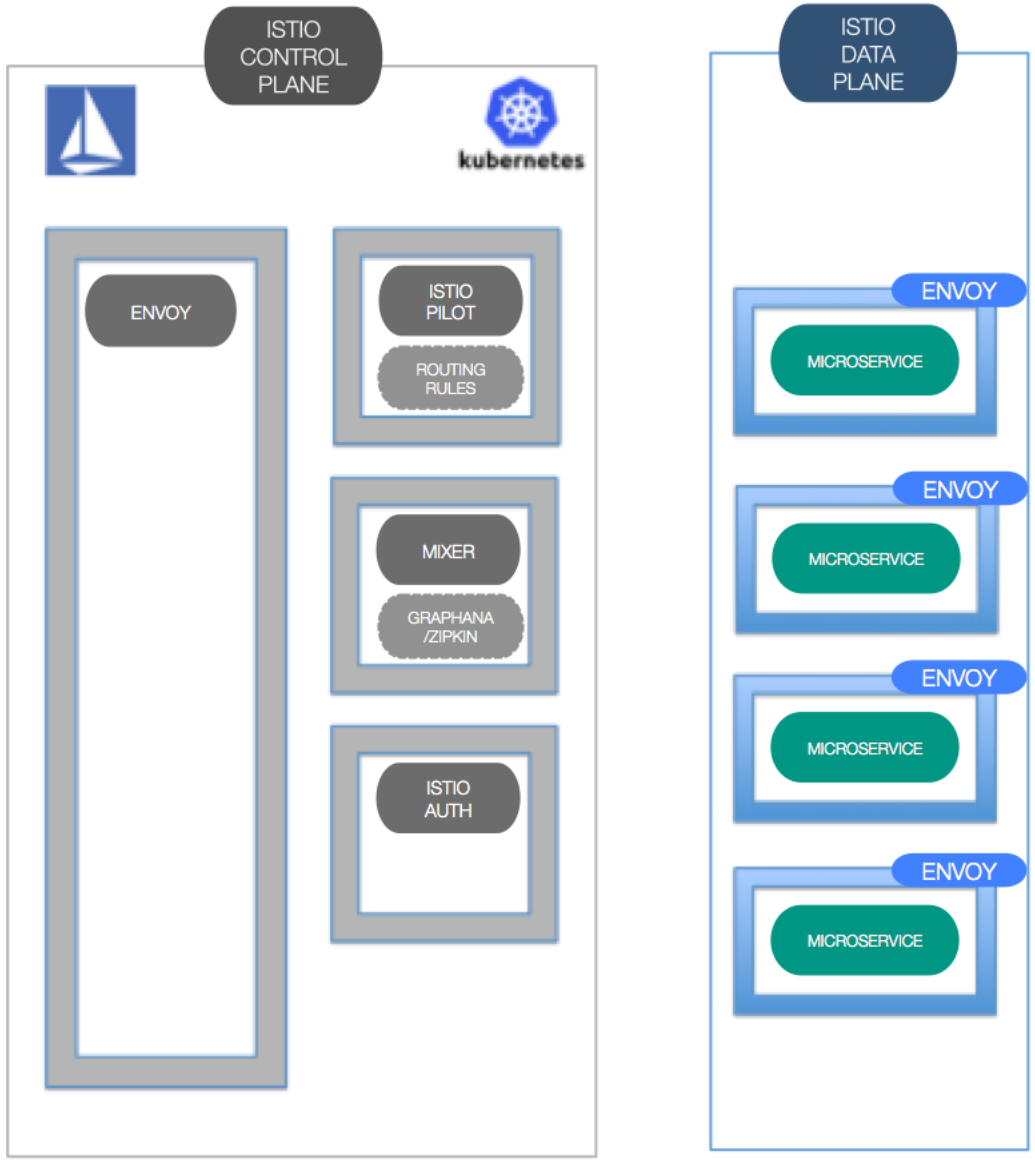
- Service authentication
- Authorization
- Rate limits
- Load shedding
- Telemetry
- Request Tracing
- Fault Injection

Istio - An open platform that provides a uniform way to connect, manage, and secure microservices

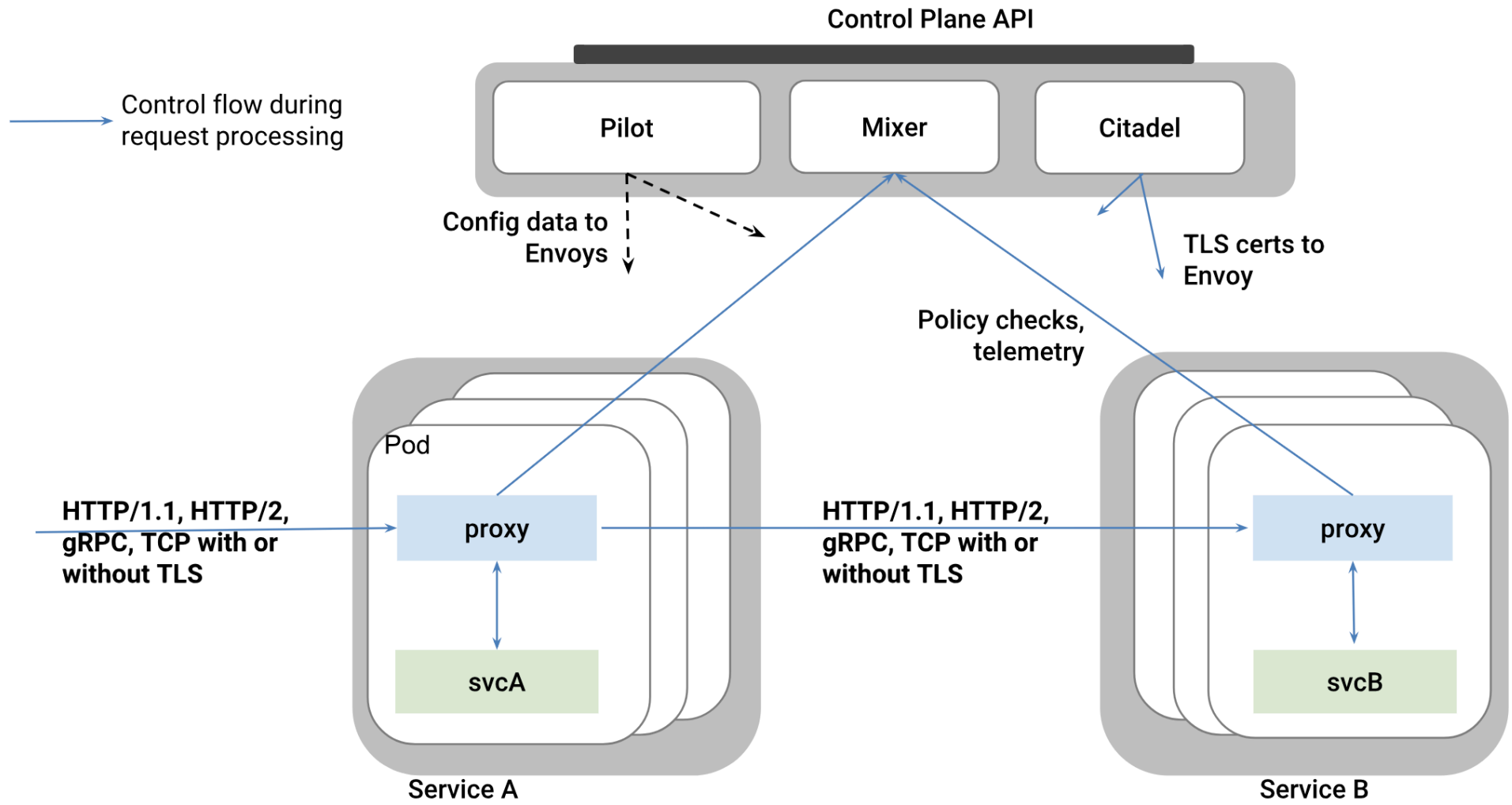
Istio supports managing traffic flows between microservices, enforcing access policies, and aggregating telemetry data, all without requiring changes to the microservice code

Istio Concepts

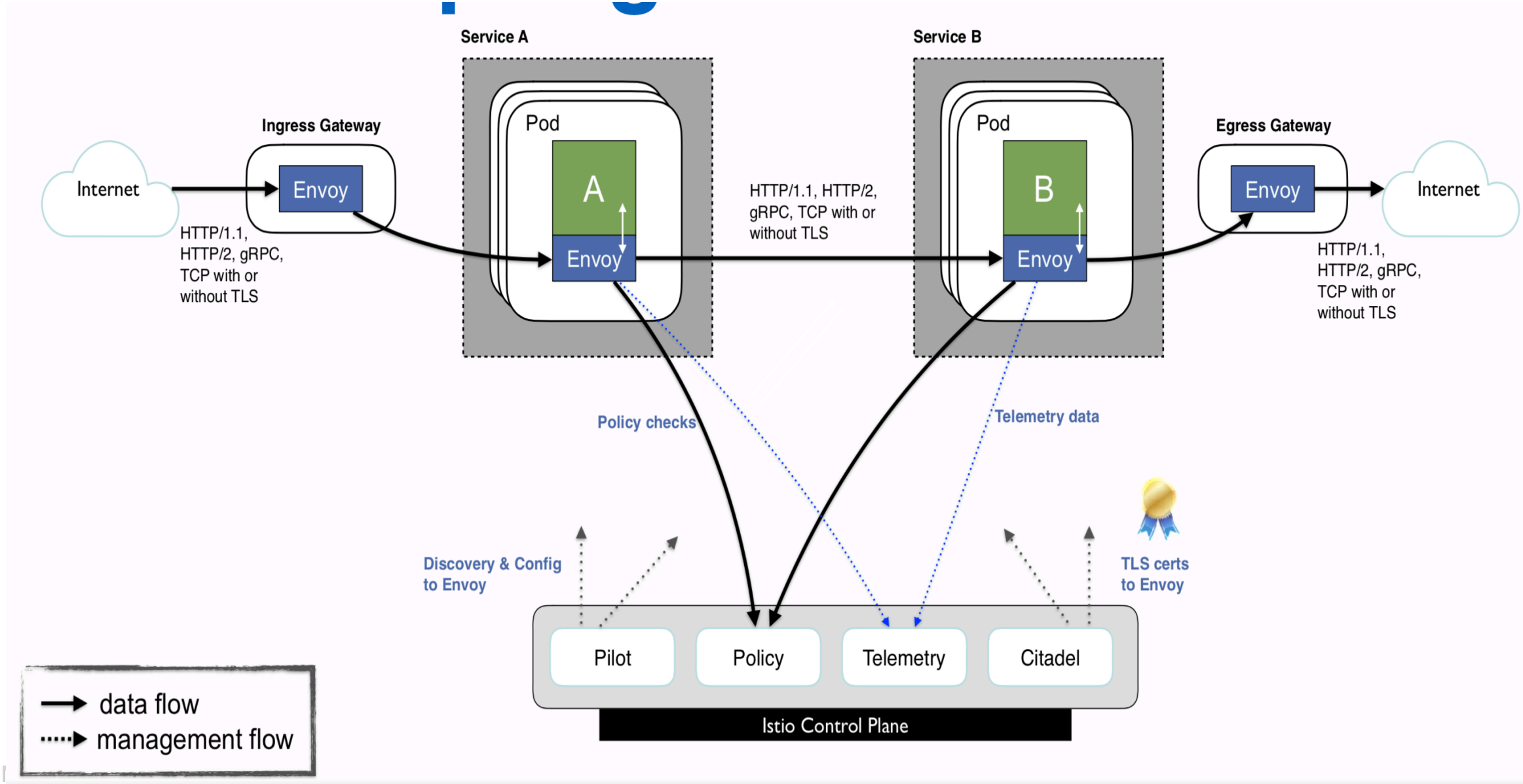
- **Pilot** - "Drives" the service mesh. Configures Istio deployments and propagate configuration to the other components of the system. Routing and resiliency rules go here.
- **Mixer** - Responsible for policy decisions and aggregating telemetry data from the other components in the system using a flexible plugin architecture
- **Proxy** – Based on Envoy, mediates inbound and outbound traffic for all Istio-managed services. It enforces access control and usage policies, and provides rich routing, load balancing, and protocol conversion.
- **Citadel** provides strong service-to-service and end-user authentication, with built-in identity and credential management. Key and certificate management. Distributes as Kubernetes Secrets.



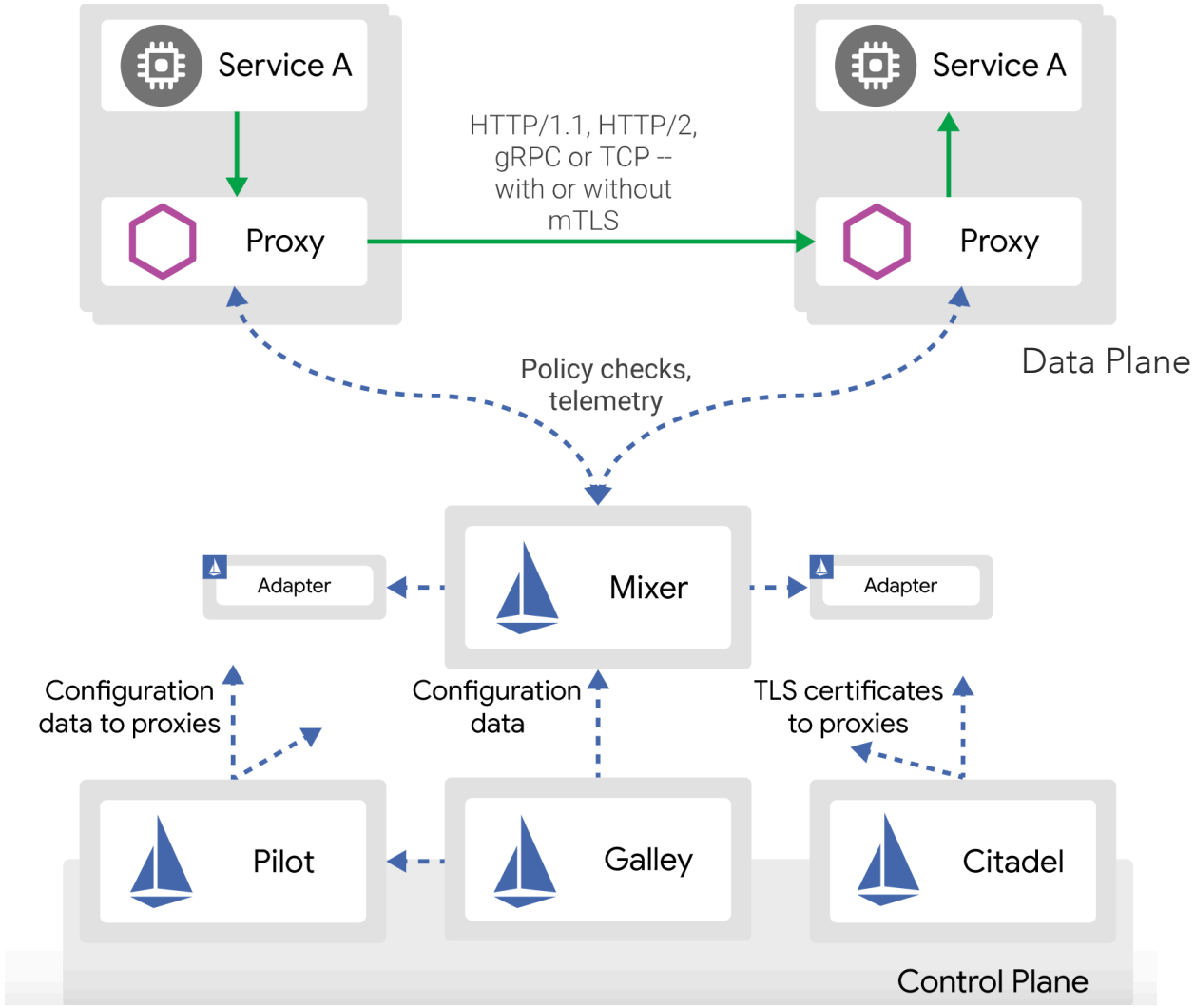
Istio Architecture - Putting it Together



Istio Architecture



Istio Architecture



Data Plane: Set of proxies deployed as sidecars on each microservice

Proxy/Envoy: High-performance multi-protocol proxy that implements policy and monitoring

Control Plane: Manages and configures the proxies in how to route traffic

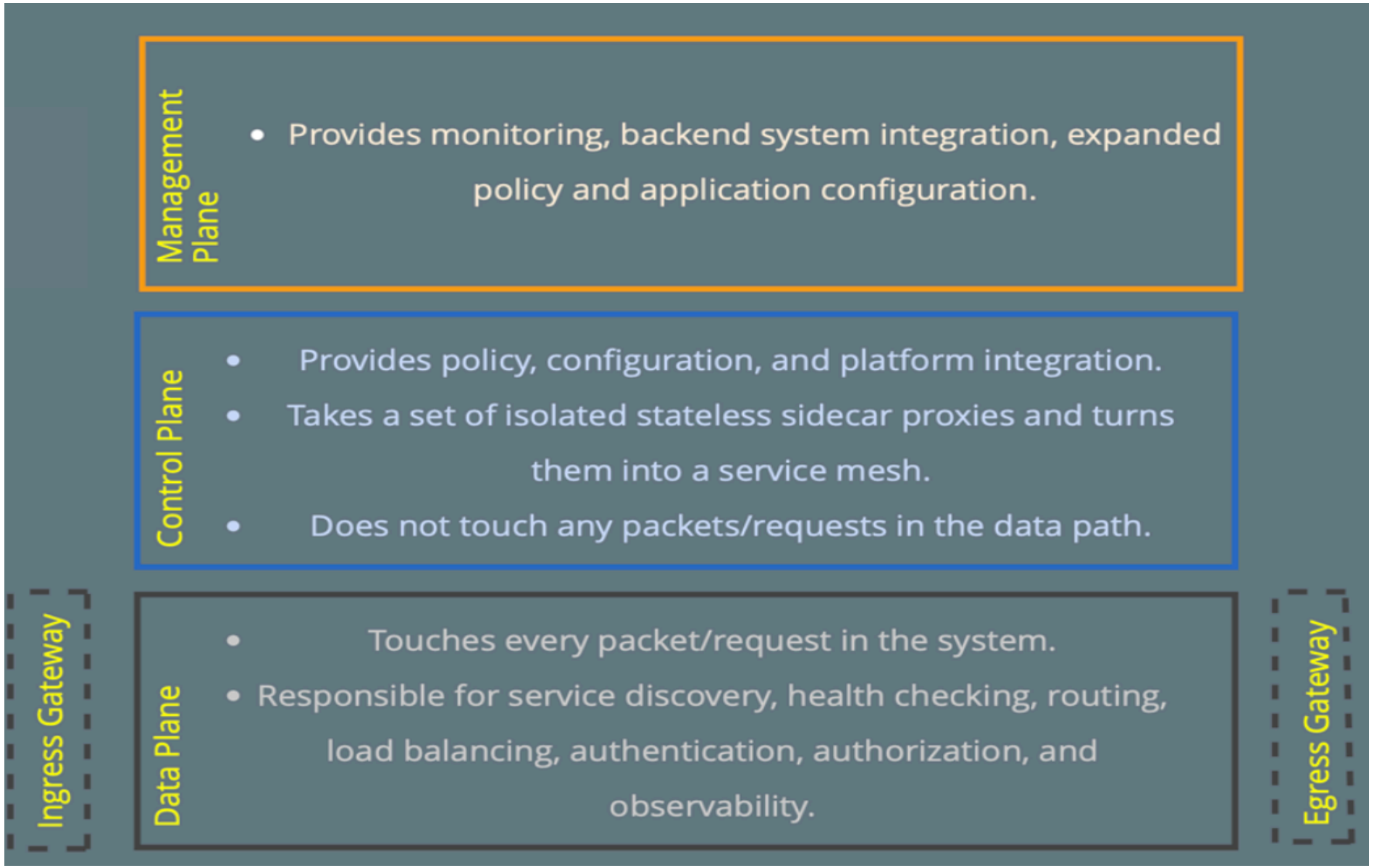
Mixer: Evaluates attributes of requests forwarded from Envoy proxy and collects metrics

Pilot: Service discovery, traffic management (canary, circuit breaker)

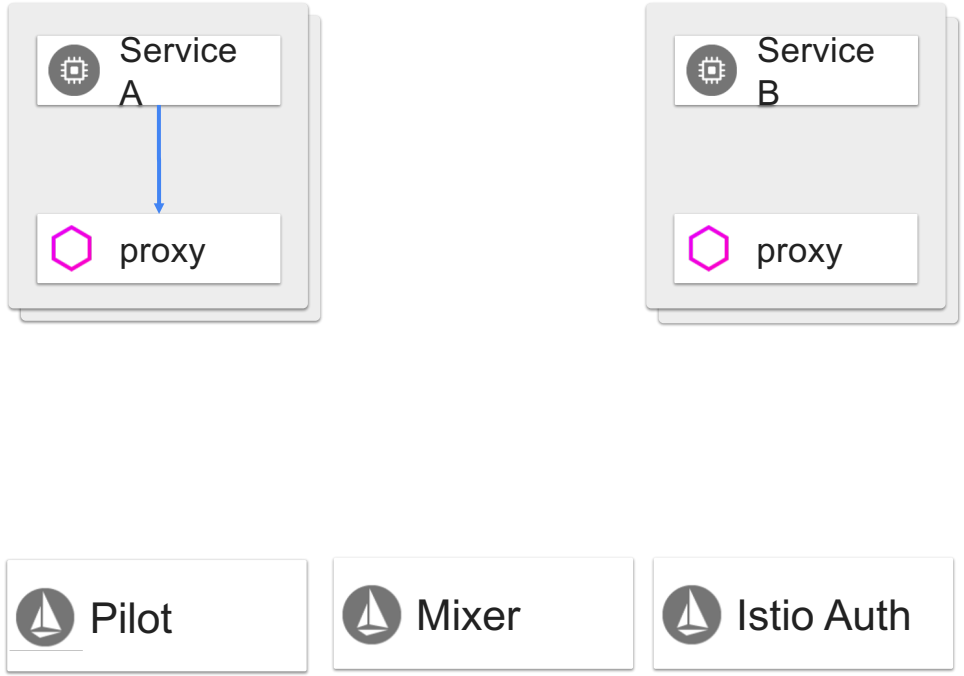
Citadel: Certificate and security management

Galley: Collect and manage configuration from underlying platform

Role of architectural components



Life of a request

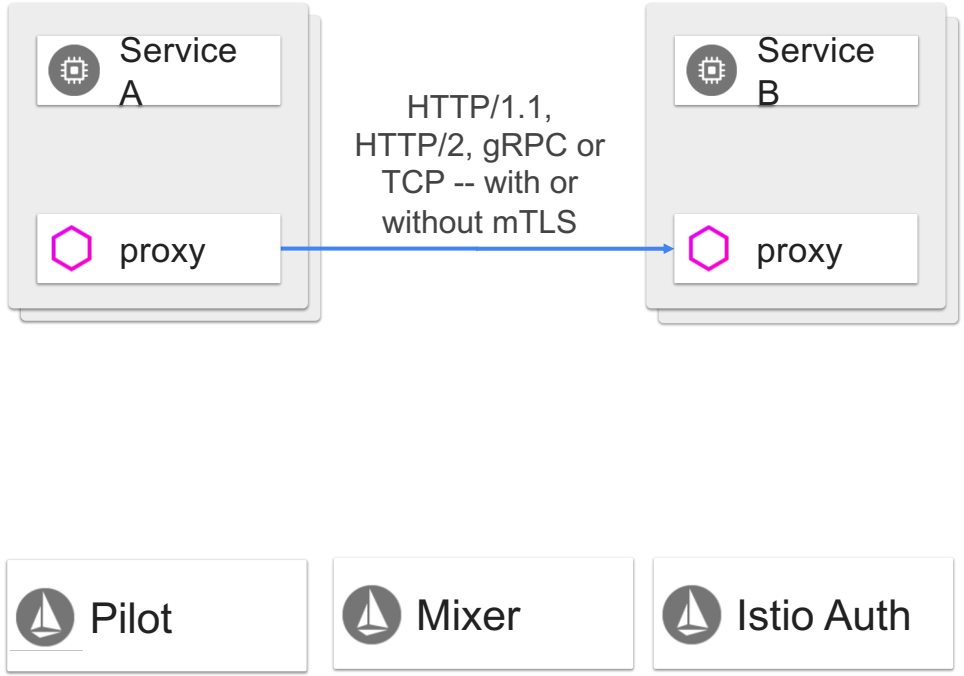


Service A places a call to http://service-b.

Client-side Envoy intercepts the call.

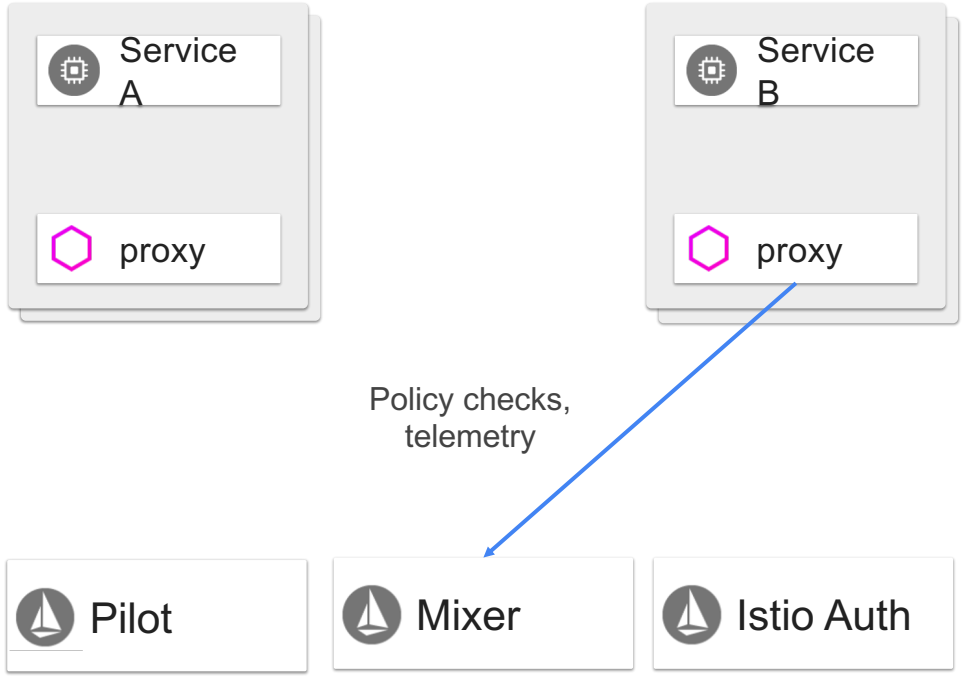
Envoy consults config (previously received from Pilot) to know how/where to route call to service B (taking into account service discovery, load balancing, and routing rules), forwards the call to the right destination.

Life of a request



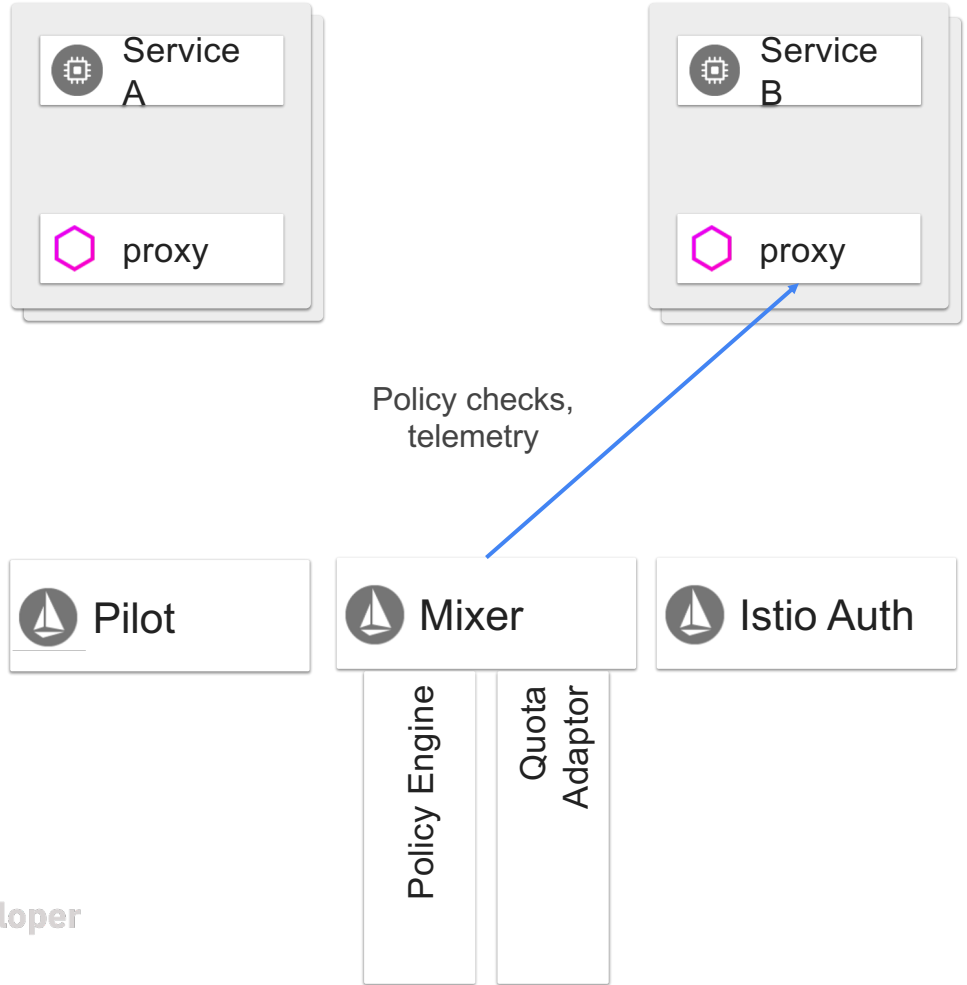
Envoy forwards request to appropriate instance of service B. There, the Envoy proxy deployed with the service intercepts the call.

Life of a request



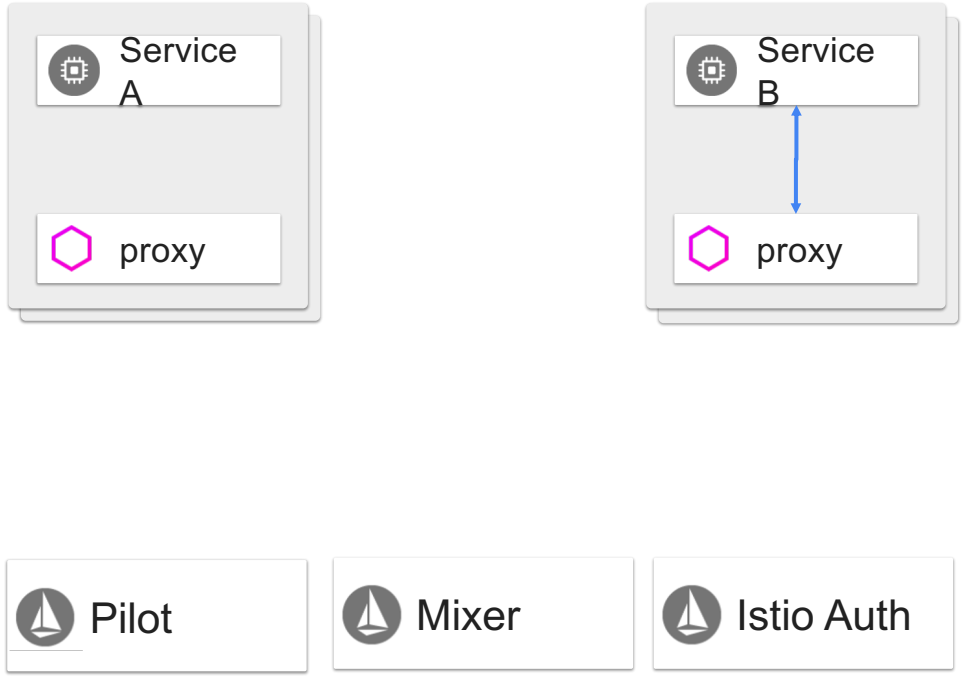
Server-side Envoy checks with Mixer to validate that call should be allowed (ACL check, quota check, etc).

Life of a request



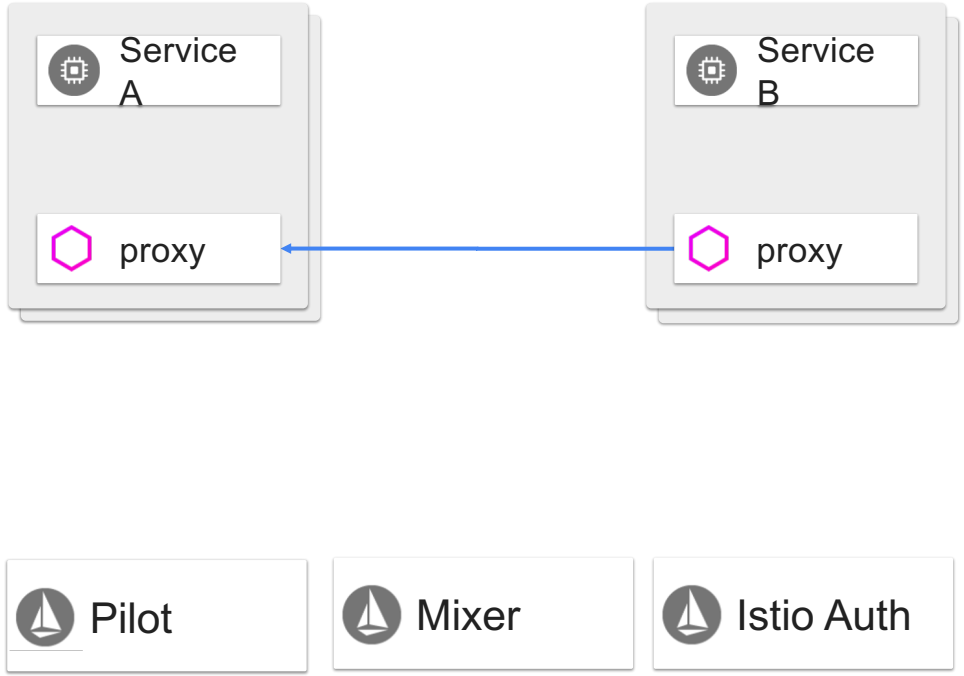
Mixer checks with appropriate adaptors (policy engine, quota adaptor) to verify that the call can proceed and returns true/false to Envoy

Life of a request



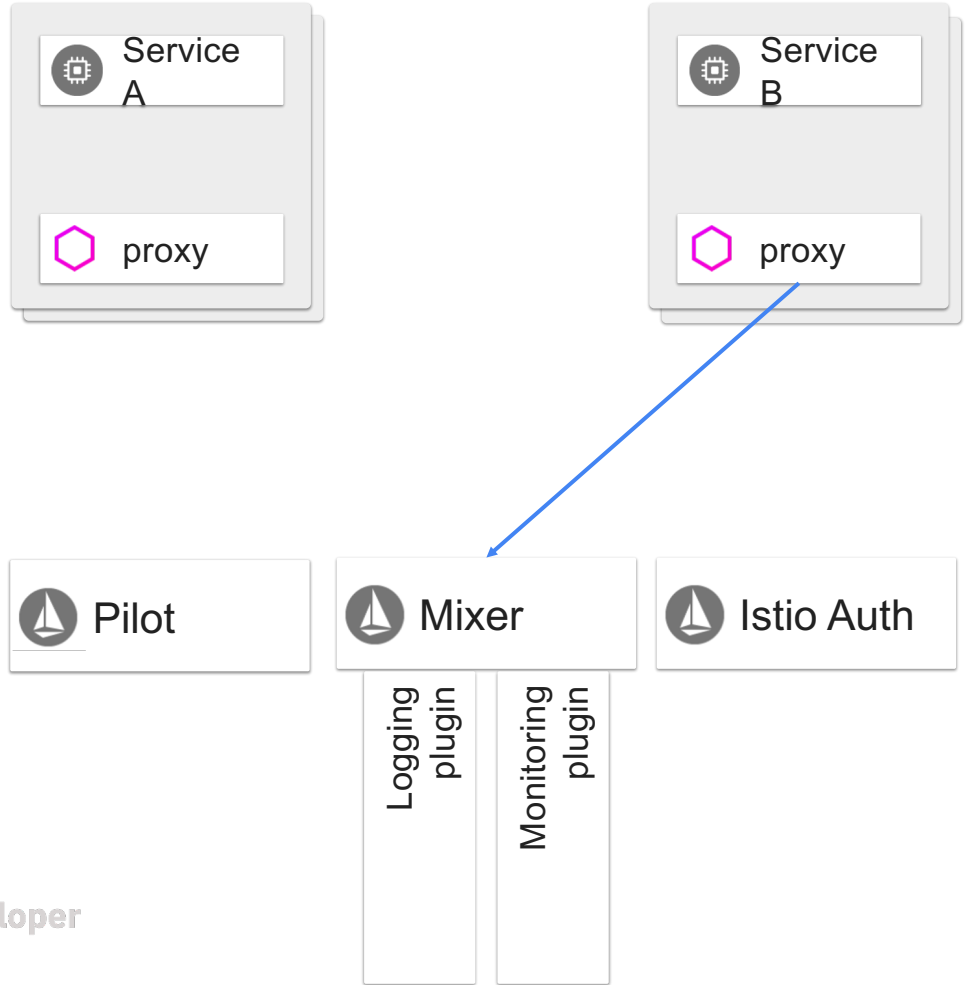
Server-side Envoy forwards request to service B, which process request and returns response

Life of a request



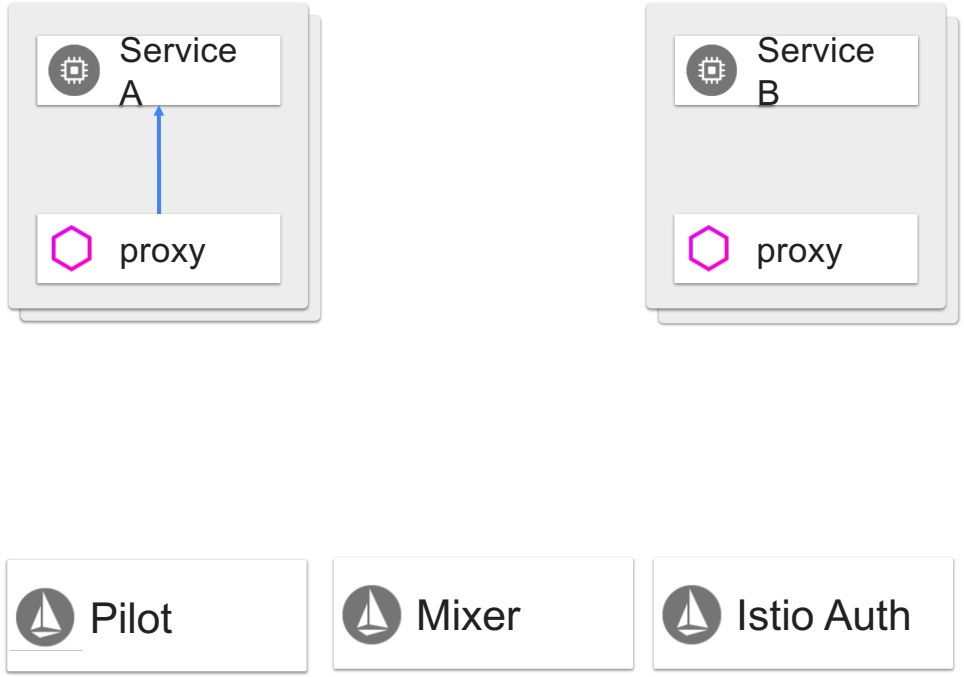
Envoy forwards response to the original caller, where response is intercepted by Envoy on the caller side.

Life of a request



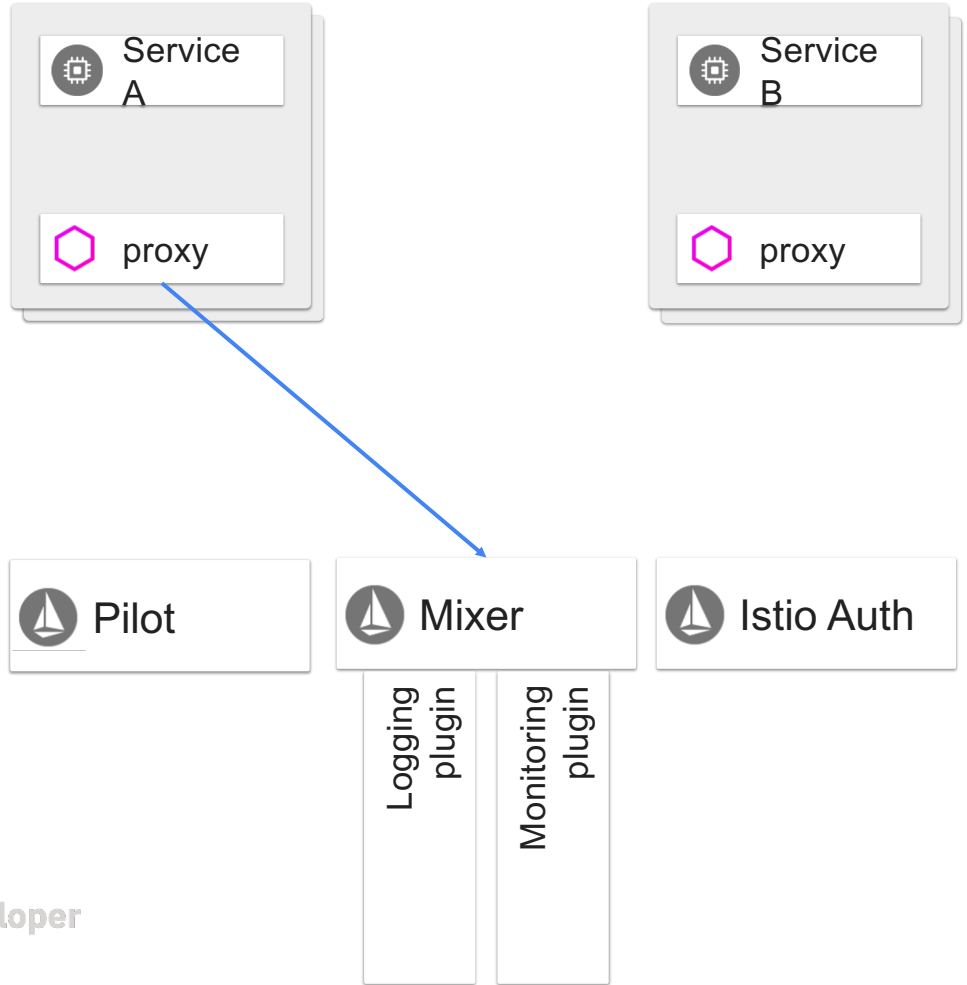
Envoy reports telemetry to Mixer, which in turn notifies appropriate plugins

Life of a request



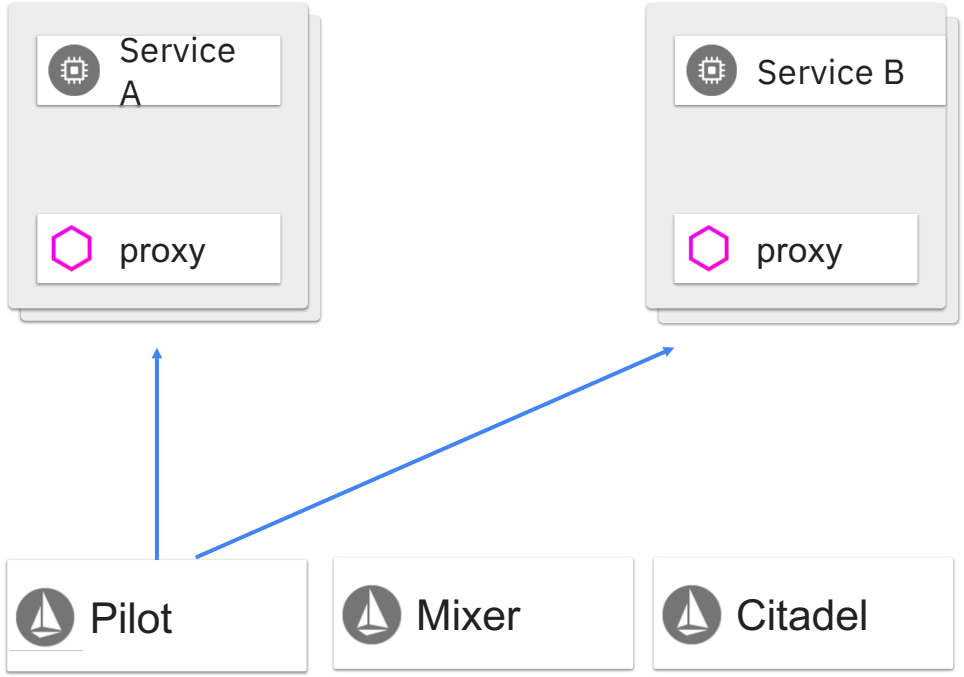
Client-side Envoy forwards response to original caller.

Life of a request



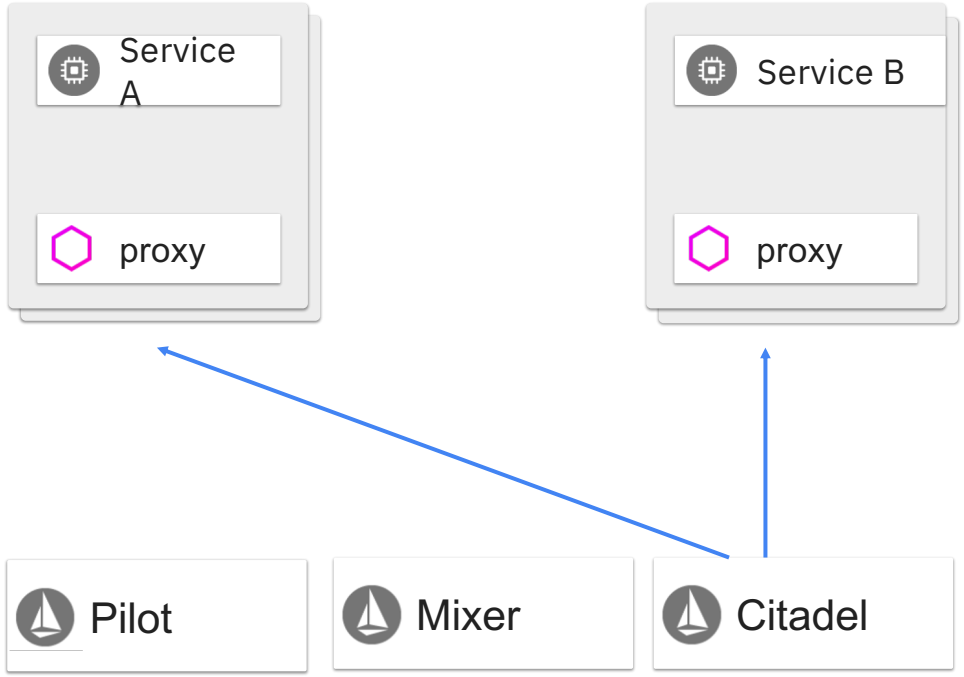
Client-side Envoy reports telemetry to Mixer (including client-perceived latency), which in turn notifies appropriate plugins

At Anytime



Pilot listens to your configuration, such as routing rules, circuit breaking and fault injection. Converts to low-level config (routing rules) and distributes to proxy side cars

At Anytime



Citadel distributes keys and certificates as Kubernetes Secrets available to sidecar containers

Istio Proxy Sidecar

- To create a service mesh with Istio, you update the deployment of the pods to add the Istio Proxy (based on the Lyft Envoy Proxy) as a side car to each pod. The Proxy is then run as a separate container that manages all communication with that pod.

– Manual Approach

– Automated Approach

- Istio modifies the individual pods and not deployments

Manual Approach

- To create a service mesh with Istio, you update the deployment of the pods to add the Istio Proxy (based on the Lyft Envoy Proxy) as a side car to each pod.

```
istioctl kube-inject -f 03hub-deployment.yaml
```

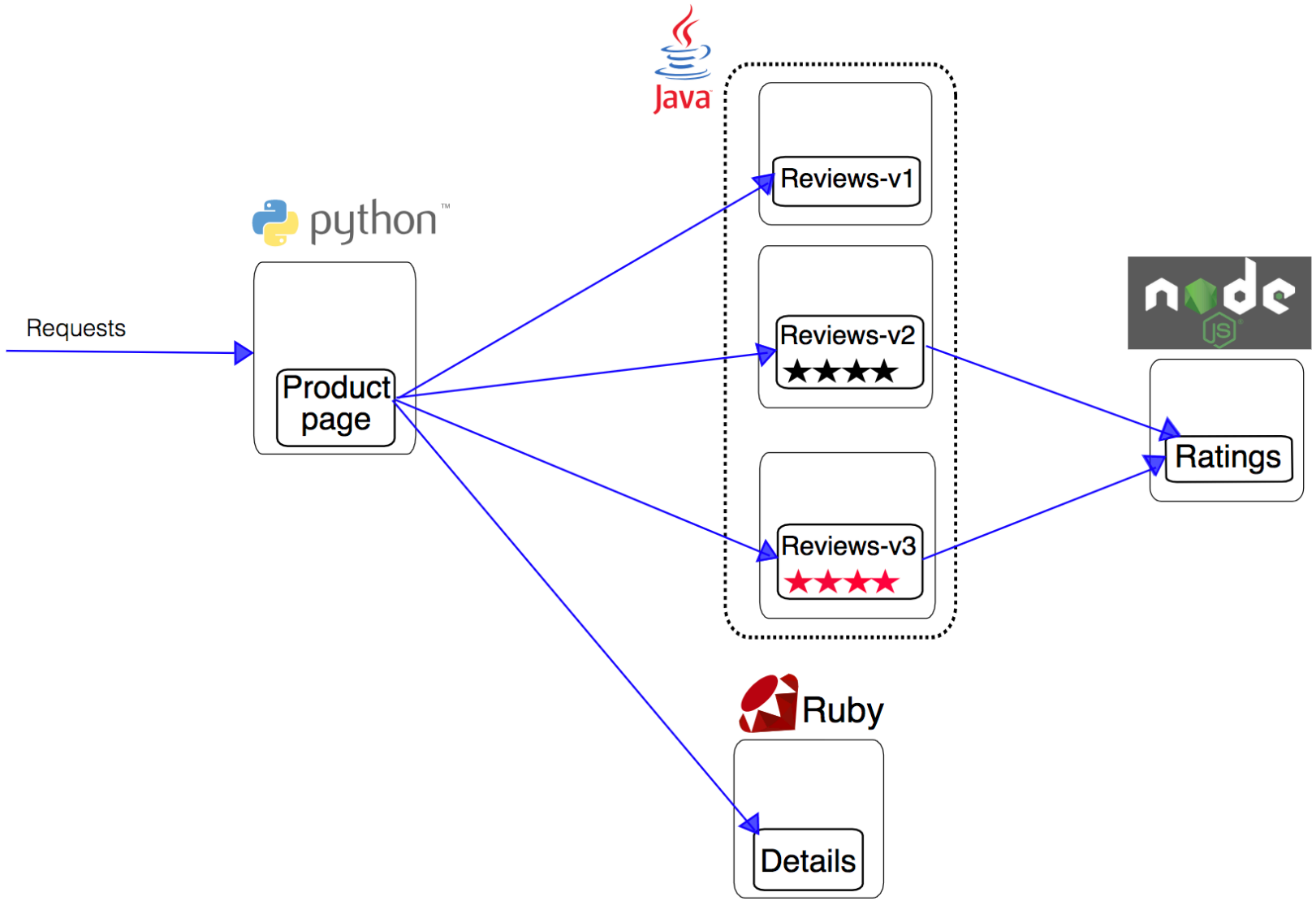
```
image: docker.io/istio/proxyv2:0.8.0
imagePullPolicy: IfNotPresent
name: istio-proxy
resources:
  requests:
    cpu: 100m
    memory: 128Mi
securityContext:
```

Automatic Approach

- Istio sidecars can also be automatically injected into a pod at creation time using a feature in Kubernetes called a mutating webhook admission controller.
- When you install istio, a deployment is automatically created called istio-sidecar-injector-* that injects proxy side-cars automatically to pods installed on namespaces that have a label istio-injection=enabled

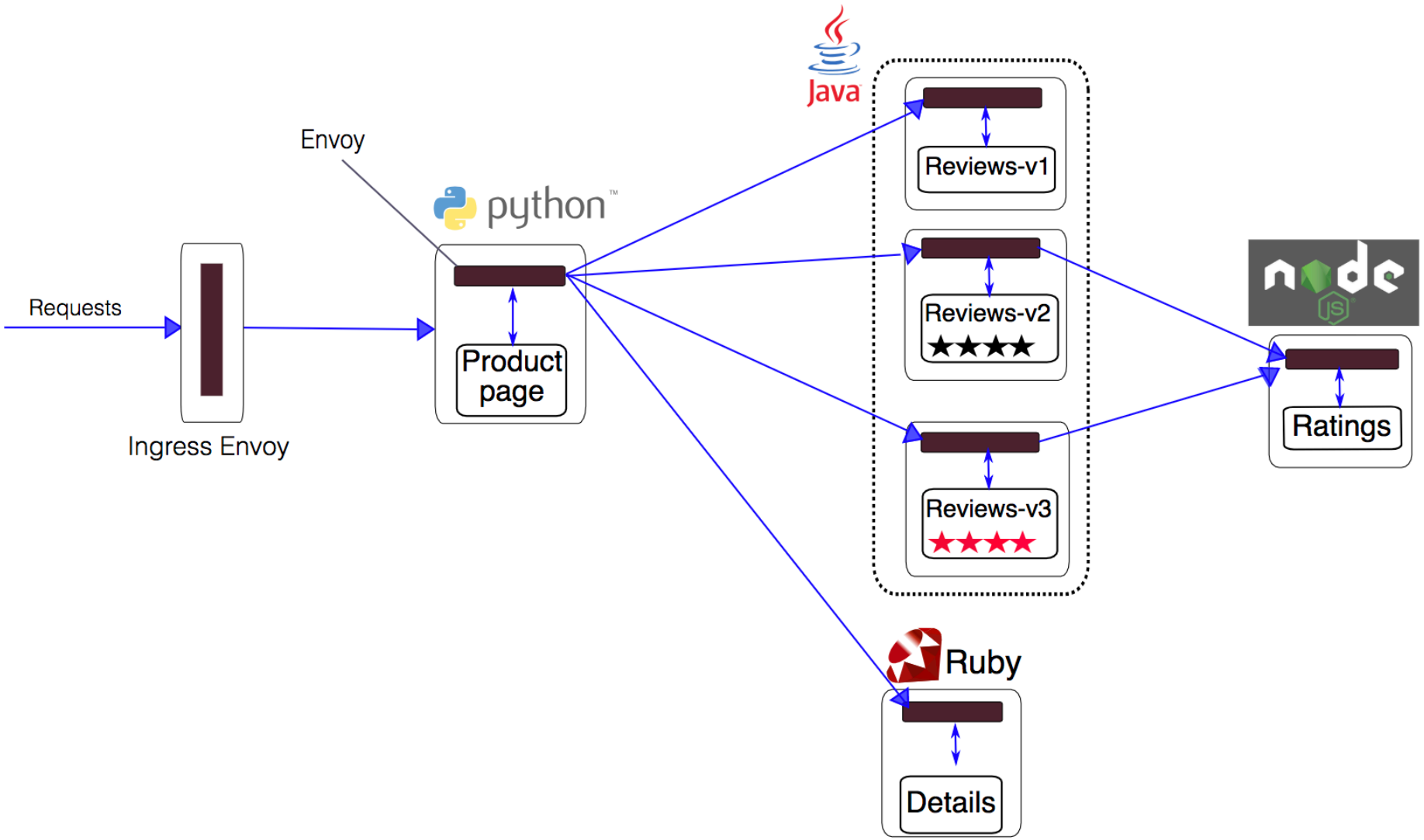
```
kubectl label namespace default istio-injection=enabled
```

Bookinfo Sample Microservices Application (without Istio)



Bookinfo Application without Istio

Bookinfo Sample Microservices Application (with Istio)



Bookinfo Application

Resiliency

- Istio adds fault tolerance to your application without any changes to code

```
// Circuit breakers
destination: serviceB.example.cluster.local
policy:
- tags:
    version: v1
  circuitBreaker
  : simpleCb:
     maxConnections: 100
     httpMaxRequests: 1000
     httpMaxRequestsPerConnection: 10
     httpConsecutiveErrors: 7
     sleepWindow: 15m
     httpDetectionInterval:
     5m
```

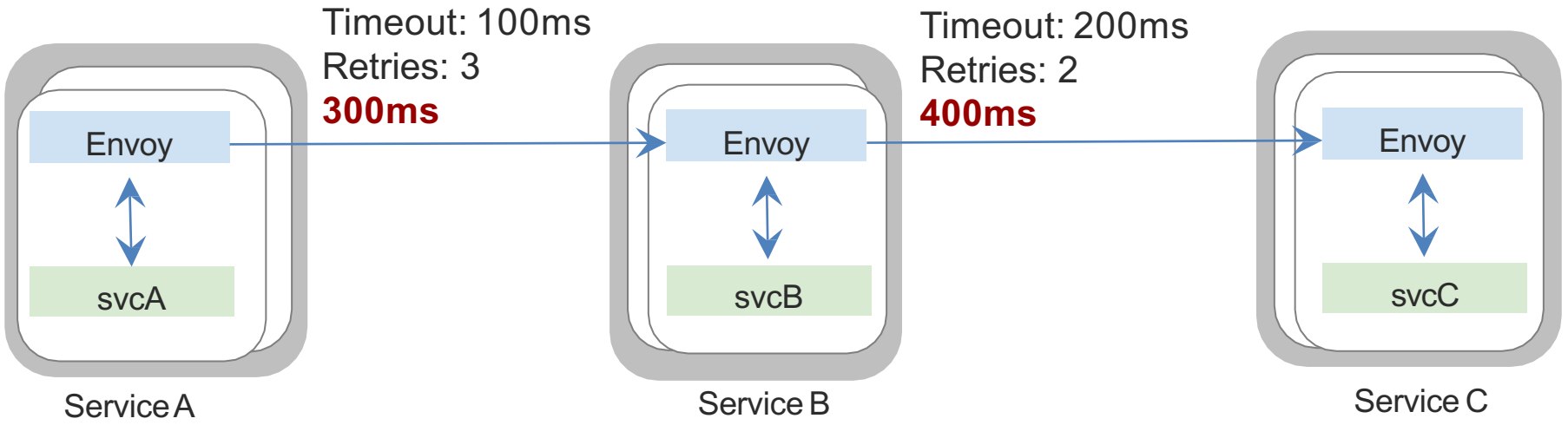
Resilience features

- Timeouts
- Retries with timeout budget
- Circuit breakers
- Health checks
- AZ-aware load balancing w/ automatic failover
- Control connection pool size and request load
- Systematic fault injection

Resiliency Testing

Systematic fault injection to identify weaknesses in failure recovery policies

- HTTP/gRPC error codes
- Delay injection



Istio Ingress Controller

Control Ingress Traffic

- **Gateway** describes a load balancer operating at the edge of the mesh receiving incoming or outgoing HTTP/TCP connections.
- **VirtualService** defines a set of traffic routing rules to apply when a host is addressed. Each routing rule defines matching criteria for traffic of a specific protocol.
- Istio ingress controller gets exposed as a normal Kubernetes service load balancer on port 80.

```
kubectl get svc -n istio-system
```

https://istio.io/docs/tasks/traffic-management/ingress/

Traffic Management
&
Canary Deployments

Core Components

The core component used for traffic management in Istio is Pilot, which manages and configures all the Envoy proxy instances deployed in a particular Istio service mesh.

VirtualService defines a set of traffic routing rules to apply when a host is addressed. Each routing rule defines matching criteria for traffic of a specific protocol. If the traffic is matched, then it is sent to a named destination service (or subset or version of it) defined in **DestinationRule**.

DestinationRule defines policies that apply to traffic intended for a service after routing has occurred. Any destination host and subset referenced in a VirtualService rule must be defined in a corresponding DestinationRule.

ServiceEntry configuration enables services within the mesh to access a service not necessarily managed by Istio. The rule describes the endpoints, ports and protocols of a white-listed set of mesh-external domains and IP blocks that services in the mesh are allowed to access. For example for accessing a Watson service.

https://istio.io/docs/reference/config/istio.networking.v1alpha3/

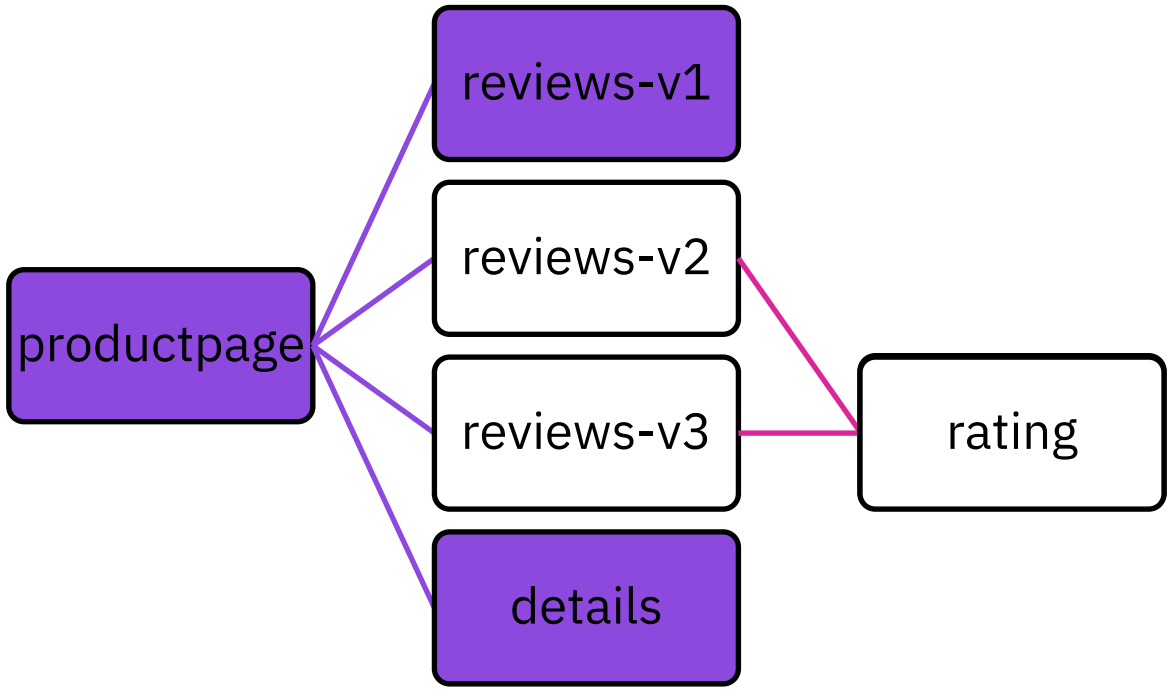
Possible Use Cases

- A/B Testing
- Canary Deployments
- Route based on header
- Fault Injection
- HTTP Retry
- Rate Limits/Polices
- Circuit Breaker

A/B Testing

- A/B testing is a method of performing identical tests against two separate service versions in order to determine which performs better.

- Enforce only v1 of reviews microservice



A/B Testing

- A/B testing is a method of performing identical tests against two separate service versions in order to determine which performs better.

- Enforce only v1 of reviews microservice

```
istioctl create -f 02route-rule-reviews-v1.yam
```

02route-rule-reviews-v1.yaml

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

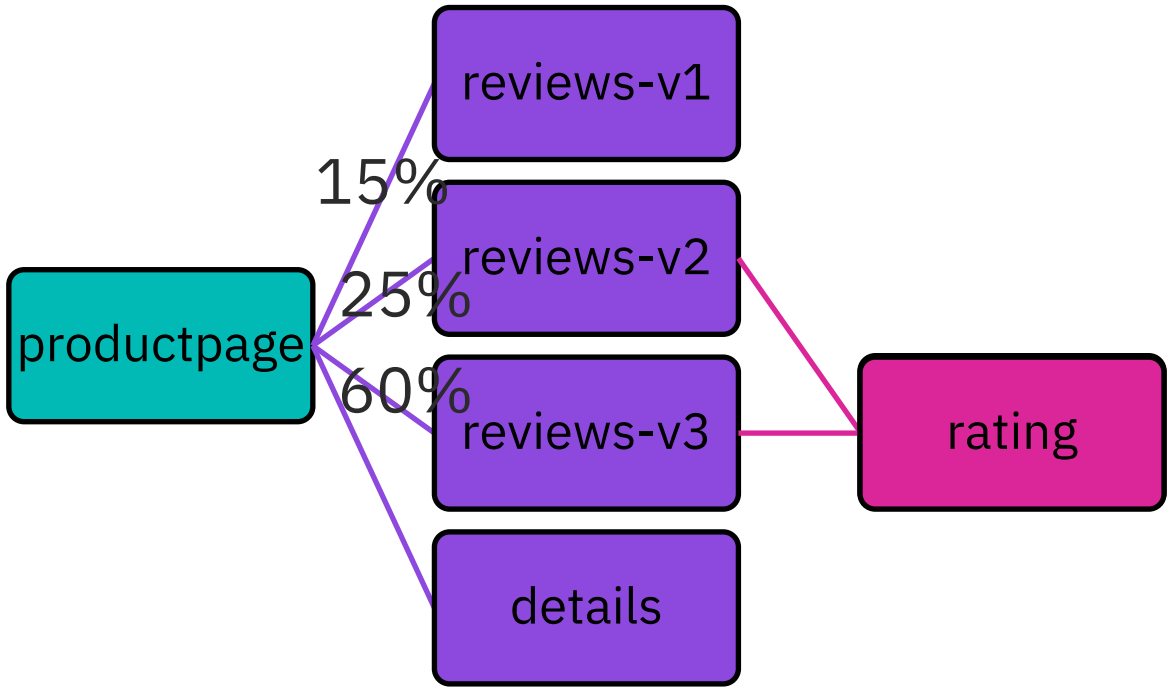
Canary Deployments

- You can do Canary Deployments with Kubernetes replicasets alone, but it’s quite complex.
- Maintaining canary traffic at small percentages requires many replicas (e.g., 1% would require a minimum of 100 replicas).
- Istio’s routing rules also provide other important advantages; you can easily control fine grain traffic percentages (e.g., route 1% of traffic without requiring 100 pods) and you can control traffic using other criteria (e.g., route traffic for specific users to the canary version).
- All the traffic inside the pod is captured by iptables commands and directed to the sidecar proxy. Then the sidecar proxy performs routing, according to routing tables it receives from Istio Pilot (a part of the Istio Control Plane)

Canary Deployments (Cont'd)

Enforce the following:

- 15% for reviews-v1
- 25% for reviews-v2
- 60% for reviews-v3



Canary Deployments (Cont'd)

Enforce the following:

- 15% for reviews-v1
- 25% for reviews-v2
- 60% for reviews-v3

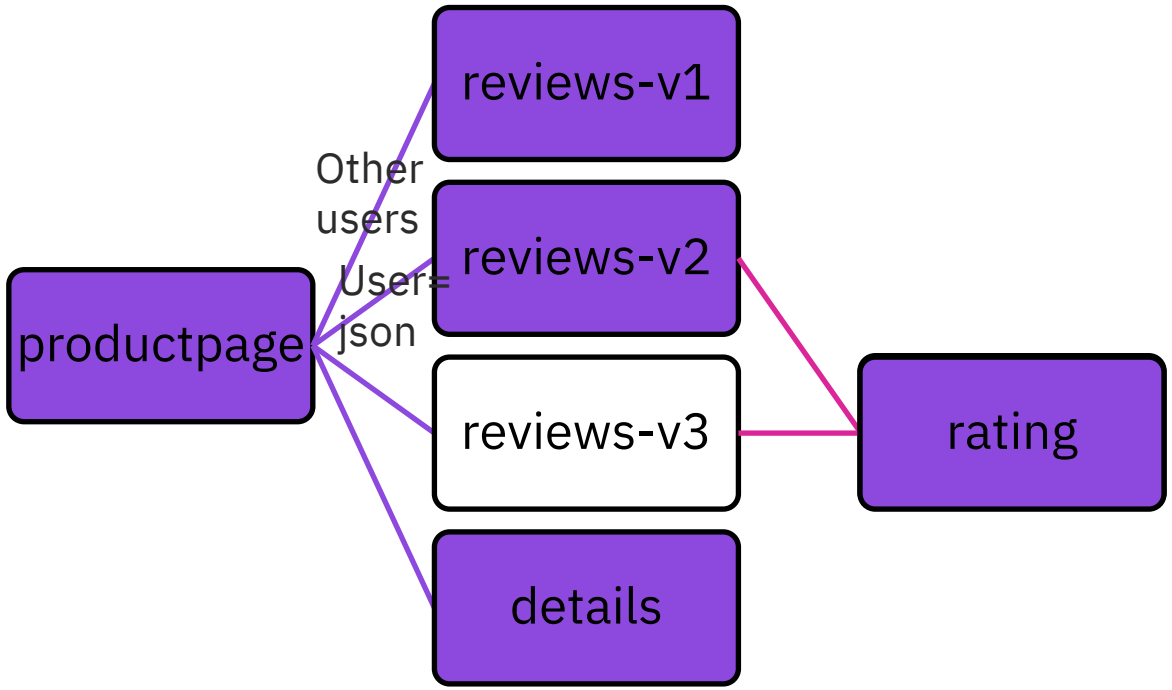
```
istioctl replace -f 03route-rule-
reviews-canary.yaml
```

03route-rule-reviews-canary.yaml

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
      weight: 15
    - destination:
        host: reviews
        subset: v2
      weight: 25
    - destination:
        host: reviews
        subset: v3
      weight: 60
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v3
    labels:
      version: v3
```

Route based on Header

- You can route to microservices based on header (i.e. based on users, geography, browser, mobile,.. etc).
- **Example:** Route requests based on cookie (user=jason) to reviews-v2, else to reviews-v1.



Route based on Header

You can route to microservices based on header (i.e. based on users, geography, browser, mobile,.. etc).

Example: Route requests based on cookie (user=jason) to reviews-v2, else to reviews-v1

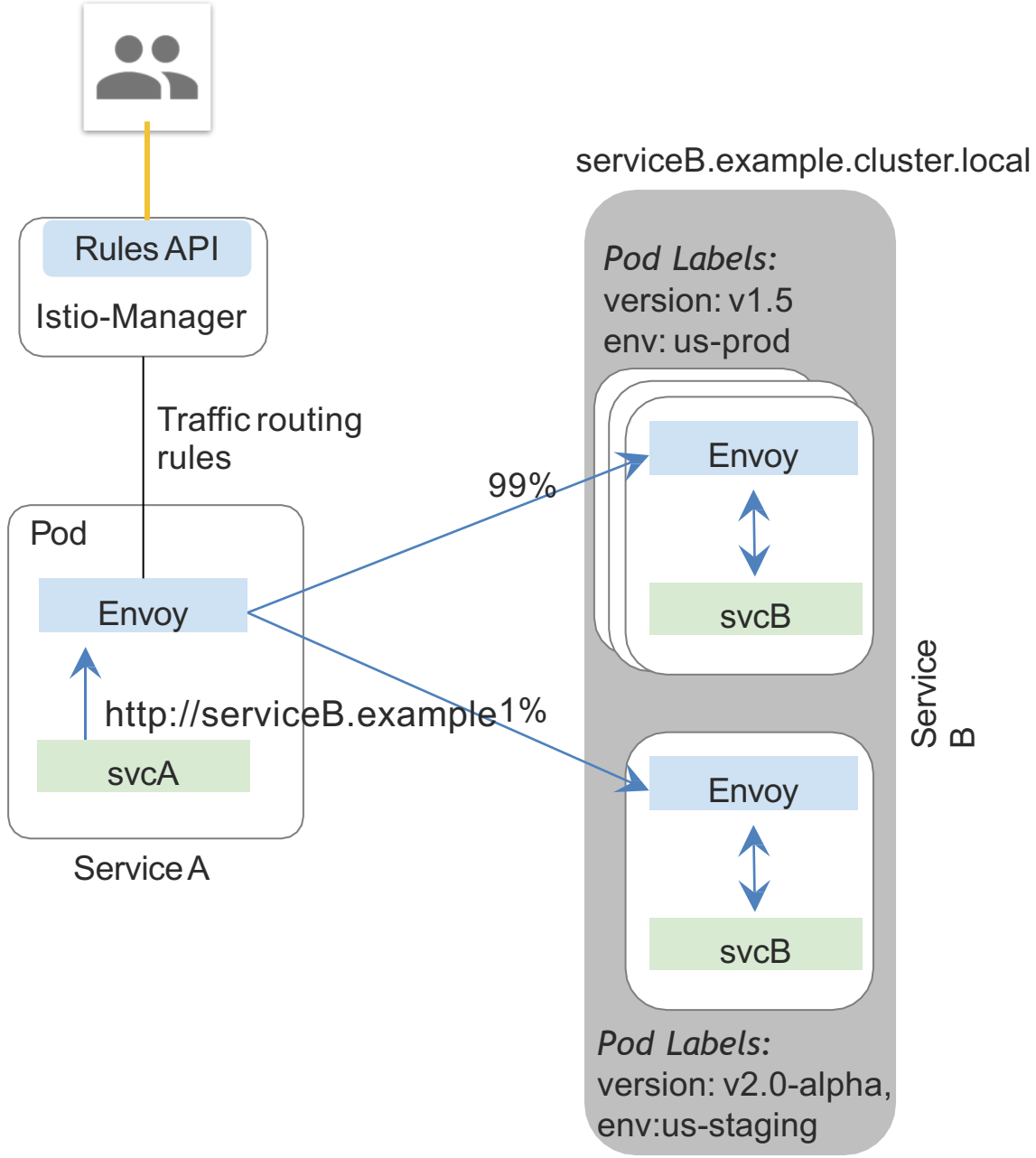
```
istioctl replace -f 04route-rule-
reviews-headers.yam
```

04route-rule-reviews-headers.yaml

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - match:
    - headers:
        cookie:
          regex: "^(.*?;)?(user=jason)(;.*)?$"
    route:
    - destination:
        host: reviews
        subset: v2
  - route:
    - destination:
        host: reviews
        subset: v1
```

Traffic Splitting

```
// A simple traffic splitting rule
destination: serviceB.example.cluster.local
match:
  source: serviceA.example.cluster.local
route:
- tags:
    version: v1.5
    env: us-prod
  weight: 99
- tags:
    version: v2.0-alpha
    env: us-staging
  weight: 1
```



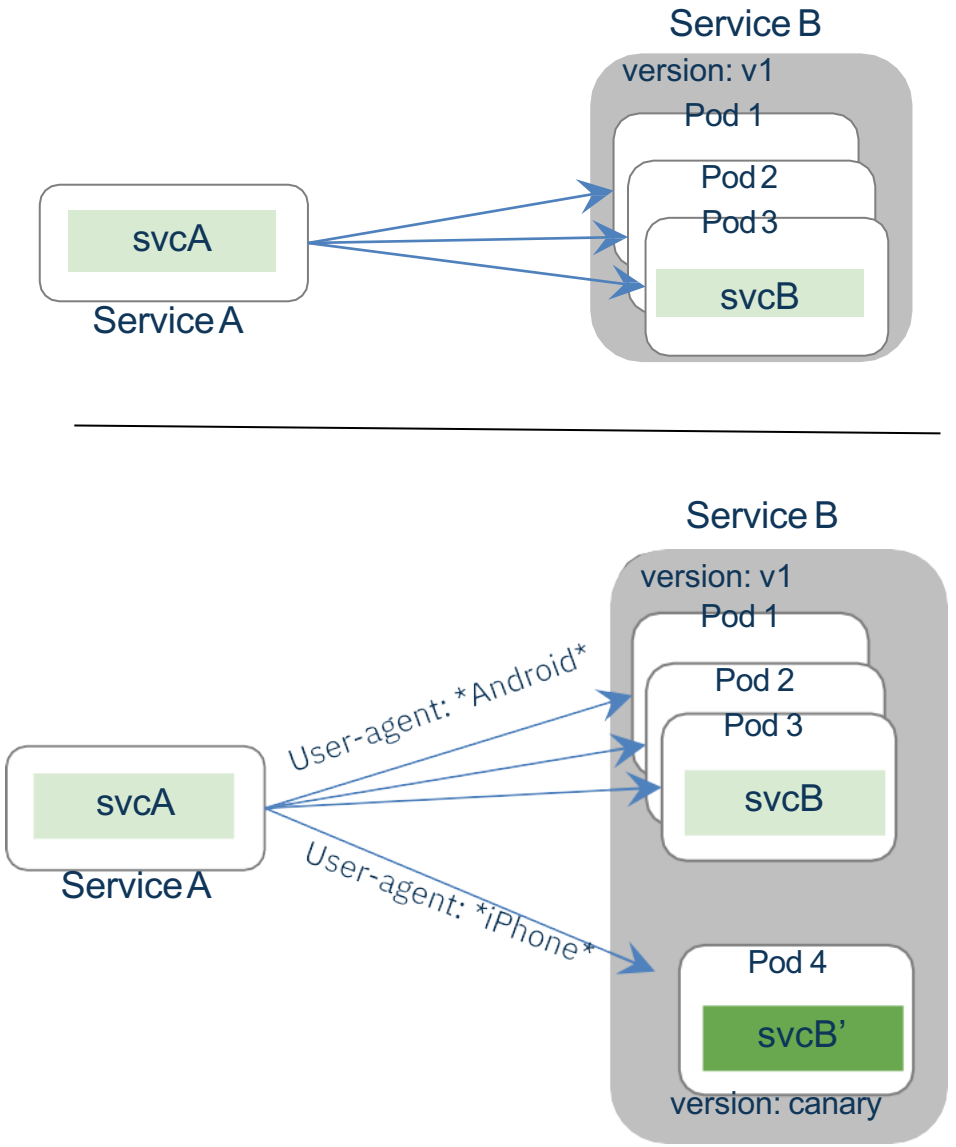
Traffic control is decoupled from infrastructure scaling

Traffic Steering

```
// Content-based traffic steering rule

destination: serviceB.example.cluster.local
match:
  httpHeaders:
  user-agent:
    regex: ^(.*?;)?(iPhone)(;.*)?$
precedence: 2
route:
- tags:
    version: canary
```

Content-based traffic steering



Fault Injection

Istio enables protocol-specific fault injection into the network.

- Configure faults to be injected into requests that match specific criteria.
- Restrict the percentage of requests that should be subjected to faults.
- Two types of faults:
  - Delay: timing failures, mimicking increased network latency, or an overloaded upstream service.
  - Abort: crash failures that mimic failures in upstream services. Aborts usually manifest in the form of HTTP error codes or TCP connection failures.

Other Traffic Management Capabilities

- **HTTP Retry** describes the retry policy to use when a HTTP request fails. For example, you can define a rule that sets the maximum number of retries to 3 when calling ratings:v1 service, with a 2s timeout per retry attempt.

- **Rate Limits** allows you to deny access to services using Mixer, or specify the rate limits.

- **Circuit Breaker** allow users to set global defaults for failure recovery on a per service and/or per service version basis. It allows developers to write applications that limit the impact of failures, and latency spikes.

- **Automatic Request Timeout** allows to set timeout for services

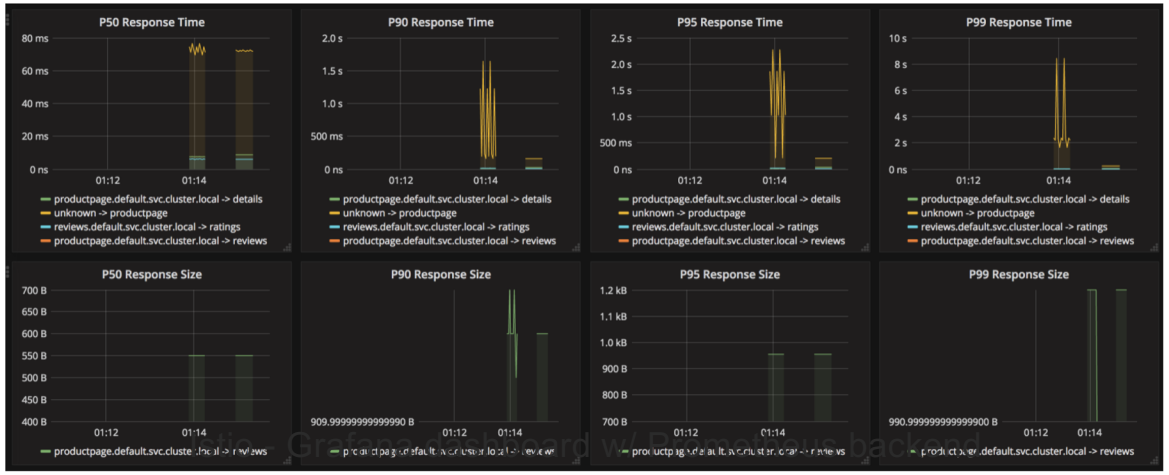
Telemetry

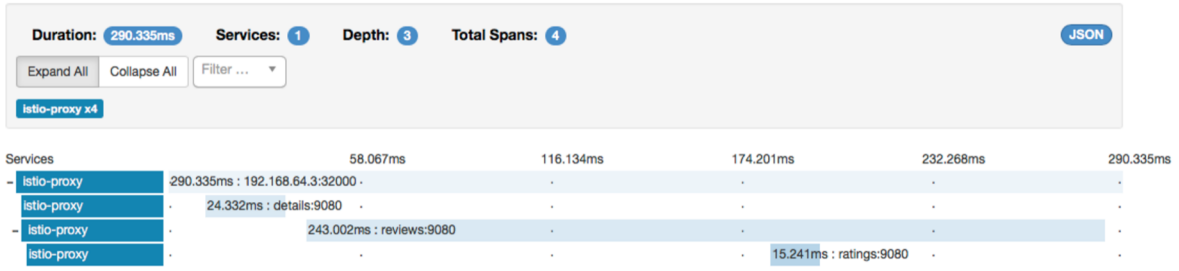
Visibility

Monitoring & tracing should not be an afterthought in the infrastructure

Goals

- Metrics without instrumenting apps
- Consistent metrics across fleet
- Trace flow of requests across services
- Portable across metric backend providers

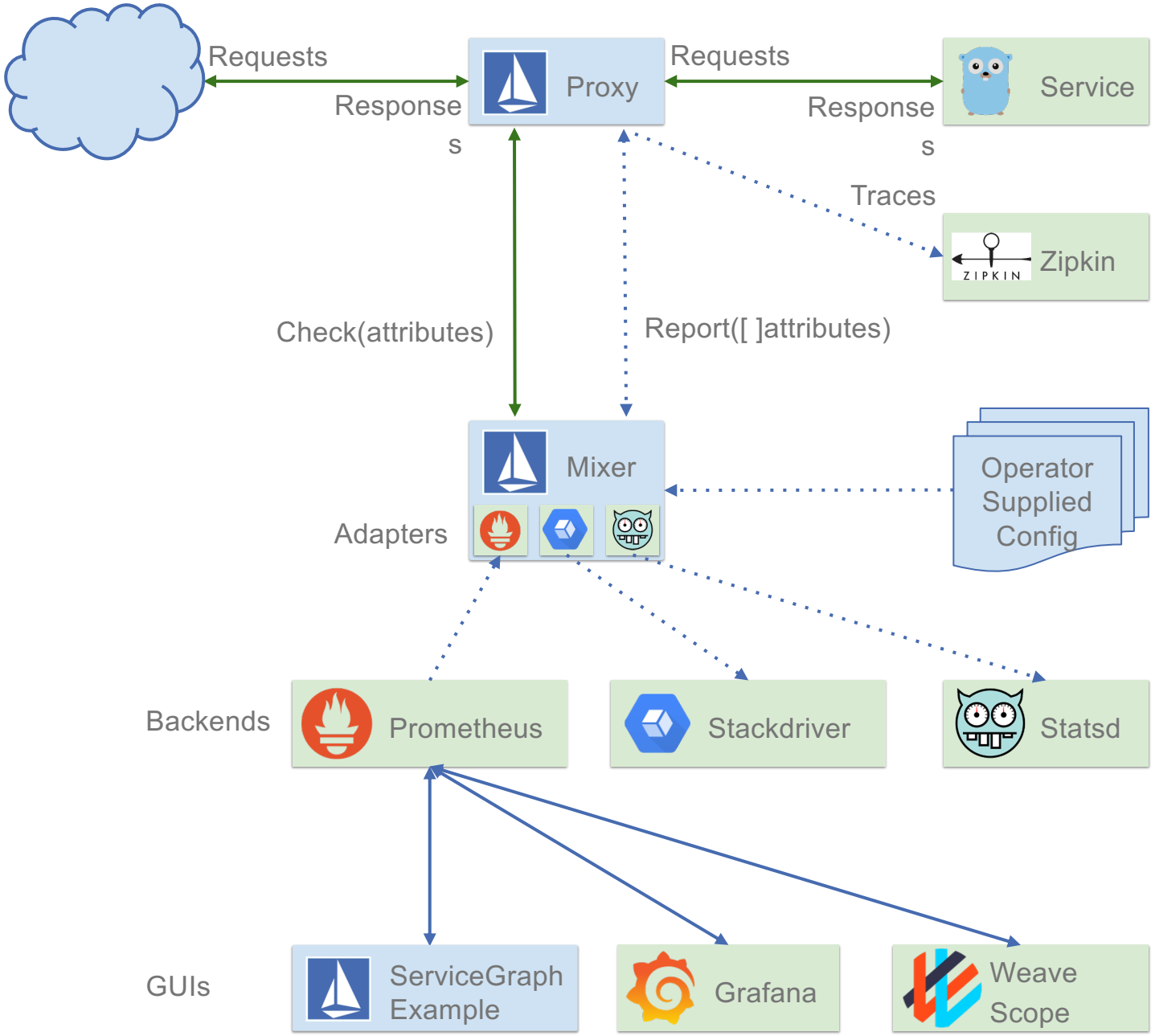




Istio Zipkin tracing dashboard

Visibility: Metrics

- Mixer collects metrics emitted by Istio Proxy
- Adapters in the Mixer normalize and forward to monitoring backends
- Proxy send traces to Mixer
- Metrics backend can be swapped at runtime

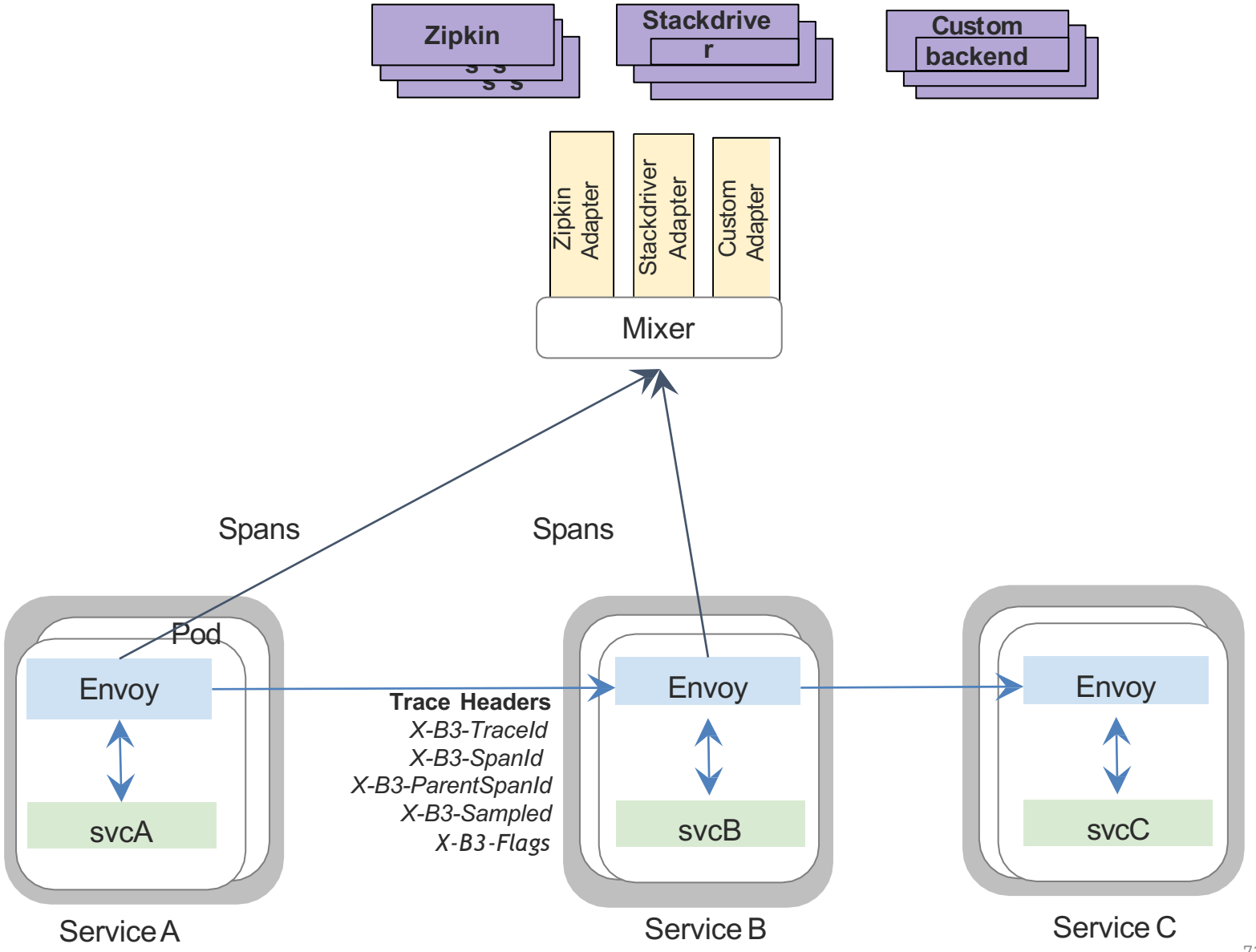


Changes needed to Legacy Applications

- Modify request headers to pass span ids
- x-request-id
- x-b3-traceid
- x-b3-spanid
- x-b3-parentspanid
- x-b3-sampled
- x-b3-flags
- x-ot-span-context

Visibility: Tracing

- Application do not have to deal with generating spans or correlating causality
- Envoys generate spans
  - Applications need to *forward* context headers on outbound calls
- Envoys send traces to Mixer
- Adapters at Mixer send traces to respective backends



Distributed Tracing: Jaegar

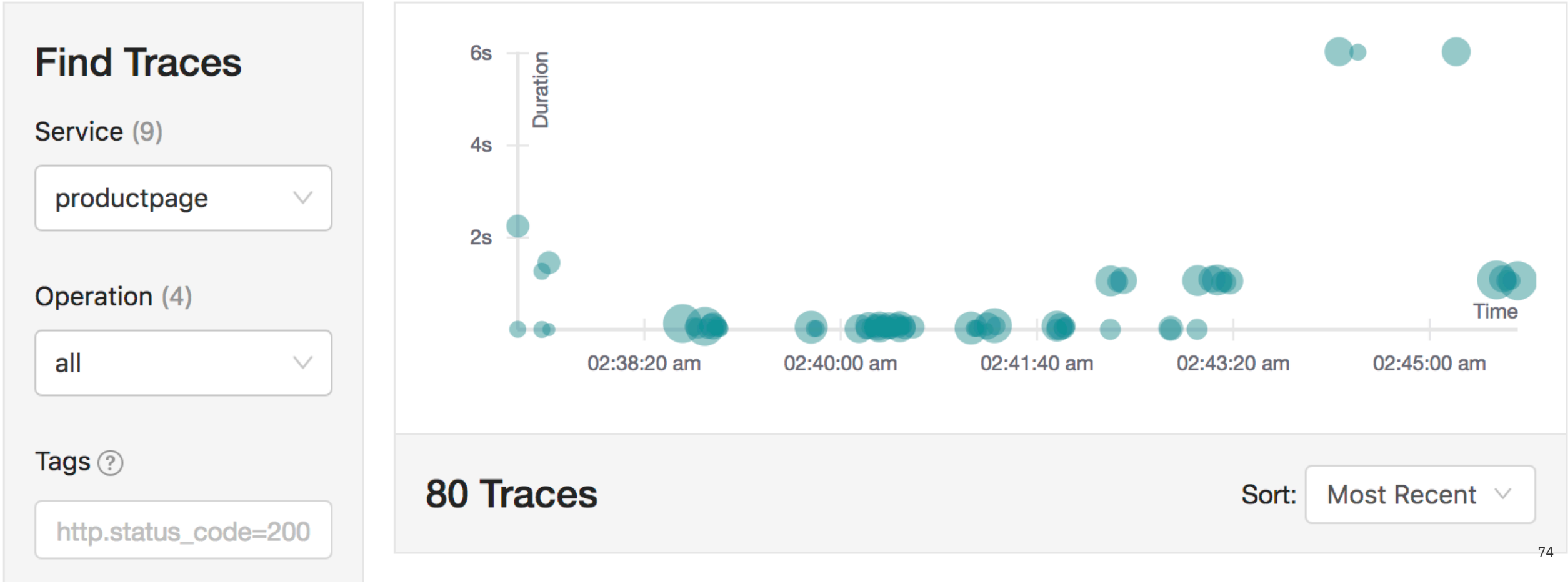
- Jaeger, a **CNCF** project inspired by Dapper and OpenZipkin is a distributed tracing system released as open source by Uber Technologies.
- Jaeger is installed by default with Istio for Tracing.
- It can be used for monitoring microservices-based distributed systems:
  - Distributed context propagation
  - Distributed transaction monitoring
  - Root cause analysis
  - Service dependency analysis
  - Performance / latency optimization

https://github.com/jaegertracing/jaeger

Jaeger: Cont'd

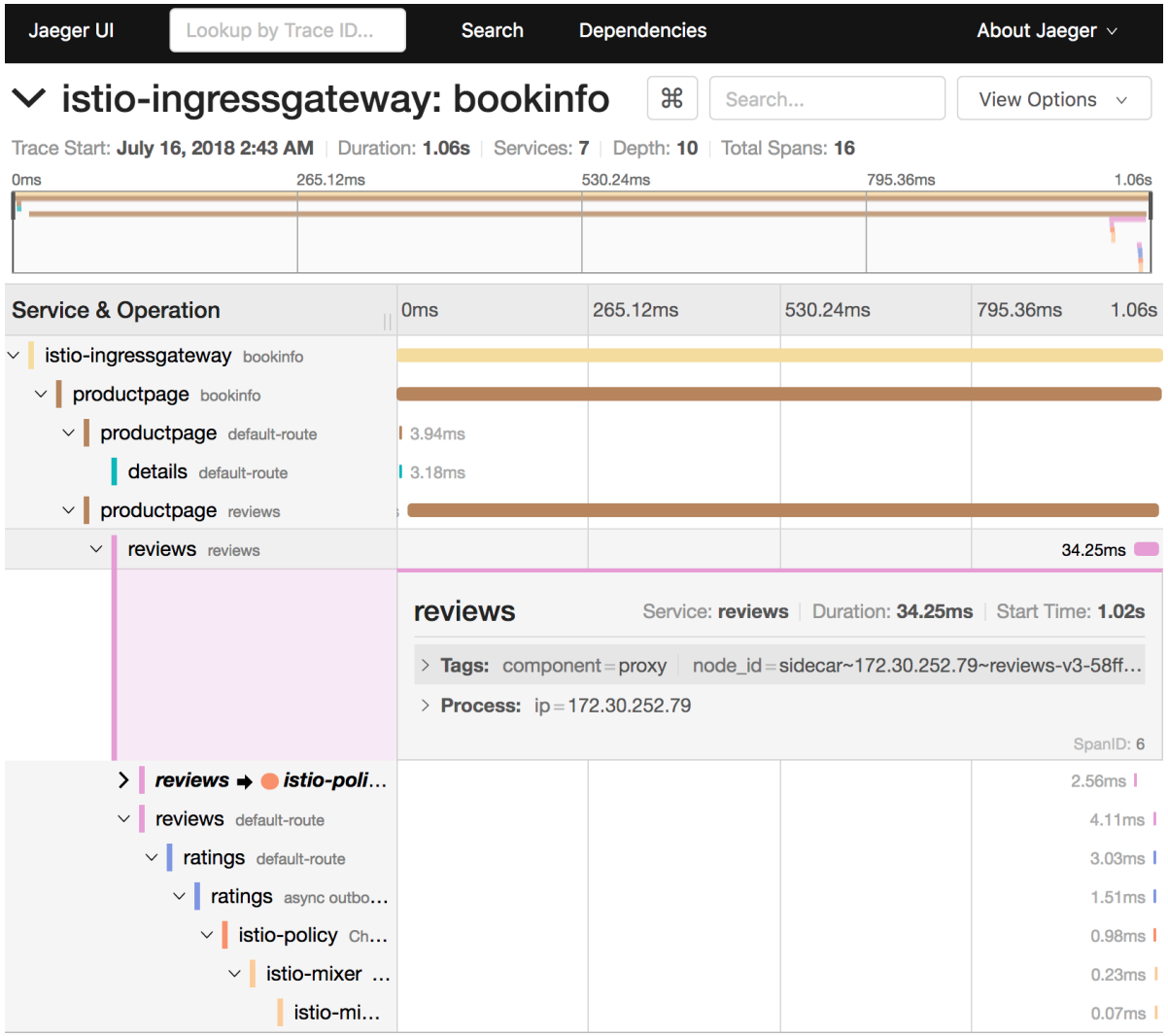
Get the tracing service details to access it

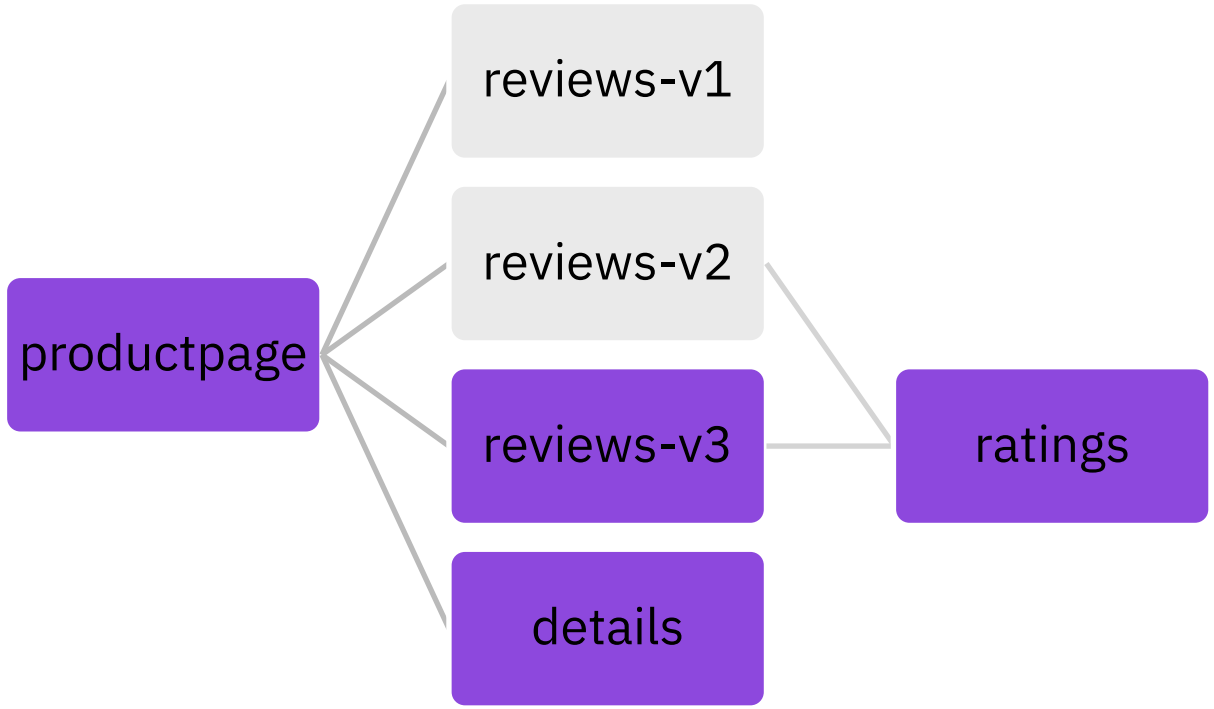
```
kubectl get svc tracing -n istio-system
```



Jaeger: Cont’d

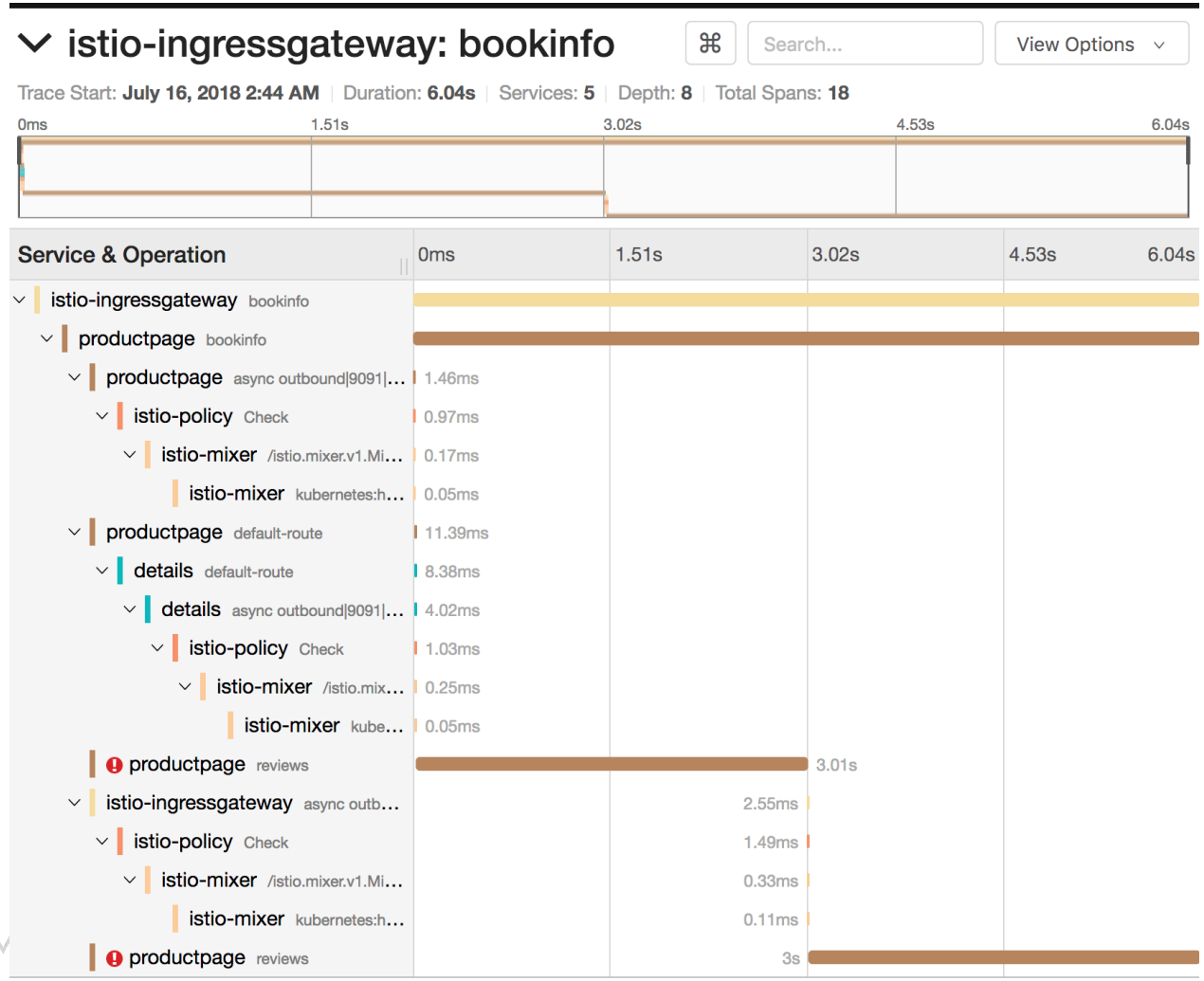
Inspect one of the requests

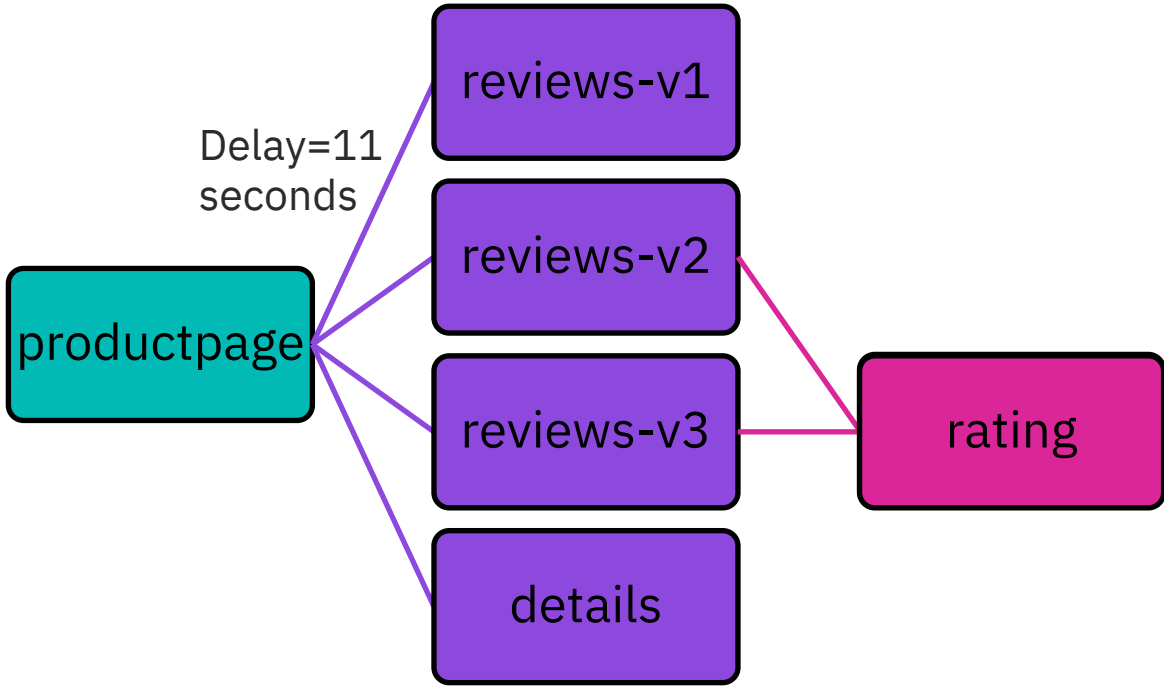




Jaeger: Cont'd

Find traces for productpage where tags = error





Monitoring: Prometheus

- Prometheus is a CNCF project which is an open-source monitoring system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach
- It’s used by default in Istio for metrics monitoring.
- Istio instructs Mixer to automatically generate and report a new metric and a new log stream for all traffic within the mesh.

https://prometheus.io/

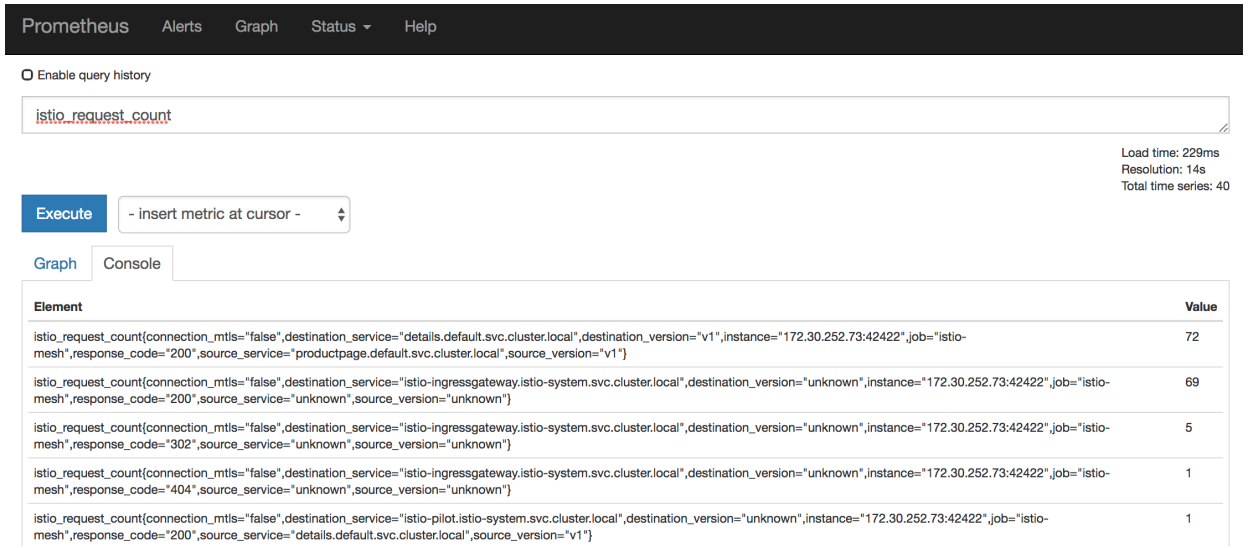
Monitoring: Prometheus

- Port forward to Prometheus service on Istio

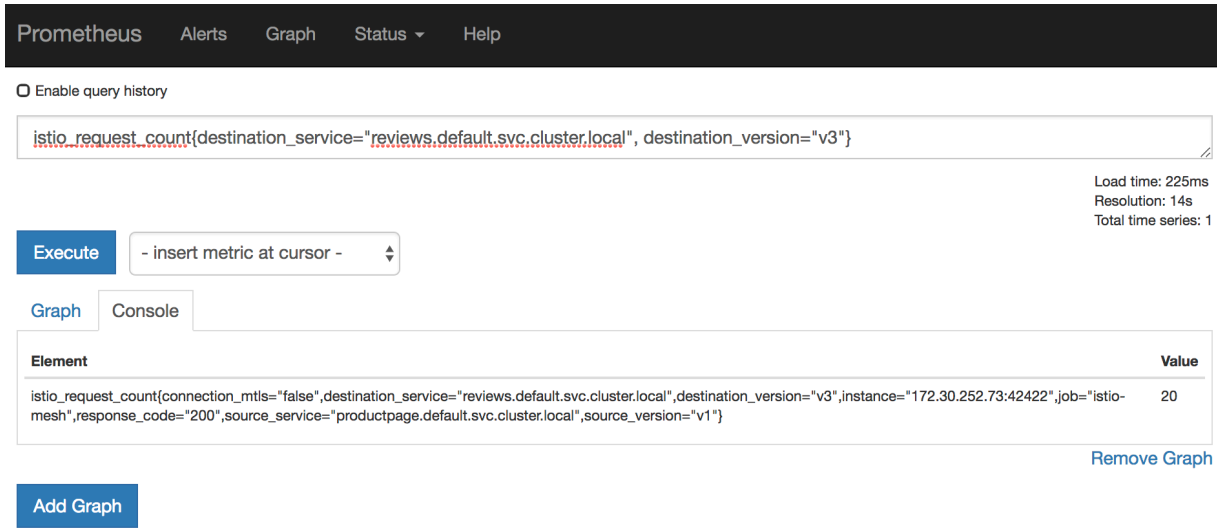
```
kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l
app=prometheus -o jsonpath='{.items[0].metadata.name}') 9090:9090
```

- Access Prometheus

http://localhost:9090/graph



istio_request_count



istio_request_count{destination_service="reviews.default.svc.cluster.local", destination_version="v3"}

Monitoring Visualization: Grafana

- **Grafana** is a data visualization & Monitoring with support for Graphite, InfluxDB, Prometheus, Elasticsearch and many more databases.

- It’s used by default in Istio for monitoring visualization

https://grafana.com/

Metrics: Prometheus

- Port forward to Grafana service on Istio

```
kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l
app=grafana -o jsonpath='{.items[0].metadata.name}') 3000:3000
```

- Access Grafana

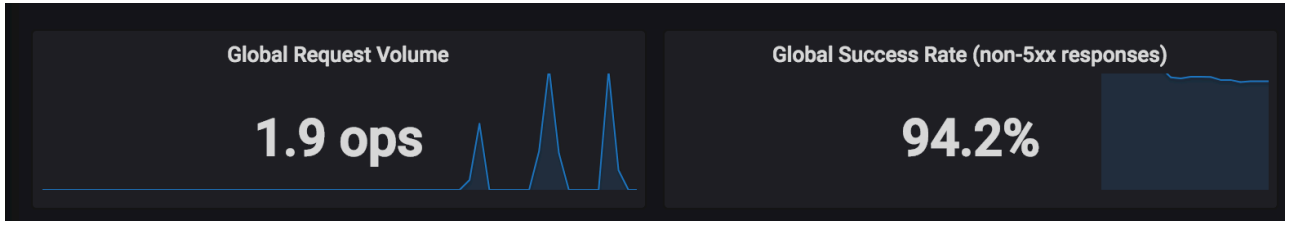
http://localhost:3000/

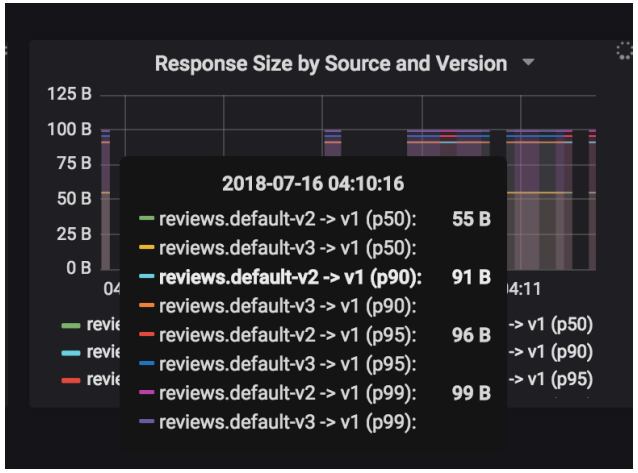
- Generate some traffic

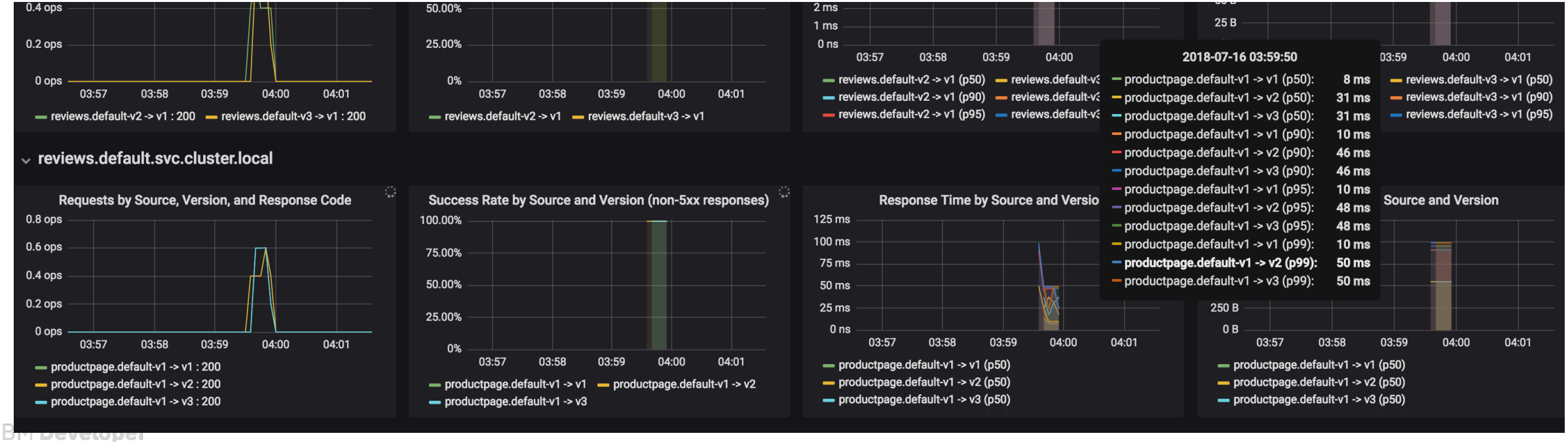
```
while sleep 3;do curl http://169.46.54.122/productpage; done
```

- The Istio Dashboard consists of three main sections:
  - Global Summary View
  - Mesh Summary View
  - Individual Services View

Metrics: Prometheus







Service Graph

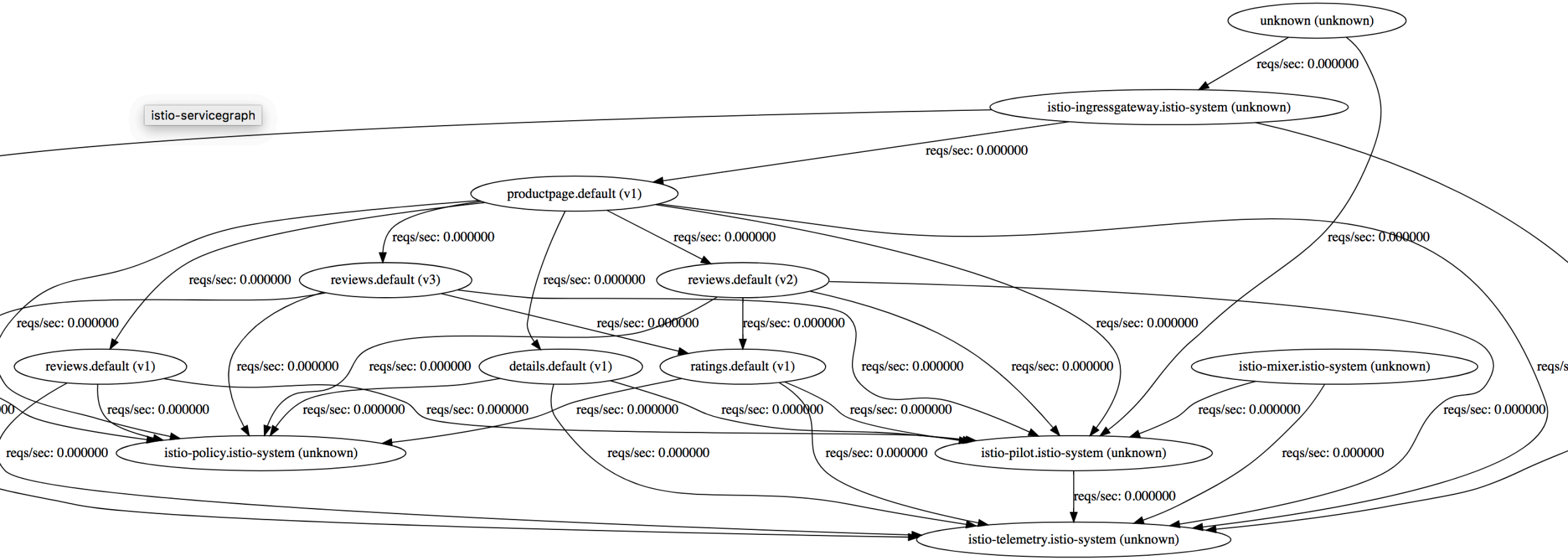
- To view a graphical representation of your mesh.
- It is built on top of Prometheus queries and depends on the standard Istio metric configuration
- Port forward to service graph service on Istio

```
kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l
app=servicegraph -o jsonpath='{.items[0].metadata.name}') 8088:8088
```

Service Graph

- Access Service Graph

`http://localhost:8088/dotviz`



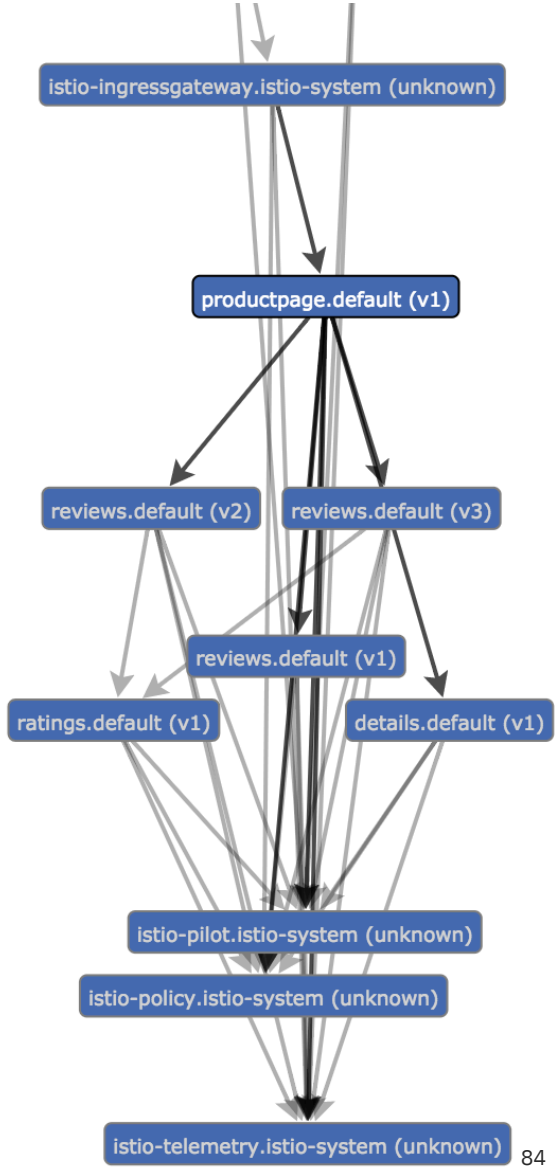
Service Graph

- Generate some traffic and access dynamic Service Graph

`http://localhost:8088/force/forcegraph.html`

Incoming Connections	Reqs/sec
istio-ingressgateway.istio-system (unknown)	0.786441

Outgoing Connections	Reqs/sec
details.default (v1)	0.786441
istio-pilot.istio-system (unknown)	0.000000
istio-policy.istio-system (unknown)	0.162712
istio-telemetry.istio-system (unknown)	0.681356
reviews.default (v1)	0.261017
reviews.default (v2)	0.261017
reviews.default (v3)	0.261017



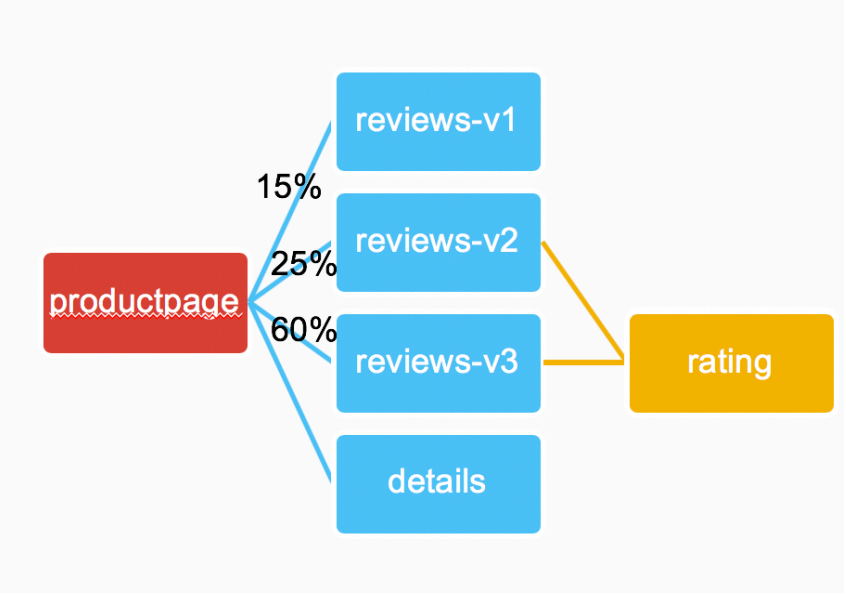
Service Graph

- Apply Istio Canary Deployment

```
istioctl create -f 03route-rule-reviews-canary.yaml
```

Incoming Connections	Reqs/sec
istio-ingressgateway.istio-system (unknown)	24.349153

Outgoing Connections	Reqs/sec
details.default (v1)	24.294915
istio-pilot.istio-system (unknown)	0.000000
istio-policy.istio-system (unknown)	0.223729
istio-telemetry.istio-system (unknown)	1.796610
reviews.default (v1)	3.488136
reviews.default (v2)	6.027119
reviews.default (v3)	14.755932

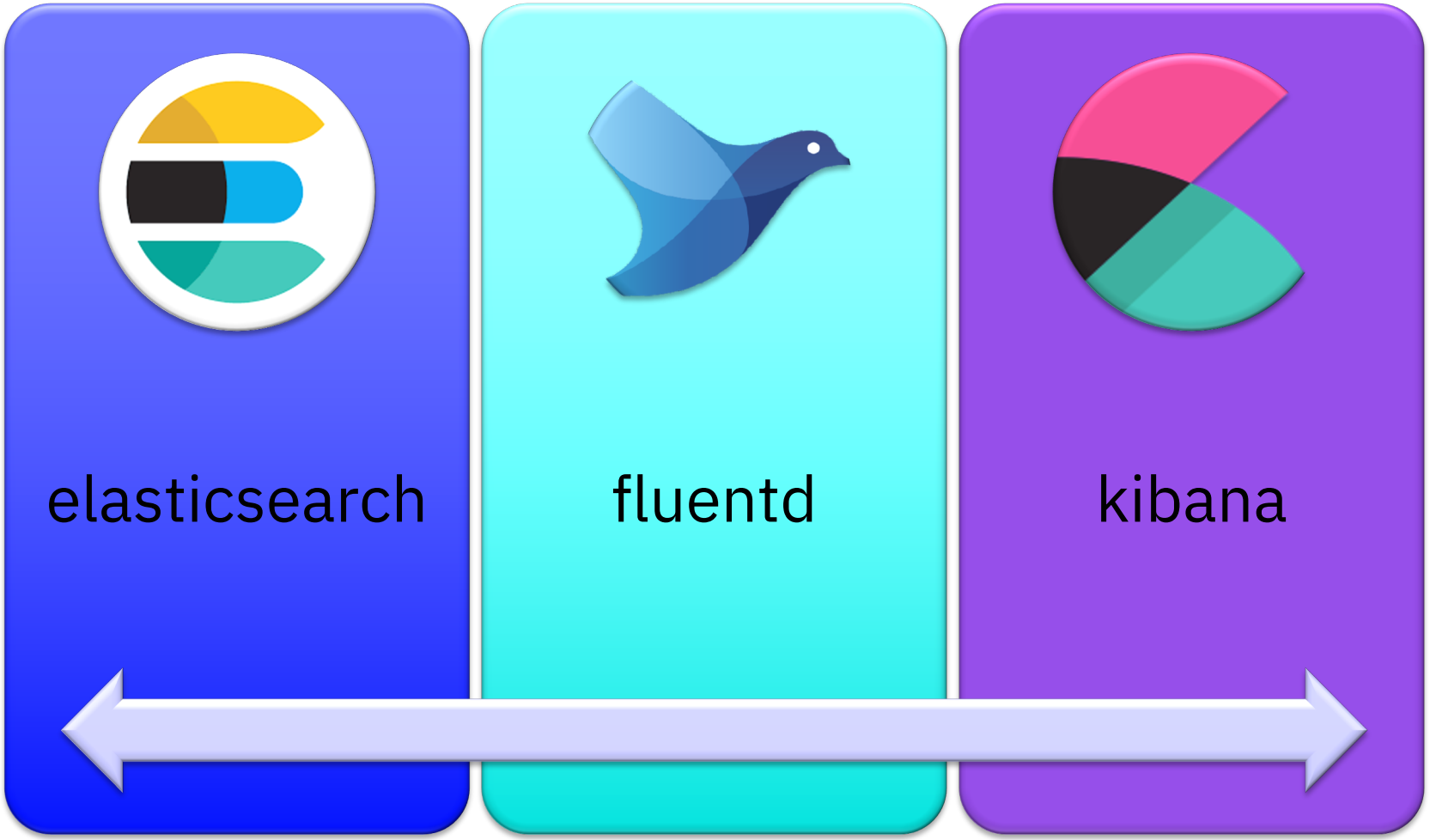


Logging

Log Aggregation

- With an increasing number of systems decoupled and scattered throughout the landscape it becomes increasingly difficult to track and trace events across all systems.
- Log aggregation solutions provides a series of benefits to distributed systems.
- The problems it tackles are:
  - Centralized, aggregated view over all log events
  - Normalization of log schema
  - Automated processing of log messages
  - Support for a great number of event sources and outputs

EFK Stack



EFK Stack

- EFK stack doesn't ship by default with Istio. However, Istio allows integration with EFK stack through its proxy.
- Deploy EFK stack in your cluster.
- `kubectl apply -f logs/logging-stack.yaml`
- Now that there is a running Fluentd daemon, add a new log entry in Istio, and send those logs to the listening daemon.
- Create a new YAML file to hold configuration for the log stream that Istio will generate and collect automatically.

`istioctl create -f logs/fluentd-istio.yaml`

fluentd-istio.yaml

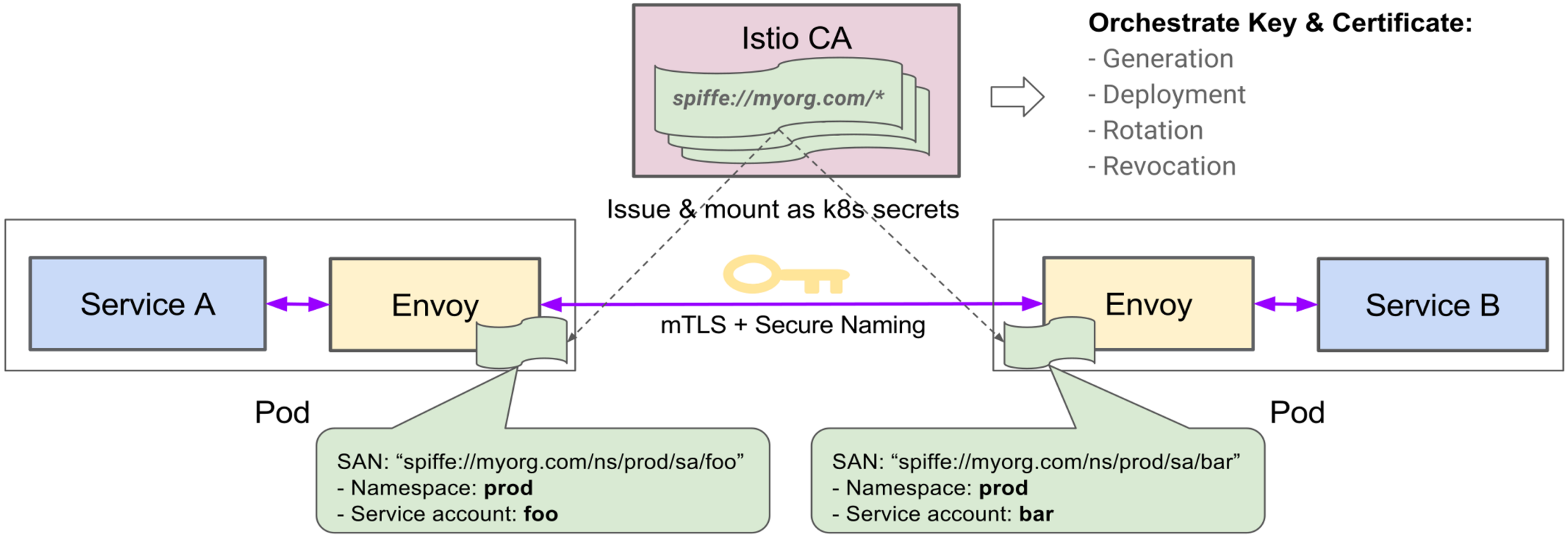
```
# Configuration for logentry instances
apiVersion: "config.istio.io/v1alpha2"
kind: logentry
metadata:
  name: newlog
  namespace: istio-system
spec:
  severity: '"info"'
  timestamp: request.time
  variables:
    source: source.labels["app"] | source.service | "unknown"
    user: source.user | "unknown"
    destination: destination.labels["app"] |
destination.service | "unknown"
    responseCode: response.code | 0
    responseSize: response.size | 0
    latency: response.duration | "0ms"
  monitored_resource_type: '"UNSPECIFIED"'
---
# Configuration for a fluentd handler
apiVersion: "config.istio.io/v1alpha2"
kind: fluentd
metadata:
  name: handler
  namespace: istio-system
spec:
  address: "fluentd-es.logging:24224"
---
# Rule to send logentry instances to the fluentd handler
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: newlogtofluentd
  namespace: istio-system
spec:
  match: "true" # match for all requests
  actions:
   - handler: handler.fluentd
     instances:
     - newlog.logentry
---
```

Security

Securing Microservices

- Verifiable identity
- Secure naming / addressing
- Traffic encryption
- Revocation

Istio Citadel - Security at Scale



spiffe.io

Istio Citadel

- Istio can secure the communication between microservices without requiring application code changes.
- Security is provided by authenticating and encrypting communication paths within the cluster.
- Delegating communication security to Istio (as opposed to implementing TLS in each microservice), ensures that your application will be deployed with consistent and manageable security policies.
- Citadel is responsible for:
    - Providing each service with an identity representing its role.
    - Providing a common trust root to allow Envoys to validate and authenticate each other.
    - Providing a key management system

What’s the difference between Red Hat OpenShift Service Mesh and Istio?

- OpenShift Service Mesh installs a multi-tenant control plane by default
- OpenShift Service Mesh extends Role Based Access Control (RBAC) features
- OpenShift Service Mesh replaces BoringSSL with OpenSSL
- Kiali and Jaeger are enabled by default in OpenShift Service Mesh

Resources

Istio Home - https://istio.io

Istio on IBM Cloud - https://www.ibm.com/cloud/istio

Free e-Book: **Istio Explained - Getting Started with Service Mesh** - https://www.ibm.com/downloads/cas/XWN1WV9Q

Code Patterns, Articles, Tutorials, Blogs in IBM Developer:

https://developer.ibm.com/components/istio/
https://developer.ibm.com/technologies/microservices/

FREE Badges:

1. Getting started with Microservices with Istio and IBM Cloud Kubernetes Service - https://www.ibm.com/training/badge/32942101-6321-4c8f-b0e1-9b687d0addc5

2. Beyond the Basics: Istio and IBM Cloud Kubernetes Service - https://www.youracclaim.com/org/ibm/badge/beyond-the-basics-istio-and-ibm-cloud-kubernetes-service