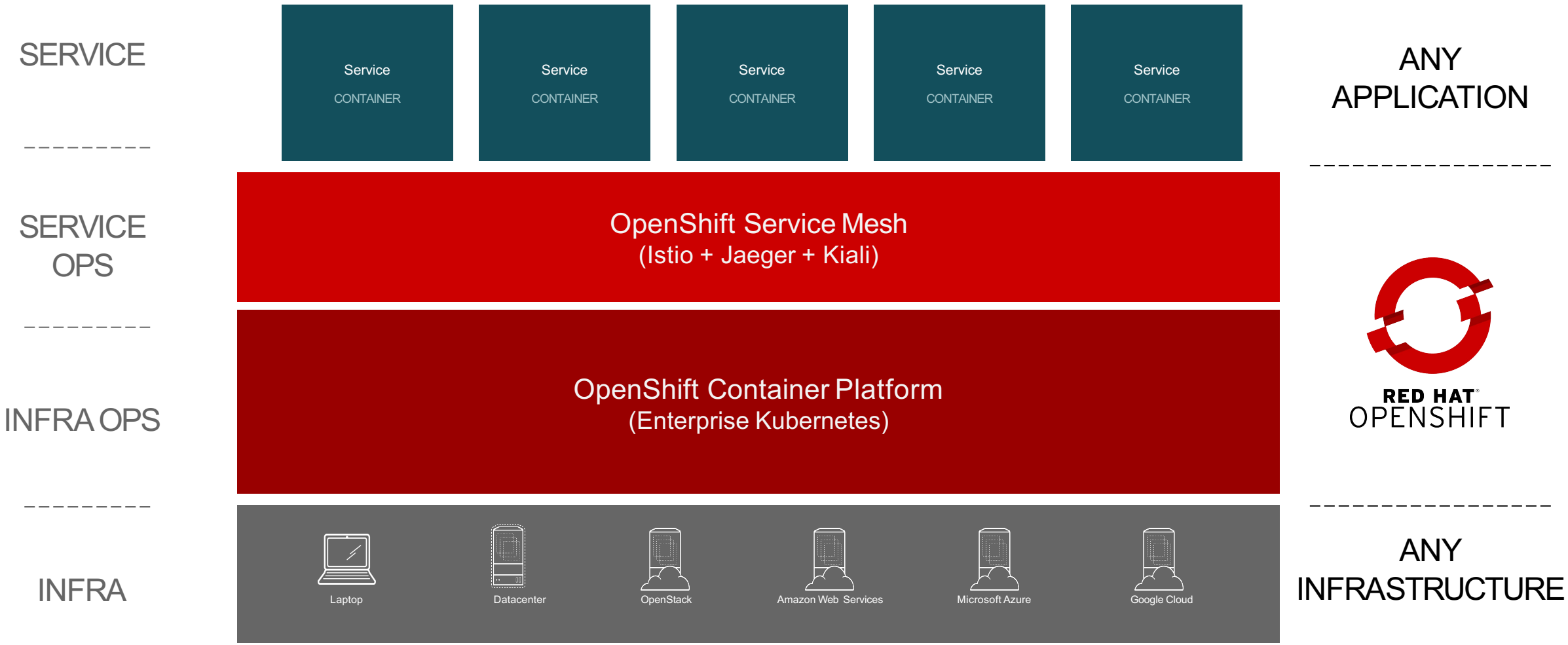
OpenShift Container Platform

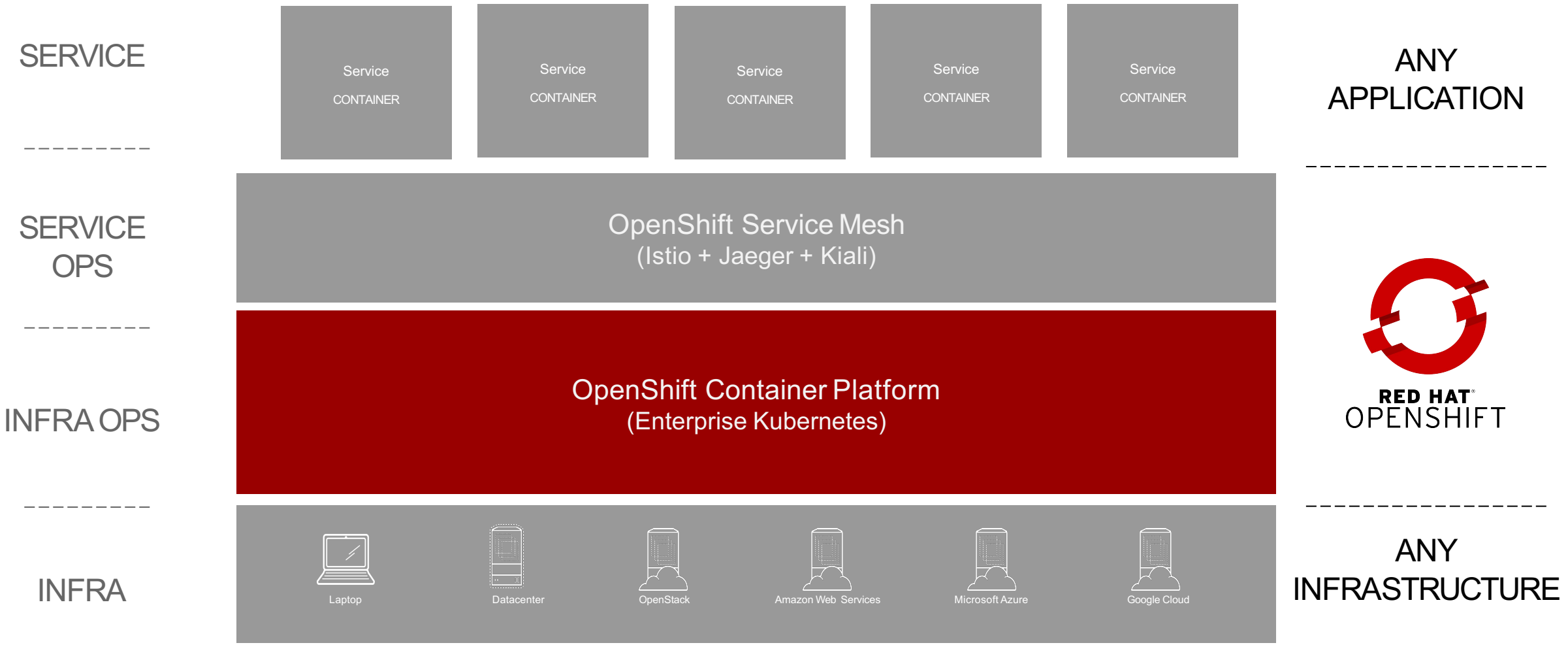
Network

Rubayat Khan,
Hakam Abdelqader

Distributed Services With Red Hat Openshift



Distributed Services With Red Hat Openshift

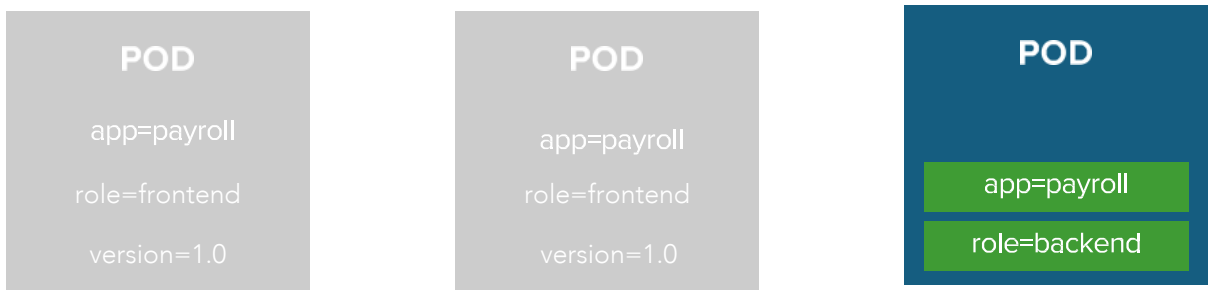


BUILT-IN SERVICE DISCOVERY INTERNAL LOAD-BALANCING

BUILT-IN SERVICE DISCOVERY INTERNAL LOAD-BALANCING

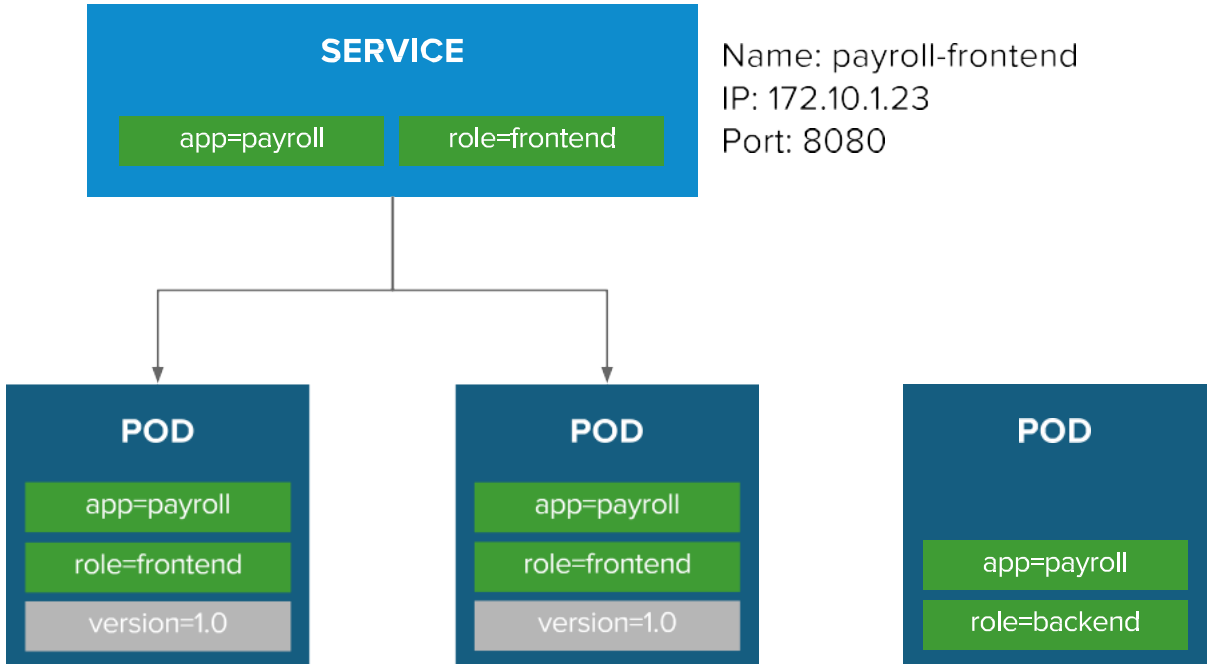

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    app: payroll
    role: frontend
  annotations:
    version: 1.0
  name: pod
spec:
  containers:
  - args: [...]
    image: busybox
    name: pod
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
```

BUILT-IN SERVICE DISCOVERY INTERNAL LOAD-BALANCING



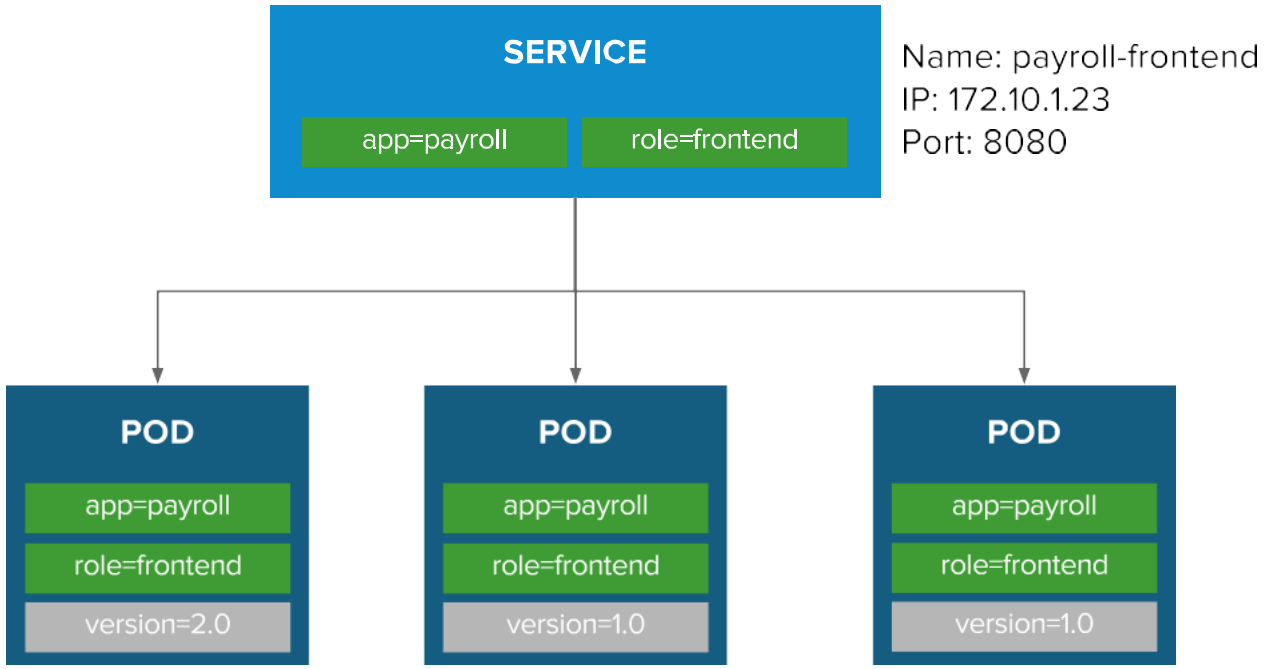

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    app: payroll
    role: backend
  name: pod
spec:
  containers:
  - args: [...]
    image: busybox
    name: pod
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
```

BUILT-IN SERVICE DISCOVERY INTERNAL LOAD-BALANCING



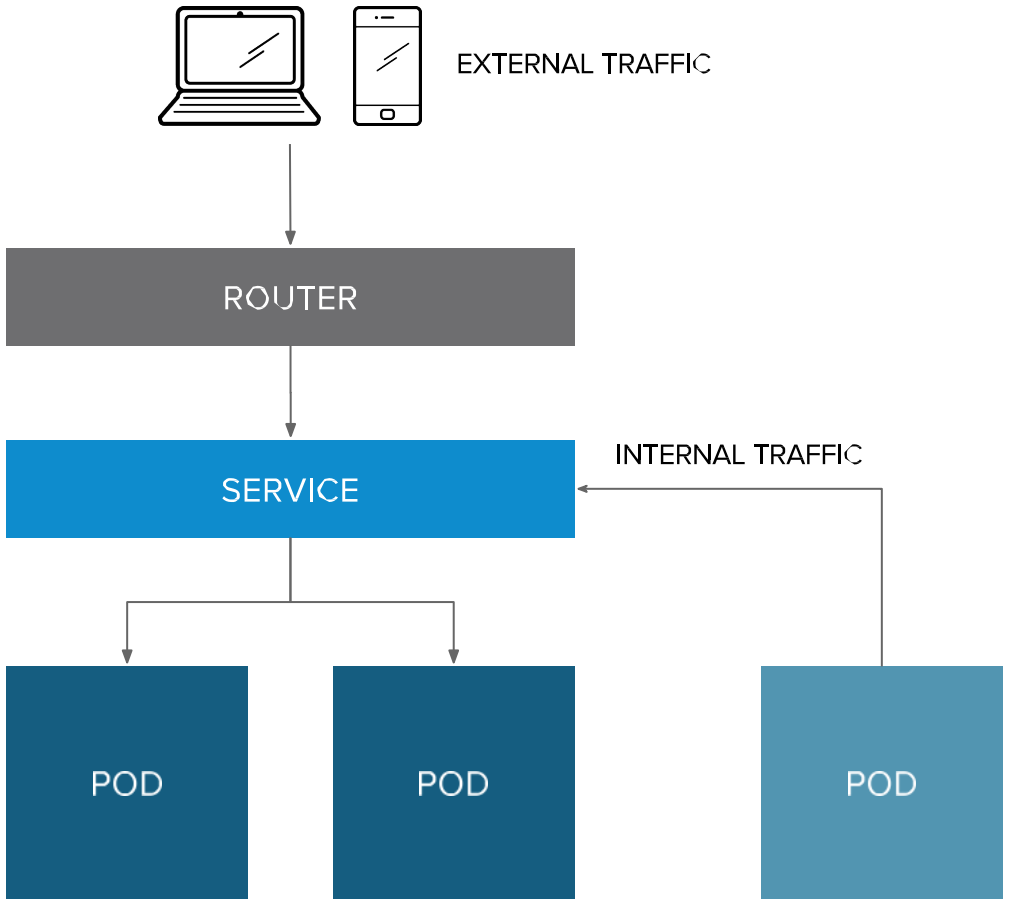

```yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: payroll
    role: frontend
  name: payroll-frontend
spec:
  clusterIP: 172.10.1.23
  ports:
    - name: payroll-frontend
      port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: payroll
    role: frontend
  type: ClusterIP
status:
  loadBalancer: {}
```

BUILT-IN SERVICE DISCOVERY INTERNAL LOAD-BALANCING



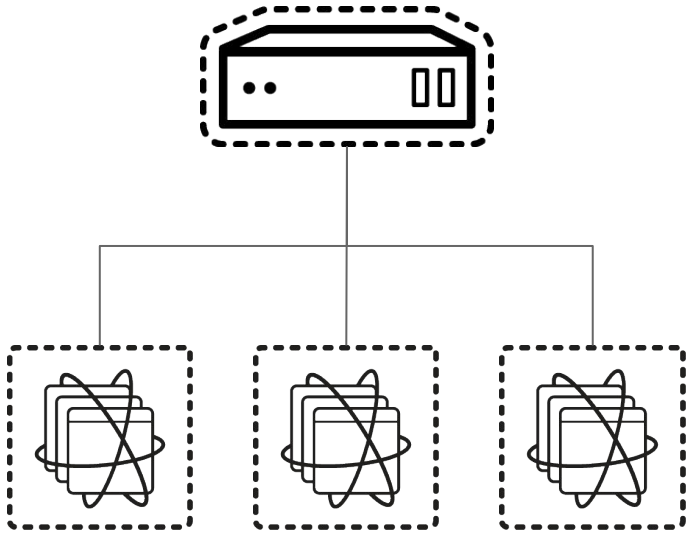

```yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: payroll
    role: frontend
  name: payroll-frontend
spec:
  clusterIP: 172.10.1.23
  ports:
    - name: payroll-frontend
      port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: payroll
    role: frontend
  type: ClusterIP
status:
  loadBalancer: {}
```

ROUTE EXPOSES SERVICES EXTERNALLY



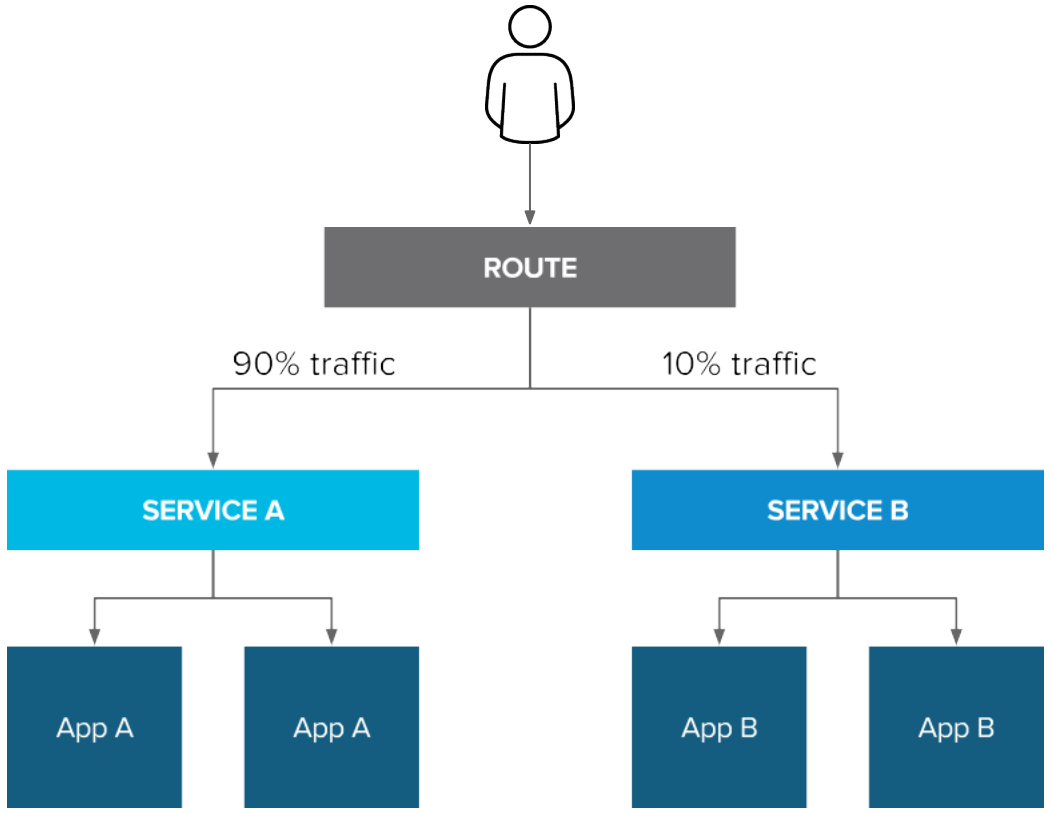
ROUTING AND EXTERNAL LOAD-BALANCING

- Pluggable routing architecture
  - HAProxy Router
  - F5 Router
- Multiple-routers with traffic sharding
- Router supported protocols
  - HTTP/HTTPS
  - WebSockets
  - TLS with SNI
- Non-standard ports via cloud load-balancers, external IP, and NodePort



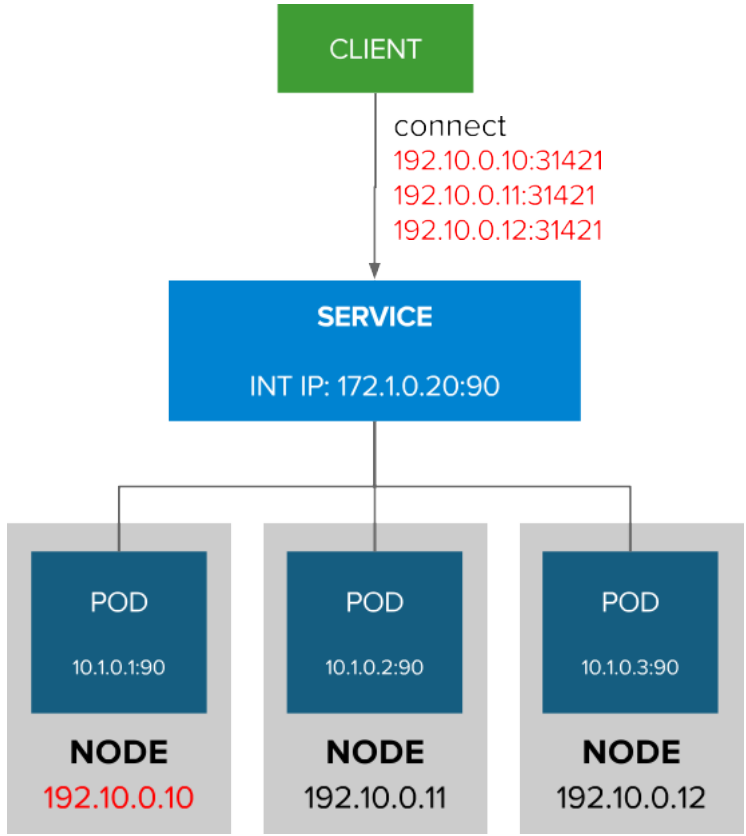
ROUTE SPLIT TRAFFIC

Split Traffic Between Multiple Services For A/B Testing, Blue/Green and Canary Deployments



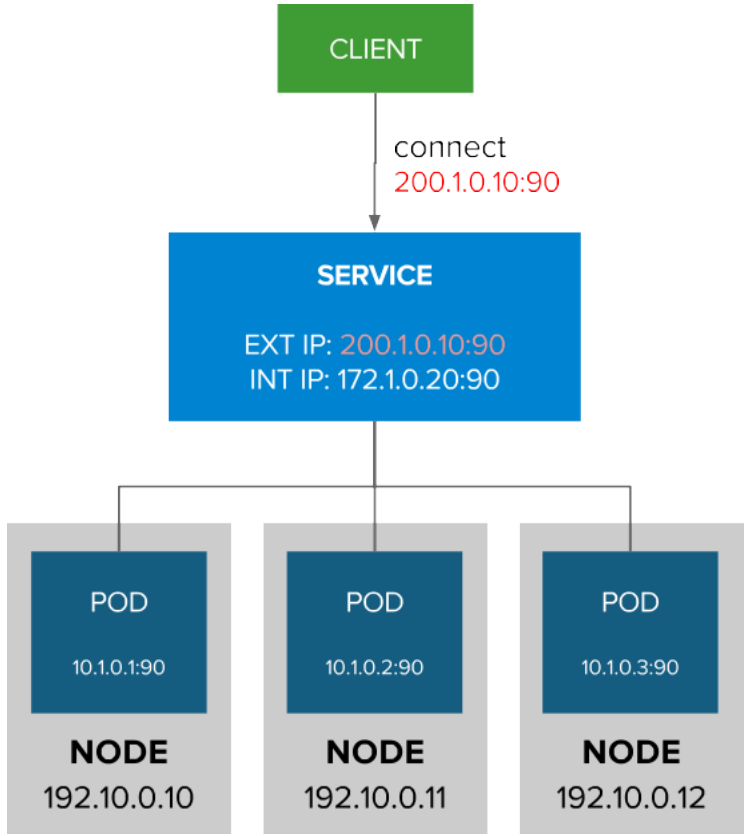
EXTERNAL TRAFFIC TO A SERVICE ON A RANDOM PORT WITH NODEPORT

- NodePort binds a service to a unique port on all the nodes
- Traffic received on any node redirects to a node with the running service
- Ports in 30K-60K range which usually differs from the service
- Firewall rules must allow traffic to all nodes on the specific port

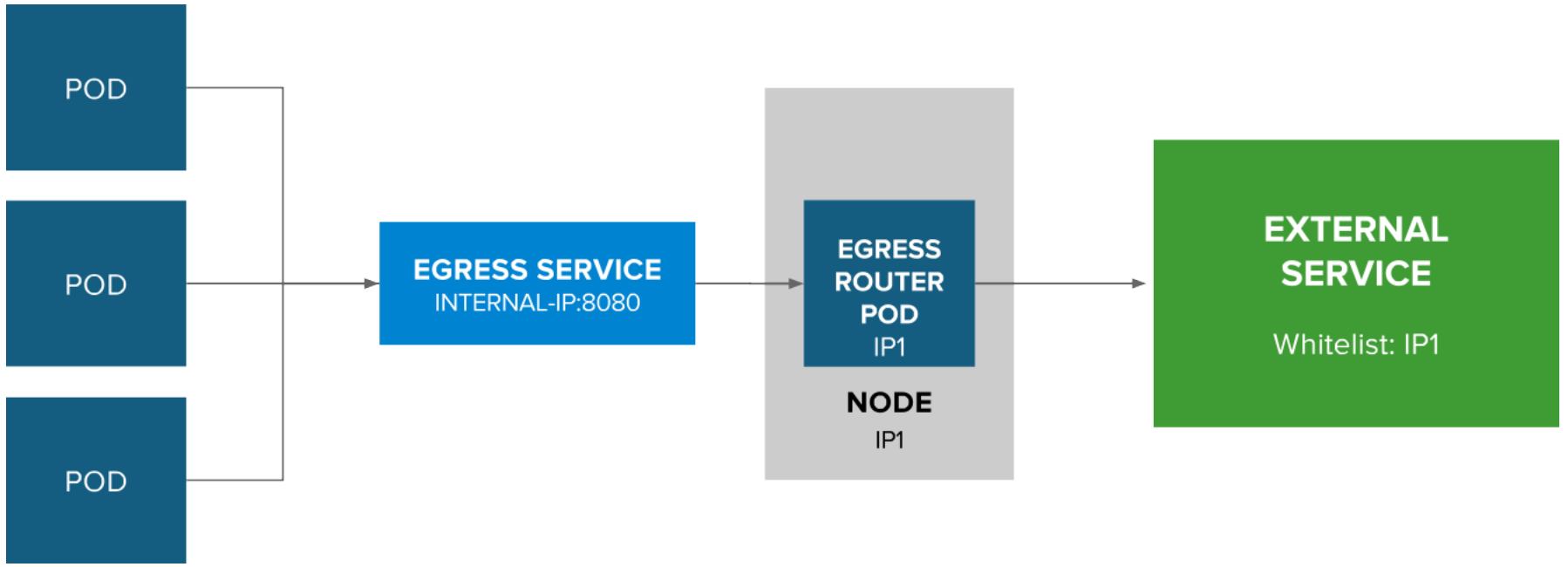


EXTERNAL TRAFFIC TO A SERVICE ON ANY PORT WITH INGRESS

- Access a service with an external IP on any TCP/UDP port, such as
  - Databases
  - Message Brokers
- Automatic IP allocation from a predefined pool using Ingress IP Self-Service
- IP failover pods provide high availability for the IP pool

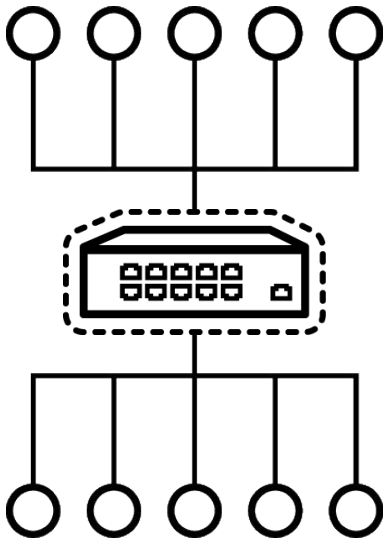


CONTROL OUTGOING TRAFFIC SOURCE IP WITH EGRESS ROUTER

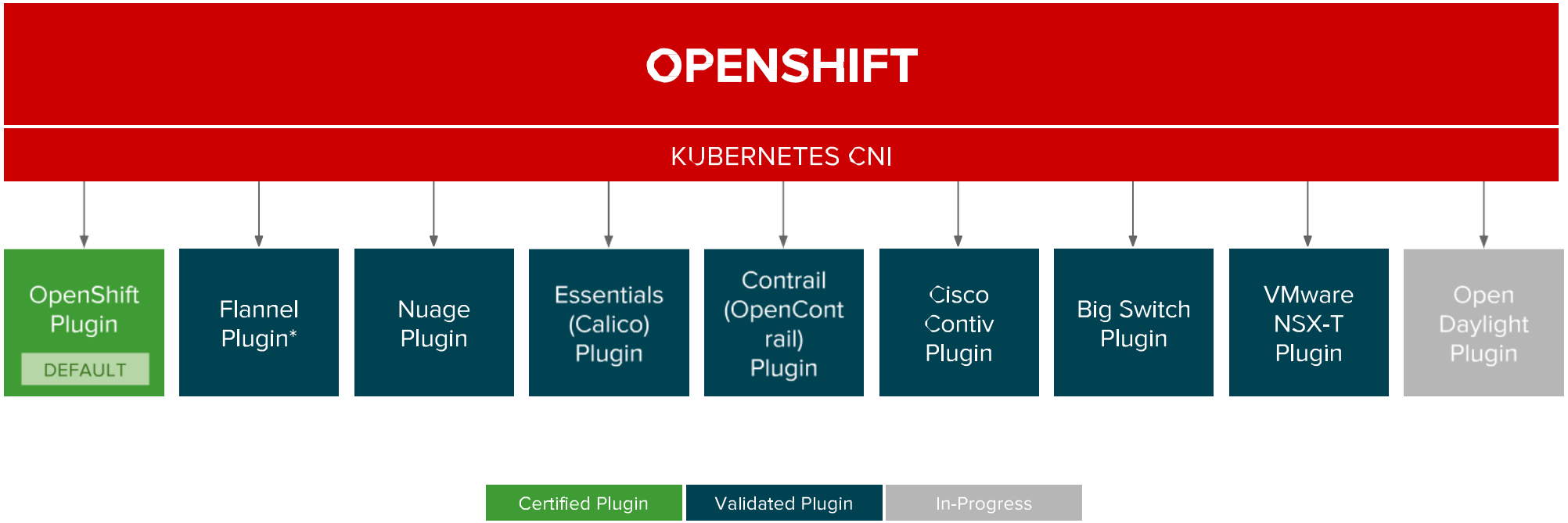


OPENSHIFT NETWORKING

- Built-in internal DNS to reach services by name
- Split DNS is supported via SkyDNS
  - Master answers DNS queries for internal services
  - Other nameservers serve the rest of the queries
- Software Defined Networking (SDN) for a unified cluster network to enable pod-to-pod communication
- OpenShift follows the Kubernetes Container Networking Interface (CNI) plug-in model

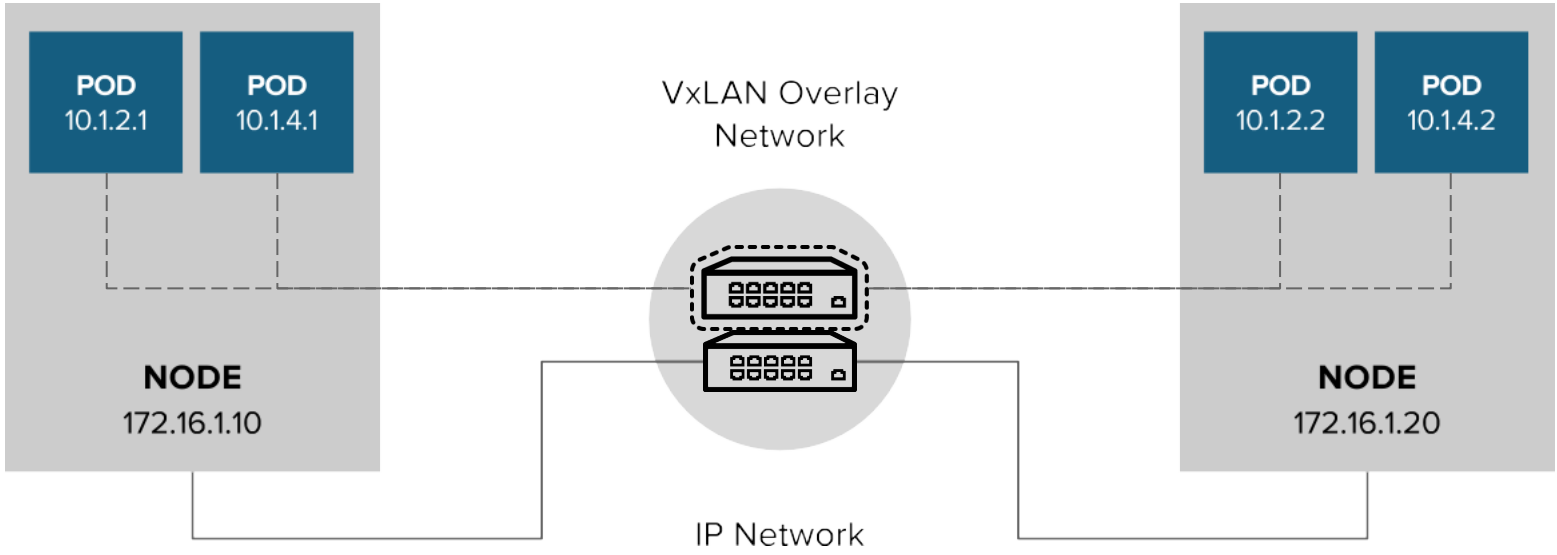


OPENSHIFT NETWORK PLUGINS



* Flannel is minimally verified and is supported only and exactly as deployed in the OpenShift on OpenStack reference architecture

OPENSHIFT NETWORKING



OPENSHIFT SDN

FLAT NETWORK (Default)

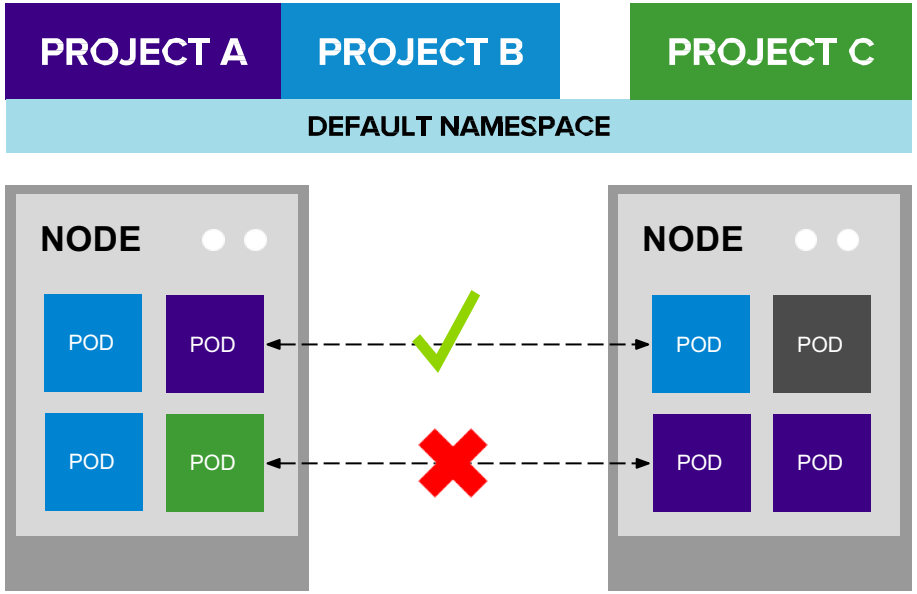
- All pods can communicate with each other across projects

MULTI-TENANT NETWORK

- Project-level network isolation
- Multicast support
- Egress network policies

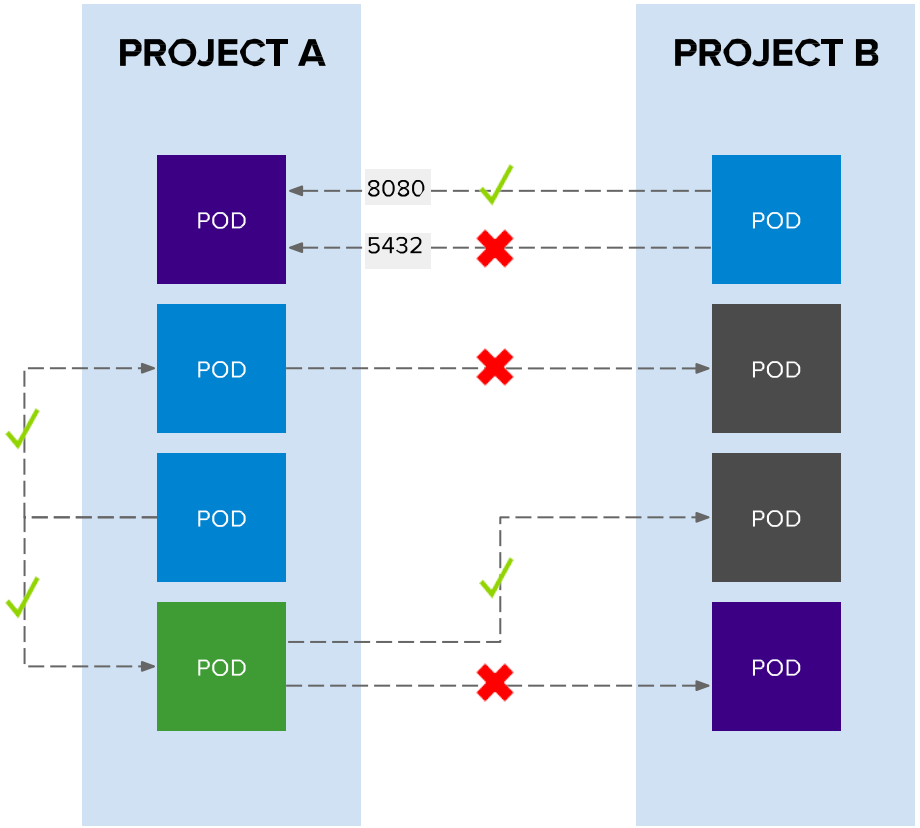
NETWORK POLICY (Tech Preview)

- Granular policy-based isolation



Multi-Tenant Network

OPENSHIFT SDN - NETWORKPOLICY



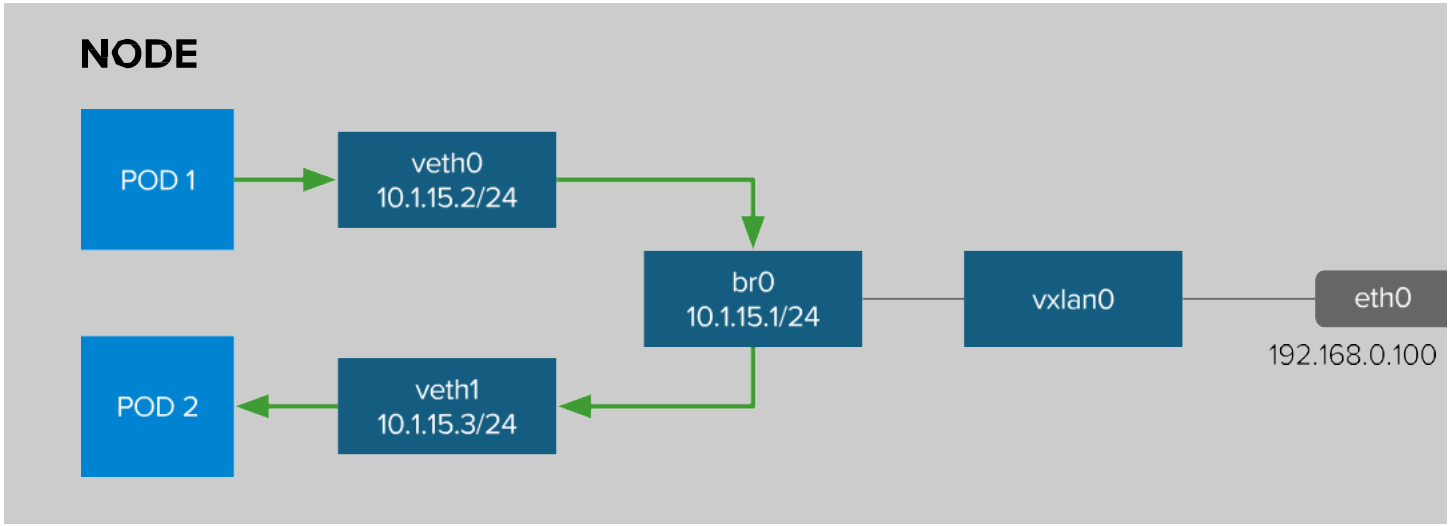
Example Policies

- Allow all traffic inside the project
- Allow traffic from green to gray
- Allow traffic to purple on 8080

```
apiVersion: extensions/v1beta1
kind: NetworkPolicy
metadata:
  name: allow-to-purple-on-8080
spec:
  podSelector:
    matchLabels:
      color: purple
  ingress:
  - ports:
    - protocol: tcp
      port: 8080
```

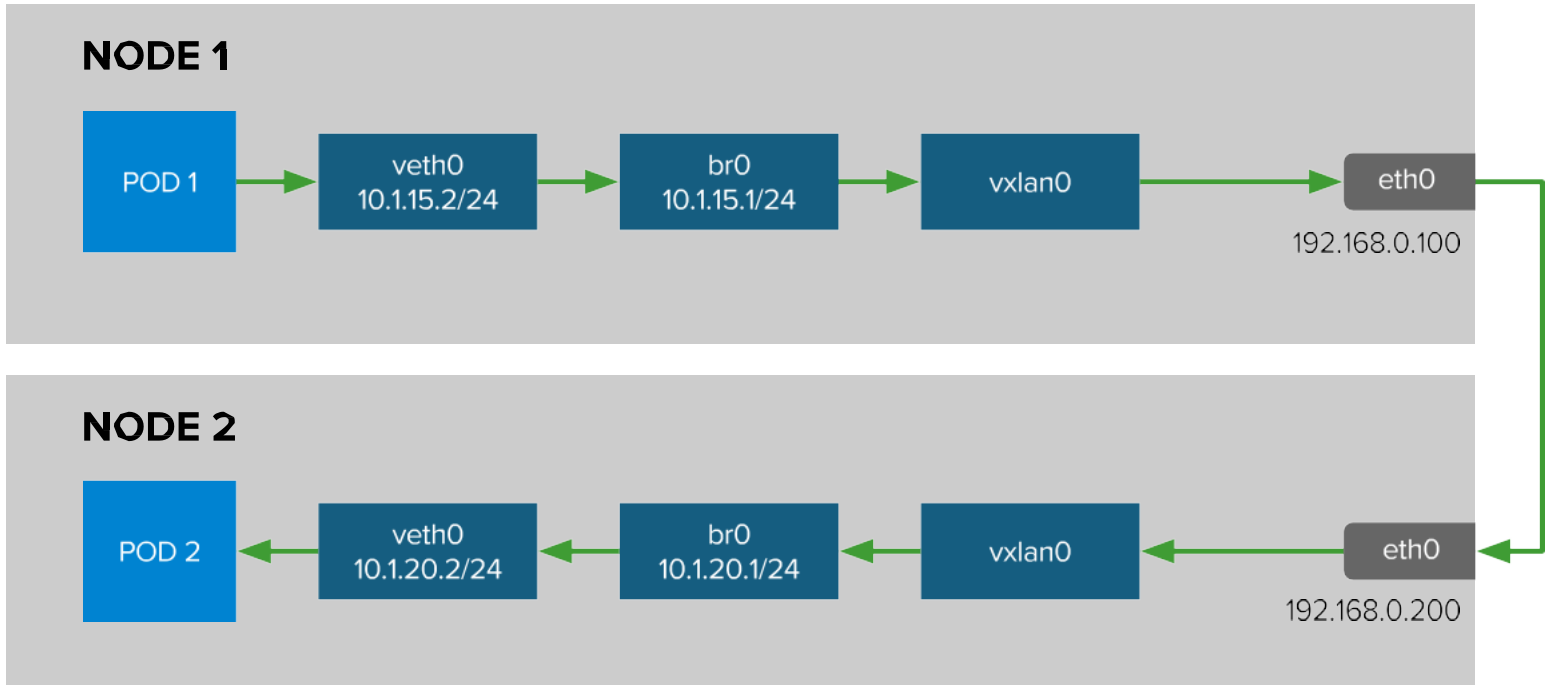
OPENSHIFT SDN - OVS PACKET FLOW

Container to Container on the Same Host



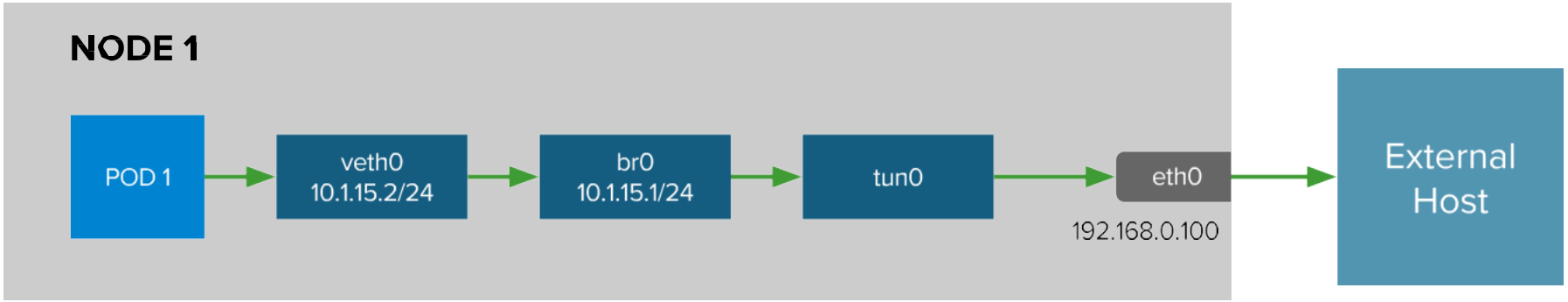
OPENSHIFT SDN - OVS PACKETFLOW

Container to Container on the Different Hosts

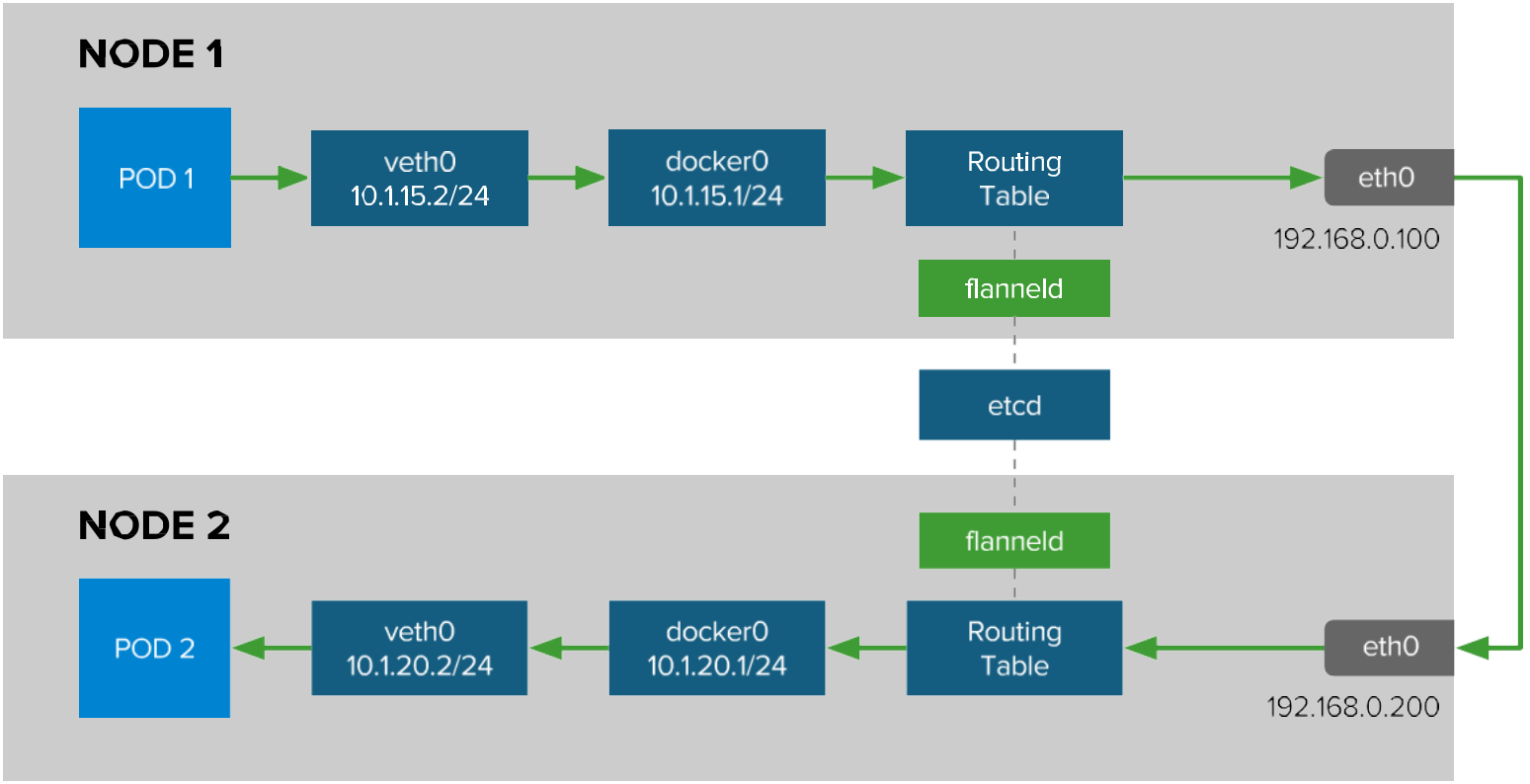


OPENSHIFT SDN - OVS PACKET FLOW

Container Connects to External Host

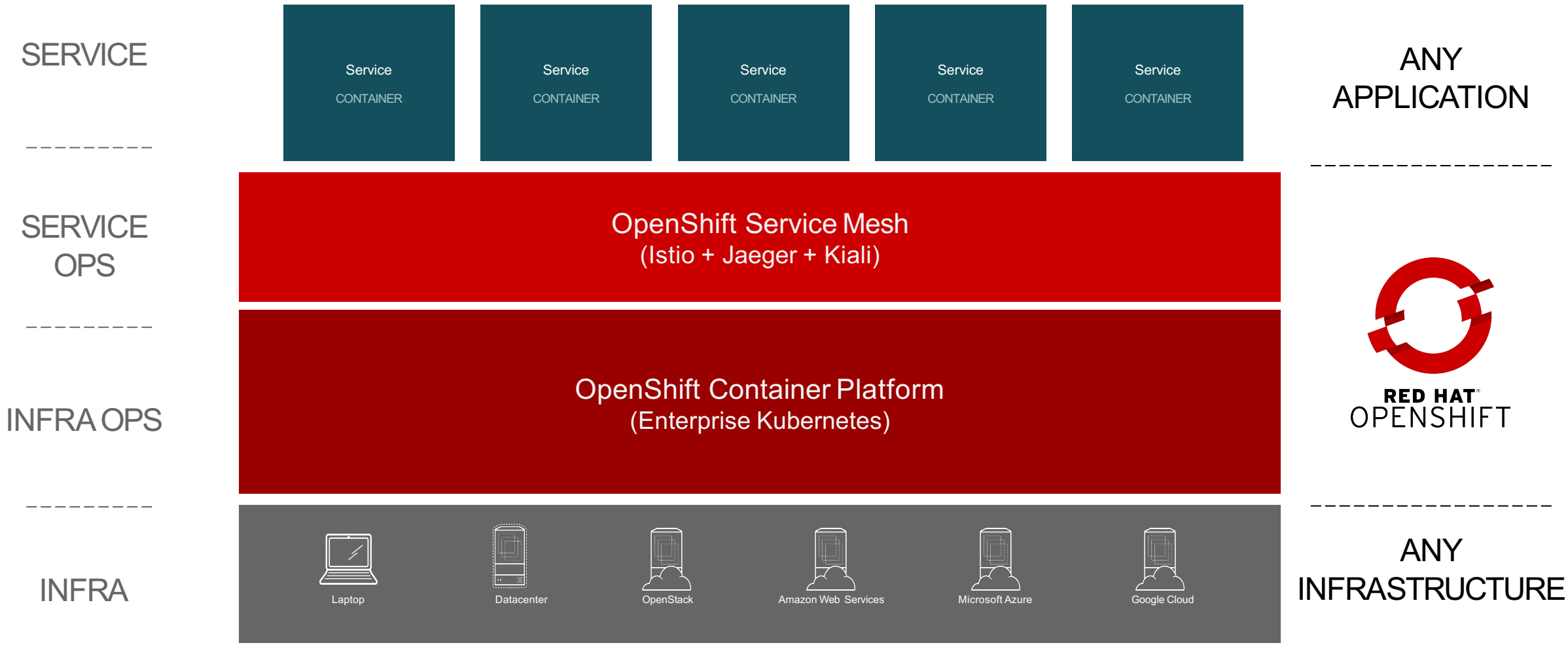


OPENSHIFT SDN WITH FLANNEL FOR OPENSTACK

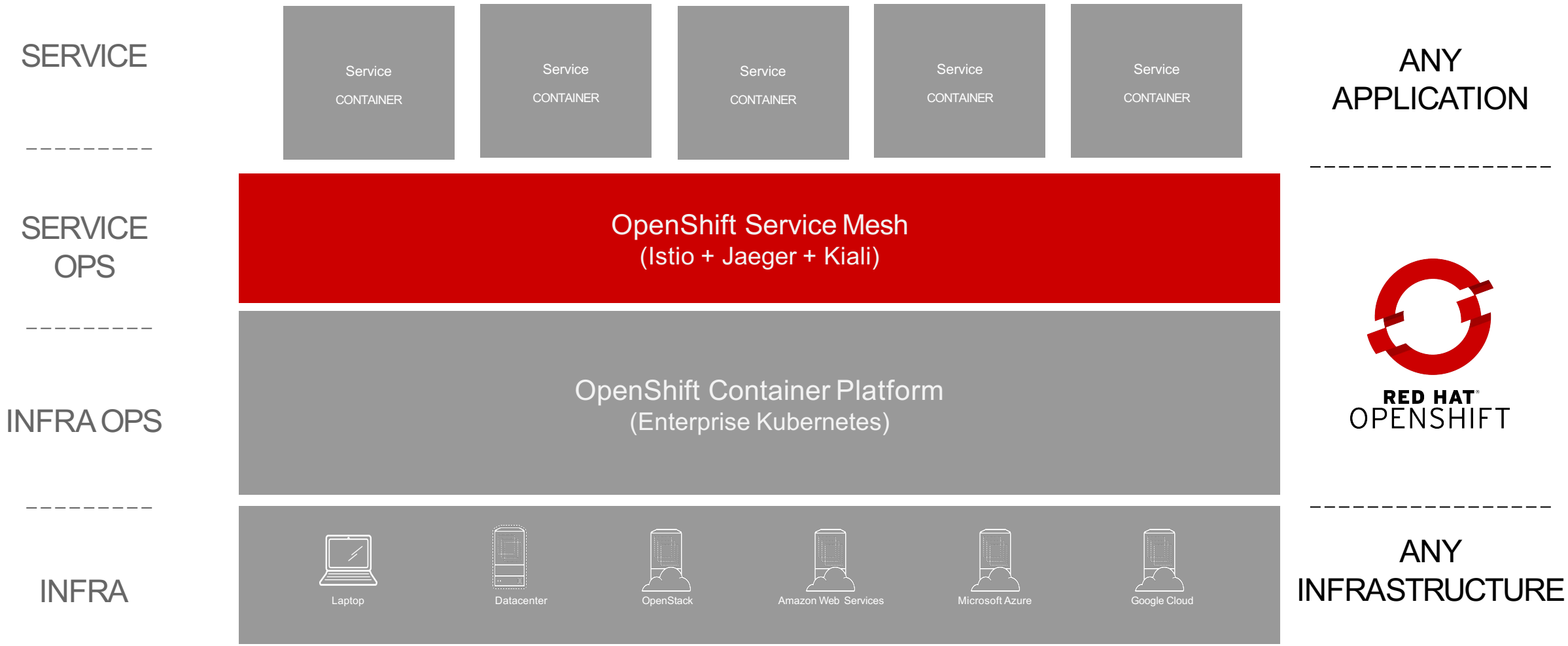


Flannel is minimally verified and is supported only and exactly as deployed in the OpenShift on OpenStack reference architecture https://access.redhat.com/articles/2743631

Distributed Services With Red Hat Openshift



Distributed Services With Red Hat Openshift



Microservice

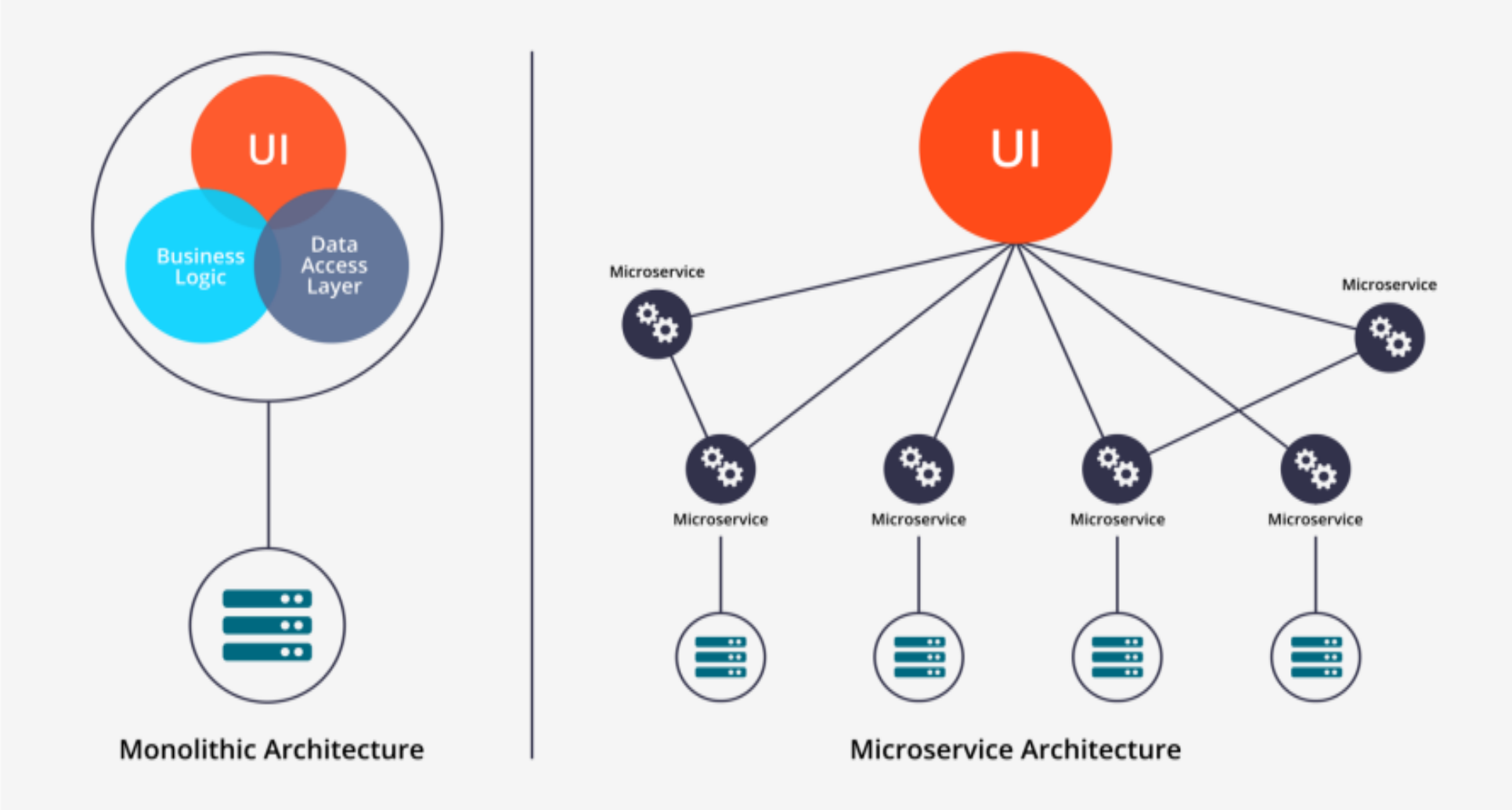
Microservice architecture is a software engineering paradigm of decomposing an application into **single function** modules with **well-defined interfaces** which are **independently** deployed and operated by **small teams** owning the **entire lifecycle** of the services

- Microservices provide a number of proven advantages over traditional monoliths
- Independent scaling of components
- Language neutral polyglot architecture
- Higher delivery velocity by minimizing communications and co-ordination between people
- etc.

Benefits come with a price

- Massively distributed systems are operationally complex

Application Architecture



Challenges With Kubernetes

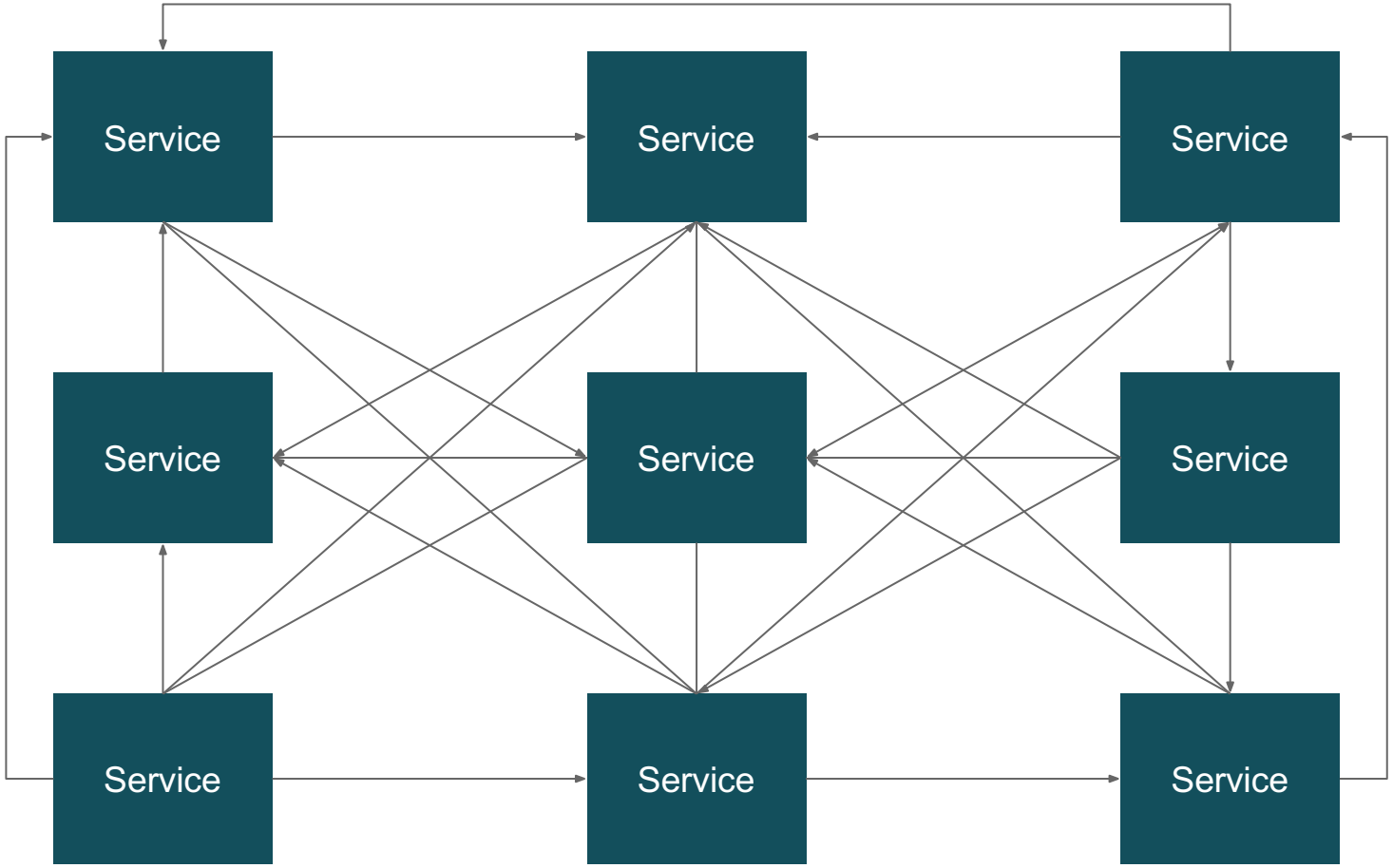
- The network among microservices may not be reliable.
- How can my microservice handle unpredictable failures and retry?
- How do I handle system degradation or topology change?
- How can I monitor and trace my microservices?
- As I develop multiple versions of my microservices, how can I easily dark launch and shift traffic?
- How can I ensure the communication among microservices is secure?
- How can I add enforce policies on my microservices?

Microservices Deployment Environments need

- Security
- A/B testing
- Blue/Green deployments
- Canary deployments
- Circuit breaking
- Fault injection
- Rate limiting
- Policy management
- In depth telemetry and reporting

Service Mesh

Service Mesh



What is a ‘Service Mesh’ ?

A network for services, not bytes

- Observability
- Resiliency
- Traffic Control
- Security
- Policy Enforcement

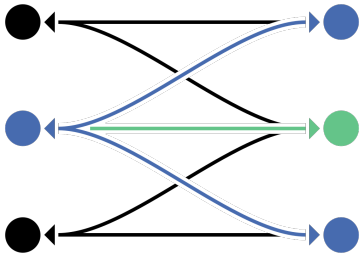
Istio

Value of Istio

An open service mesh platform to connect, observe, secure, and control microservices

Connect

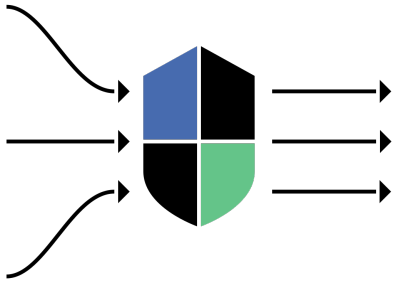
Traffic Control, Service Discovery, Load Balancing, Resiliency



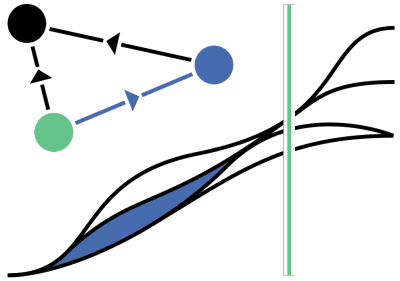
Secure

Encryption (TLS), Authentication, and Authorization of service-to-service communication

Control



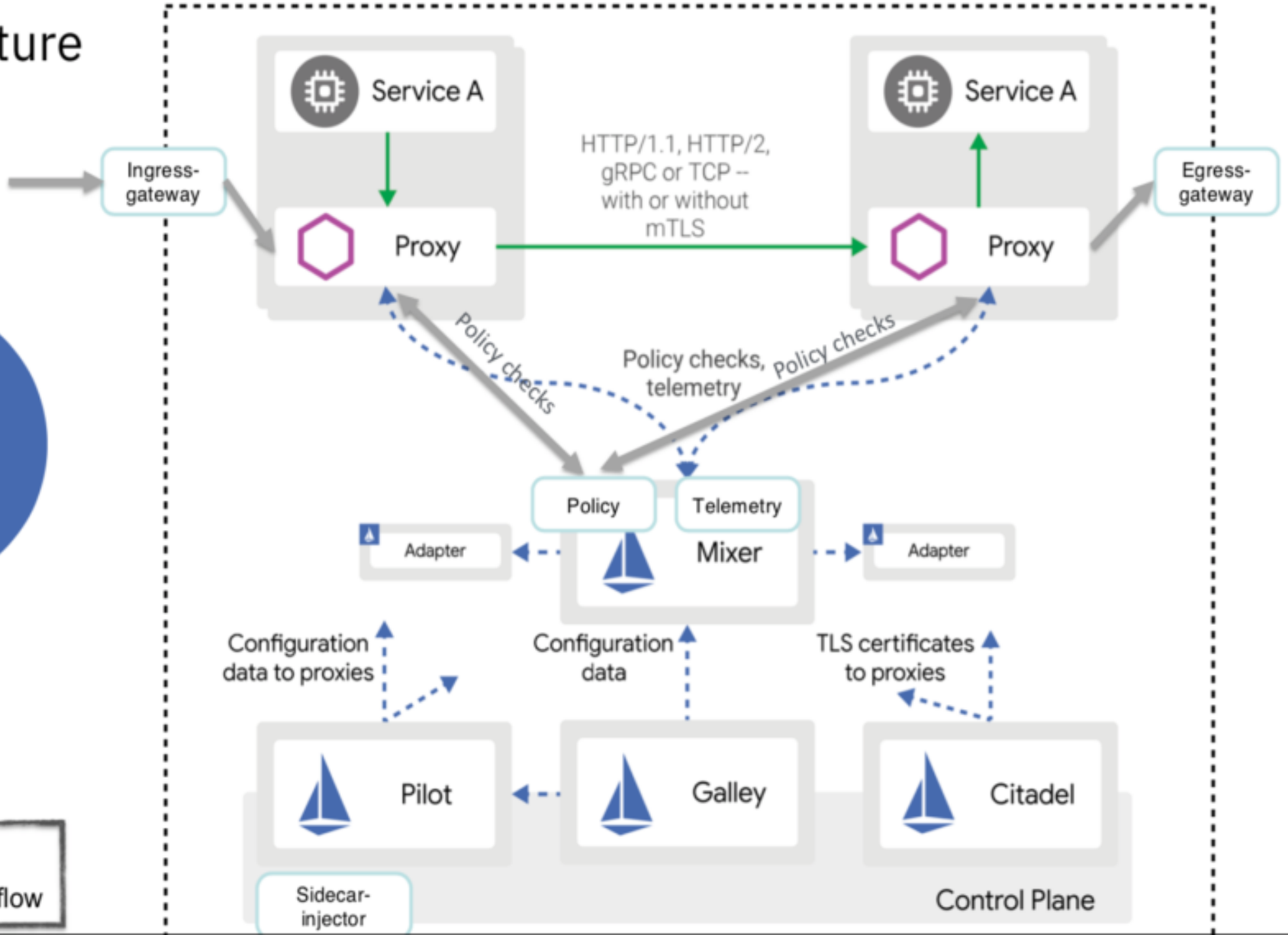
Policies enforcement – Authorization, Am I allowed to call that service?

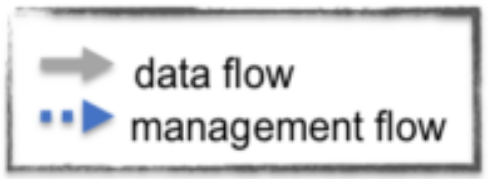


Observe

Automatic tracing, monitoring and logging of services

Istio Architecture

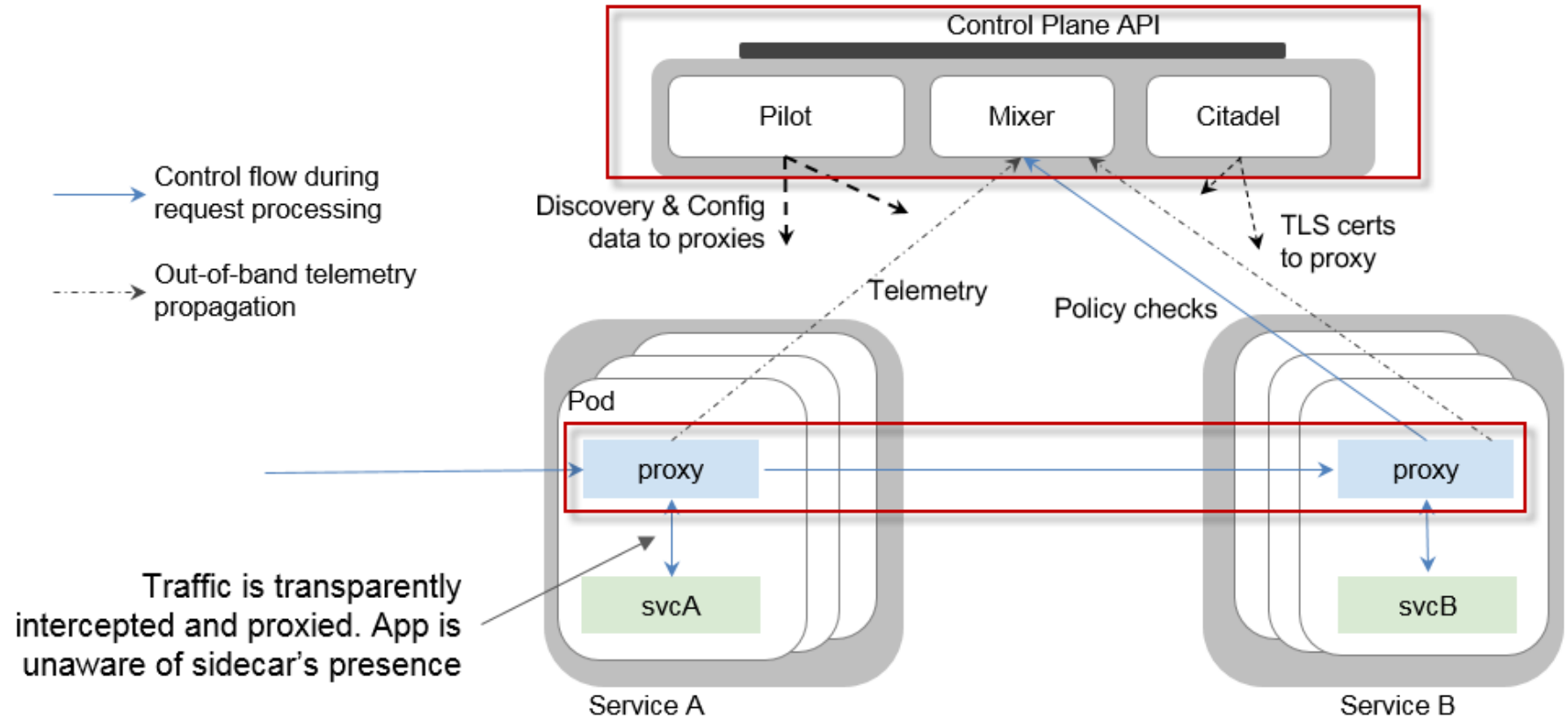




How Istio Works

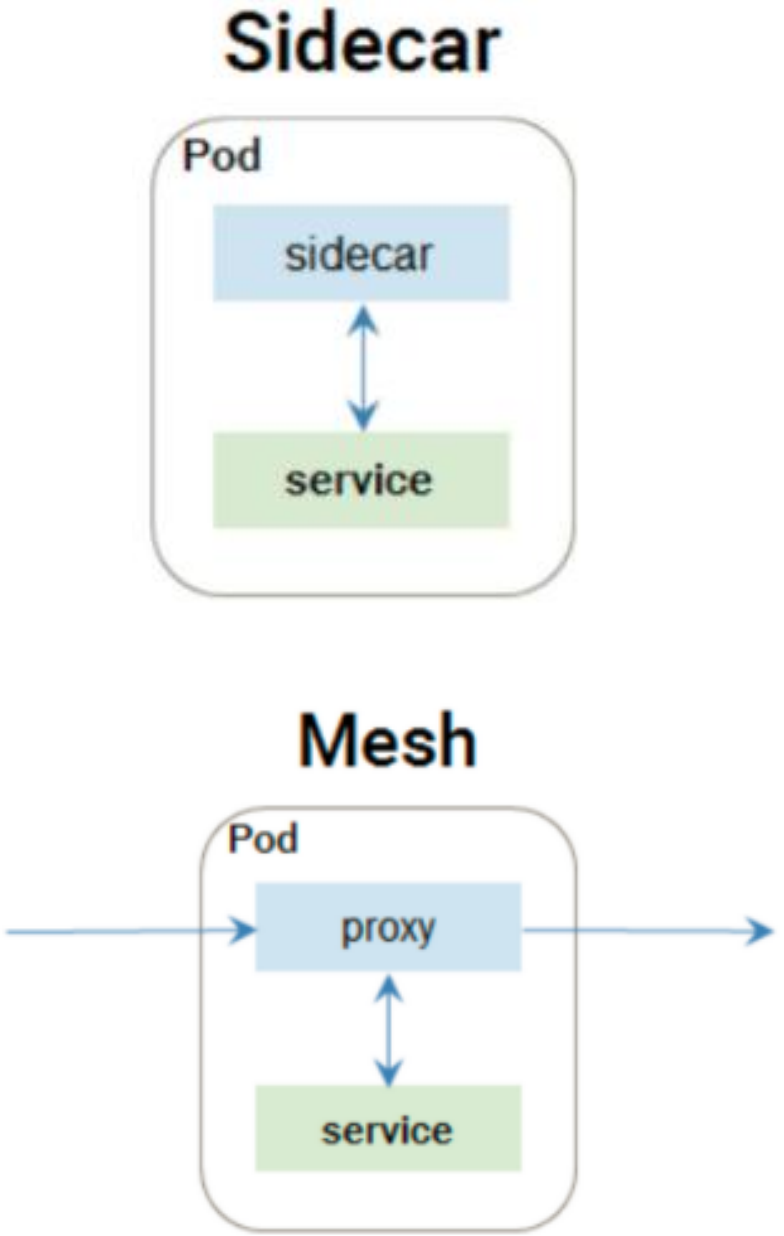
An Istio service mesh is logically split into a **data plane** and a **control plane**.

- The **data plane** is composed of a set of **intelligent proxies** (Envoy) deployed as **sidecars** that mediate and control all network communication among microservices.
- The **control plane** is responsible for managing and configuring proxies to **route traffic** and **enforcing policies** at runtime.



How Istio Works – Sidecar Proxy

- A proxy controls access to another object.
- Istio uses proxies between services and clients.
- It enables the **service mesh** to manage interactions
- A sidecar adds behavior to a container without changing it.
- Istio uses an extended version of the Envoy Proxy (from Lyft)



How Istio Works – Envoy sidecar

- Envoy is deployed as a **sidecar** to the relevant service in the same Kubernetes pod
  - The sidecar proxy model allows you to add Istio capabilities to an existing deployment **without rewriting** application code

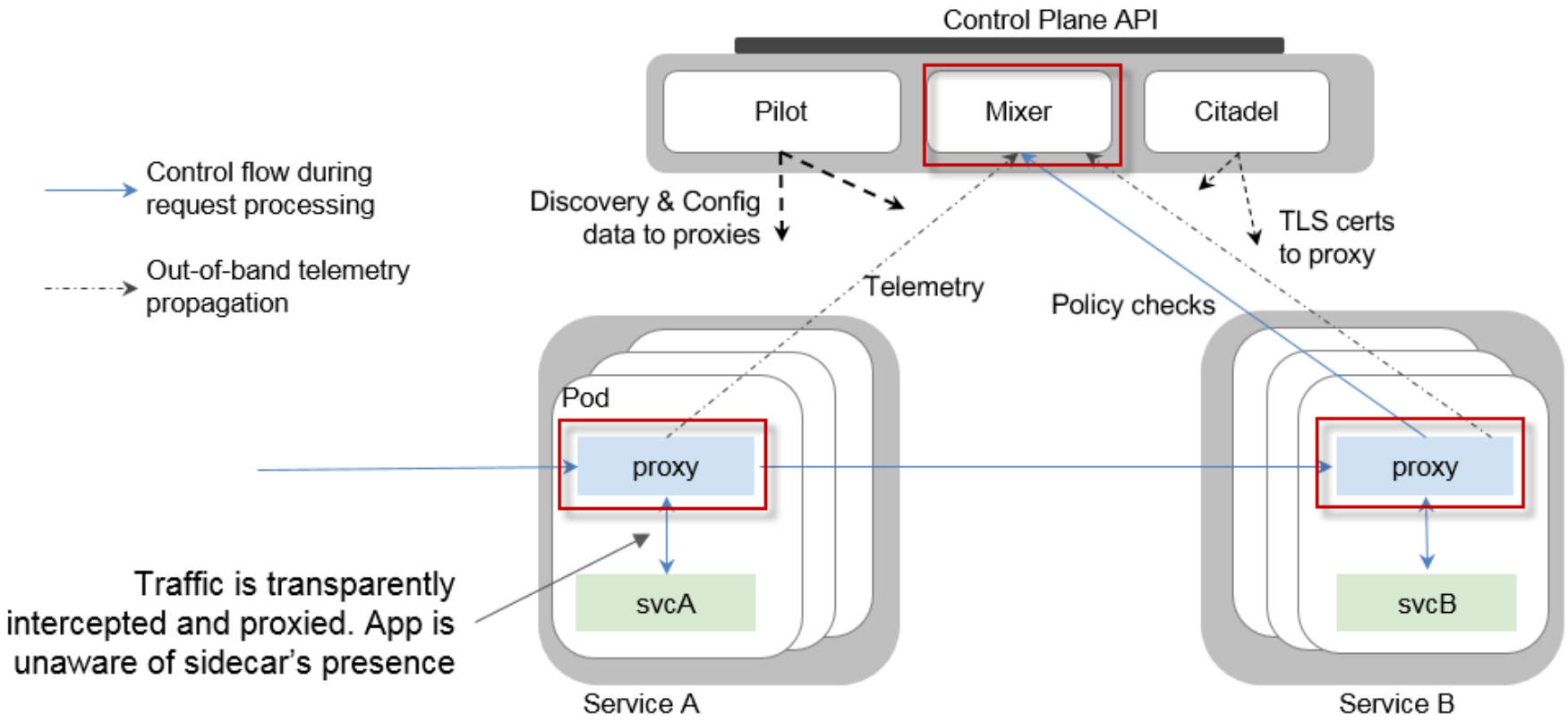
- The sidecar architecture allows Istio to extract information about traffic behavior
  - Information is then used in **Mixer** to enforce **policy** decisions
  - Sent to **monitoring** systems to provide information about the behavior of the entire mesh

- Istio uses Envoy's many built-in features
  - dynamic service discovery
  - load balancing
  - TLS termination
  - circuit breakers
  - health checks
  - staged rollouts with %-based traffic split, fault injection
  - rich metrics
  - transparent HTTP/1.1 to HTTP/2 proxy in both directions
  - Supports distributed tracing through 3rd party providers (Jeager for example)

How Istio Works – Mixer

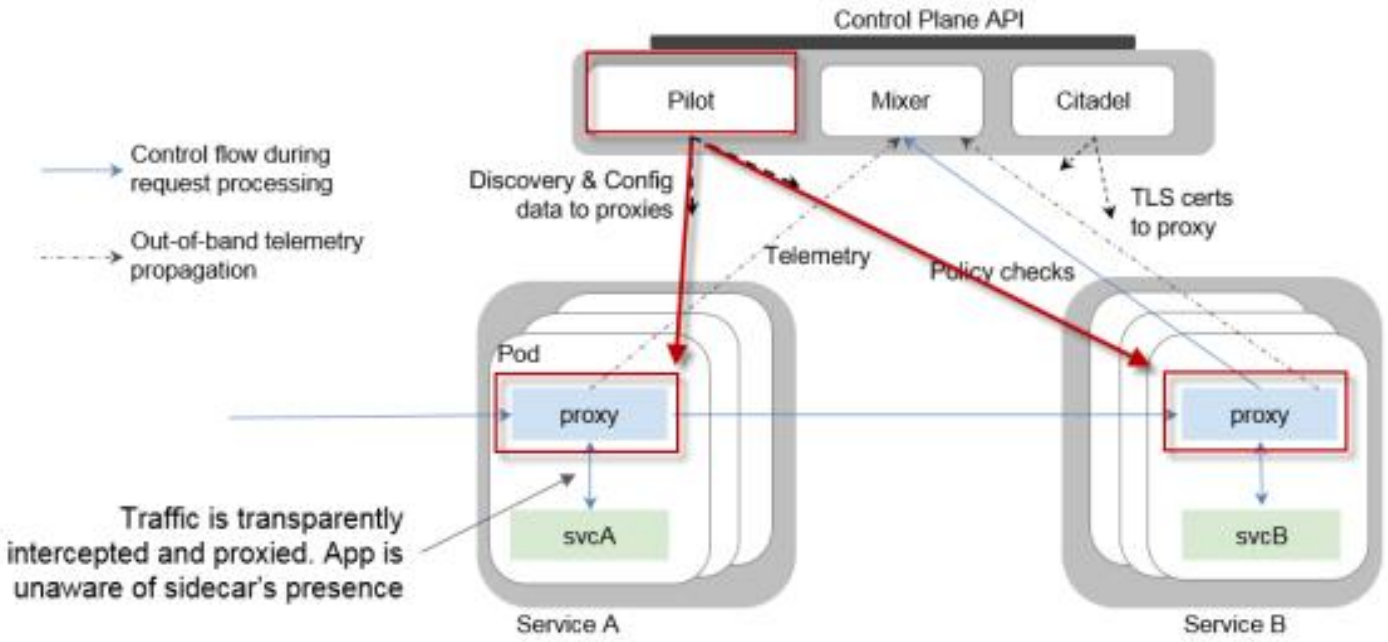
- Mixer is responsible for **enforcing access control** and usage policies across the service mesh, and **collecting telemetry data** from the Envoy proxy

- The Envoy proxy extracts request level attributes, which are sent to Mixer for evaluation



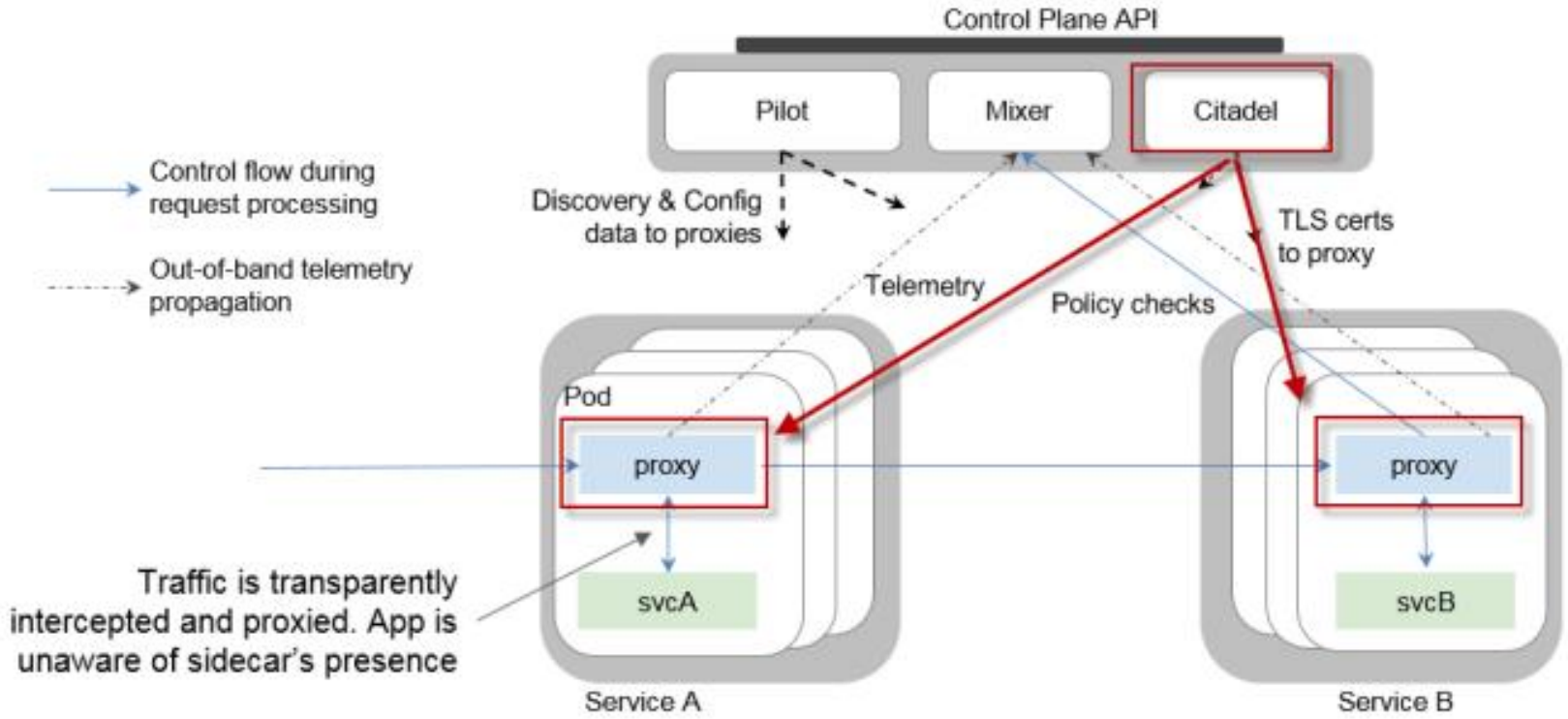
How Istio Works – Pilot

- Pilot provides
  - **service discovery** for the Envoy sidecars
  - **traffic management** capabilities for intelligent routing (for example, A/B tests and canary deployments)
  - **resiliency** (timeouts, retries, circuit breakers, and more)
- Pilot converts a high-level routing rules that control traffic behavior into Envoy-specific configurations and propagates them to the sidecars at runtime

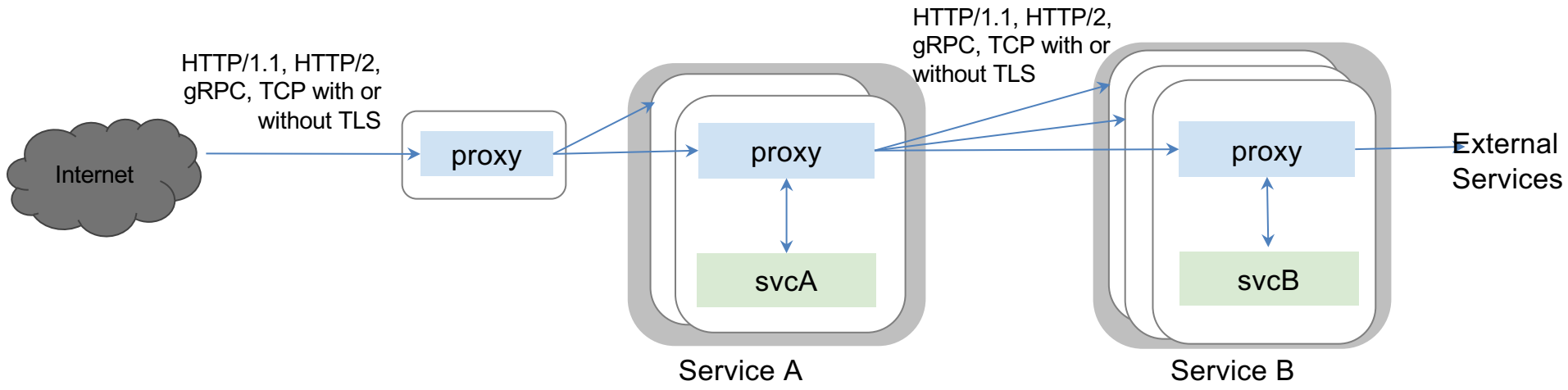


How Istio Works – Citadel

- Citadel enables strong service-to-service and end-user authentication with built-in identity and credential management.
  - Secure pod-to-pod or service-to-service communication at the network and application layers
  - Can be used to upgrade unencrypted traffic in the service mesh
  - Can provide operators the ability to enforce policy that is based on service identity rather than layer 3 or layer 4 network identifiers



Weaving the Mesh with Sidecars



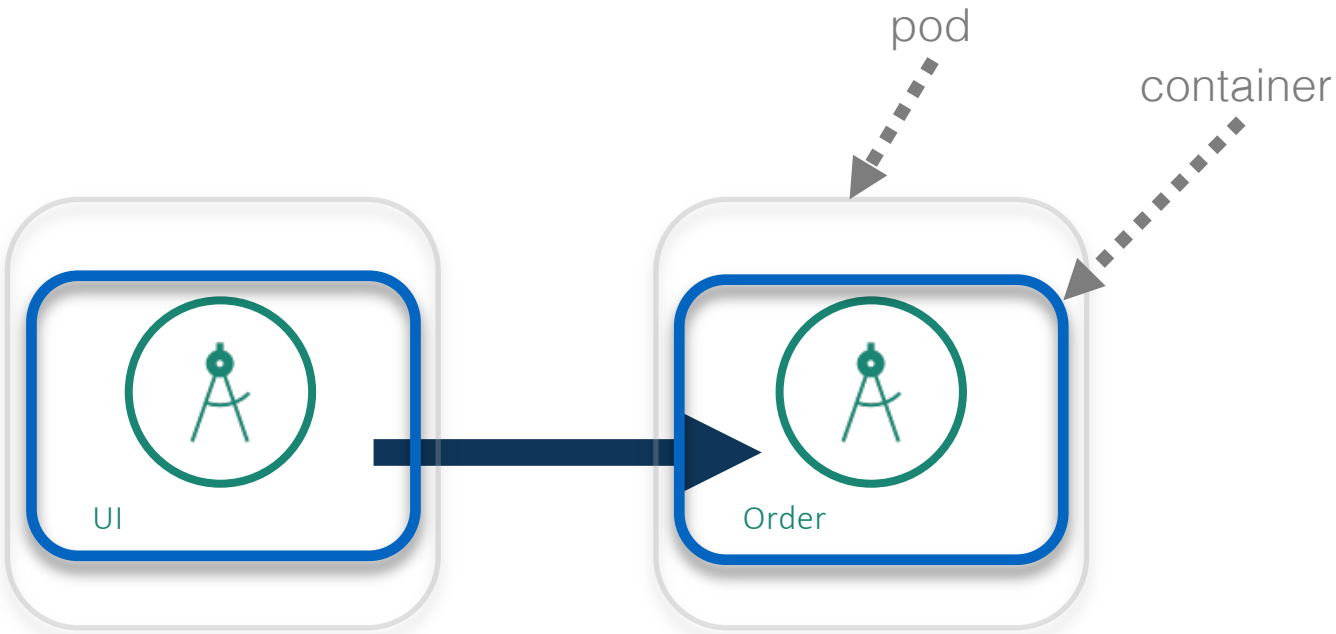
Inbound features:

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

Outbound features:

- Service authentication
- Load balancing
- Timeouts, retries and circuit breakers
- Connection pool sizing
- Fine-grained routing
- Telemetry
- Request Tracing
- Fault Injection

How does it work?



Without Istio:

- when service A (UI) talks to service B (Orders), it can use the local kube dns to find and talk to it directly.
- If there are multiple instances of the Order, it uses standard round robin.

Envoy Request Interception

Istio deploys a proxy, using a sidecar pattern, that sits next to each of the services

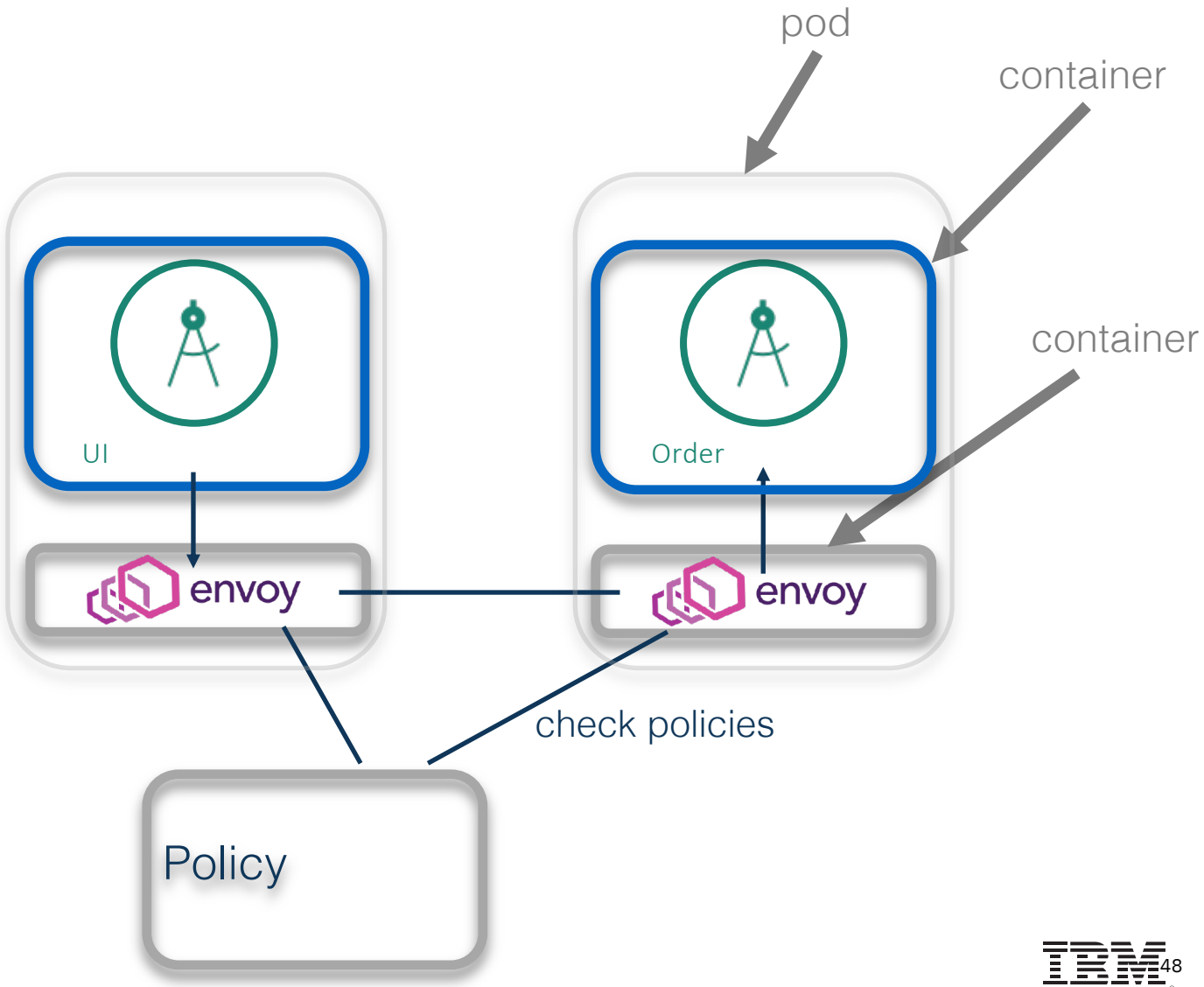
- **Service A -> Service B**

Client side

a) Locally. envoy traps the requests, using IP Tables
b) Envoy looks at that request, figures where we're going and then makes a client-side decision on where it is going to send that request
c) Envoy will find the **destination B host** and send the request

Server Side

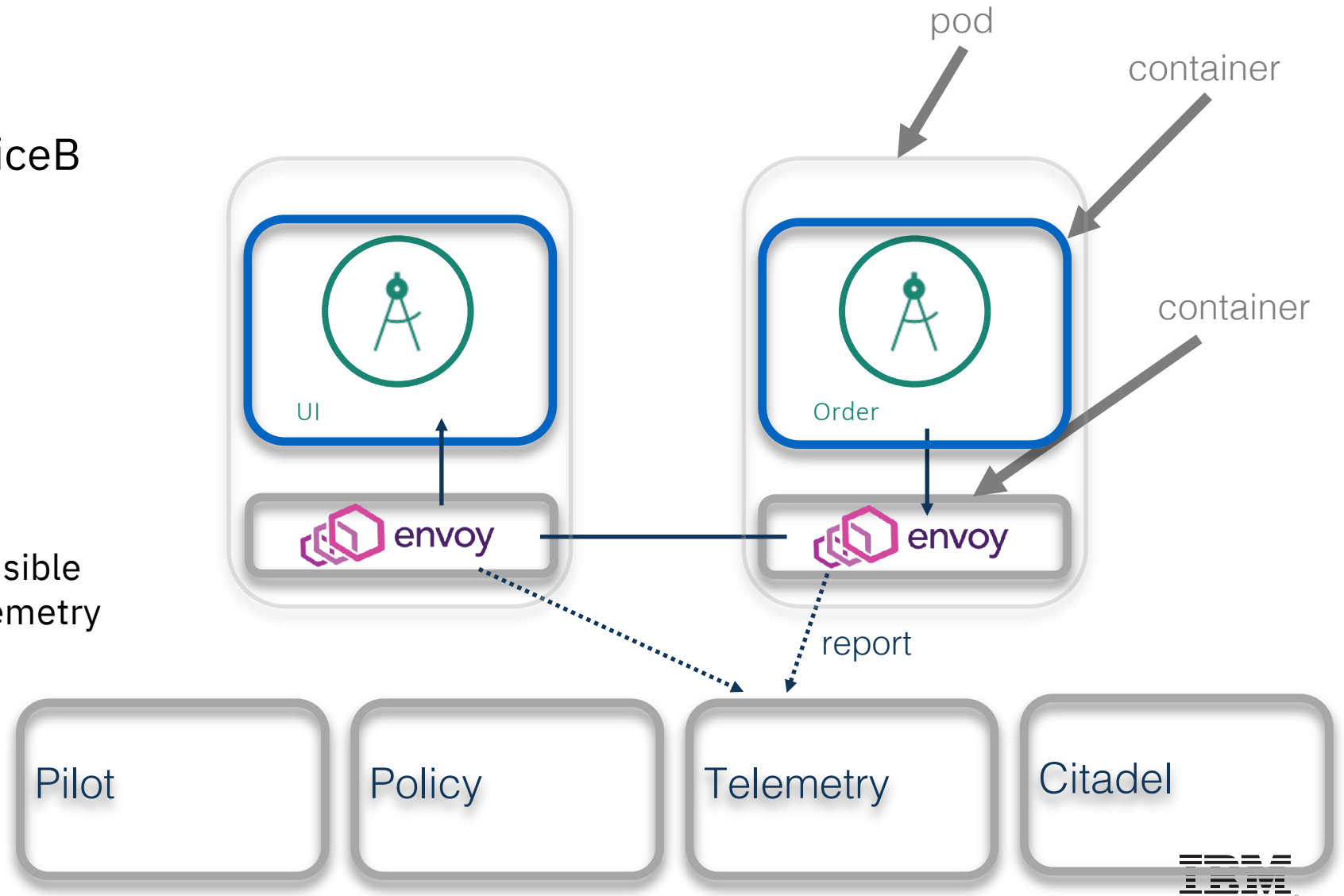
a) **Checks policies in Mixer** that this call is allowed, and responds to the B service request



Telemetry – Request and Response

- Both client and server asynchronously send data about the request for serviceB to the Mixer.
- Data is provided for both request and response for service B

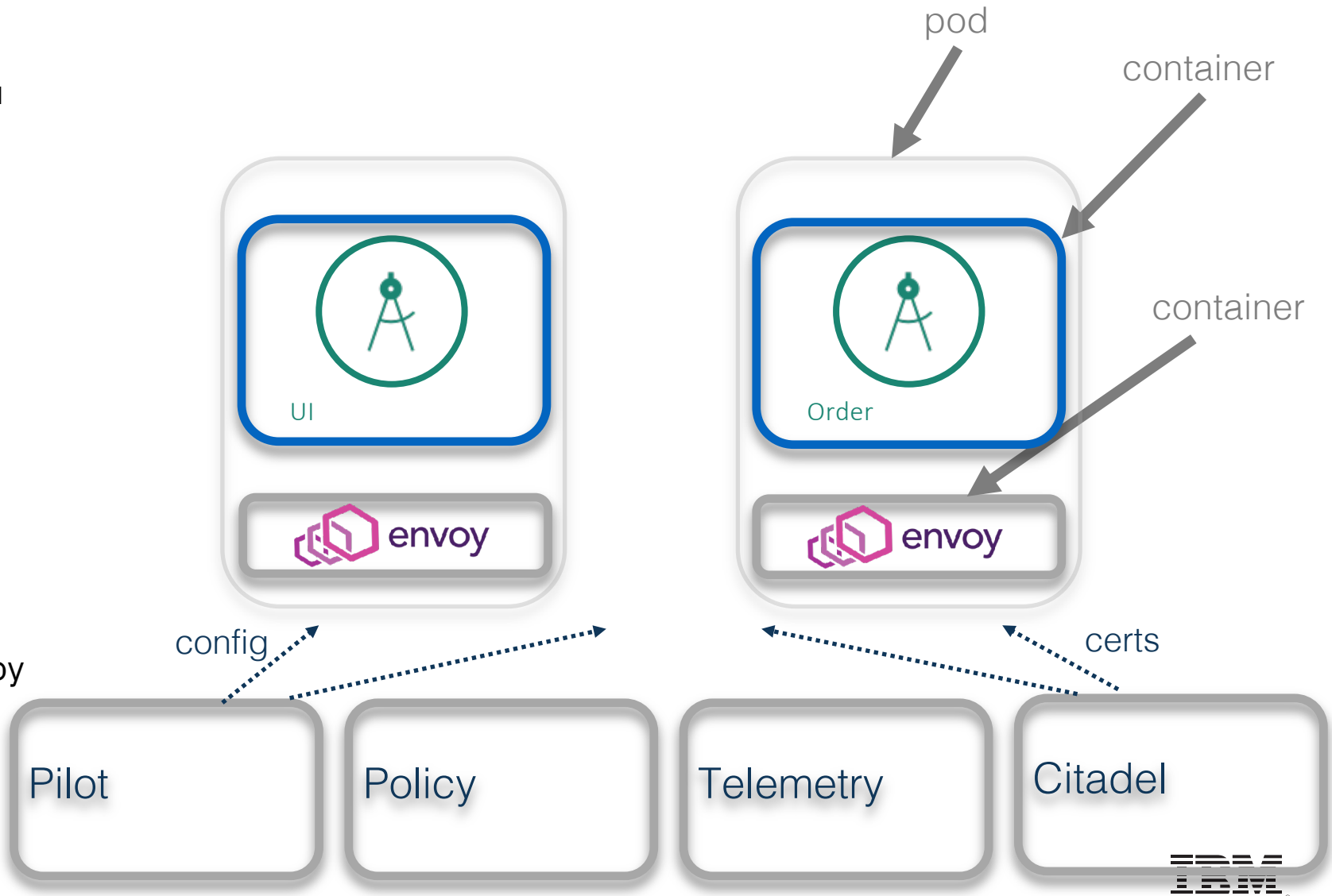
Mixer is the Istio component responsible for providing policy controls and telemetry collection:



Piloting Traffic

The core component used for traffic management in Istio is **Pilot**

- Pilot lets you specify which rules you want to use to **route traffic** between Envoy proxies
- You configure **failure recovery features** such as timeouts, retries, and circuit breakers.
- Pilot also maintains a canonical model of all the services in the mesh
- Pilot uses this model to let Envoy instances know about the other Envoy instances in the mesh for **service discovery**



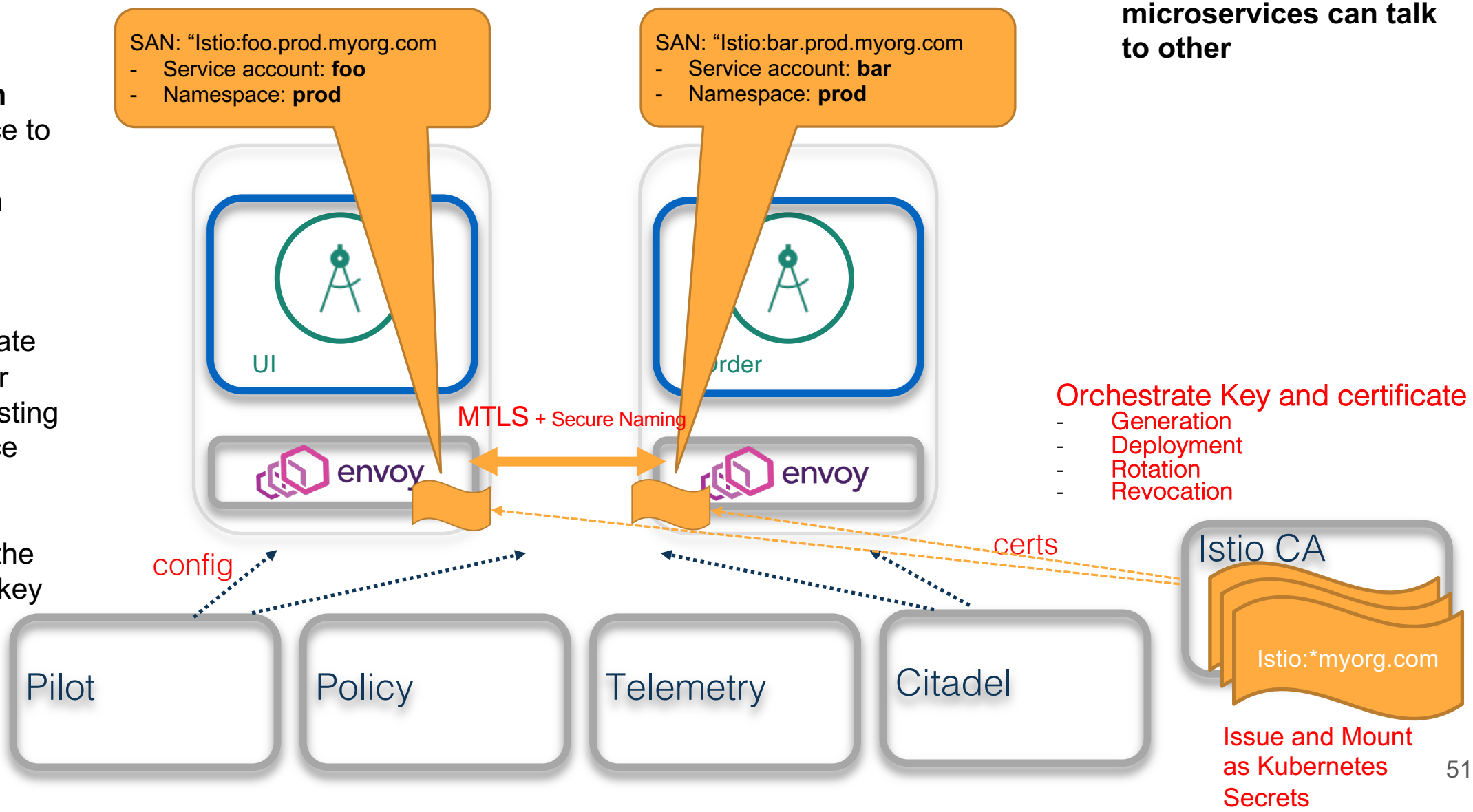
Securing Traffic (Citadel)

Istio provides **mutual TLS authentication** between service to service communication

Automatically creates certificate and key pair for each of the existing and new service accounts.

Citadel stores the certificate and key pairs as [Kubernetes secrets](#).

Istio can control which microservices can talk to other



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:

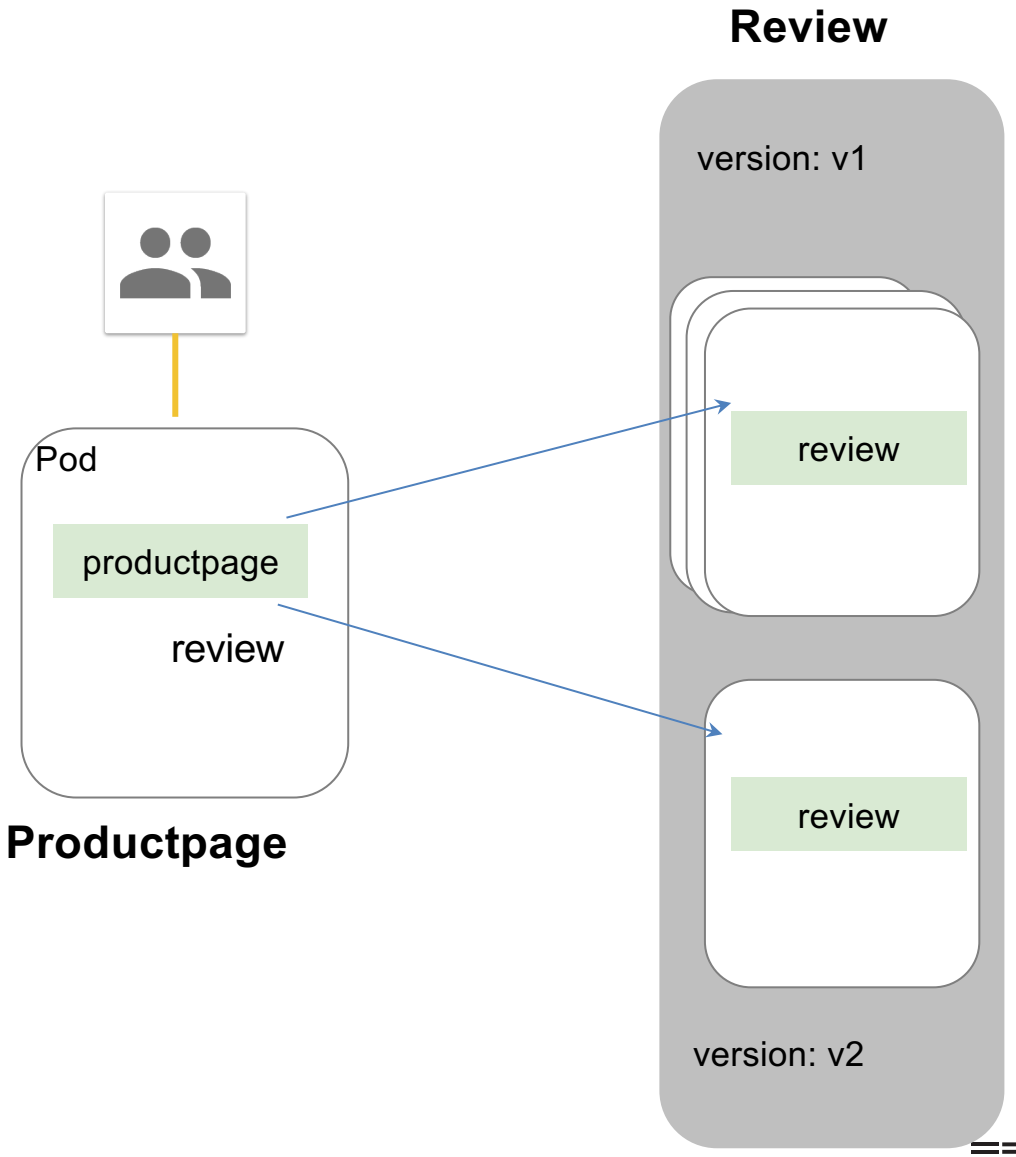
```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
```

deployment.yaml and DestinationRule (excerpts)

```
# Reviews service
apiVersion: v1
kind: Service
metadata:
  name: reviews
  labels:
    app: reviews
    service: reviews
spec:
  ports:
  - port: 9080
    name: http
  selector:
    app: reviews
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: reviews-v1
  labels:
    app: reviews
    version: v1
<omitted>
-
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: reviews-v2
  labels:
    app: reviews
    version: v2
<omitted>
```

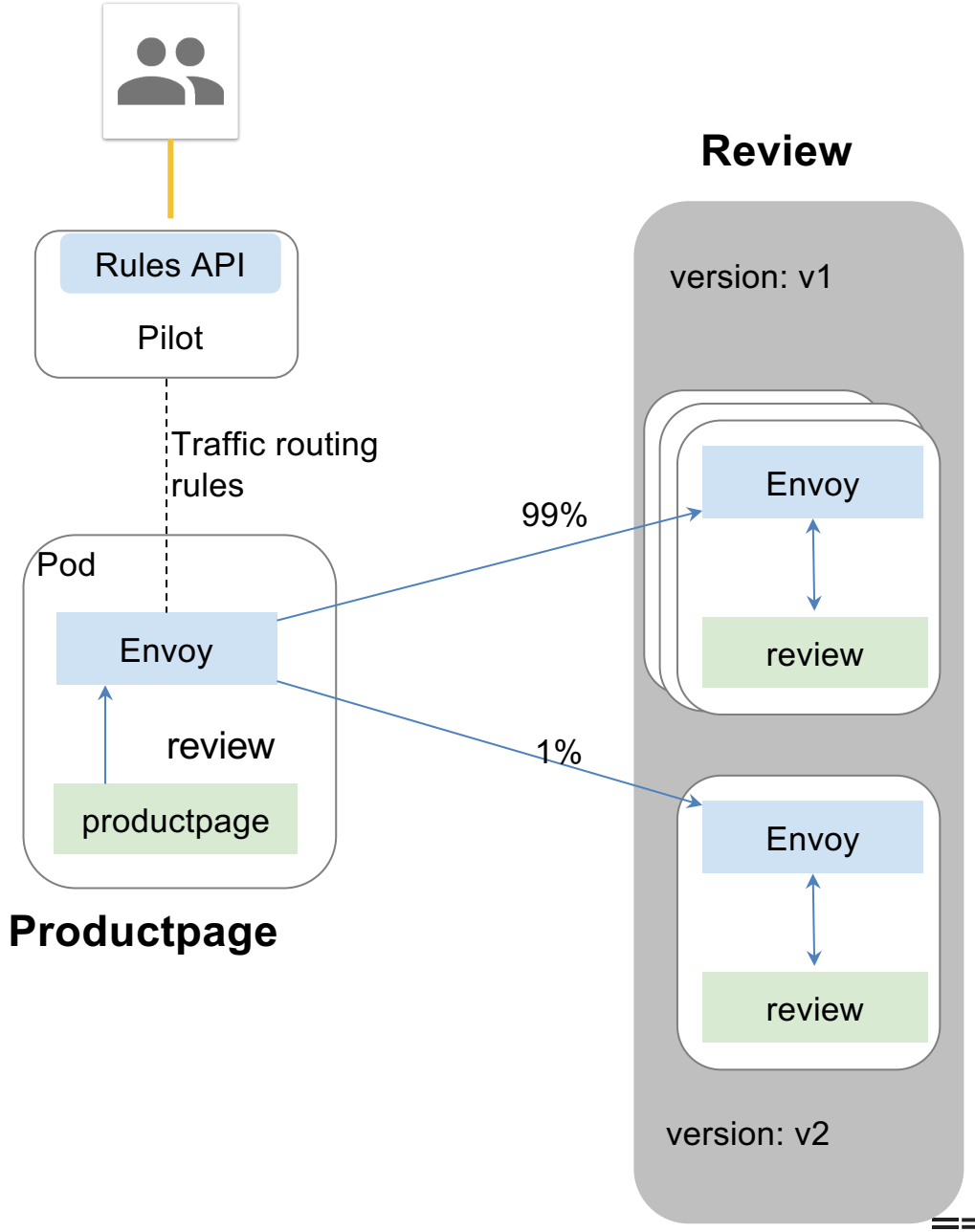
```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: productpage
spec:
  host: productpage
  subsets:
  - name: v1
    labels:
      version: v1

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
```



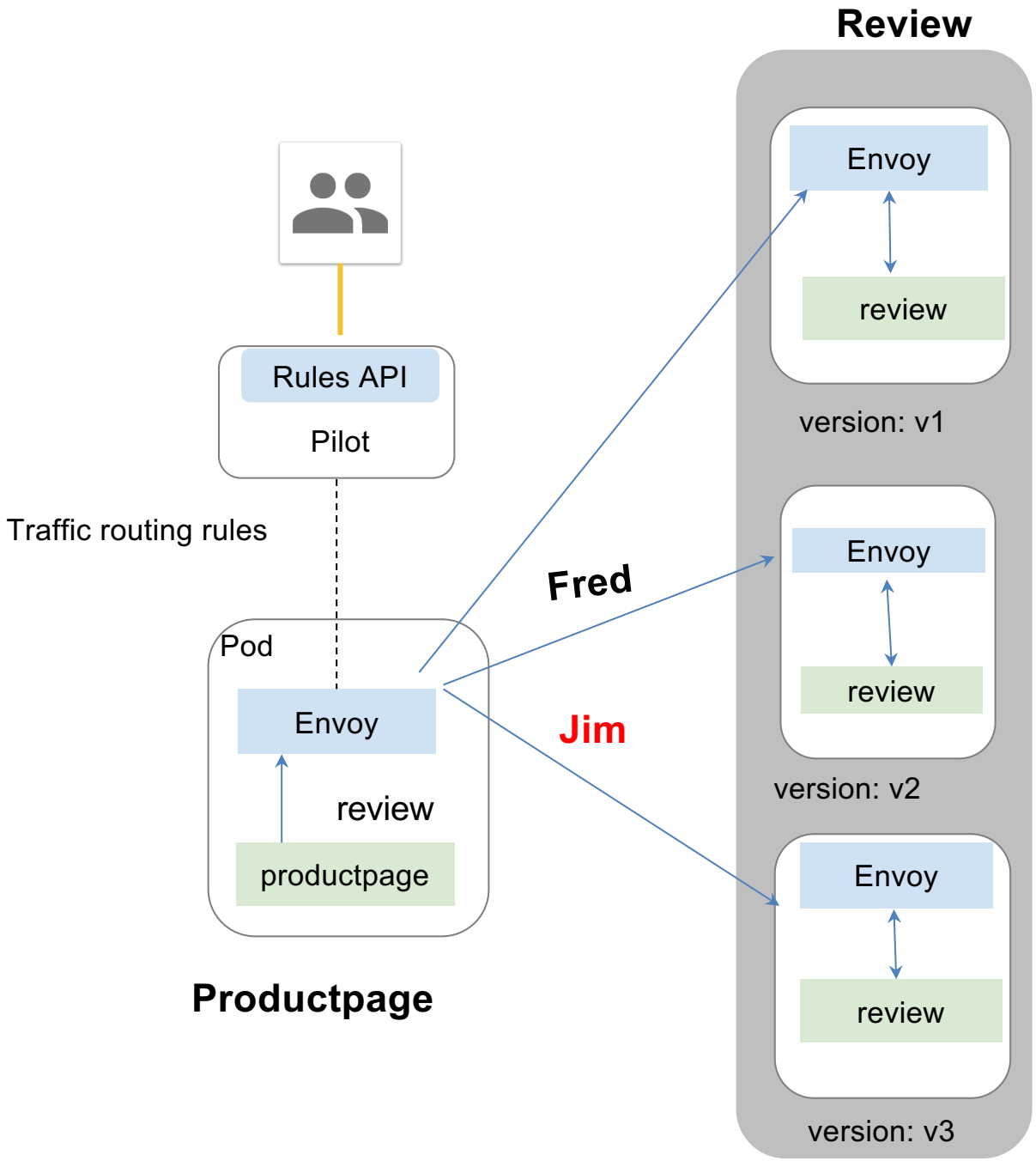
Traffic Splitting - Weight Based

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
      weight: 99
    - destination:
        host: reviews
        subset: v2
      weight: 1
```



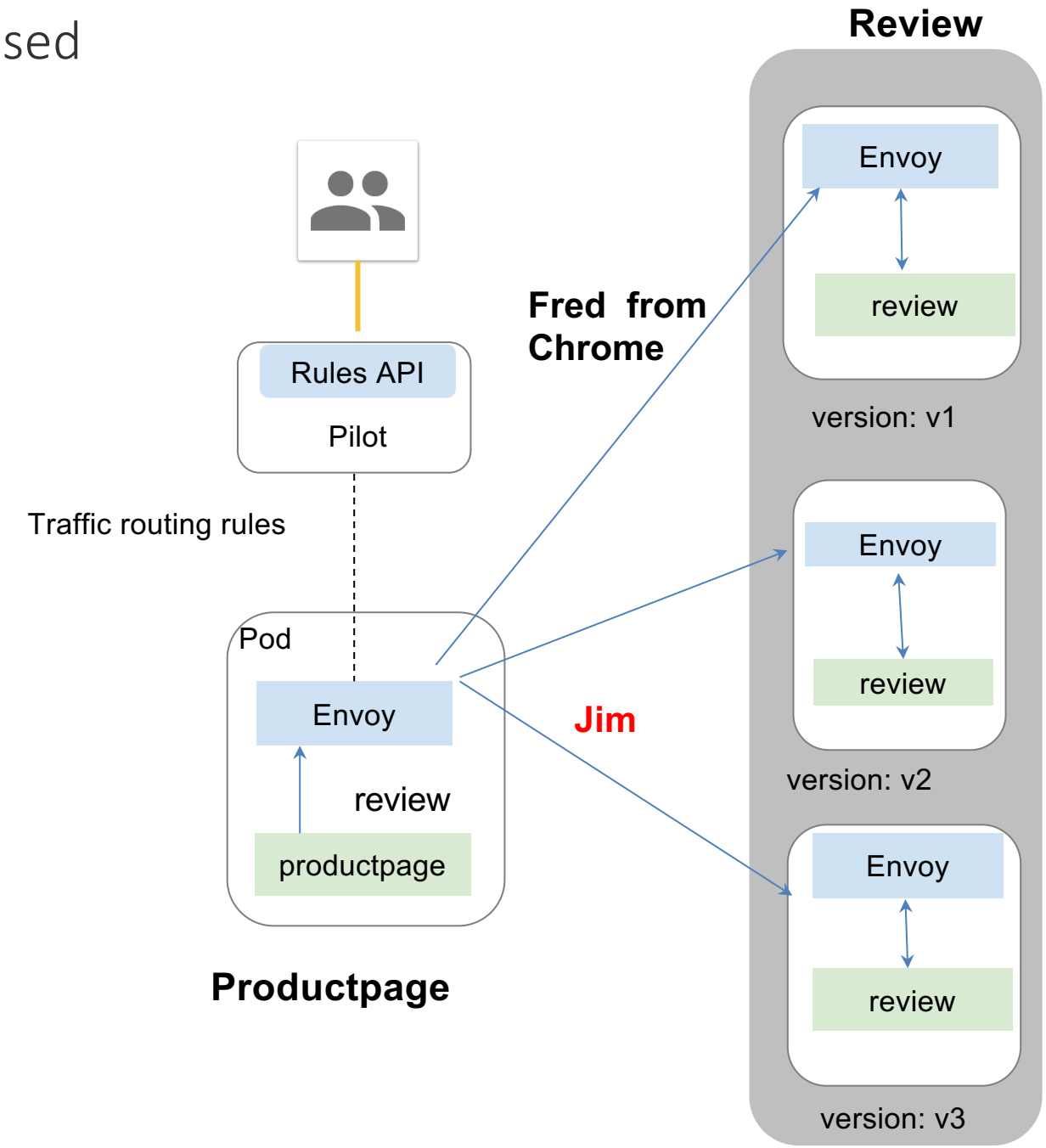
Traffic Splitting – Content Based

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - match:
     -headers:
          end-user:
             regex: (Fred|Don\ Andres)
  - route:
    - destination:
        host: reviews
        subset: v2
 -match:
   -headers:
          end-user:
             exact: Jim
- route:
   - destination:
        host: reviews
        subset: v3
- route:
   - destination:
        host: reviews
        subset: v1
```



Traffic Splitting – Complex Content Based

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - match:
     -headers:
          user-agent
            regex: ".*Chrome.*"
          end-user:
             regex: (Fred|Don\ Andres)
  - route:
    - destination:
        host: reviews
        subset: v2
 -match:
   -headers:
          end-user:
             exact: Jim
 - route:
    - destination:
        host: reviews
        subset: v3
 - route:
    - destination:
        host: reviews
        subset: v1
```



Istio Resiliency

- 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

Istio adds **resiliency and fault-tolerance** to your application without any change to source code

Circuit Breaker

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: httpbin
spec:
  host: httpbin
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
    outlierDetection:
      consecutiveErrors: 1
      interval: 1s
      baseEjectionTime: 3m
      maxEjectionPercent: 100
```

Time Out request after 1 second

Testing Failover - Delay Injection

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

Inject 7 seconds of delay only for Jason – which will exceed the 1 sec Circuit Breaker (prior slide)

Testing Failover - Fault Injection

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
  ...
spec:
  hosts:
  - ratings
  http:
  - fault:
      abort:
        httpStatus: 500
        percent: 100
    match:
    - headers:
        end-user:
          exact: jason
    route:
    - destination:
        host: ratings
        subset: v1
  - route:
    - destination:
        host: ratings
        subset: v1
```

Return 500 error code only for Jason

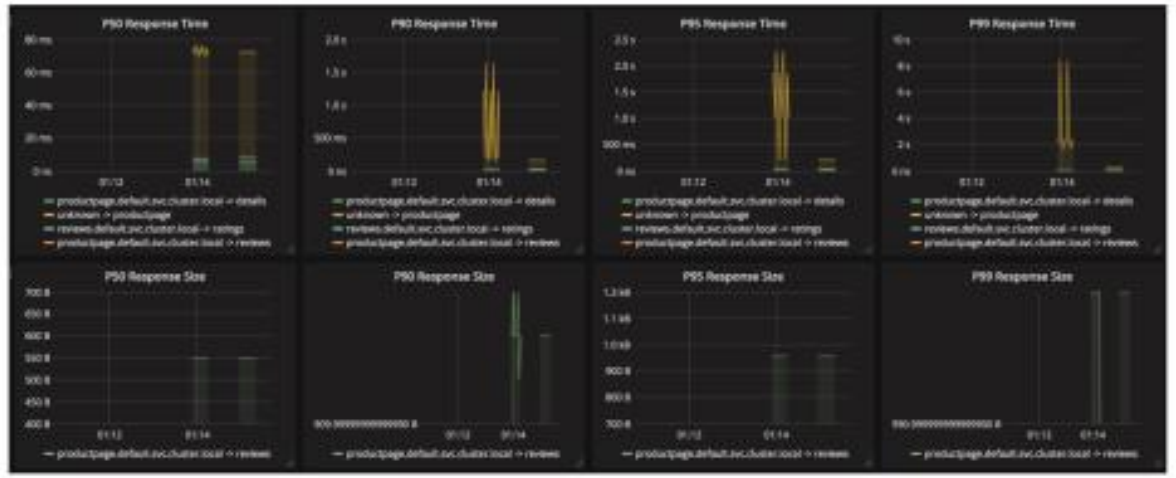
Visibility

Monitoring *&* ***tracing*** *should not be an afterthought in the infrastructure*

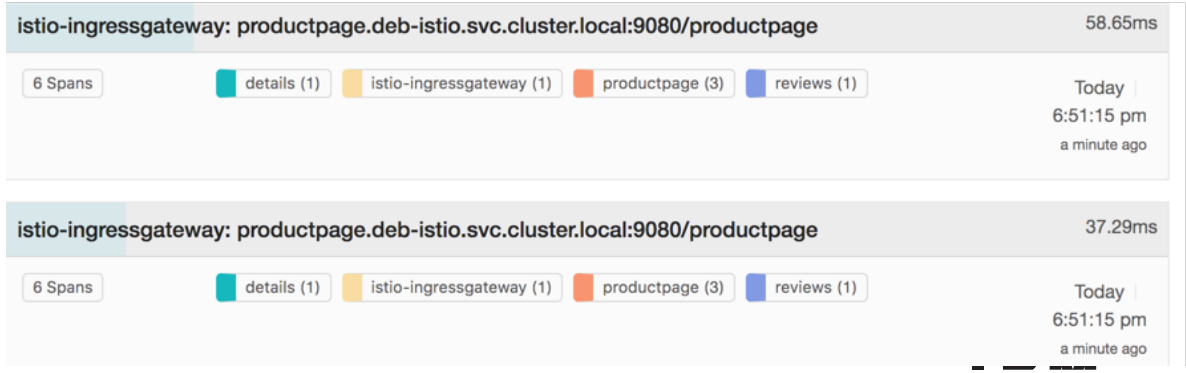
Goals

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

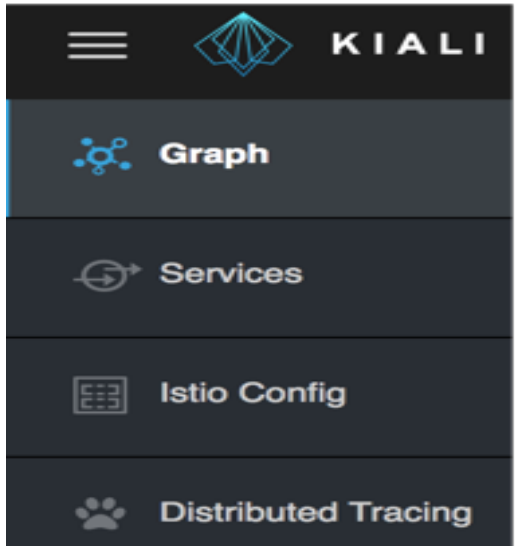
Istio - Grafana dashboard w/ Prometheus backend

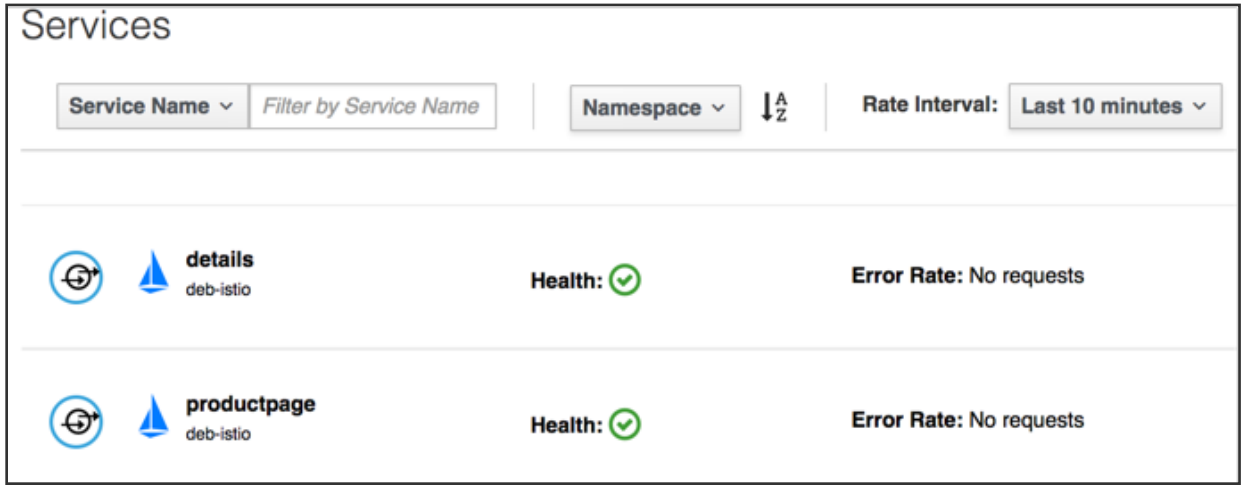


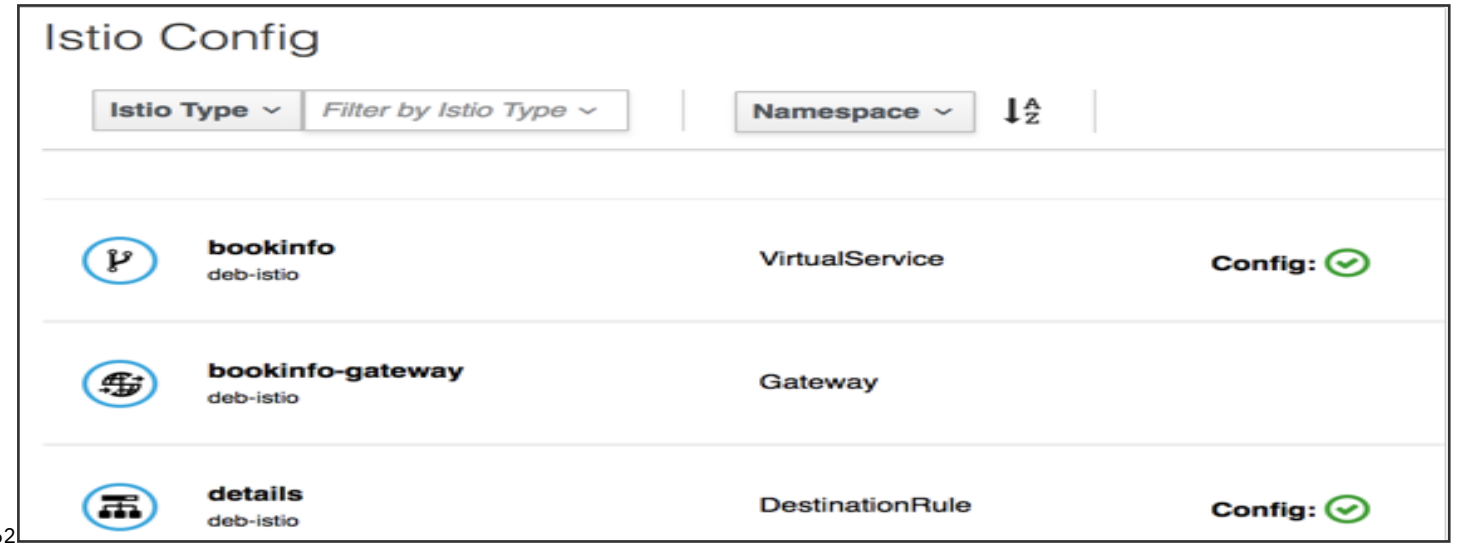
Istio - Jaegar tracing dashboard

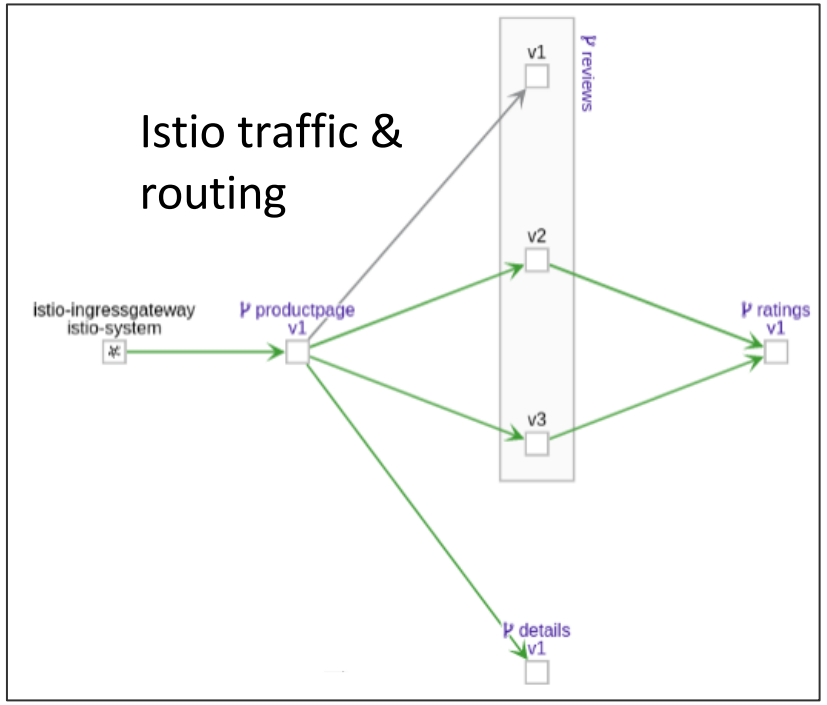


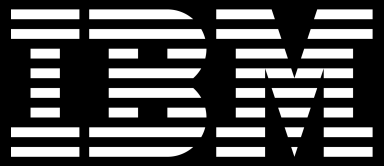
KIALI Provides Service Mesh Observability











WHAT YOU NEED FOR MICROSERVICES?

IBM

WHAT YOU NEED FOR MICROSERVICES?

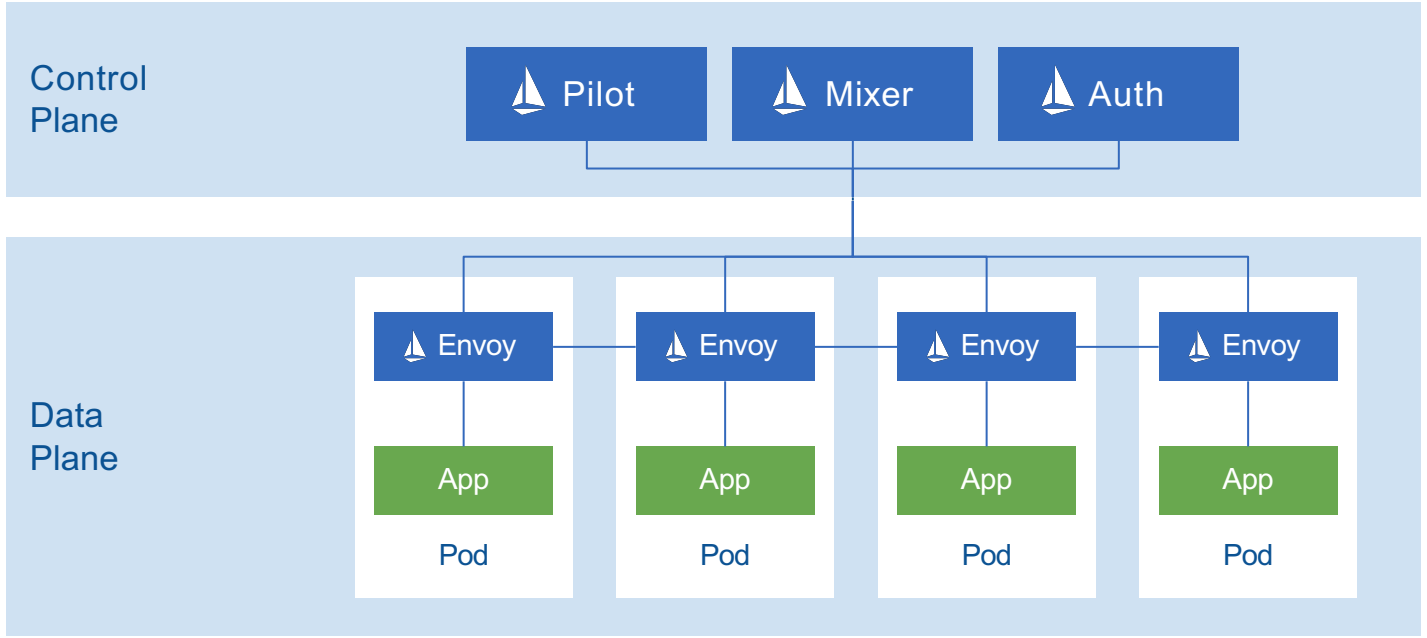
- Visibility & Reporting
- Resilience & Fault Tolerance
- Routing & Traffic Control
- Identity & Security
- Policy Enforcement

Istio

IBM

WHAT IS ISTIO?

a service mesh to connect, manage, and secure microservices



RED HAT® OPENSHIFT

COMING SOON

IBM