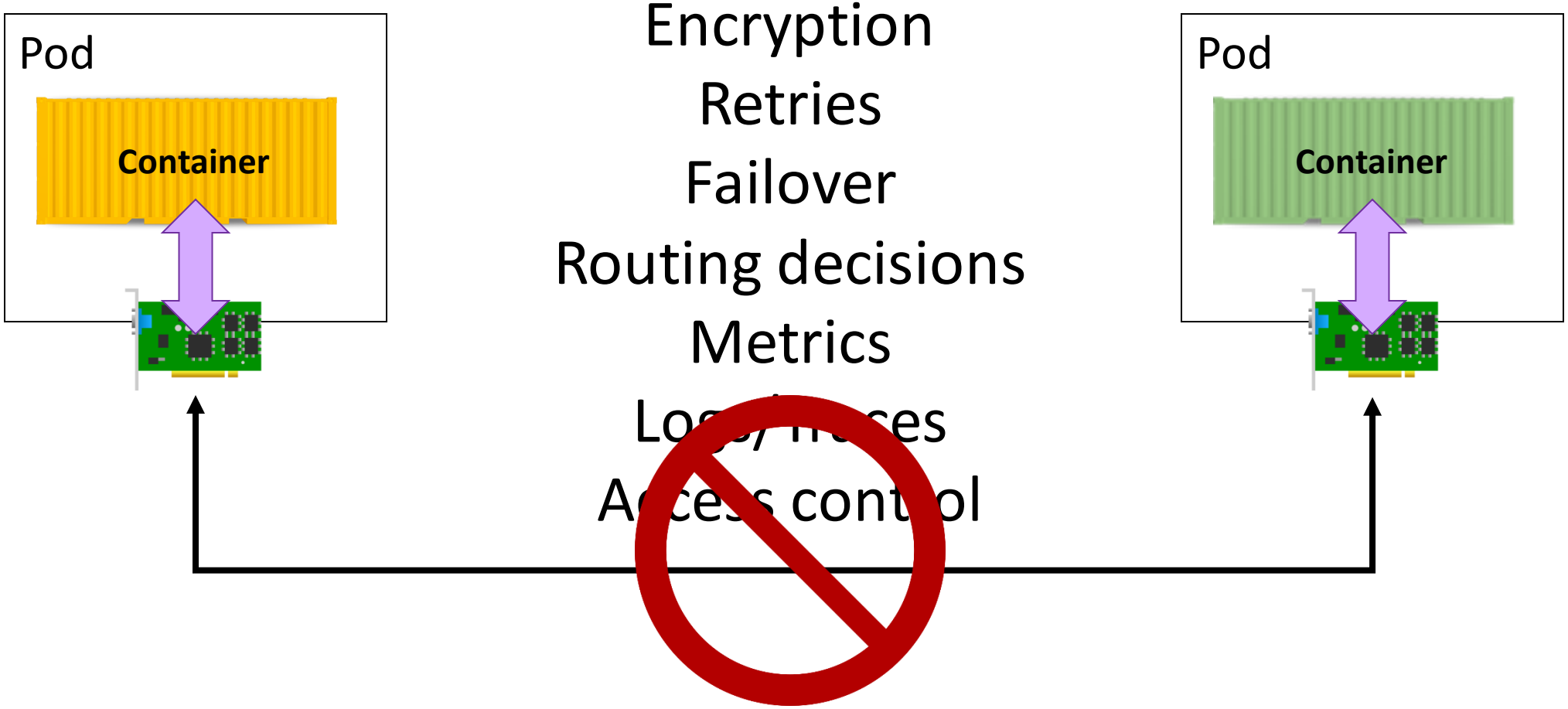
Istio

Microservices and Service Mesh

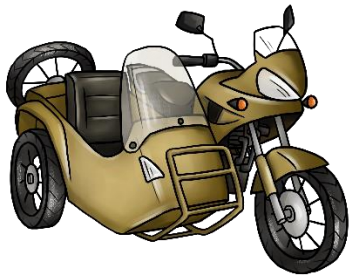
- An architectural style that decompose an application into loosely coupled services
- Each service implements a single functional concern
- Services communicate with teach other over the network
- A collection of these services fulfil a request
  - Service Mesh
- Services need to discover other services
- Service authentication and access control
- Dealing with network issues
- Collecting logs and metrics
- Ability to trace a request as it is fulfilled by multiple services

Microservices Communications

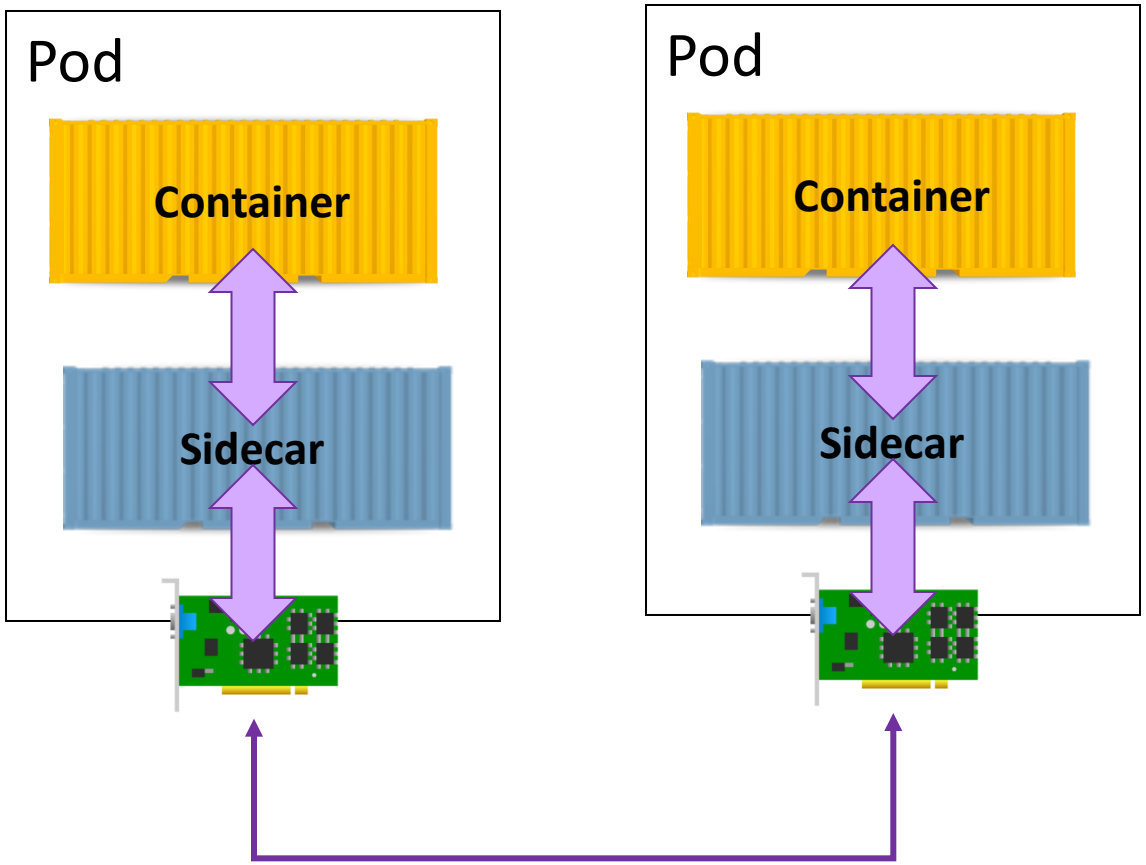


Encryption
Retries
Failover
Routing decisions
Metrics
Logs/traces
Access control

Sidecar



- Bundle observability, tracing, load balancing, etc. into a separate container
- Inject the container (sidecar) into every Pod
  - Sidecars are part of the infrastructure
- All communications in/out of the Pod is proxied across the sidecar
- Communicate with sidecar for configuration and data collection



What is Istio?

- Istio is a service mesh platform providing
  - Observability
  - Traffic management
  - Security
- Install on Kubernetes as a set of Pod and services to `istio-system` namespace

Istio Architecture

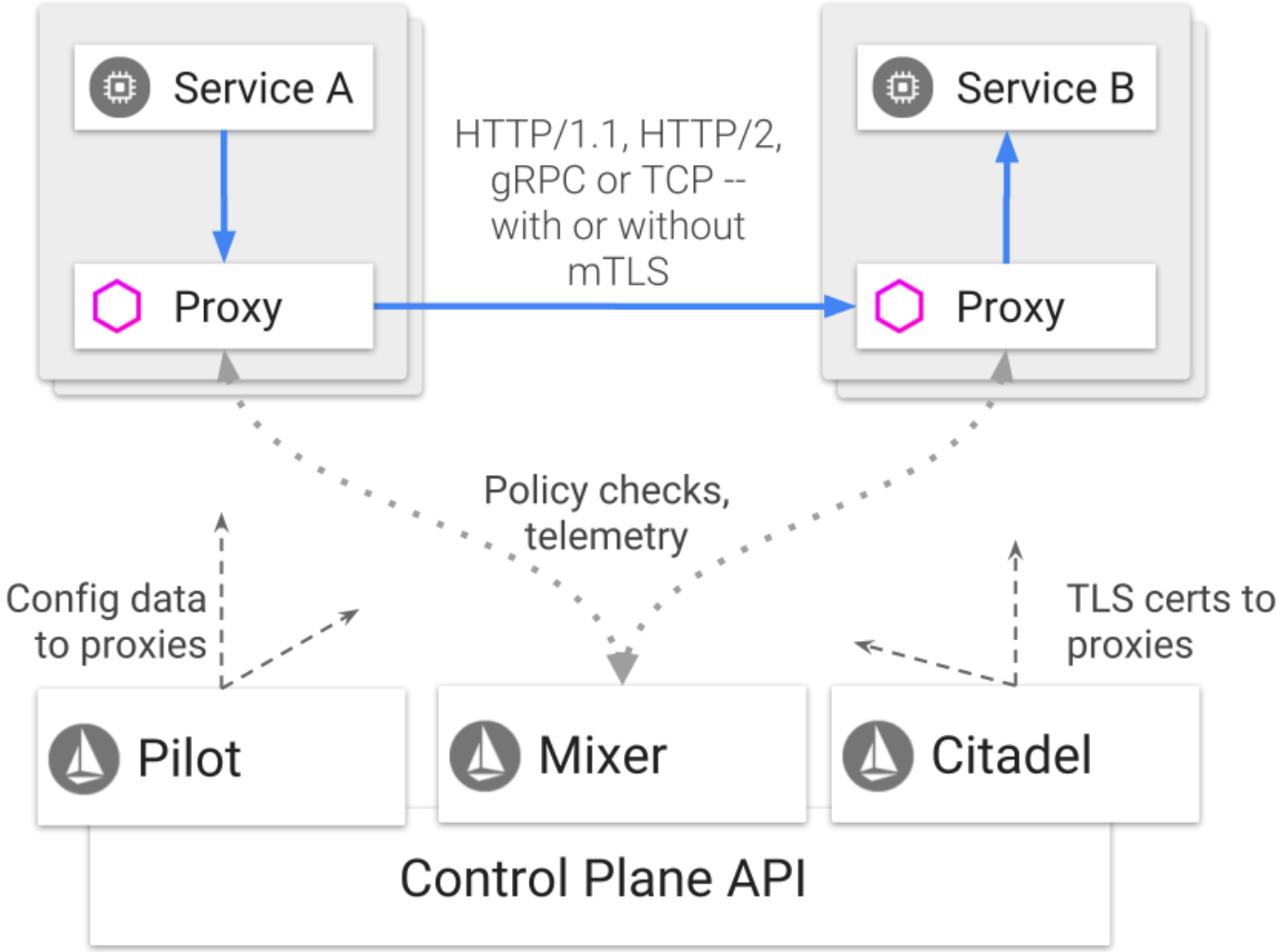


Image from https://istio.io/docs/concepts/what-is-istio/arch.svg

Istio Architecture

Data Plane

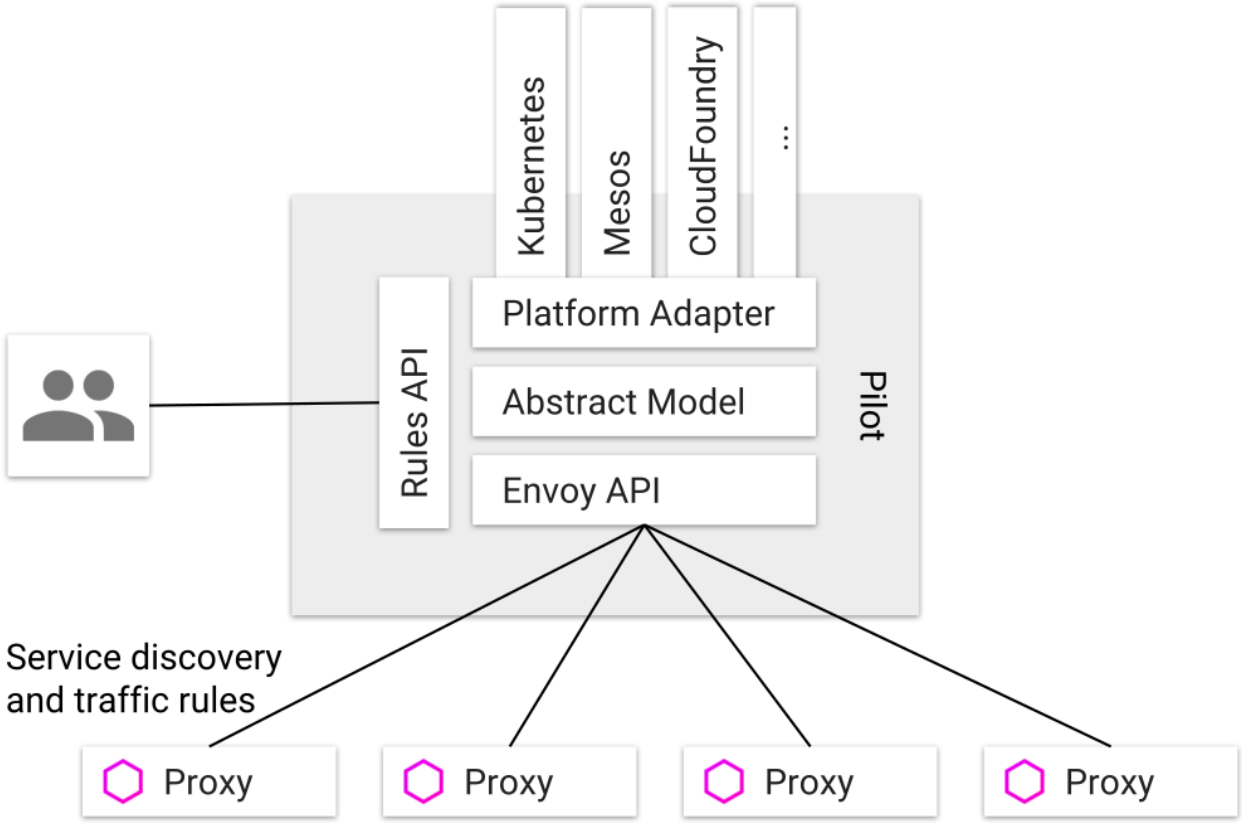
- A set of sidecars (proxies)
- Sidecars functionalities
  - Dynamic service discovery
  - Client side load balancing
  - TLS termination
  - HTTP/2 and gRPC proxies
  - Circuit breakers
  - Health checks
  - Metrics
  - Weighted traffic split

Control Plane

- Manages and configures the sidecars
- Enforces policies
- Has the following components
  - Mixer enforces access control and usage policies across the service mesh
  - Pilot provides service discovery for the sidecars
  - Citadel provides service to service and end-user authentication

Pilot and Proxy

- Pilot is part of Istio's control plane
- Use to configure the proxy in each of the Pod
  - Timeout information
  - How to handle failures
  - Routing policies
- Proxy enforces the configurations and policies from Pilot



Installing Istio - with Helm

- Download and unzip/untar Istio
  - https://github.com/istio/istio/releases
- Create custom resources
  - In `install/kubernetes/helm` directory

```
helm install istio-init \
   --name istio-init \
   --namespace istio-system
```

- Create Istio resources
  - See https://istio.io/docs/setup/kubernetes/additional-setup/config-profiles for list of installed components

```
helm install istio \
   --values istio/values-istio-demo.yaml
```

Istio Pods and Services

- Istio installs its artefacts into `istio-system` namespace
- Pods include (not exhaustive)
  - `istio-citadel`
  - `istio-ingressgateway`
  - `istio-egressgateway`
  - `istio-tracing`
- Services include (not exhaustive)
  - `istio-egressgateway`
  - `istio-ingressgateway` - traffic enters the cluster via this
  - `jaeger-query`

Deploying to Istio

- Create a deployment and service object as per normal
- Except inject sidecar into Pods

```
istioctl kube-inject -f deployment.yml > \
    deployment-with-sidecar.yml
kubectl apply -f deployment-with-sidecar.yml
```

or

```
kubectl apply -f \
    <(istioctl kube-inject -f deployment.yml)
```

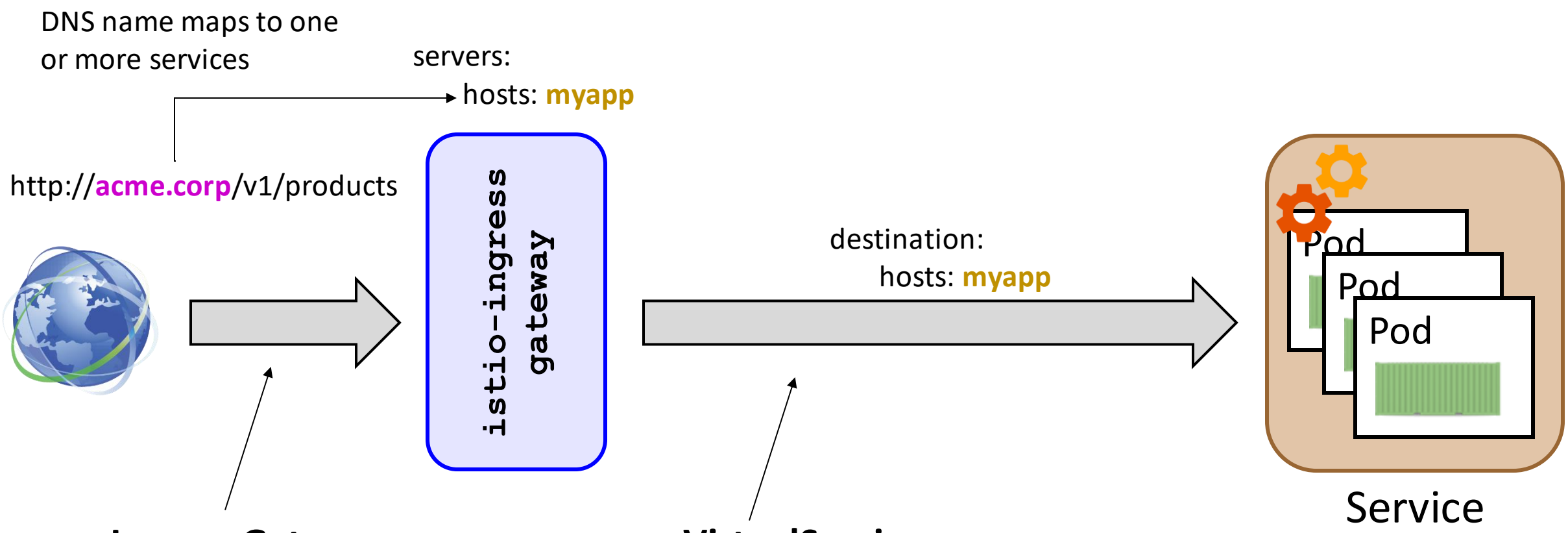
- Deploy service

```
kubectl apply -f service.yml
```

Istio Routing

- Uses `Gateway` and `VirtualService` inplace of `Ingress`
  - `Gateway` L4 to L6 load balancer
    - Configures sidecar for ingress
  - `VirtualService` is a L7 router
    - Bound to a gateway, route traffic to services
- Decoupling of traffic with `Gateway` and `VirtualService` provides different traffic management features configured at the `VirtualService`
  - Eg. A/B testing, canary releases, gradual rollouts, etc.
- Routing can be applied to both ingress and egress traffic

Gateway and VirtualService - Ingress



Ingress Gateway
allow traffic, eg. HTTP(S),
into the cluster

VirtualService
route traffic to services

Image idea from https://rinormaloku.com/istio-practice-routing-virtualservices/

Define a Gateway

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: myapp-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - myapp.ns.svc.cluster.local
```

Use Istio ingress to handle this

Accept traffic to this gateway from one or more ports

Traffic to this host (service) or * for any service. Must be FQDN

Define a Virtual Service

```yaml
apiVersion: networking.stio.io/v1alpha3
kind: VirtualService
metadata:
   name: myapp
spec:
   hosts:
   - "*"
   gateways:
   - myapp-gateway
   http:
   - match:
     - uri:
         exact: /v1
     route:
     - destination:
         hosts: myapp.ns.svc.cluster.local
     rewrite:
         uri: /
```

Receive traffic from the list of gateways

All request with literal match

Route to this service. Must be FQDN

Rewrite the resource from /v1 to /

Istio Routing

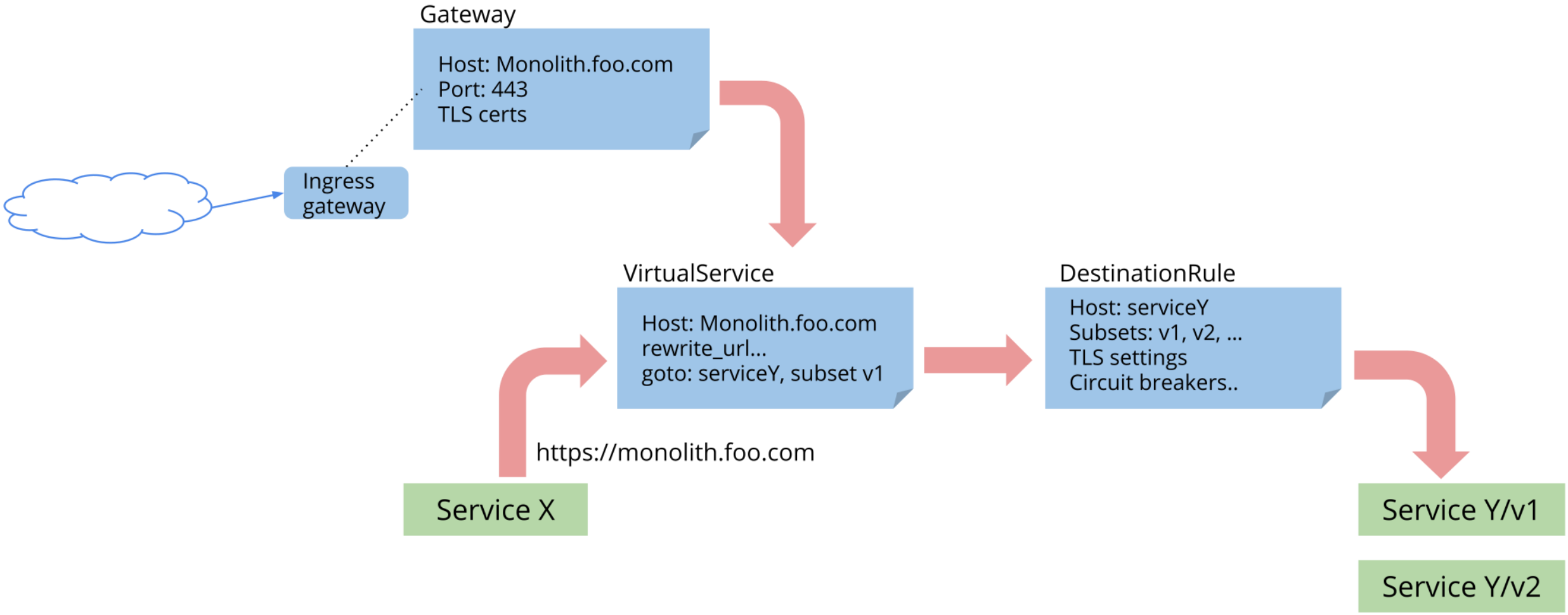


Image from https://archive.istio.io/v0.8/blog/2018/img/virtualservices-destrules.svg

Default Route

```yaml
apiVersion: networking.stio.io/v1alpha3
kind: VirtualService
metadata:
   name: myapp
spec:
   hosts:
   - "*"
   gateways:
   - myapp-gateway
   http:
   ...
```

```yaml
http:
- match:
  - uri:
      exact: /v1
  route:
  - destination:
      hosts: myapp
  rewrite:
      uri: /
- route:
  - destination:
      hosts: myapp_v2
      port:
         number: 8080
```

One or more matched route

Default route

Request with `/v1` are routed to `myapp`, all other request defaults to `myapp_v2`

Weight Based Routing

```yaml
apiVersion: networking.stio.io/v1alpha3
kind: VirtualService
spec:
  gateways:
  - myapp-gateway
  http:
  - match:
    - uri:
        exact: /v1
    route:
    - destination:
        hosts: myapp
        weight: 75
    - destination:
        hosts: myapp-beta
        weight: 25
```

If there is only 1 destination, the weight property can be omitted

Traffic is split between the 2 services. Assume `myapp-beta` is a new version that we are testing

Routing Based on HTTP Header

```
apiVersion: networking.stio.io/v1alpha3
kind: VirtualService
spec:
  hosts:
  - myapp
  gateways:
  - myapp-gateway
  http:
  - match:
    - headers:
      authorization: ^.*Bearer .+$
    route:
    - destination:
      hosts: myapp
```

Route bases on the value of HTTP headers

Routing Based on Pod Labels

```yaml
apiVersion: networking.stio.io/v1alpha3
kind: VirtualService
spec:
  hosts:
  - myapp
  gateways:
  - myapp-gateway
  http:
  - match:
    - sourceLabels:
        app: myotherapp
        version: v2
    route:
    - destination:
      hosts: myapp
```

Only allow request from Pods with the following labels

Configure CORS

```yaml
apiVersion: networking.stio.io/v1alpha3
kind: VirtualService
spec:
   hosts:
   - myapp
   gateways:
   - myapp-gateway
   http:
   - match:
     - headers:
         authorization: ^.*Bearer .+$
     route:
     - destination:
         hosts: myapp
     corsPolicy:
         allowOrigin:
         - *
         allowMethods:
         - GET
         - POST
```

Returns the following CORS headers in the response

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET, POST
```

Load Balancing and Health Checks

- Istio Pilot gets service information from Kubernetes
- Proxy (sidecar) discovers these services from Pilot
- Proxy keeps a list of the pods in the services internally
- Proxy load balances across the Pods in a service
- Proxy also does health checks on the Pods
  - Removes unhealthy Pods from its lists

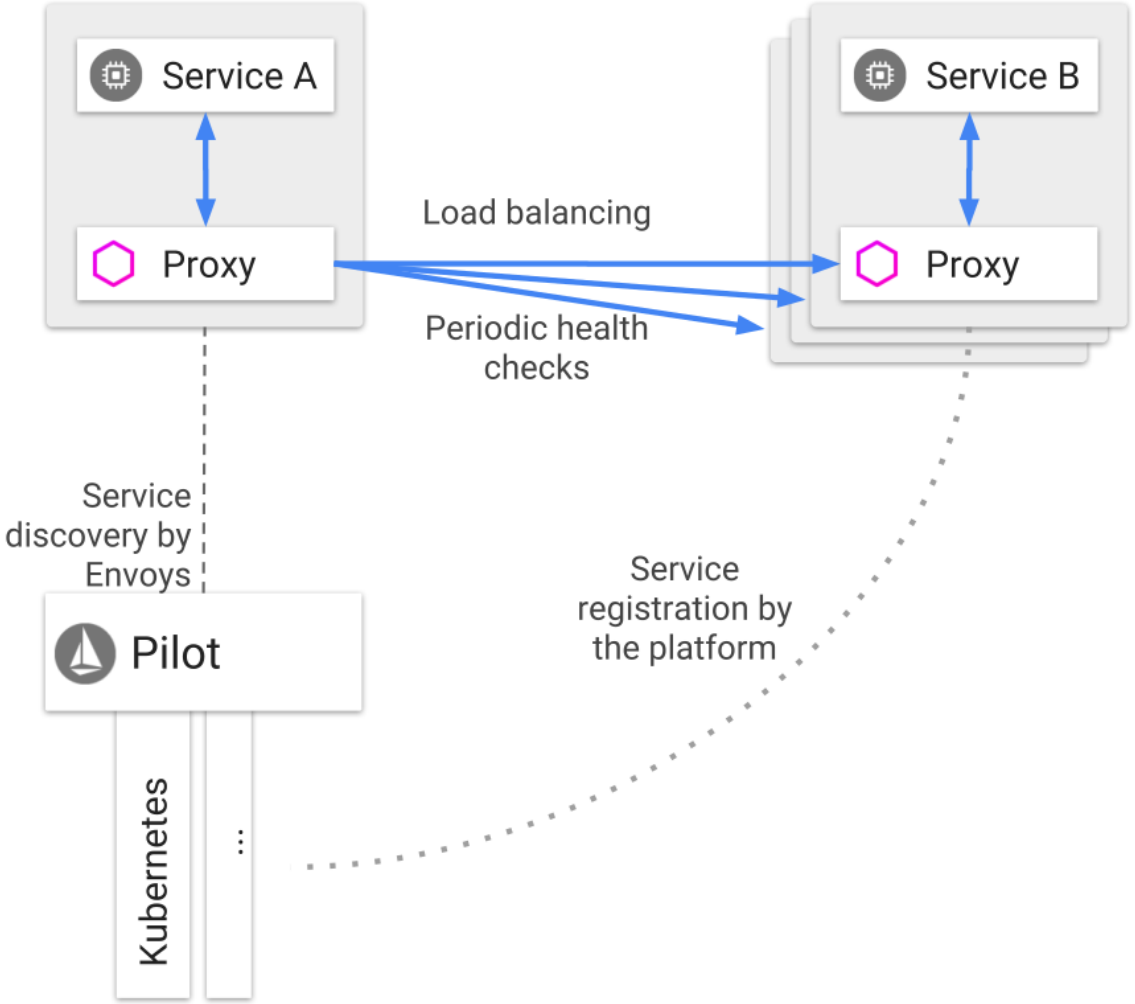


Image from https://istio.io/docs/concepts/traffic-management/#discovery-and-load-balancing

Define a DestinationRule

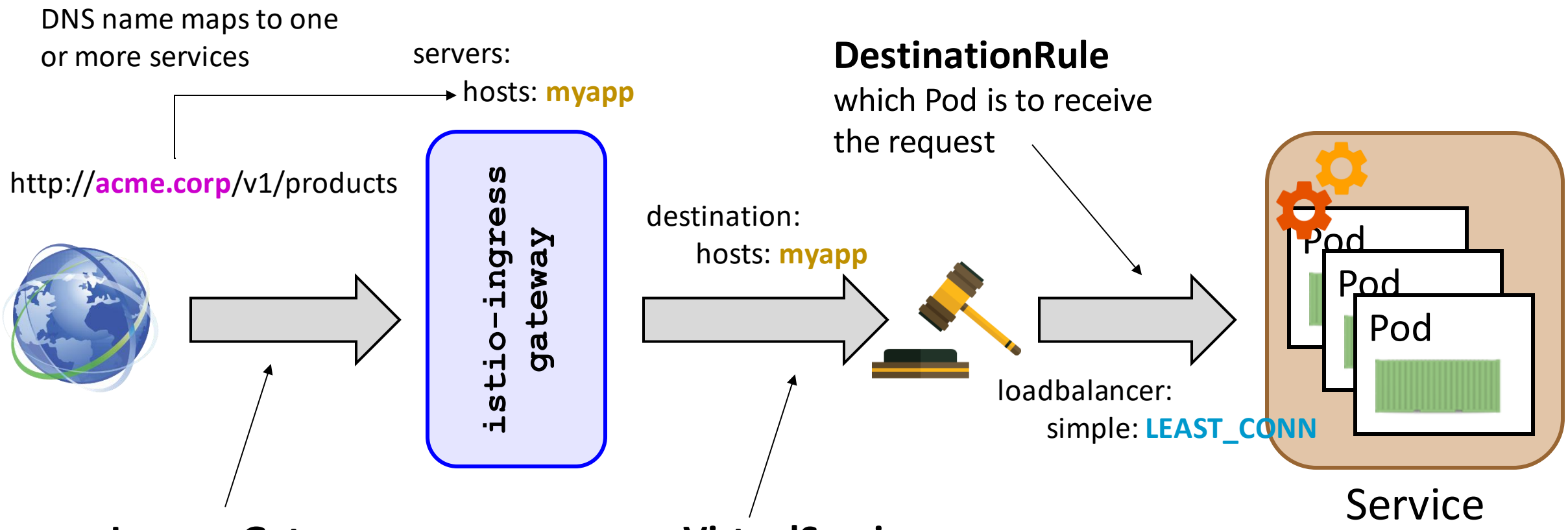
- Applies to traffic bound for a particular service
  - After routing has occurred

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: myapp
spec:
  host: myapp
  trafficPolicy:
    loadBalancer:
      simple: LEAST_CONN
```

Load balancer policy

- ROUND_ROBIN
- LEAST_CONN - least busy host
- RANDOM - randomly selects a healthy host
- PASSTHROUTH - pass directly to the requested host

Gateway and VirtualService - Ingress



Ingress Gateway
allow traffic, eg. HTTP(S),
into the cluster

VirtualService
route traffic to services

Image idea from https://rinormaloku.com/istio-practice-routing-virtualservices/

VirtualService with DestinationRule - 2

```yaml
apiVersion: networking.stio.io/v1alpha3
kind: VirtualService
metadata:
  name: myapp
spec:
  hosts:
  - myapp
  gateways:
  - myapp-gateway
  http:
  - match:
    - uri:
        exact: /v1
    route:
    - destination:
        hosts: myapp
        subset: v1rule
```

Destination rule name

VirtualService with DestinationRule - 2

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: myapp
spec:
  host: *
  subsets:
  - name: v1rule
    labels:
      app: myapp
      version: v1
    trafficPolicy:
      loadBalancer:
        simple: LEAST_CONN
```

Destination rule name

Apply to Pods with these labels

One or more named destination rule

VirtualService Route Matching Example

Ingress Resource	Rewrite To
/v1	/
/v1/	/
/v1/products	/products
/v1/products/tv	/products/tv

```
http:
- match:
  - uri:
     exact: /v1
  rewrite:
     uri: /
   route:
   - destination:
      host: myapp
- match:
  - uri:
     prefix: /v1
  rewrite:
     uri: " "
   route:
   - destination:
      host: myapp
```

Order of the match is important

Egress

`fetch()` made from with a container in a Pod

Sidecar will attempt to resolve this name.
Not found in Pilot's internal registry

```
fetch('https://hacker-news.firebaseio.com/v0/item/1234.json')
  .then(response => {
```

- All traffic from container(s) within a Pod are routed through the sidecar
- The sidecar the route the traffic to other services via service discovery
- Pod will not be able to make external calls viz. to resources outside of the service mesh

Handling Egress Traffic

- Two ways to allow traffic out of Istio
  - Creating an service entry to allow individual Pods to access to destination
  - Going through an egress gateway

Service Entry

- Manually add external resources into the Istio's service registry
  - Maintained by Pilot
- Allow sidecar to discover the service
- Describes the service characteristics
  - Eg. host, ports, TLS, etc

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: myapp-egress
spec:
  hosts:
  - hacker-news.firebaseio.com
  - api.openweathermap.org
  ports:
  - number: 80
    protocol: HTTP
    name: http
  - number: 443
    protocol: HTTPS
    name: https
  location: MESH_EXTERNAL
  resolution: DNS
```

List of external host resolvable by DNS

One or more ports associated with the external service

The service is outside of the mesh

Mode in which the service is resolved

Distributed Tracing

- To follow the request as it travers the application mesh
  - Provides observability as a request is being fulfilled by various micro services
- Captures the entire call chain
  - From when the request enters the service mesh and all the services that the request interacts with
- For monitoring and profiling micro services
- OpenTracing is an open standard for distributed tracing
  - Jaeger is an implementation of OpenTracing

OpenTracing

- Trace covers the entire request across all services that the request pass thru
- Trace consist of one or more Spans
- Span is a logical chunk of work
  - Spans can contain other spans
- Micro services propagate specific HTTP headers to outgoing request to enable tracing
  - Eg. `X-B3-TraceId`, `X-B3-SpanId`, `X-B3-Sampled`, etc.

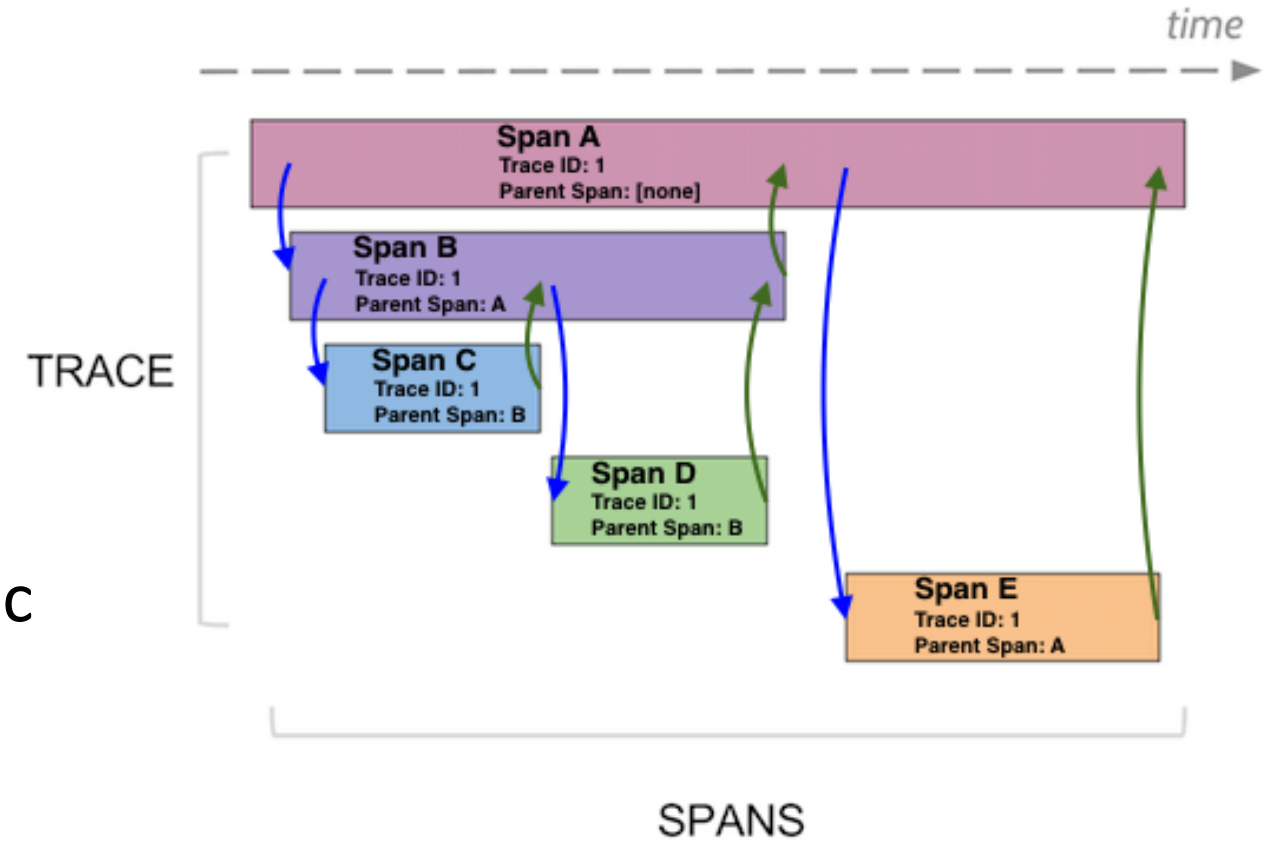


Image from https://medium.com/nikeengineering/hit-the-ground-running-with-distributed-tracing-core-concepts-ff5ad47c7058

Tracing Console

- Find the distributed tracing pod, starts with `istio-tracing-XXXX`

```
kubectl get pod -n istio-system | grep tracing
```

- Port forward the console

```
kubectl port-forward istio-tracing-XXX 16686:16686
```

Example of a Trace

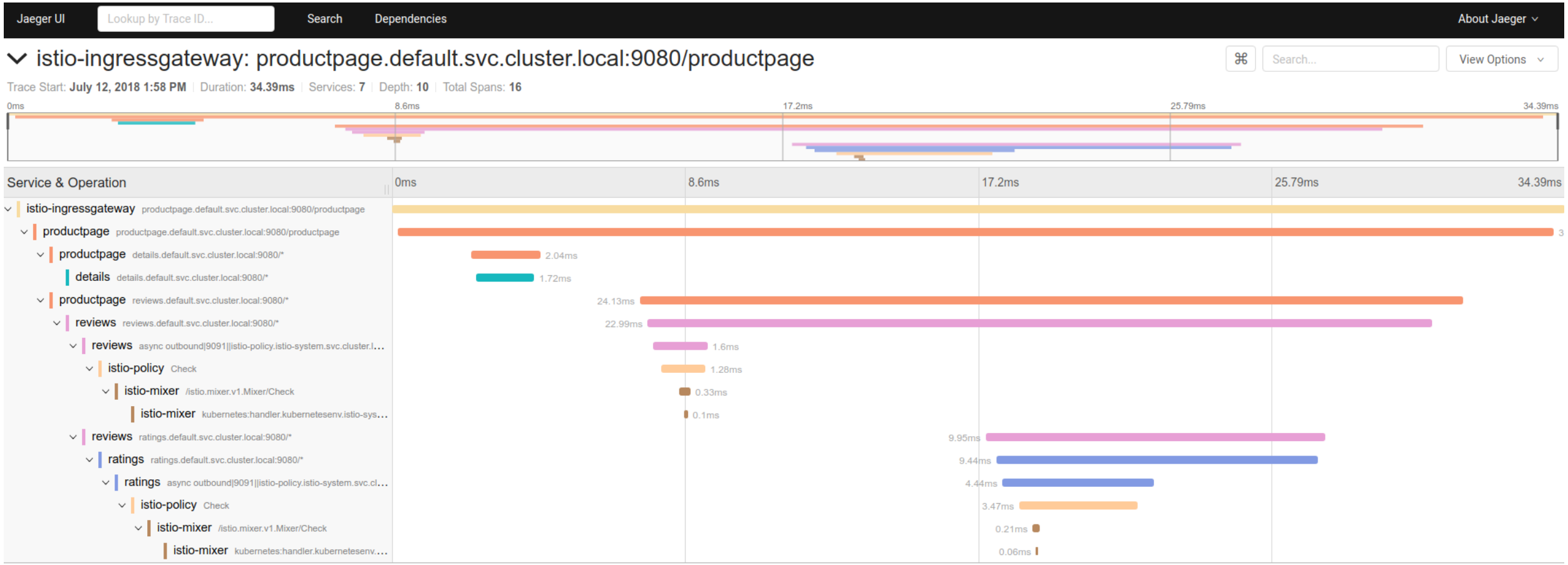


Image from https://istio.io/docs/tasks/telemetry/distributed-tracing/istio-tracing-details.png

Appendix

Installing Istio - with kubectl

- Download and unzip/untar Istio
  - https://github.com/istio/istio/releases
- Create custom resources

```
kubectl apply -f
<istio_home>/install/kubernetes/helm/istio/templates/crds.
yaml
```

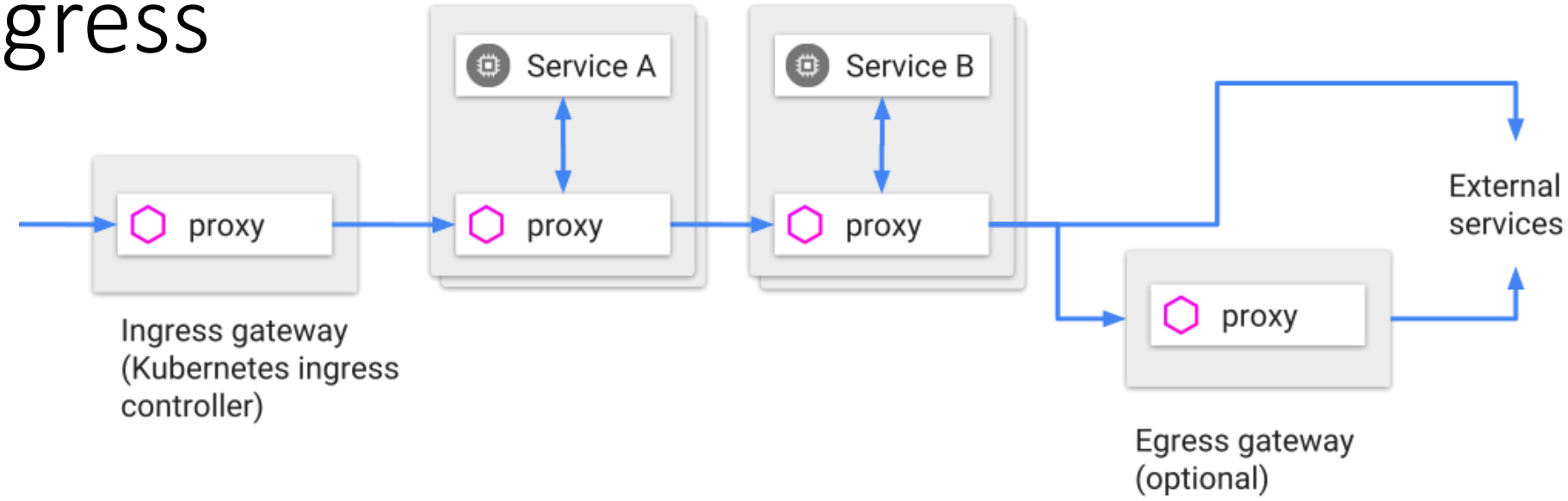
- Create Istio resources
  - Without authentication

```
kubectl apply -f <istio_home>/install/kubernetes/istio-
demo.yaml
```

  - With authentication

```
kubectl apply -f <istio_home>/install/kubernetes/istio-
demo-auth.yaml
```

Egress



- Egress connections for sidecar is forwarded to an egress gateway
- Egress gateway allows for greater control eg.
  - Logging
  - Converting all non TLS traffic to TLS
  - Cache results
  - Circuit breaking

Image from https://istio.io/docs/concepts/traffic-management

Routing to External Service - Gateway

```yaml
apiVersion: .../v1alpha3
kind: Gateway
metadata:
   name: myapp-egress
spec:
   selector:
      istio: egressgateway
   servers:
   - hosts:
     - api.openweathermap.org
     - hacker-news.firebaseio.com
     port:
        number: 443
        protocol: HTTPS
        name: https
     tls:
        mode: SIMPLE
        serverCertificate: /etc/certs/cert-chain.pem
        privateKey: /etc/certs/key.pem
        caCertificates: /etc/certs/root-cert.pem
```

SSL connections for hosts

Default list of certificate in the `egressgateway` Pod. To examine the certs, find the `egressgateway` Pod name in `istio-system` namespace and exec into it

Routing to External Service - VirtualService

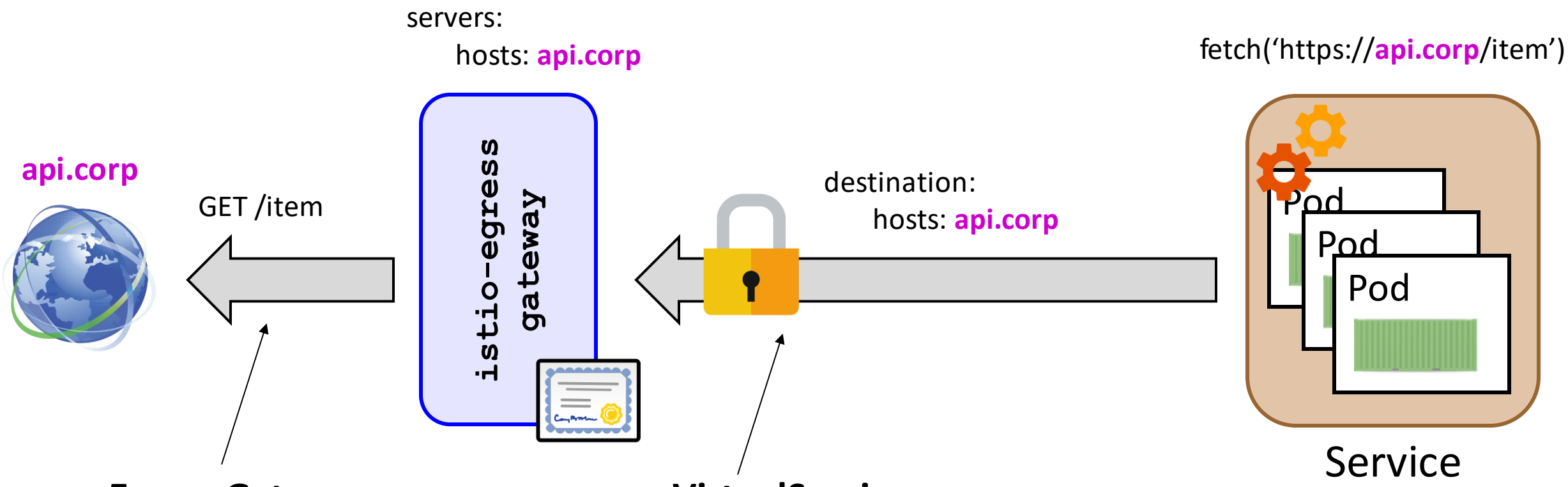
```yaml
apiVersion: .../v1alpha3
kind: VirtualService
metadata:
   name: myapp-egress
spec:
   hosts:
   - api.openweathermap.org
   gateways:
   - mesh
   - myapp-egress
   http:
   - match:
     - gateways:
       - mesh
       port: 80
     route:
     - destination:
          host: istio-egress-istio-system.svc.cluster.local
```

List of gateways to use

Traffic going to api.openweathermap.org is required to use myapp-egress gateway

```yaml
   - match:
     - gateways:
       - myapp-egress
       port: 80
     route:
     - destination:
          host: api.openweather.map
```

Gateway and VirtualService - Egress

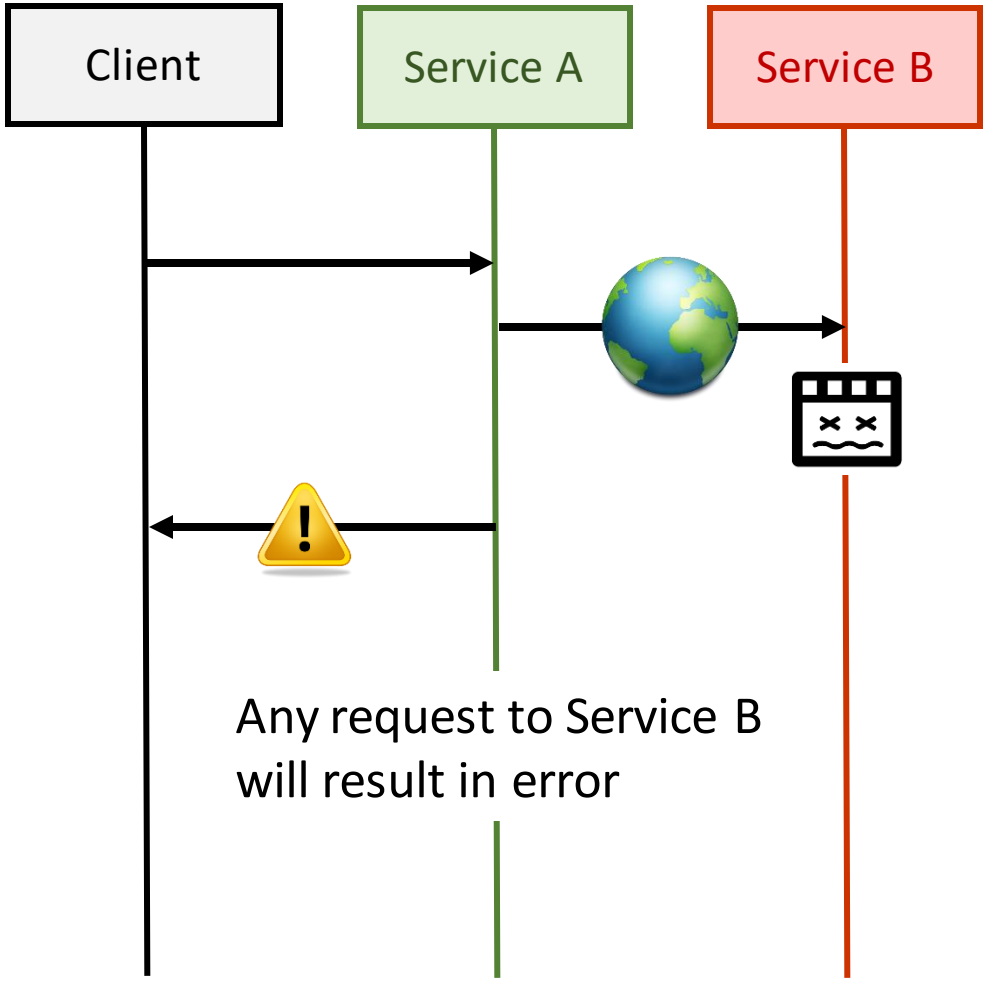


Egress Gateway
allow traffic, eg. HTTP(S), into the cluster

VirtualService
route traffic to services

Image idea from https://rinormaloku.com/istio-practice-routing-virtualservices/

Circuit Breaker



- Lots of things can go wrong when connecting to a remote connection
  - Not under your control
  - Eg. network issue, service down, etc.
- If Service B is down, then there is no point in making any further calls to it until it becomes available
  - Continually make calls from Service A may lead to cascading failures

Circuit Breaker

- Remote calls are proxied by a circuit breaker object
- Circuit breaker monitors the availability of remote services
- Once a failure threshold is breached, the circuit is open
- Any subsequent call to the service via the circuit breaker will result in an immediate error

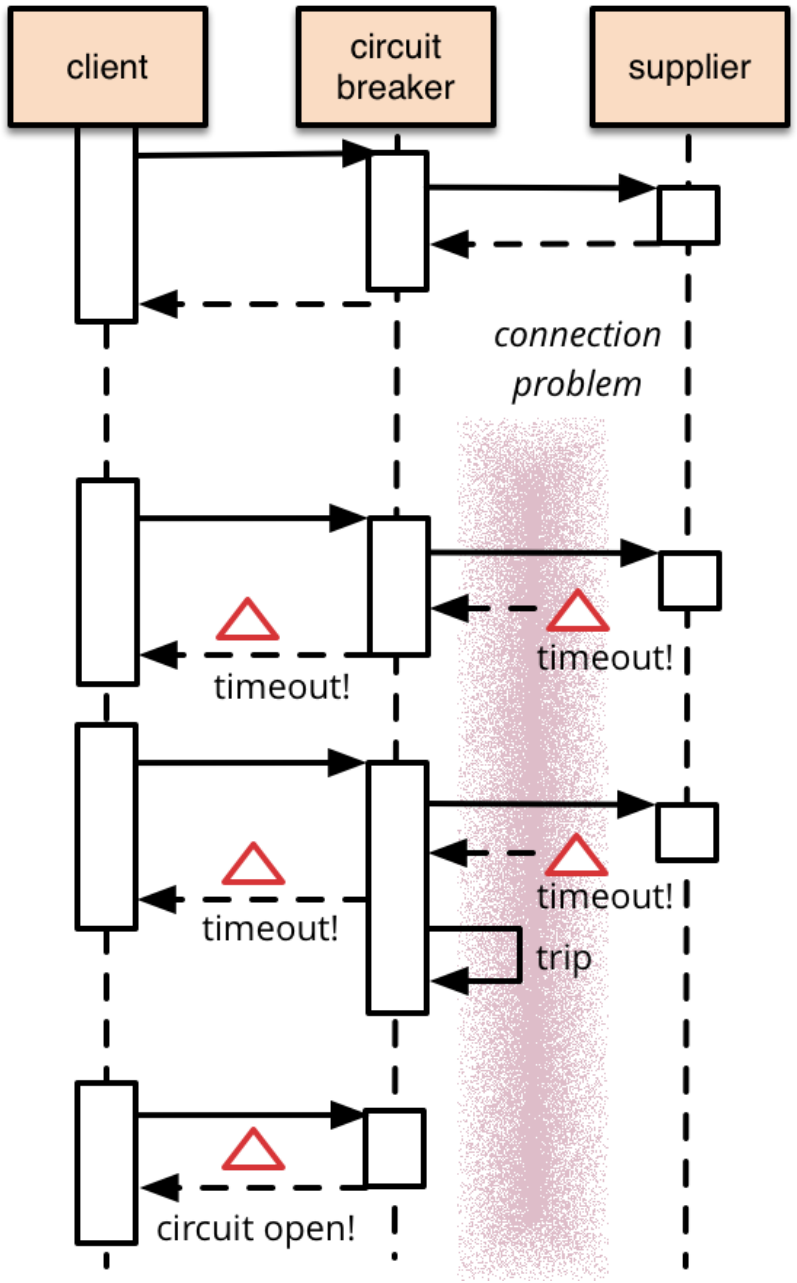


Image from https://martinfowler.com/bliki/CircuitBreaker.html

Circuit Breaker Algorithm

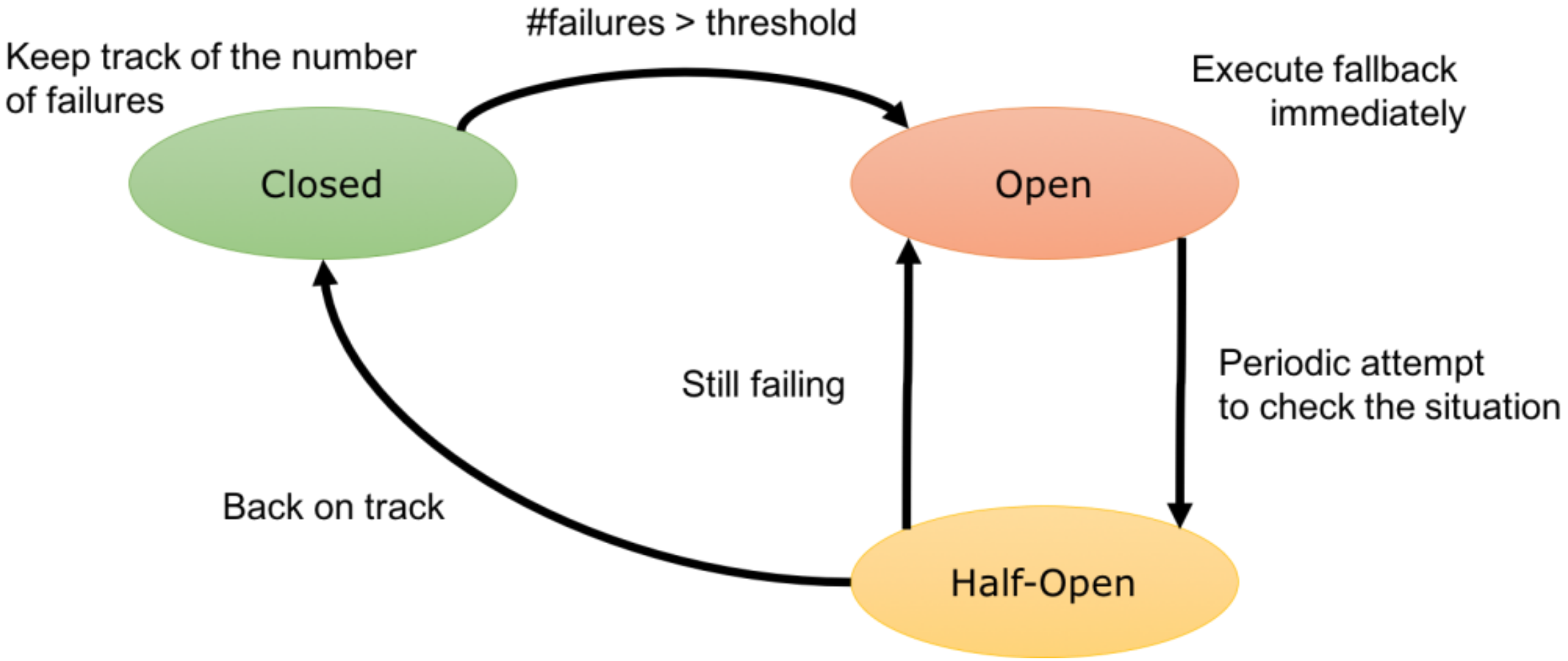


Image from https://developers.redhat.com/blog/wp-content/uploads/2018/03/circuit-breaker-1024x430.png

Defining a Circuit Breaker

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: myapp-egress
spec:
  host: api.openweathermap.org
  trafficPolicy:
    connectionPool:
      maxConnection: 10
    outlierDetection:
      interval: 5m
      consecutiveErrors: 5
      baseEjectionTime: 15m
```

Destination

Optional. Number of egress connection in the gateway

Number of consecutive 5XX errors in a period of 5 minutes, the destination will be ejected for 15 minutes