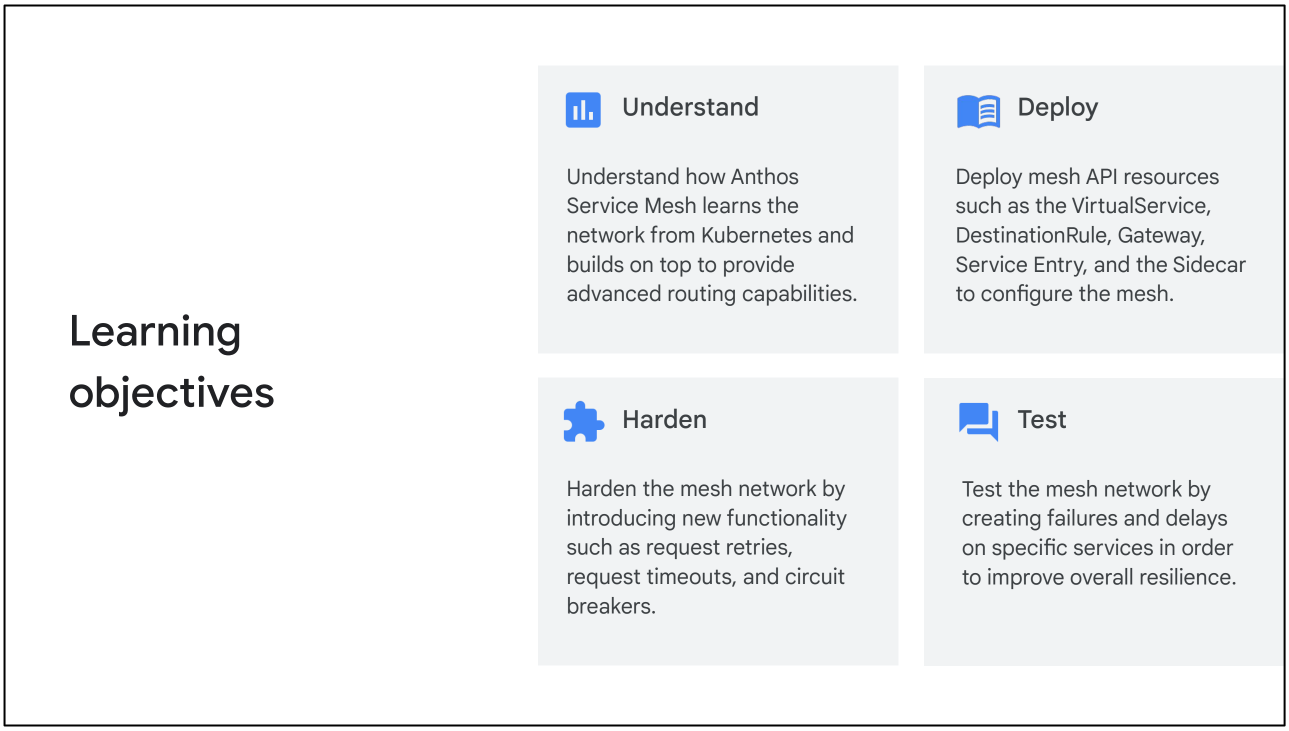
Managing Traffic Flow with Anthos Service Mesh

Welcome to the introduction of Anthos Service Mesh.



Learning objectives

Understand

Understand how Anthos Service Mesh learns the network from Kubernetes and builds on top to provide advanced routing capabilities.

Deploy

Deploy mesh API resources such as the VirtualService, DestinationRule, Gateway, Service Entry, and the Sidecar to configure the mesh.

Harden

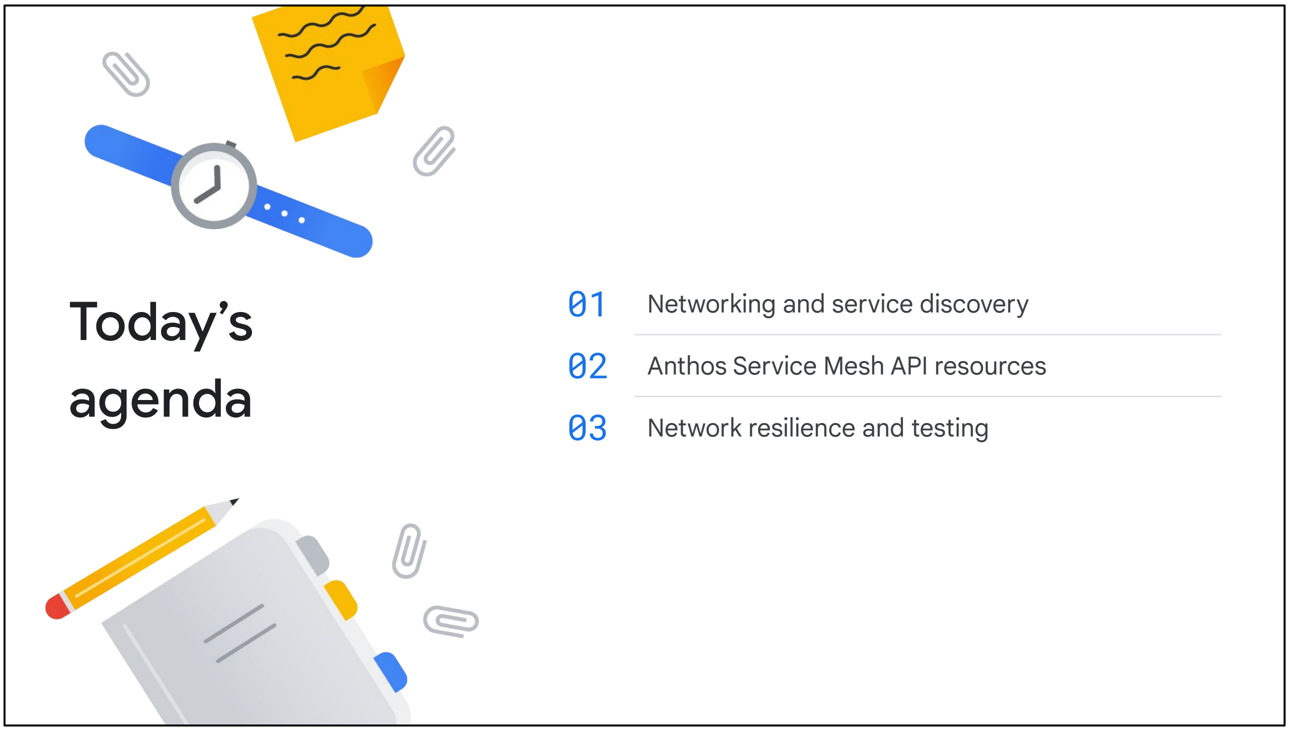
Harden the mesh network by introducing new functionality such as request retries, request timeouts, and circuit breakers.

Test

Test the mesh network by creating failures and delays on specific services in order to improve overall resilience.

In this module, you:

- Understand how Anthos Service Mesh learns the network from Kubernetes and builds on top to provide advanced routing capabilities.
- Deploy mesh API resources such as the VirtualService, DestinationRule, Gateway, Service Entry, and the Sidecar to configure the mesh.
- Harden the mesh network by introducing new functionality such as request retries, request timeouts, and circuit breakers.
- Test the mesh network by creating failures and delays on specific services, so that you can improve overall resilience.



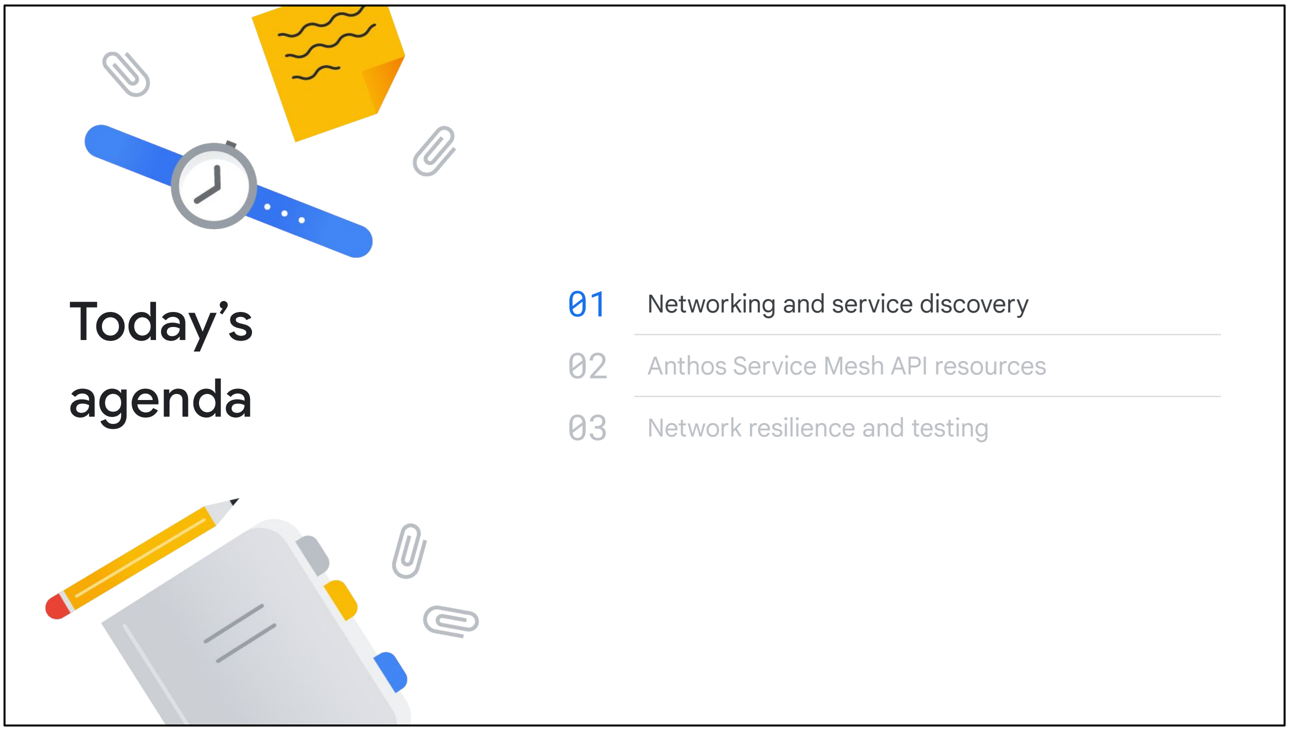
Today's agenda

01 Networking and service discovery

02 Anthos Service Mesh API resources

03 Network resilience and testing

This is our agenda for the module, shown here on the slide. Let's get started.

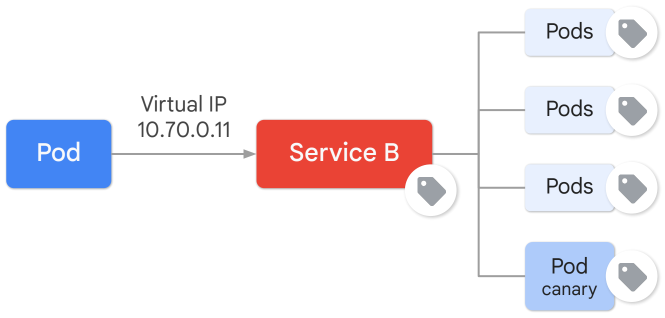


Today's agenda

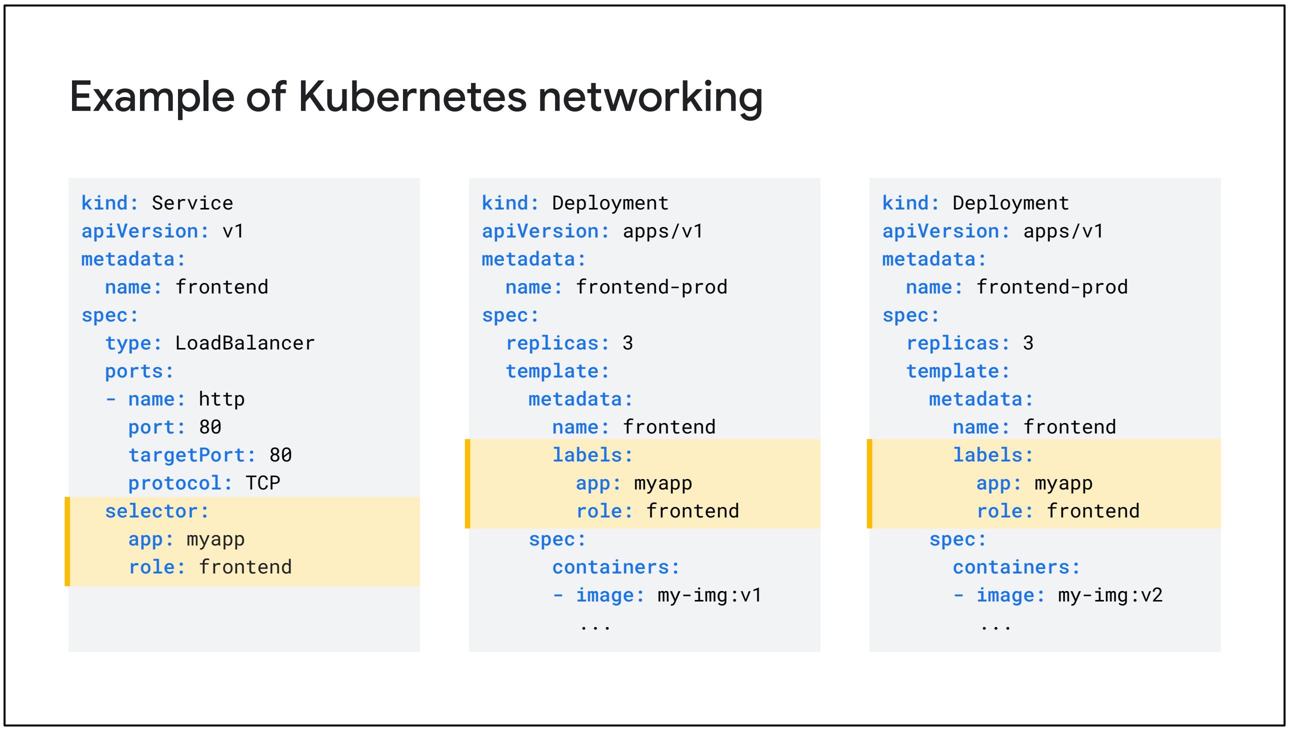
We start by discussing the need of a service mesh.

Let’s recap how Kubernetes networking works

- Kubernetes traffic is shaped by:
  - Services as network endpoint abstraction
  - Labels as compute endpoint abstraction
  - Selectors for the services to decide the labeled workloads to which traffic is routed



Let’s recap how Kubernetes networking works. If you want to route traffic to your workloads, you first need to create a Kubernetes Service. This Service is issued a static Virtual IP address. When a request hits that Service VIP, the request is forwarded to pods under management using a round-robin strategy. Services use selectors, where they describe which labels pods should have in order to receive traffic from that service. Once a service is configured, an Endpoint is created for each one of the pods under management.



Example of Kubernetes networking

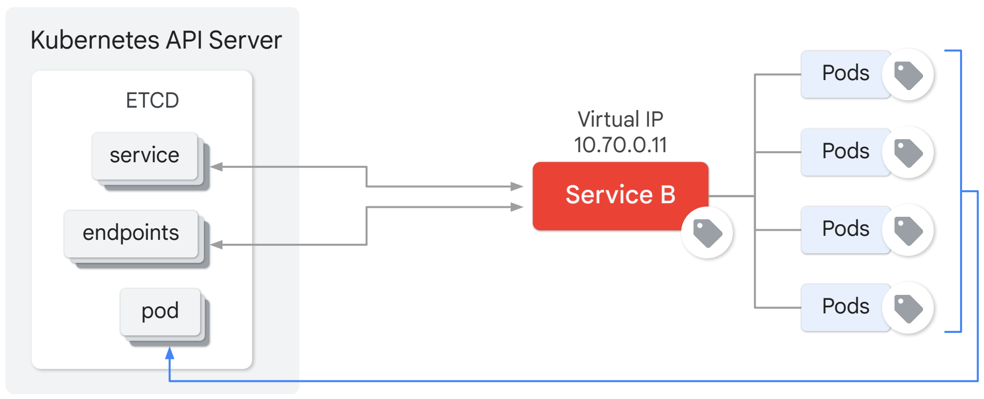
```
kind: Service
apiVersion: v1
metadata:
  name: frontend
spec:
  type: LoadBalancer
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: myapp
    role: frontend
```

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: frontend-prod
spec:
  replicas: 3
  template:
    metadata:
      name: frontend
      labels:
        app: myapp
        role: frontend
    spec:
      containers:
      - image: my-img:v1
        ...
```

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: frontend-prod
spec:
  replicas: 3
  template:
    metadata:
      name: frontend
      labels:
        app: myapp
        role: frontend
    spec:
      containers:
      - image: my-img:v2
        ...
```

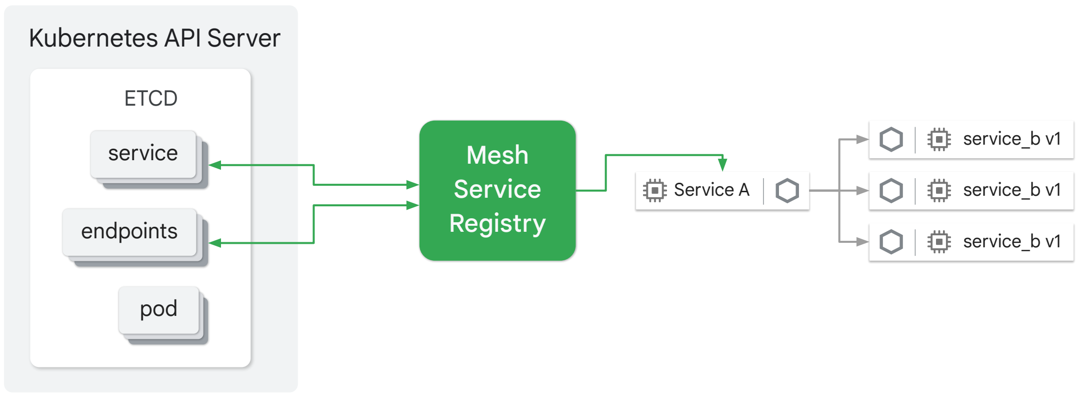
Consider the Kubernetes networking example shown here. Notice how the selector in the Service is pointing to the same labels defined in the Deployments. On the slide, these are all highlighted. That way requests are routed from the Service VIP into the pods.

All Kubernetes resources are stored in the control plane



All Kubernetes resources including deployments, pods, services, and endpoints are stored in the Kubernetes control plane's etcd store.

Anthos Service Mesh leverages Kubernetes for service discovery



In order to direct traffic within your mesh, Anthos Service Mesh (or ASM) needs to know where all your endpoints are, and which services they belong to.

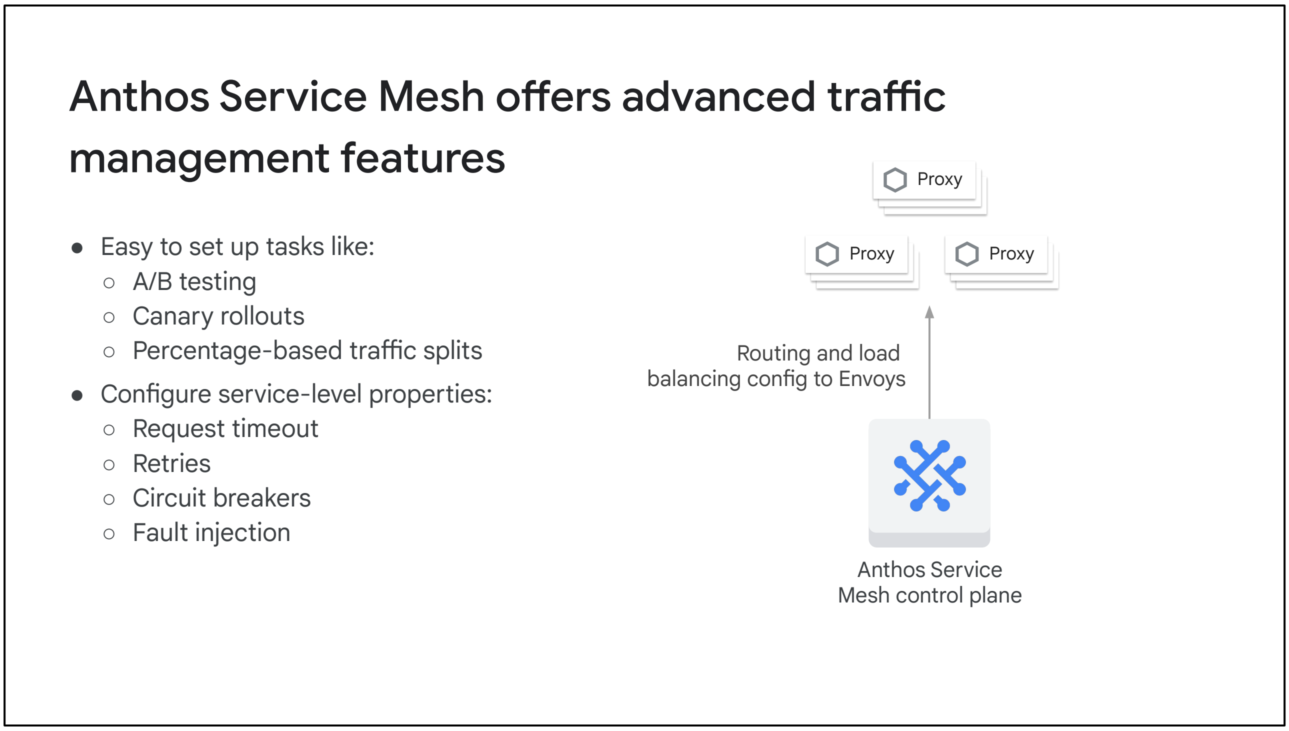
To populate its own service registry, ASM connects to the service discovery system of the underlying platform such as Kubernetes, Consul, or plain DNS.

For example, if you've installed ASM on a Kubernetes cluster, then ASM automatically detects the services and endpoints in that cluster.

The service mesh control plane adapters read the Kubernetes topology under it and register it in a service registry.

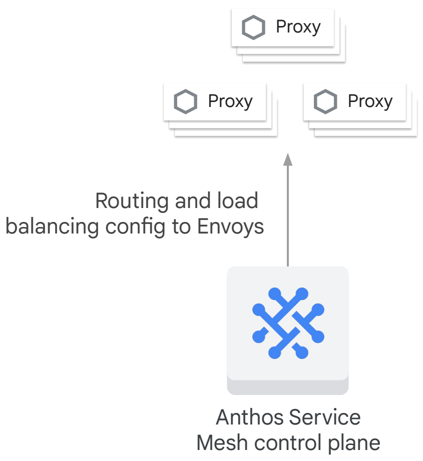
All that information is propagated to the Envoy proxies, where load balancing is done against a service's endpoints.

Once the proxy chose a specific endpoint, the packet travels directly to it, subject to policy authorization from the Envoy Filters.

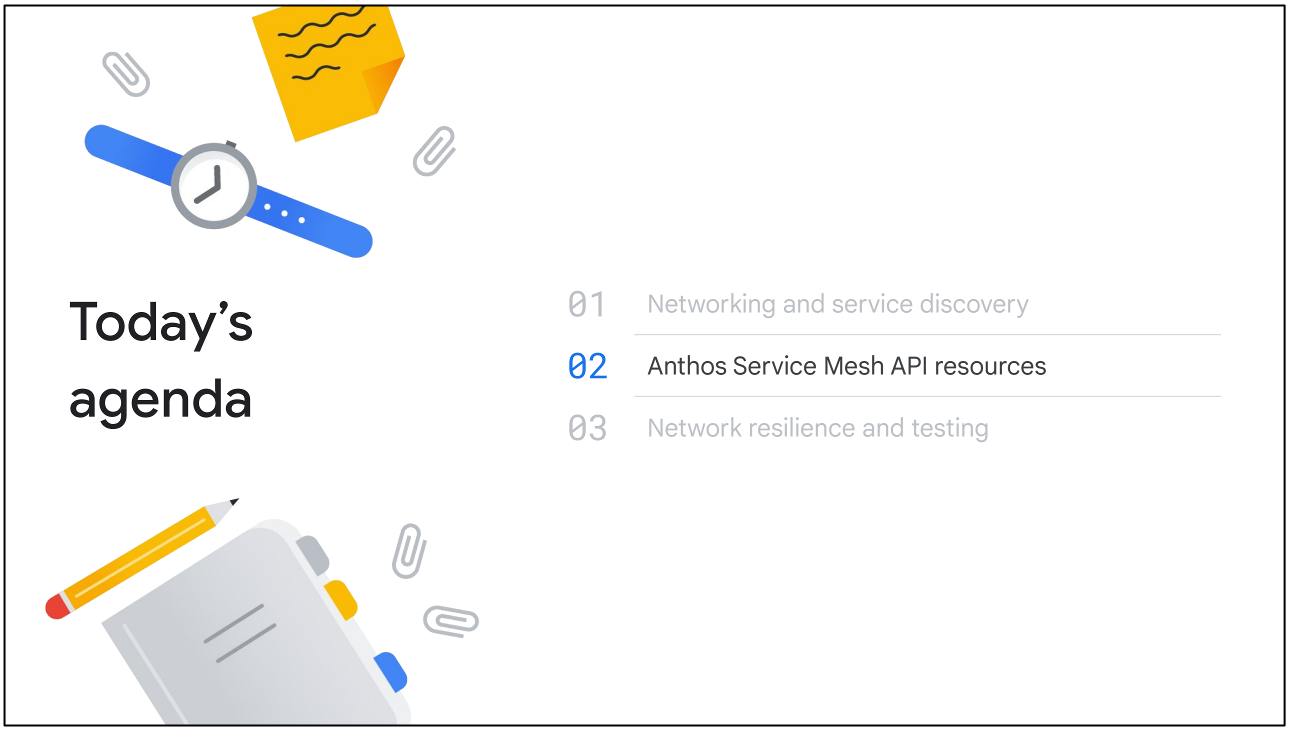


Anthos Service Mesh offers advanced traffic management features

- Easy to set up tasks like:
  - A/B testing
  - Canary rollouts
  - Percentage-based traffic splits
- Configure service-level properties:
  - Request timeout
  - Retries
  - Circuit breakers
  - Fault injection

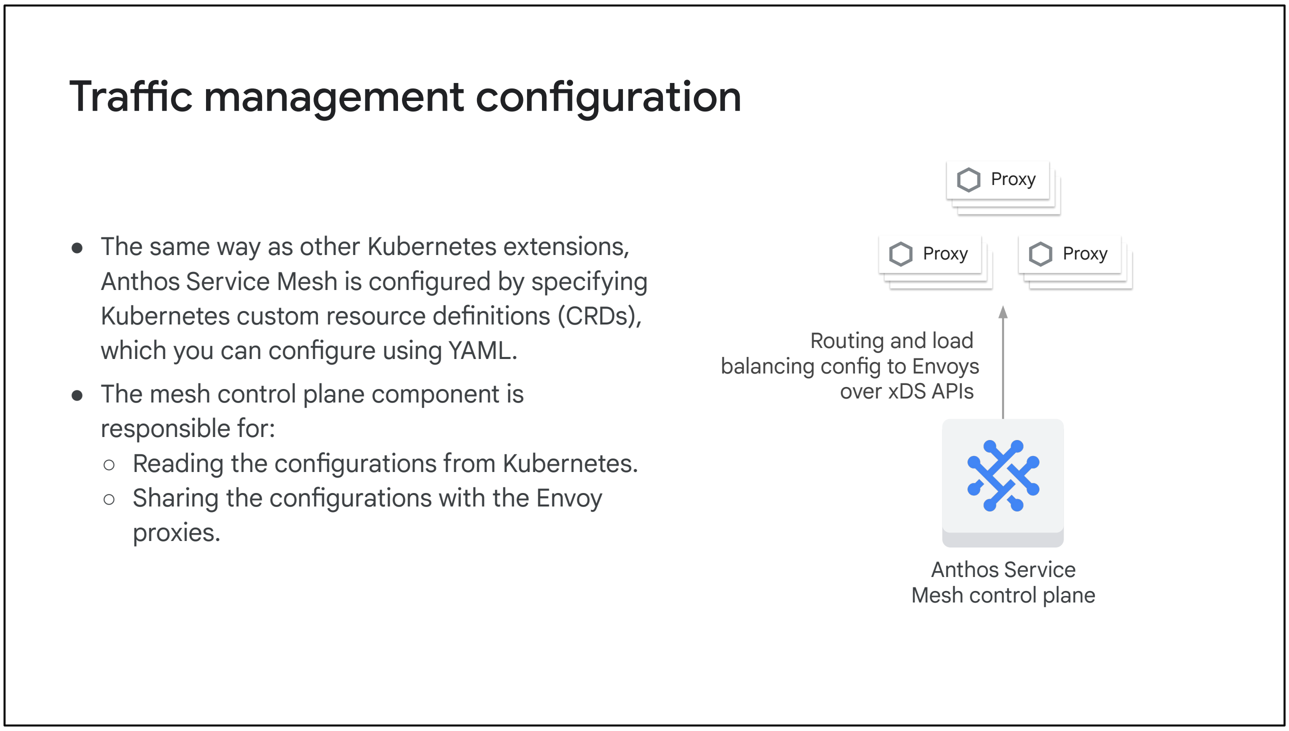

Proxy
Proxy
Proxy
Routing and load balancing config to Envoys
Anthos Service Mesh control plane

Anthos Service Mesh offers advanced traffic management features. It lets you easily control the flow of traffic and API calls between services with much greater granularity than standard Kubernetes. It also simplifies configuration of service-level properties like circuit breakers, timeouts, and retries, and makes it easy to set up important tasks like A/B testing, canary rollouts, and staged rollouts with percentage-based traffic splits. Beyond that, it provides out-of-box failure recovery features that help make your application more robust against failures of dependent services or the network.



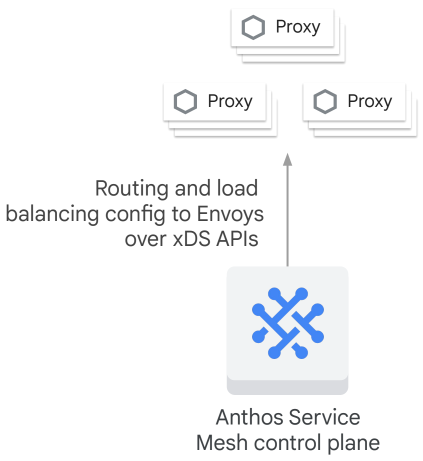
Today's agenda

Let's learn how we can use the Anthos Service Mesh API resources to configure routing.



Traffic management configuration

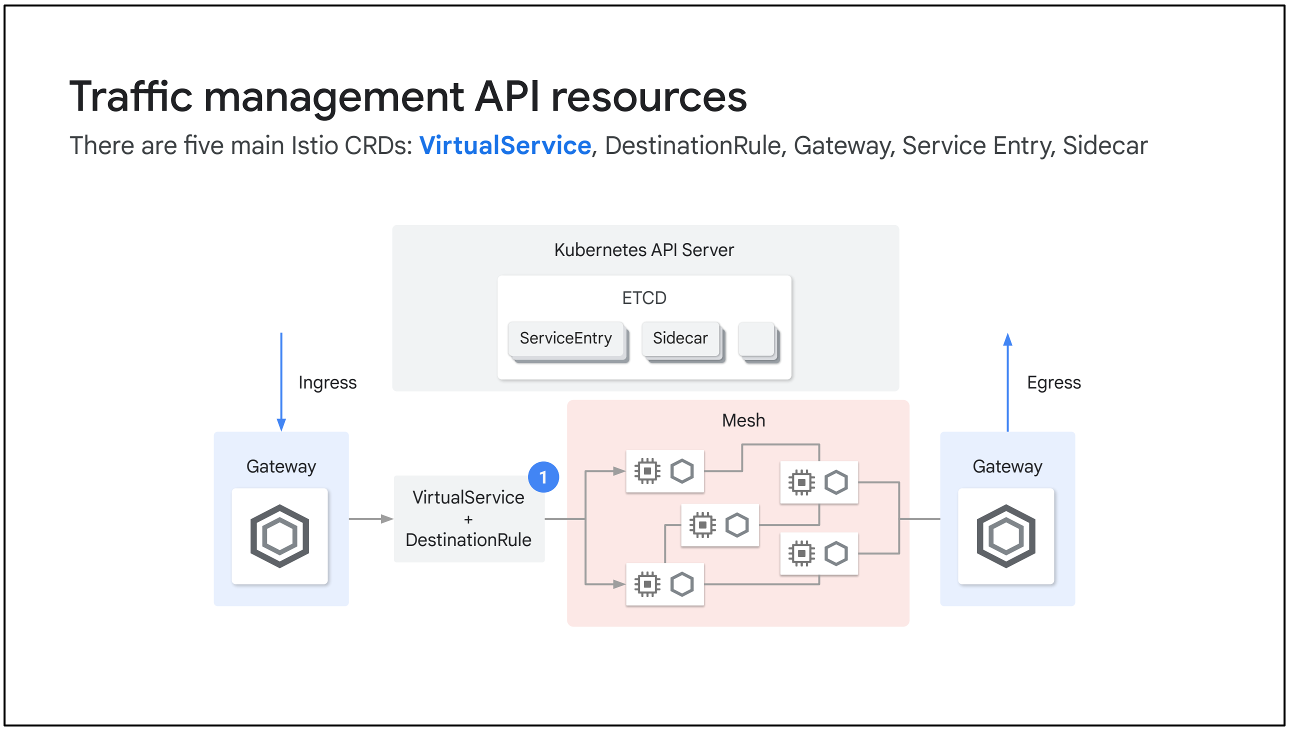
- The same way as other Kubernetes extensions, Anthos Service Mesh is configured by specifying Kubernetes custom resource definitions (CRDs), which you can configure using YAML.
- The mesh control plane component is responsible for:
  - Reading the configurations from Kubernetes.
  - Sharing the configurations with the Envoy proxies.


Proxy
Proxy
Proxy
Routing and load balancing config to Envoys over xDS APIs
Anthos Service Mesh control plane

The same way as other Kubernetes extensions, Anthos Service Mesh is configured by specifying Kubernetes custom resource definitions, or CRDs, which you can configure using YAML.

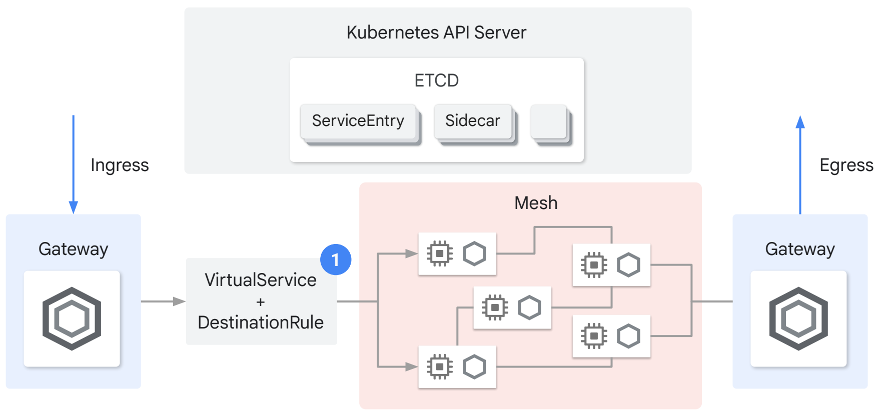
The mesh control plane component is responsible for reading those configurations from Kubernetes and sharing them with the Envoy proxies and does so via the Envoy xDS API.

Let’s take a look at the service mesh resources.



Traffic management API resources

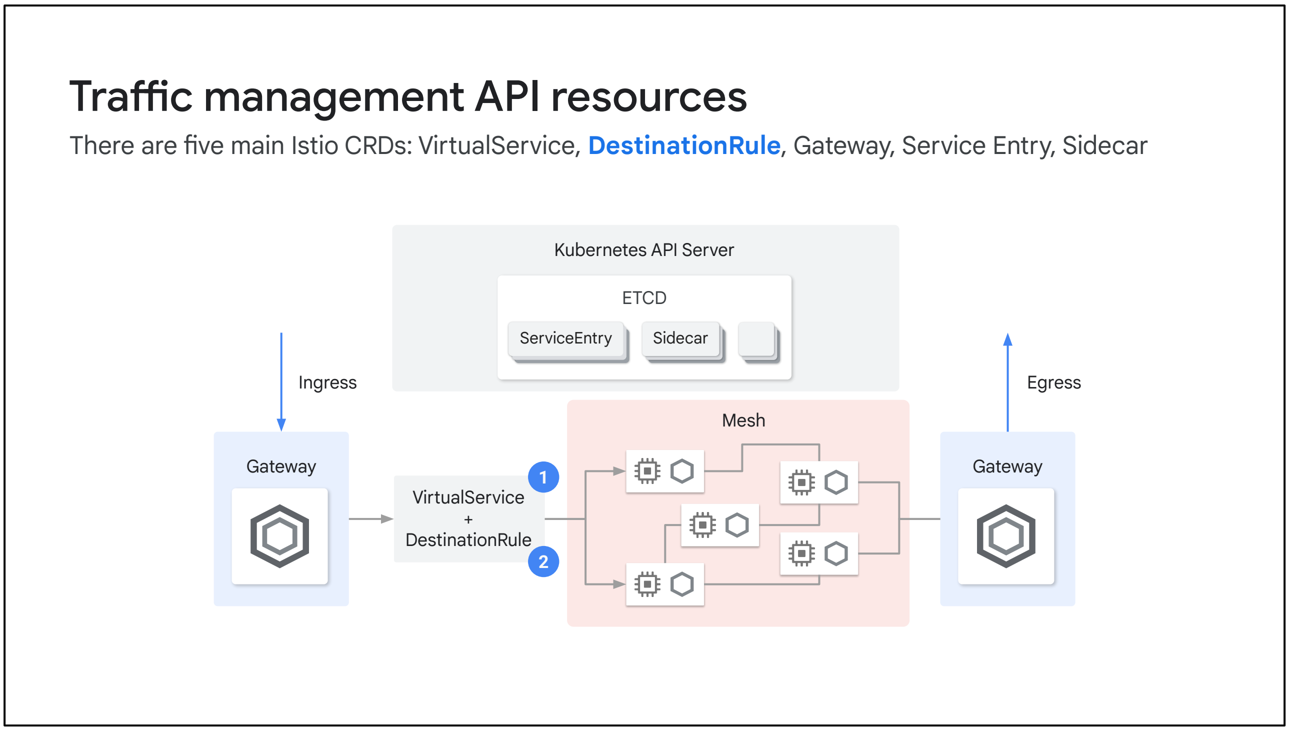
There are five main Istio CRDs: **VirtualService**, DestinationRule, Gateway, Service Entry, Sidecar



You can think of the VirtualService as the ‘Where’ and the Destination Rule as the ‘How’.

A Virtual Service:

- Defines how requests for a service are routed within an Istio service mesh.
- Can use routing rules to determine the destination to which a request should be sent.
- Can use rules to route traffic based on things like request headers, target host name, url path, etc.
- Can use target destinations as a service in the mesh registry using proxies, a subset of the service (specific group of pods), or a non-proxied service registered via a ServiceEntry.



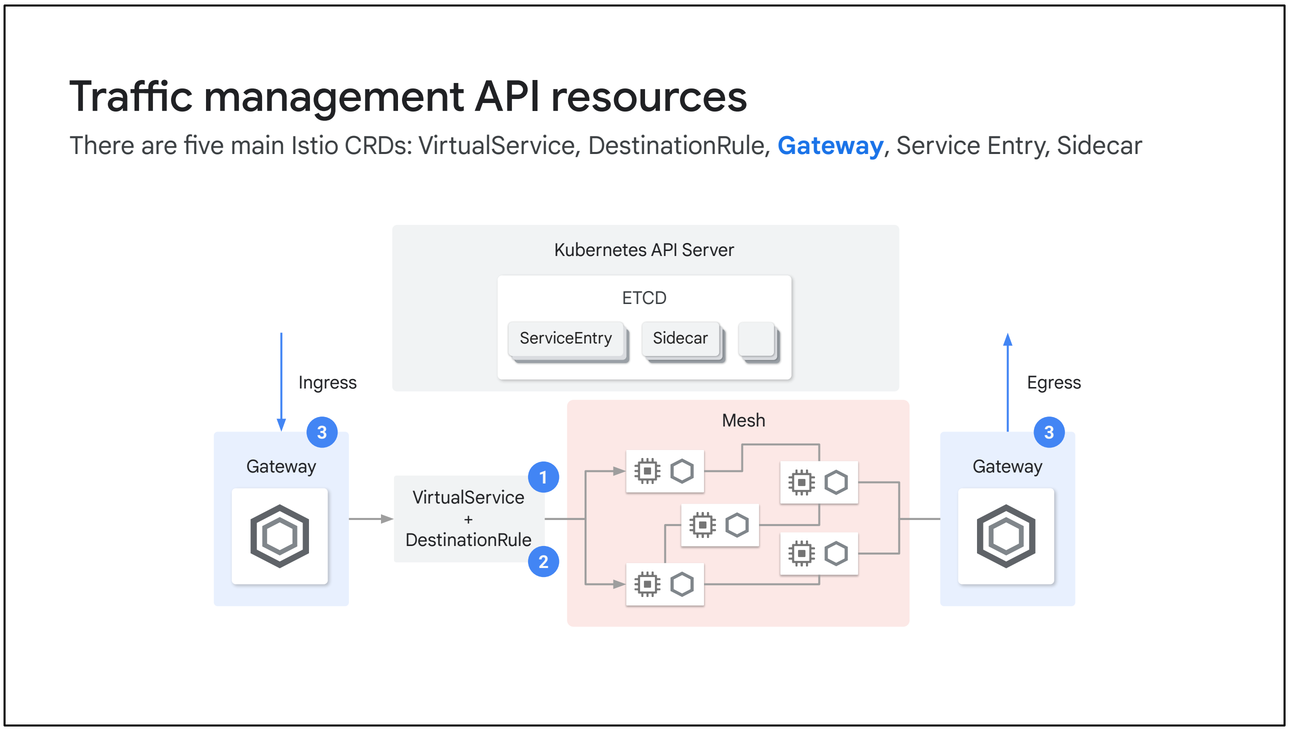
Traffic management API resources

There are five main Istio CRDs: VirtualService, **DestinationRule**, Gateway, Service Entry, Sidecar

The Destination Rule:

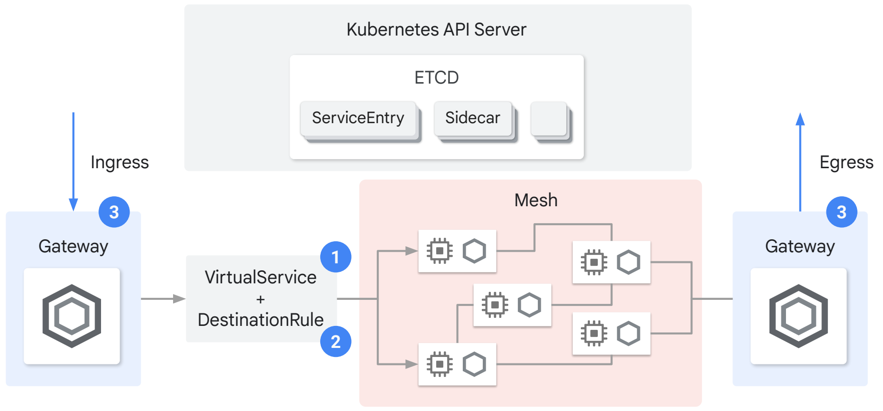
- Defines how traffic aimed at a particular destination gets handled.
- Can break a single service into multiple subsets - in other words, subcollections of pods using labels to select. The VirtualService can then route to the subset.
- Can specify load balancing behavior within the destination - random, round robin, weighted, etc.
- And it can specify TLS security mode, circuit breaker settings, etc.

Thus, a request coming from a client will select the pod to connect to by using both the VirtualService settings (to pick a destination) and DestinationRule settings (to select how to connect to that destination).



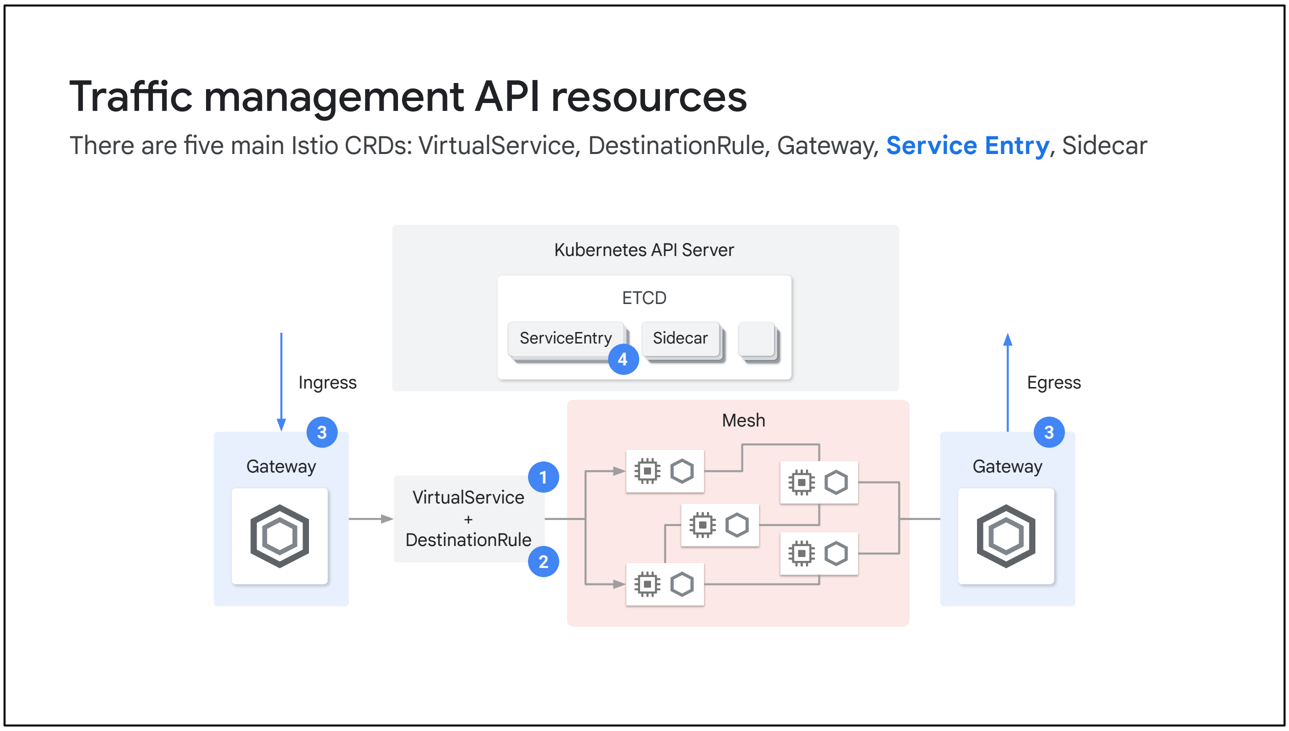
Traffic management API resources

There are five main Istio CRDs: VirtualService, DestinationRule, **Gateway**, Service Entry, Sidecar



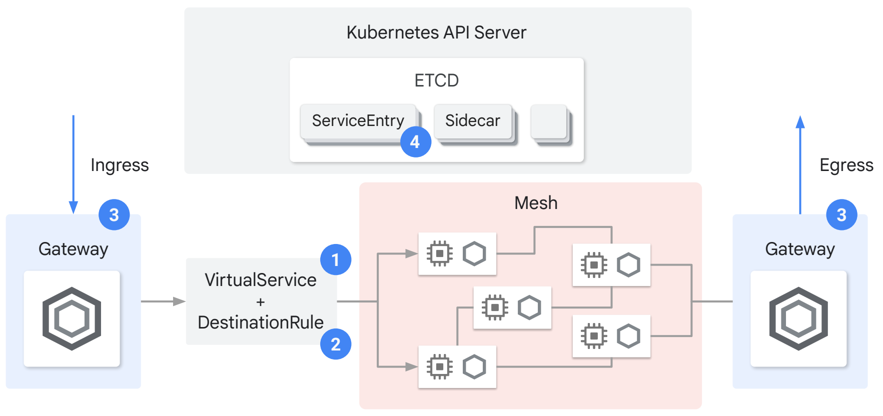
Gateways

- Gateways are used to manage inbound and outbound traffic for the mesh.
- The configuration settings are applied to a deployment on Envoy poxy pods running on the edge of the mesh, not as sidecars to particular workloads.
- A gateway can, for instance, allow incoming HTTP traffic for a specific host.
- You then bind a VirtualService to the Gateway, and the Gateway uses the VirtualService information to route the incoming request to the correct destination.

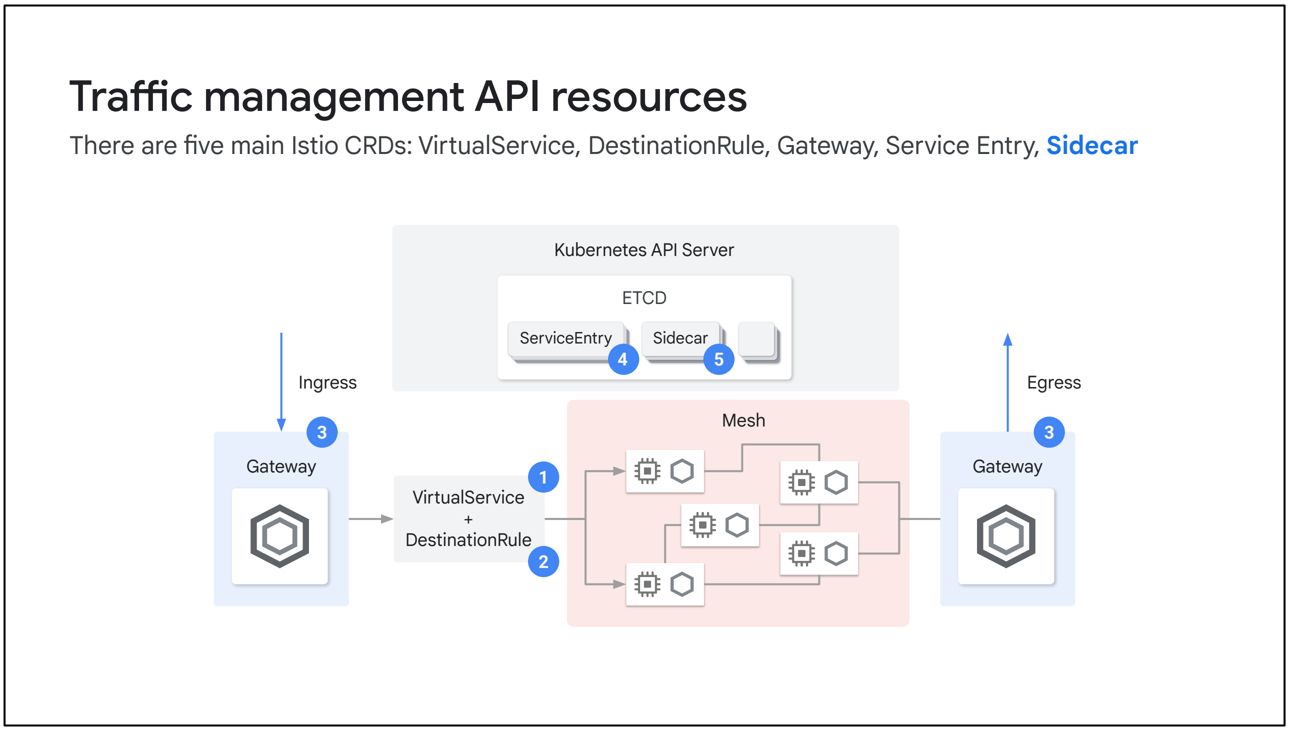


Traffic management API resources

There are five main Istio CRDs: VirtualService, DestinationRule, Gateway, **Service Entry**, Sidecar

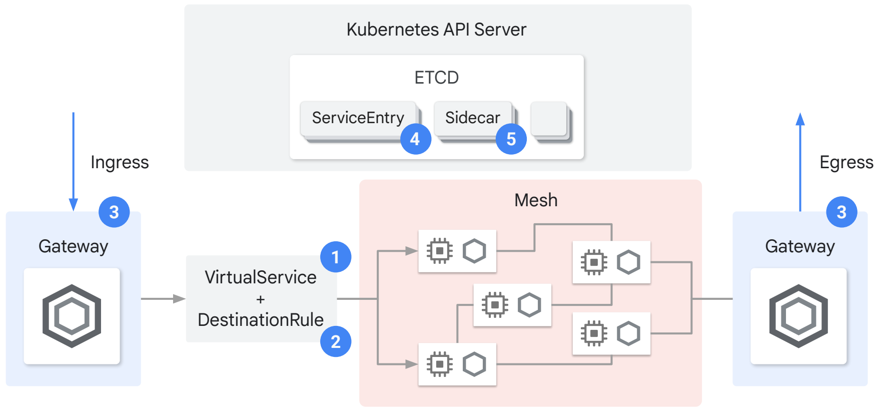


The Service Entry is commonly used to enable requests to services outside of an Istio service mesh.



Traffic management API resources

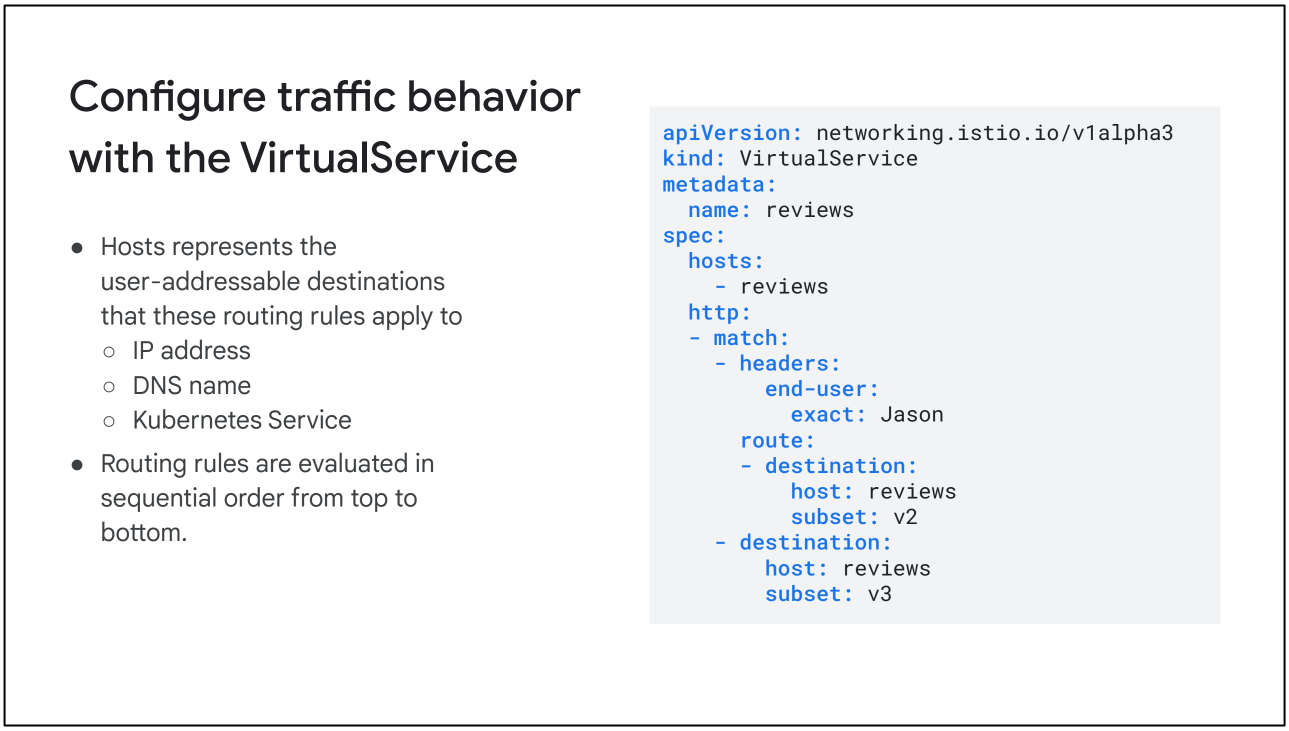
There are five main Istio CRDs: VirtualService, DestinationRule, Gateway, Service Entry, **Sidecar**



Sidecar configurations are used to fine-tune Envoy proxy settings.

By default, sidecar proxies are configured to accept traffic on all ports used by workload, and can reach every workload in the mesh.

If you want to limit the protocols, ports, or reachable services in the mesh, you do this with sidecar configurations.



Configure traffic behavior with the VirtualService

- Hosts represents the user-addressable destinations that these routing rules apply to
  - IP address
  - DNS name
  - Kubernetes Service
- Routing rules are evaluated in sequential order from top to bottom.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - match:
    - headers:
        end-user:
          exact: Jason
      route:
      - destination:
          host: reviews
          subset: v2
    - destination:
        host: reviews
        subset: v3
```

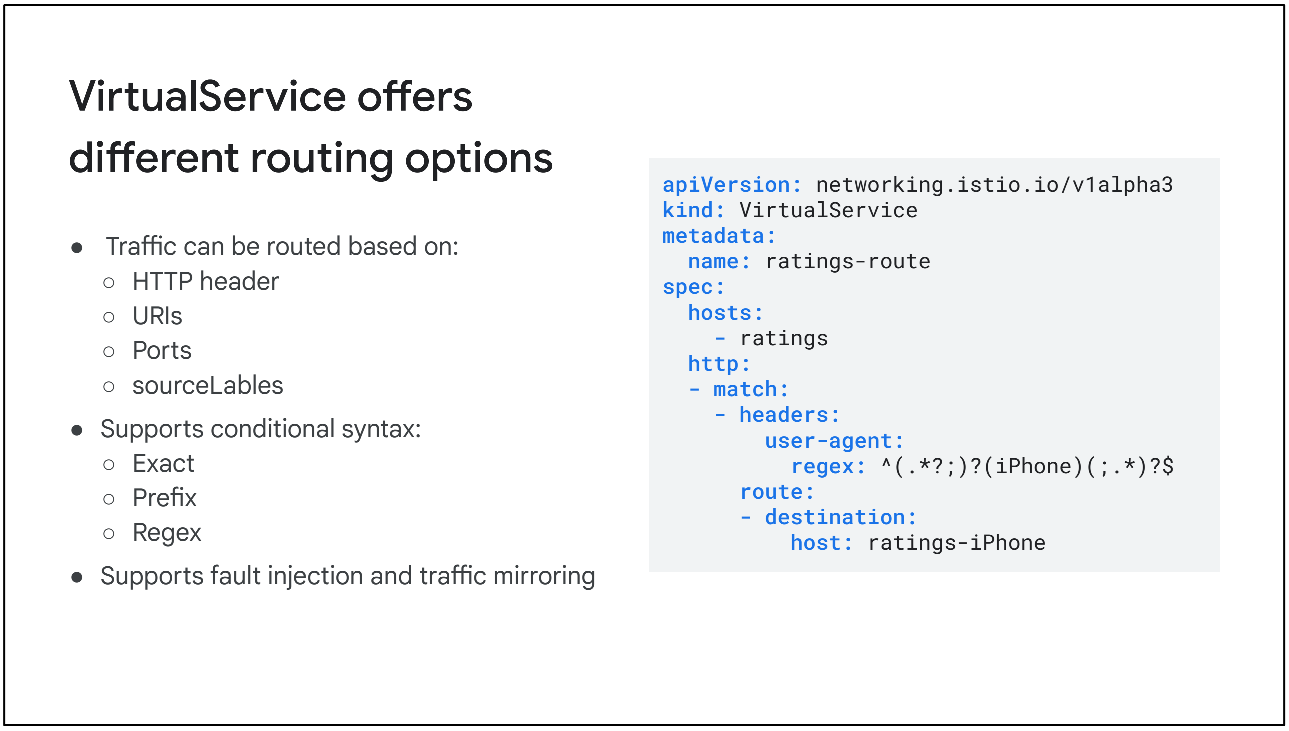
By default, the Envoy proxies distribute traffic across each service's load balancing pool using a round-robin model, where requests are sent to each pool member in turn, returning to the top of the pool once each service instance has received a request.

You can improve this behavior with what you know about the workloads. For example, some might represent a different version. This can be useful in A/B testing, where you might want to configure traffic routes based on percentages across different service versions, or to direct traffic from your internal users to a particular set of instances.

You configure traffic behavior with the VirtualService.

The virtual service hostname can be an IP address, a DNS name, or, depending on the platform, a short name (such as a Kubernetes service short name) that resolves, implicitly or explicitly, to a fully qualified domain name, or FQDN. Using short names like this only works if the destination hosts and the virtual service are actually in the same Kubernetes namespace. Because using the Kubernetes short name can result in misconfigurations, we recommend that you specify fully qualified host names in production environments. You can also use wildcard prefixes, for example, *.myapp.com, letting you create a single set of routing rules for all matching services. Virtual service hosts don't actually have to be part of the Istio service registry, they are simply virtual destinations. This lets you model traffic for virtual hosts that don't have routable entries inside the mesh.

In this example, requests that contain the user Jason are sent to v2, while all traffic that doesn’t match this condition is sent to v3, as rules are evaluated in sequential order from top to bottom.



VirtualService offers different routing options

- Traffic can be routed based on:
  - HTTP header
  - URIs
  - Ports
  - sourceLables
- Supports conditional syntax:
  - Exact
  - Prefix
  - Regex
- Supports fault injection and traffic mirroring

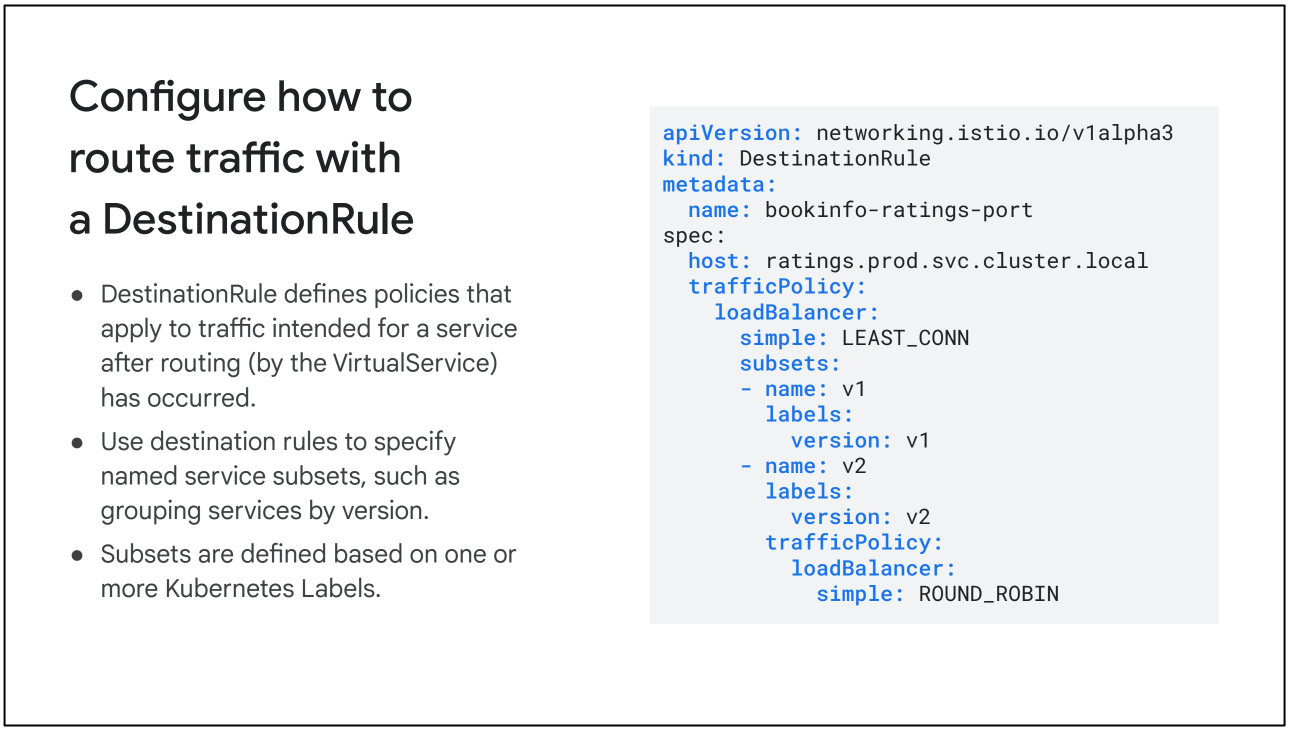
```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings-route
spec:
  hosts:
    - ratings
  http:
  - match:
    - headers:
        user-agent:
          regex: ^(.*?;)?(iPhone)(;.*)?$
      route:
      - destination:
          host: ratings-iPhone
```

VirtualService offers different routing options.

In this example, we use the conditional syntax with a regex expression in order to inspect the user-agent HTTP header and direct iPhone users to a specific service that handles their workload.

You can also route traffic based on HTTP header, URIs, Ports, and sourceLabels.

VirtualServices can also modify the traffic by injecting faults to test how your application performs under an unstable environment, and duplicate the traffic with traffic mirroring to test shadow deployments or analyze traffic to detect network intrusions.



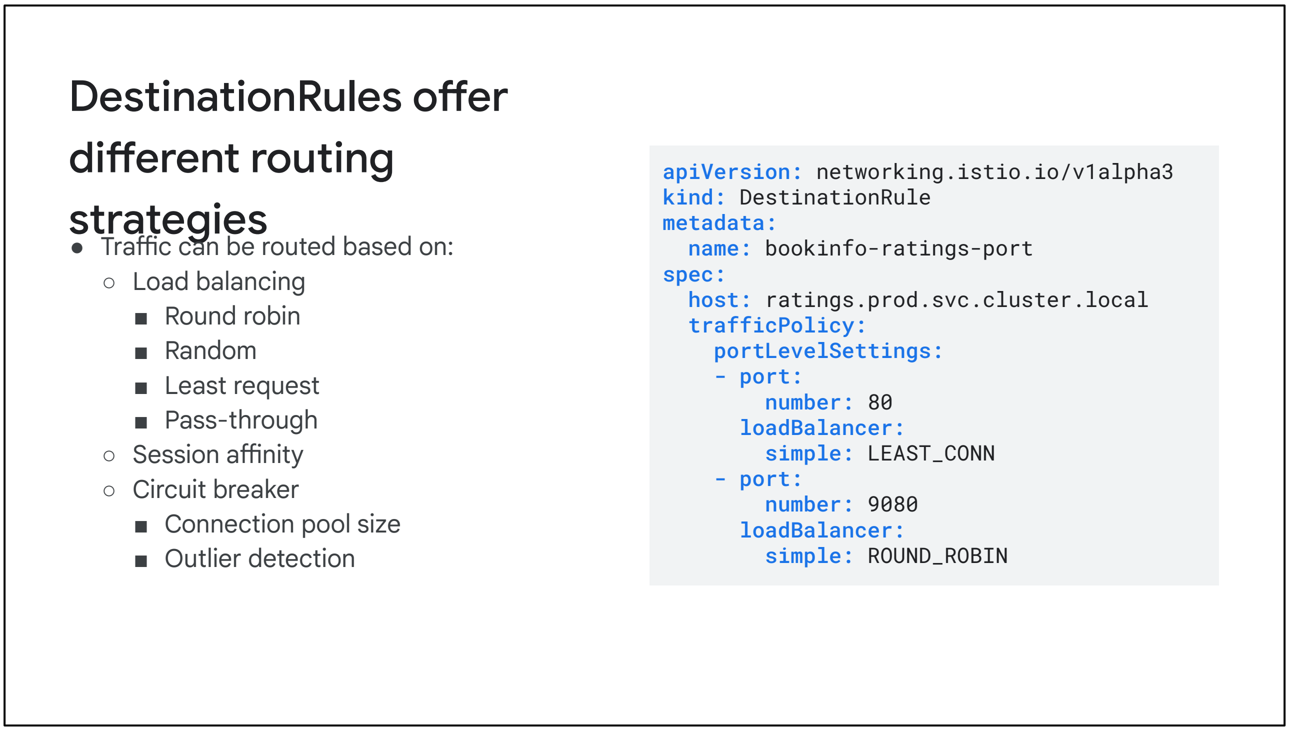
Configure how to route traffic with a DestinationRule

- DestinationRule defines policies that apply to traffic intended for a service after routing (by the VirtualService) has occurred.
- Use destination rules to specify named service subsets, such as grouping services by version.
- Subsets are defined based on one or more Kubernetes Labels.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: bookinfo-ratings-port
spec:
  host: ratings.prod.svc.cluster.local
  trafficPolicy:
    loadBalancer:
      simple: LEAST_CONN
      subsets:
      - name: v1
        labels:
          version: v1
      - name: v2
        labels:
          version: v2
        trafficPolicy:
          loadBalancer:
            simple: ROUND_ROBIN
```

A DestinationRule is usually coupled with a VirtualService. Once the routing has been determined by the VirtualService, the destination rule determines *how* to route the traffic. In particular, you use destination rules to specify named service subsets, such as grouping all a given service's instances by version. You can then use these service subsets in the routing rules of virtual services to control the traffic to different instances of your services.

Each subset is defined based on one or more labels, which in Kubernetes are key/value pairs that are attached to objects such as pods. These labels are applied in the Kubernetes Service's Deployment as metadata to identify different versions.



DestinationRules offer different routing strategies

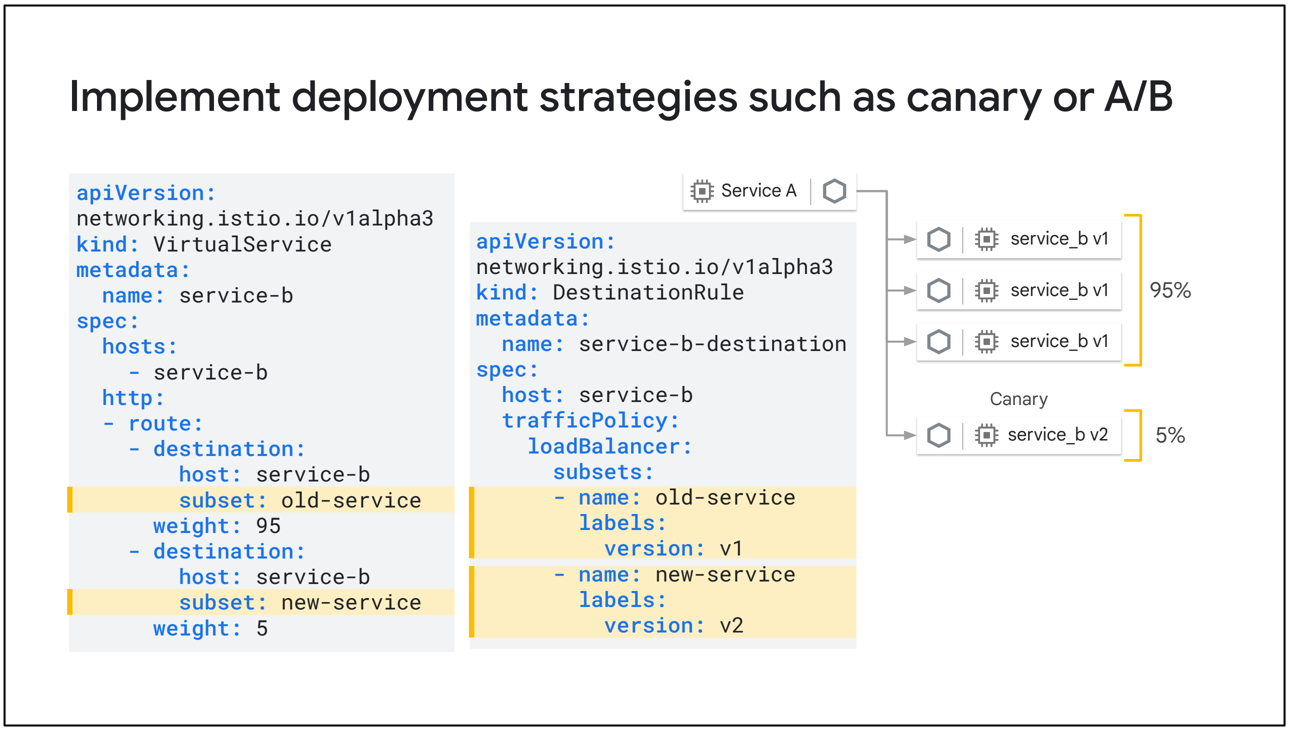
- Traffic can be routed based on:
  - Load balancing
    - Round robin
    - Random
    - Least request
    - Pass-through
  - Session affinity
  - Circuit breaker
    - Connection pool size
    - Outlier detection

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: bookinfo-ratings-port
spec:
  host: ratings.prod.svc.cluster.local
  trafficPolicy:
    portLevelSettings:
    - port:
        number: 80
      loadBalancer:
        simple: LEAST_CONN
    - port:
        number: 9080
      loadBalancer:
        simple: ROUND_ROBIN
```

DestinationRules offer different routing strategies. For example, it configures load balancing, with round-robin, random, least request, and pass-through, and allows you to configure session affinity, so that the clients get consistently routed to the same service.

DestinationRules also allow you to configure circuit-breaking capabilities. Circuit breaking allows you to write applications that limit the impact of failures, latency spikes, and other undesirable effects of network peculiarities.

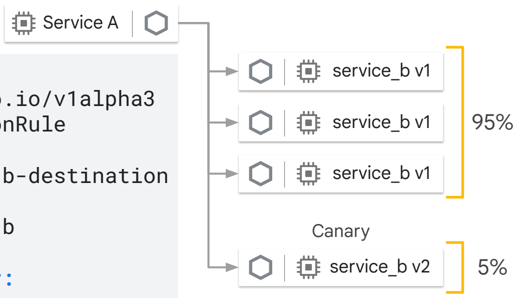
For example, you can set the connections pool size from the sidecar to limit 100 connection to a Redis service, or configure outlier detection settings to detect and evict unhealthy hosts from the load balancing pool.



Implement deployment strategies such as canary or A/B

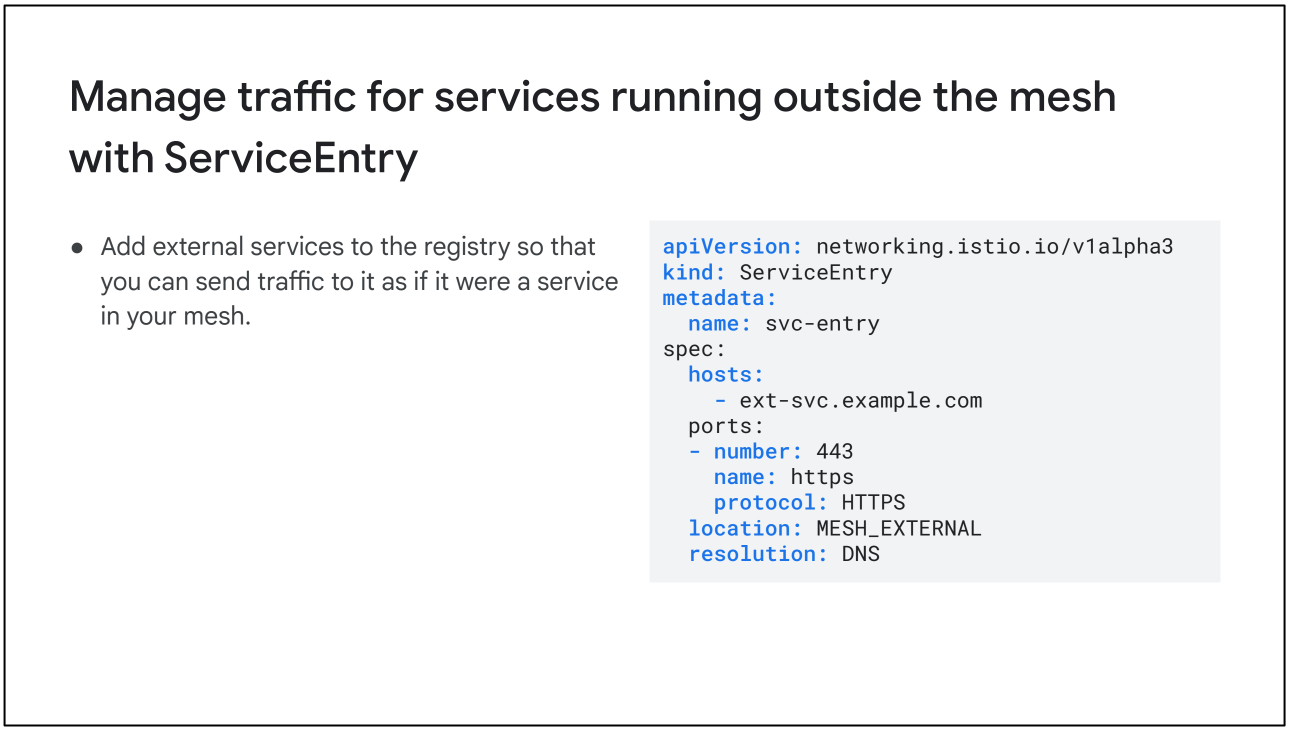
```
apiVersion:
networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: service-b
spec:
  hosts:
    - service-b
  http:
  - route:
    - destination:
        host: service-b
        subset: old-service
      weight: 95
    - destination:
        host: service-b
        subset: new-service
      weight: 5
```

```
apiVersion:
networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: service-b-destination
spec:
  host: service-b
  trafficPolicy:
    loadBalancer:
      subsets:
      - name: old-service
        labels:
          version: v1
      - name: new-service
        labels:
          version: v2
```



Having multiple destinations in the route attribute provides traffic splitting functionality to your Virtual Service. A subset defines any subset of a service, either a canary, A/B or blue-green subsets. Notice that the labels in the destination rule must match your pod labels for this configuration to take effect. Also, the subset field in the VirtualService must match the subsets in the DestinationRule.

Instead of depending on the number compute instances in the infrastructure to split traffic, we can rely on the traffic splitting functionality and set a specific percentage of traffic regardless of the number of compute instances in each subset, resulting in traffic control that is decoupled from infrastructure scaling.

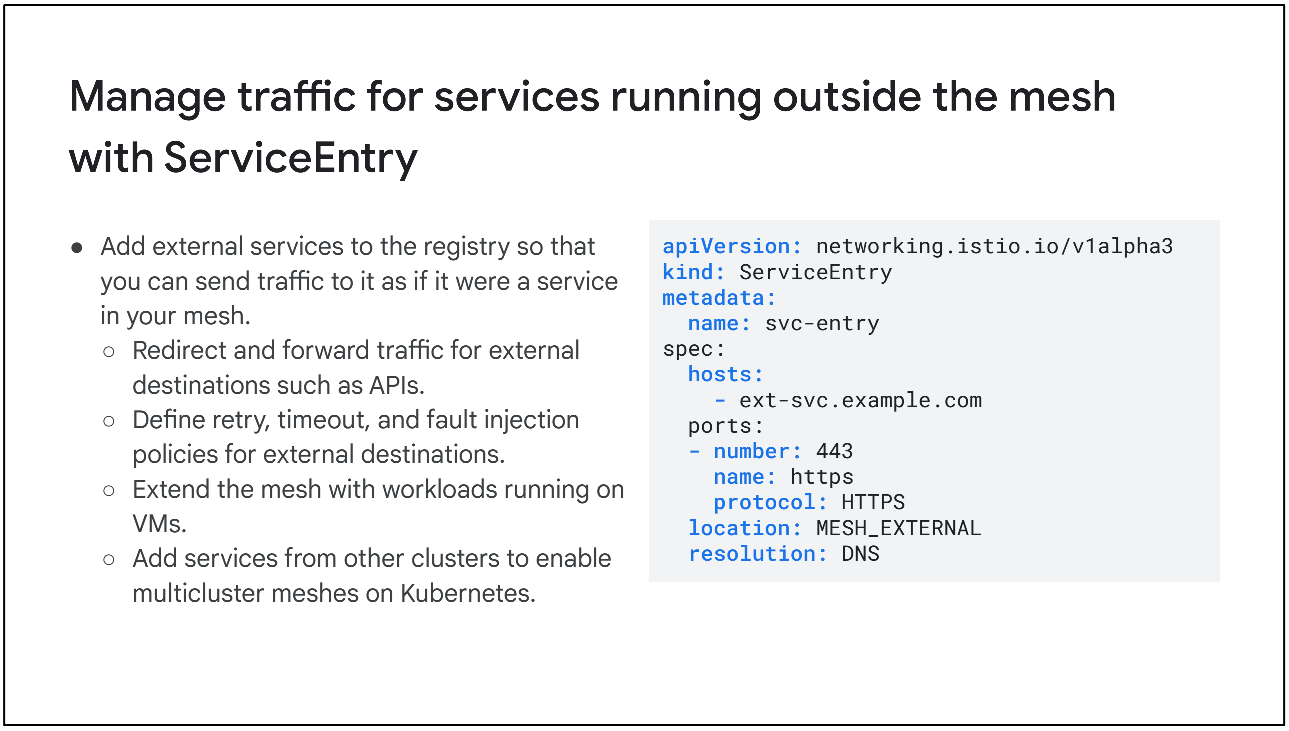


Manage traffic for services running outside the mesh with ServiceEntry

- Add external services to the registry so that you can send traffic to it as if it were a service in your mesh.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: svc-entry
spec:
  hosts:
    - ext-svc.example.com
  ports:
  - number: 443
    name: https
    protocol: HTTPS
  location: MESH_EXTERNAL
  resolution: DNS
```

You use a service entry to add an entry to the service registry that the service mesh maintains internally. After you add the service entry, the Envoy proxies can send traffic to the service as if it was a service in your mesh. Configuring service entries allows you to manage traffic for services running outside of the mesh, including the following tasks:



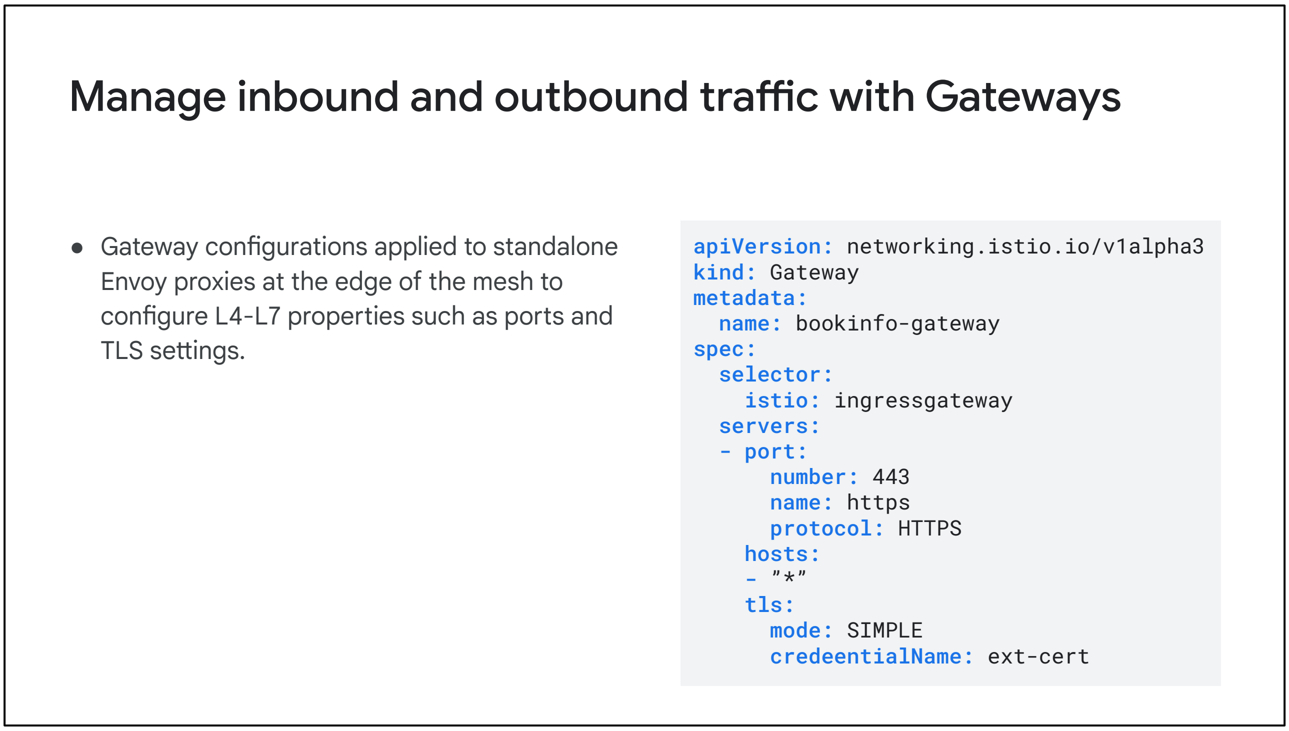
Manage traffic for services running outside the mesh with ServiceEntry

- Add external services to the registry so that you can send traffic to it as if it were a service in your mesh.
  - Redirect and forward traffic for external destinations such as APIs.
  - Define retry, timeout, and fault injection policies for external destinations.
  - Extend the mesh with workloads running on VMs.
  - Add services from other clusters to enable multicluster meshes on Kubernetes.

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: svc-entry
spec:
  hosts:
    - ext-svc.example.com
  ports:
  - number: 443
    name: https
    protocol: HTTPS
  location: MESH_EXTERNAL
  resolution: DNS
```

- Redirect and forward traffic for external destinations, such as APIs consumed from the web, or traffic to services in legacy infrastructure.
- Define retry, timeout, and fault injection policies for external destinations.
- Run a mesh service in a Virtual Machine, or VM, by adding VMs to your mesh.
- Logically add services from a different cluster to the mesh to configure a multicluster Istio mesh on Kubernetes.

You don’t need to add a service entry for every external service that you want your mesh services to use. By default, Anthos Service Mesh configures the Envoy proxies to passthrough requests to unknown services. However, you can’t use Anthos Service Mesh features to control the traffic to destinations that aren’t registered in the mesh.



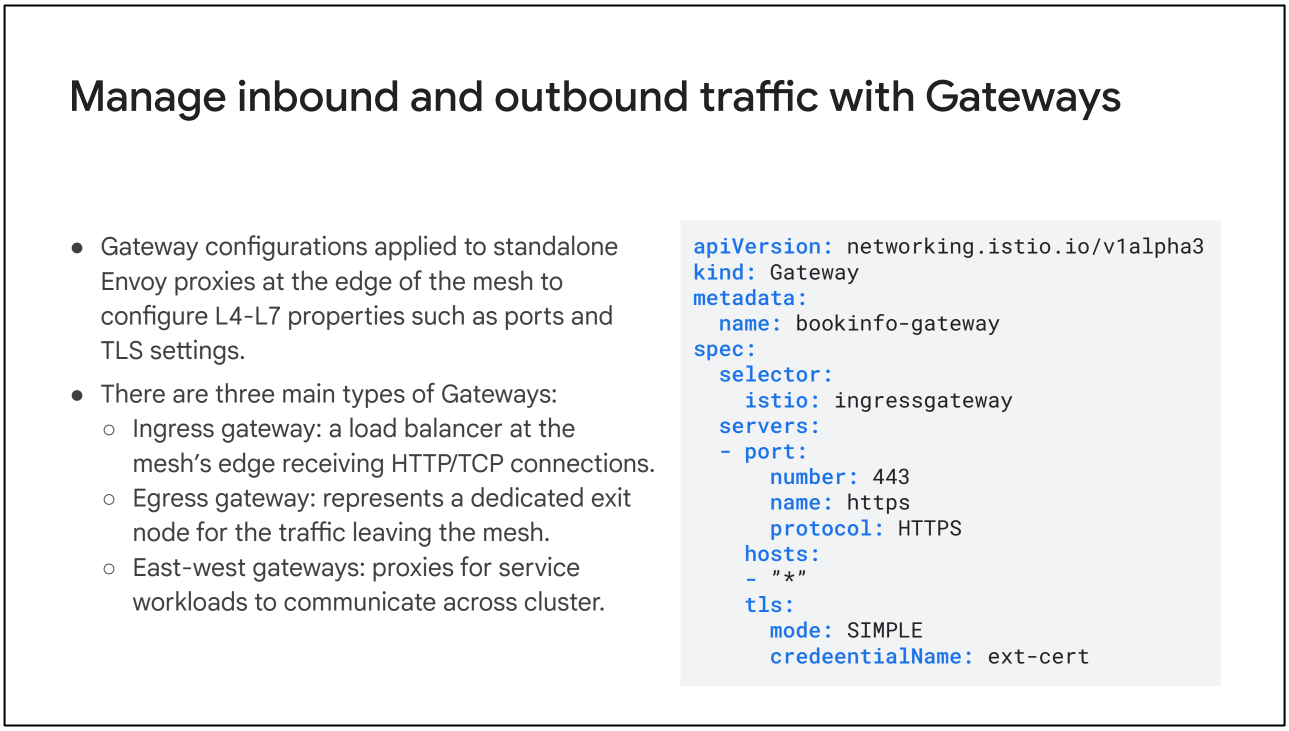
Manage inbound and outbound traffic with Gateways

- Gateway configurations applied to standalone Envoy proxies at the edge of the mesh to configure L4-L7 properties such as ports and TLS settings.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    hosts:
    - "*"
    tls:
      mode: SIMPLE
      credeentialName: ext-cert
```

Along with support for Kubernetes Ingress, Anthos Service Mesh offers Gateways to manage inbound and outbound traffic to the mesh. A Gateway provides more extensive customization and flexibility than Ingress, and allows service mesh features such as monitoring and route rules to be applied to traffic entering the cluster.

You use a gateway to manage inbound and outbound traffic for your mesh, letting you specify which traffic you want to enter or leave the mesh. Gateway configurations are applied to standalone Envoy proxies that are running at the edge of the mesh, rather than sidecar Envoy proxies running alongside your service workloads. Gateways let you configure layer 4-6 load balancing properties such as ports to expose, TLS settings, and so on. Then, instead of adding application-layer traffic routing (L7) to the same API resource, you bind a VirtualService to the gateway. This lets you basically manage gateway traffic like any other data plane traffic in an Anthos Service Mesh.



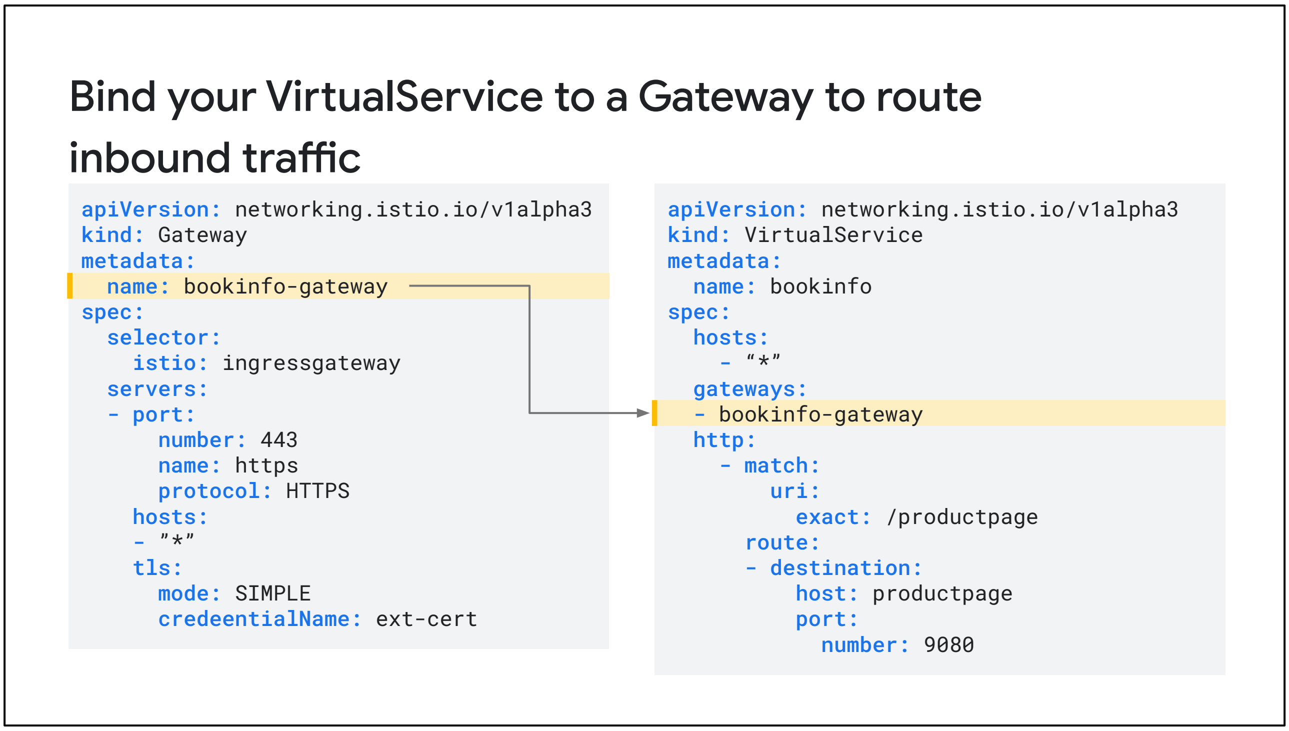
Manage inbound and outbound traffic with Gateways

- Gateway configurations applied to standalone Envoy proxies at the edge of the mesh to configure L4-L7 properties such as ports and TLS settings.
- There are three main types of Gateways:
  - Ingress gateway: a load balancer at the mesh's edge receiving HTTP/TCP connections.
  - Egress gateway: represents a dedicated exit node for the traffic leaving the mesh.
  - East-west gateways: proxies for service workloads to communicate across cluster.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    hosts:
    - "*"
    tls:
      mode: SIMPLE
      credeentialName: ext-cert
```

There are three main types of gateways:

- Ingress gateways: describe a load balancer operating at the edge of the mesh receiving incoming HTTP/TCP connections.
- Egress gateways: represent a dedicated exit node for the traffic leaving the mesh, letting you limit which services can or should access external networks, or to enable secure control of egress traffic to add security to your mesh, for example. You can also use a gateway to configure a purely internal proxy.
- East-west gateways are proxies for east-west traffic to allow service workloads to communicate across cluster boundaries in a multi-primary mesh on different networks.



Bind your VirtualService to a Gateway to route inbound traffic

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    hosts:
    - "*"
    tls:
      mode: SIMPLE
      credeentialName: ext-cert
```

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo
spec:
  hosts:
    - “*”
  gateways:
  - bookinfo-gateway
  http:
    - match:
        uri:
          exact: /productpage
      route:
      - destination:
          host: productpage
          port:
            number: 9080
```

Bind your VirtualService to a Gateway to route inbound traffic from outside the mesh. That way, the Gateway can provide routing configuration for layers 4-6 while the VirtualService can be configured a the application layer or L7.

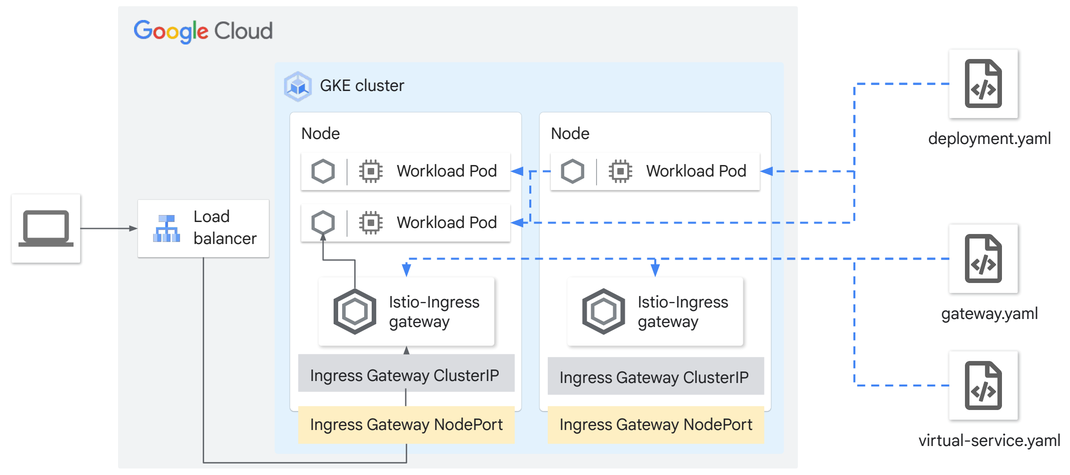
In this example, the Gateway enables HTTPS connection and links to an external certificate. The link takes place in the VirtualService, specifically in the field called gateways, where it links to the bookinfo-gateway.

Also, notice that hosts is set to asterisk, which means traffic to any destination.

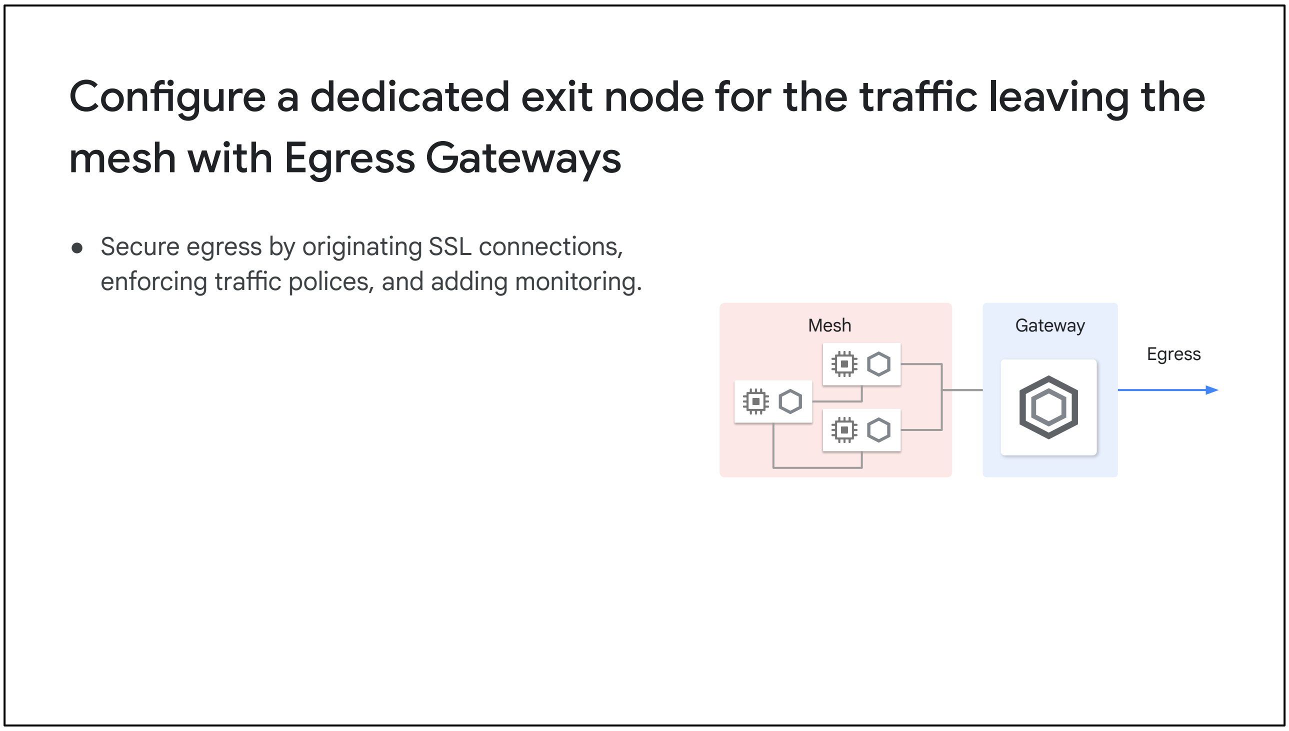
When the incoming traffic enters from the Gateway, the VirtualService evaluates the conditions set under the match attribute and routes the traffic based on the route attribute.

The host attribute, on spec.http.match.route.destination.host is actually the Kubernetes service.

Inbound traffic flow from a load balancer into the mesh

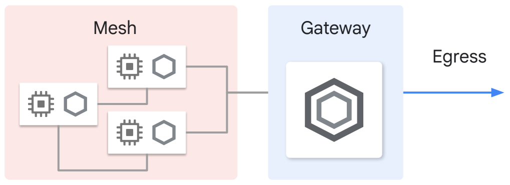


The istio-ingressgateway is a Kubernetes deployment and Kubernetes Service of type LoadBalancer. Therefore, when the ingress gateway is created, Google creates a Google Cloud Load Balancer. The Load Balancer forwards the traffic to a NodePort on a worker node, and the worker node forwards the traffic to the ingress gateway pod which handles ingress functionality. The ingress gateway deployment is comprised of a single Envoy container per Pod. That Envoy is configured with VirtualServices and DestinationRules, the same way as any other Envoys in the mesh. Once the Ingress Gateway's Envoy processes the request, it forwards it to the right destination.



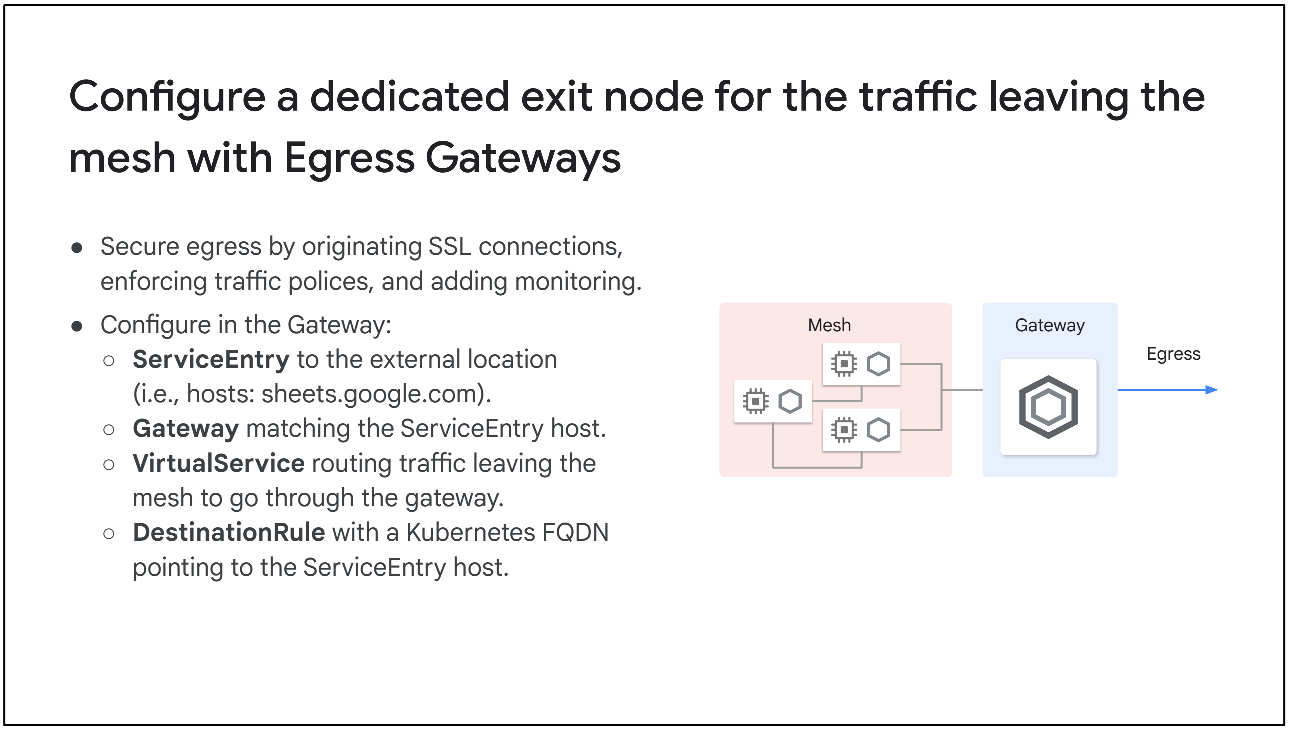
Configure a dedicated exit node for the traffic leaving the mesh with Egress Gateways

- Secure egress by originating SSL connections, enforcing traffic polices, and adding monitoring.



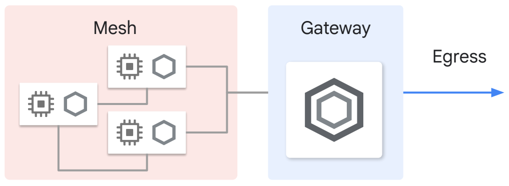
You can also configure Egress Gateways for controlling the traffic leaving your mesh.

Consider an organization that has a strict security requirement that all traffic leaving the service mesh must flow through a set of dedicated nodes. These nodes run on dedicated machines, separated from the rest of the nodes running applications in the cluster. These special nodes serve for policy enforcement on the egress traffic and are monitored more thoroughly than other nodes. Another use case is a cluster where the application nodes don't have public IPs, so the in-mesh services that run on them cannot access the internet. Defining an egress gateway, directing all the egress traffic through it, and allocating public IPs to the egress gateway nodes allows the application nodes to access external services in a controlled way.



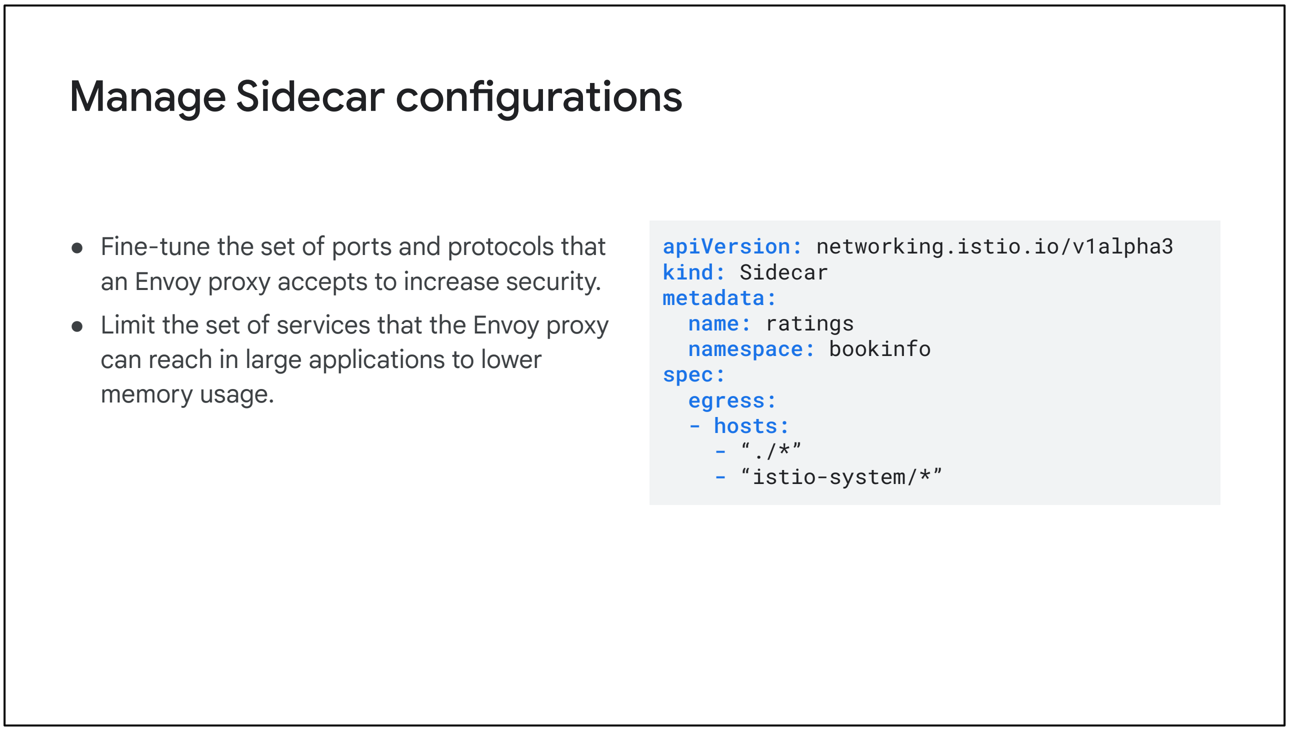
Configure a dedicated exit node for the traffic leaving the mesh with Egress Gateways

- Secure egress by originating SSL connections, enforcing traffic polices, and adding monitoring.
- Configure in the Gateway:
  - **ServiceEntry** to the external location (i.e., hosts: sheets.google.com).
  - **Gateway** matching the ServiceEntry host.
  - **VirtualService** routing traffic leaving the mesh to go through the gateway.
  - **DestinationRule** with a Kubernetes FQDN pointing to the ServiceEntry host.



There are four steps required to configure an Egress Gateway:

- Add a ServiceEntry to the external location. For example, imagine you want to access the Google Sheets API, you would point the host field to sheets.google.com.
- Create a Gateway resource, using the same host as the ServiceEntry.
- Optionally, specify routing configuration or load balancing strategies using VirtualServices and DestinationRules.



Manage Sidecar configurations

- Fine-tune the set of ports and protocols that an Envoy proxy accepts to increase security.
- Limit the set of services that the Envoy proxy can reach in large applications to lower memory usage.

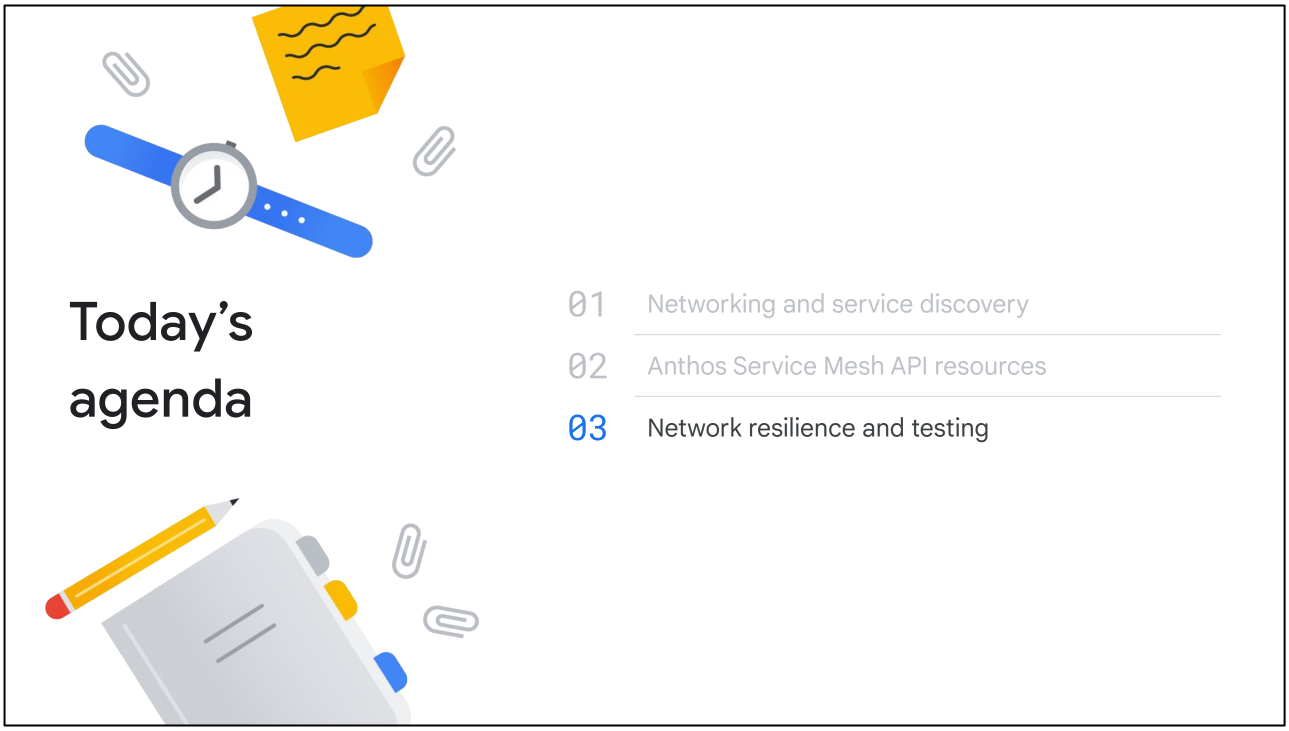
```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Sidecar
metadata:
  name: ratings
  namespace: bookinfo
spec:
  egress:
  - hosts:
    - “./*”
    - “istio-system/*”
```

You can also manage Sidecar configurations. By default, Anthos Service Mesh configures every Envoy proxy to accept traffic on all the ports of its associated workload, and to reach every workload in the mesh when forwarding traffic. You can use a sidecar configuration to do the following:

- Fine-tune the set of ports and protocols that an Envoy proxy accepts.
- Limit the set of services that the Envoy proxy can reach.

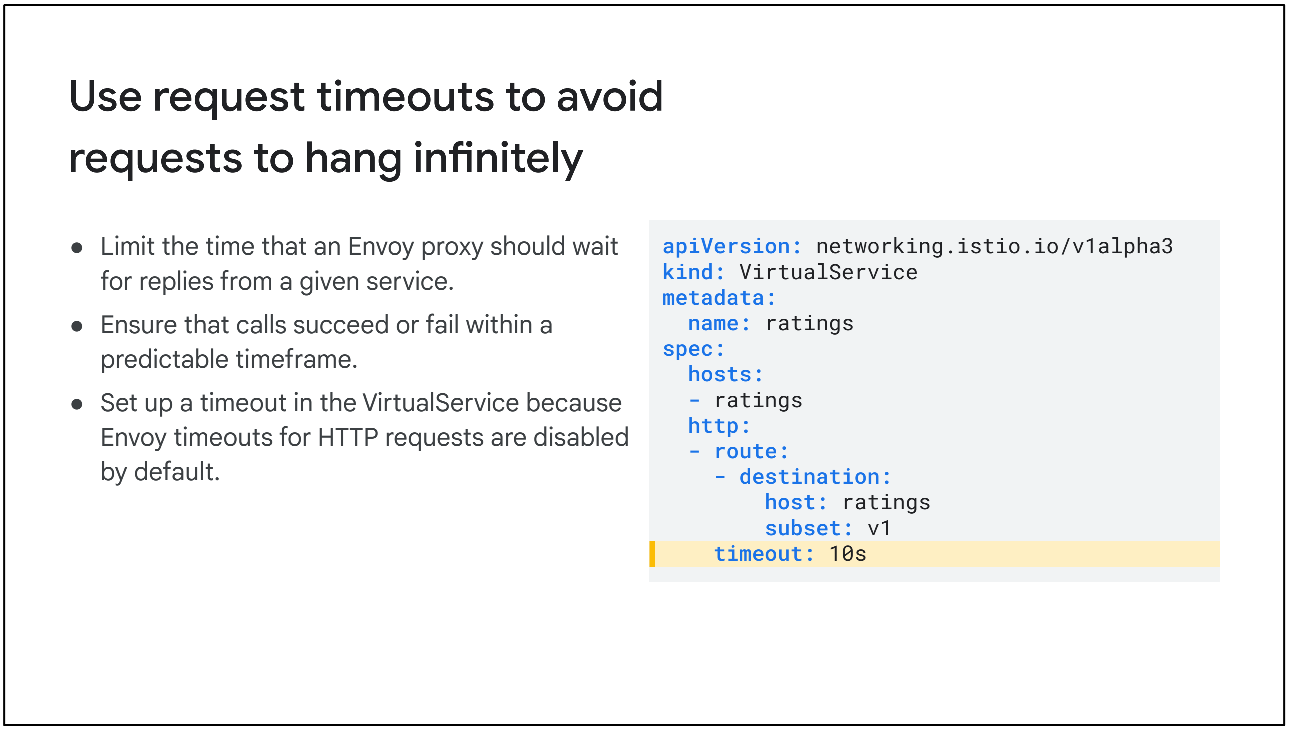
You might want to limit sidecar reachability like this in larger applications, where having every proxy configured to reach every other service in the mesh can potentially affect mesh performance due to high memory usage.

You can specify that you want a sidecar configuration to apply to all workloads in a particular namespace, or choose specific workloads using a workloadSelector. For example, the following sidecar configuration configures all services in the bookinfo namespace to only reach services running in the same namespace and the mesh control plane.



Today's agenda

As well as helping you direct traffic around your mesh, Anthos Service Mesh provides opt-in failure recovery and fault injection features that you can configure dynamically at runtime. Using these features helps your applications operate reliably, ensuring that the service mesh can tolerate failing nodes and preventing localized failures from cascading to other nodes.



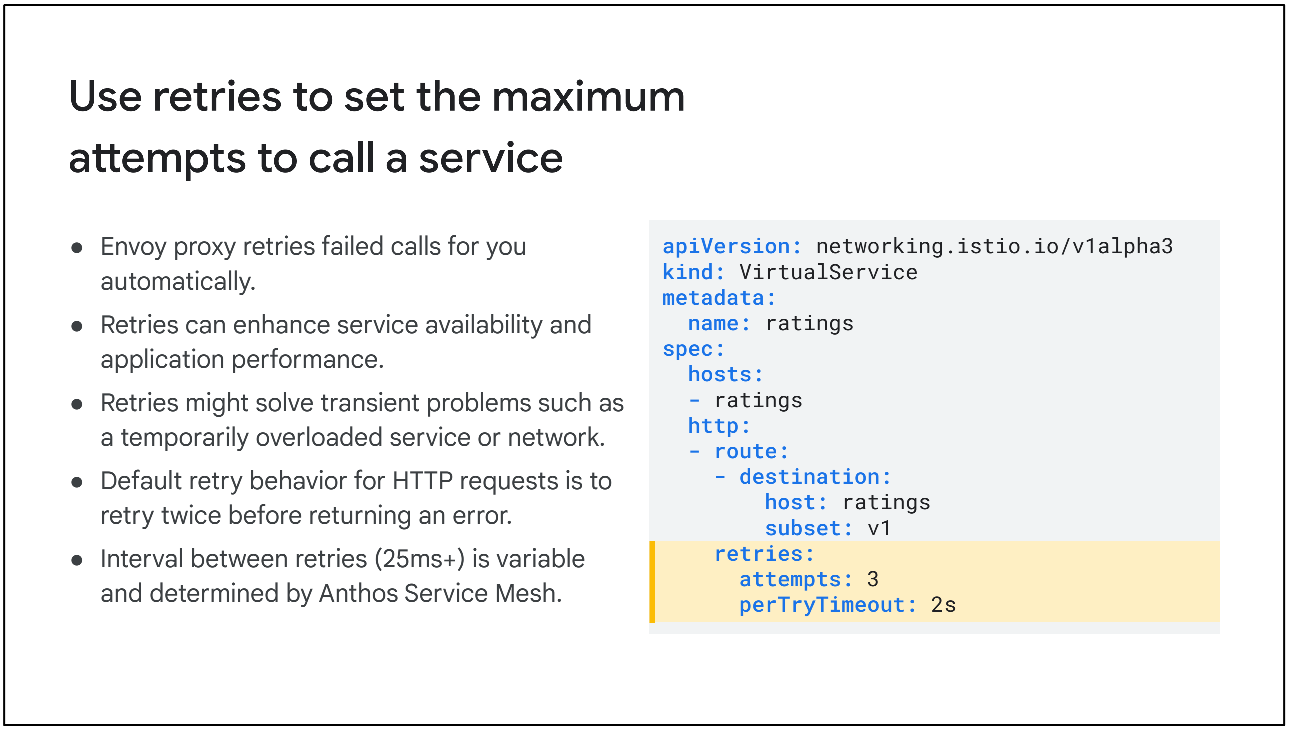
Use request timeouts to avoid requests to hang infinitely

- Limit the time that an Envoy proxy should wait for replies from a given service.
- Ensure that calls succeed or fail within a predictable timeframe.
- Set up a timeout in the VirtualService because Envoy timeouts for HTTP requests are disabled by default.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
    timeout: 10s
```

A timeout is the amount of time that an Envoy proxy should wait for replies from a given service, ensuring that services don't hang around waiting for replies indefinitely and that calls succeed or fail within a predictable timeframe.

A timeout that is too long could result in excessive latency from waiting for replies from failing services, while a timeout that is too short could result in calls failing unnecessarily while waiting for an operation involving multiple services to return. To find and use your optimal timeout settings, VirtualServices lets you easily adjust timeouts dynamically on a per-service basis without having to edit your service code. Here's an example that specifies a 10-second timeout for calls to the service_b service. Notice that Envoy timeouts for HTTP requests are disabled by default.



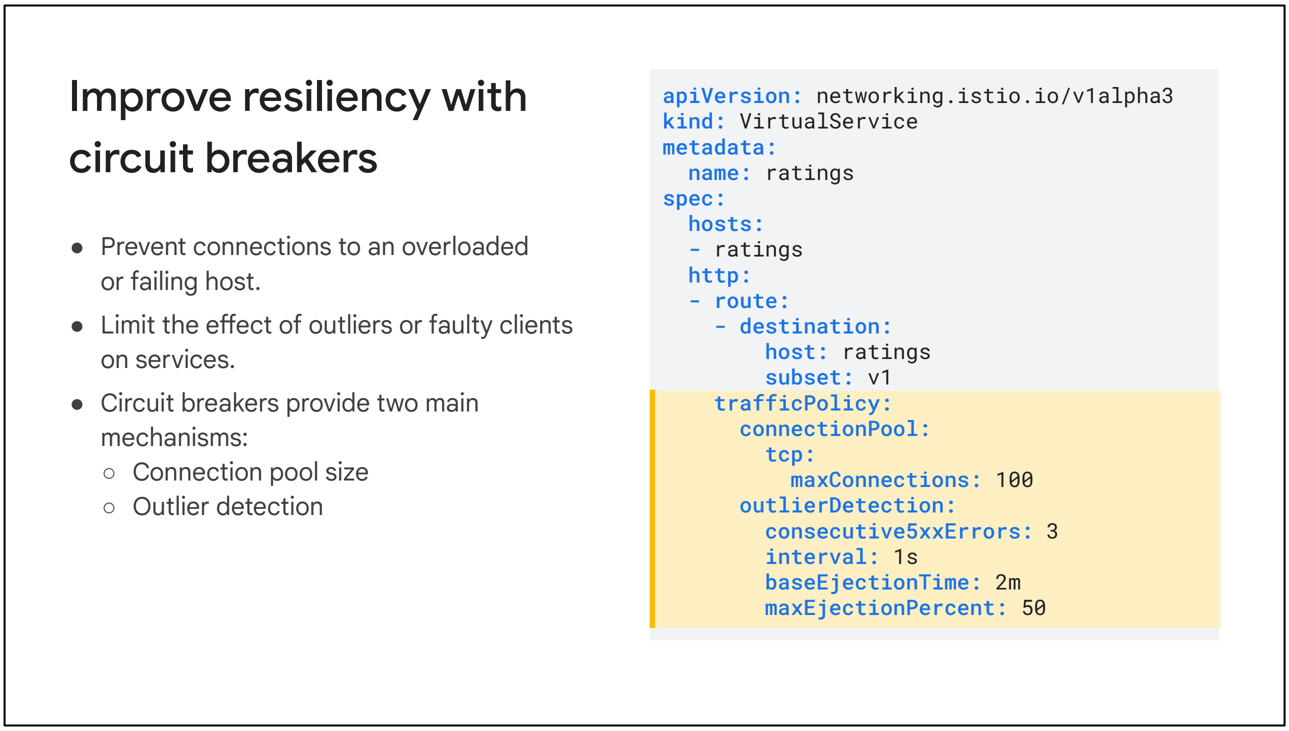
Use retries to set the maximum attempts to call a service

- Envoy proxy retries failed calls for you automatically.
- Retries can enhance service availability and application performance.
- Retries might solve transient problems such as a temporarily overloaded service or network.
- Default retry behavior for HTTP requests is to retry twice before returning an error.
- Interval between retries (25ms+) is variable and determined by Anthos Service Mesh.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
    retries:
      attempts: 3
      perTryTimeout: 2s
```

A retry setting specifies the maximum number of times an Envoy proxy attempts to connect to a service if the initial call fails. Retries can enhance service availability and application performance by making sure that calls don't fail permanently because of transient problems such as a temporarily overloaded service or network. The interval between retries (25 milliseconds or higher) is variable and determined automatically by Anthos Service Mesh, preventing the called service from being overwhelmed with requests. The default retry behavior for HTTP requests is to retry twice before returning the error.

Like timeouts, Anthos Service Mesh's default retry behavior might not suit your application needs in terms of latency or availability - for example, too many retries to a failed service can slow things down. Also like timeouts, you can adjust your retry settings on a per-service basis in virtual services without having to touch your service code. You can also further refine your retry behavior by adding per-retry timeouts, specifying the amount of time you want to wait for each retry attempt to successfully connect to the service. This example configures a maximum of three retries to connect to this service subset after an initial call failure, each with a 2-second timeout.



Improve resiliency with circuit breakers

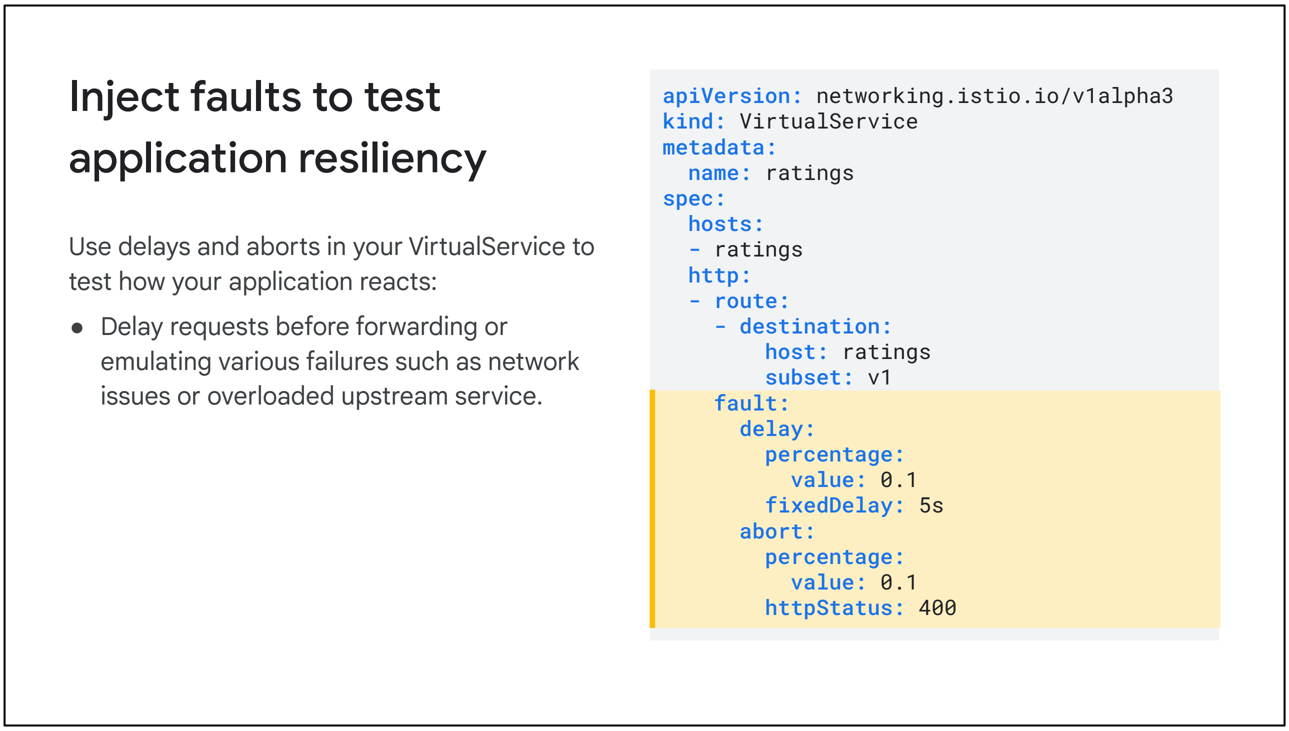
- Prevent connections to an overloaded or failing host.
- Limit the effect of outliers or faulty clients on services.
- Circuit breakers provide two main mechanisms:
  - Connection pool size
  - Outlier detection

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
    trafficPolicy:
      connectionPool:
        tcp:
          maxConnections: 100
      outlierDetection:
        consecutive5xxErrors: 3
        interval: 1s
        baseEjectionTime: 2m
        maxEjectionPercent: 50
```

Circuit breakers are another useful mechanism for creating resilient microservice-based applications. In a circuit breaker, you set limits for calls to individual hosts within a service, such as the number of concurrent connections or how many times calls to this host have failed. Once that limit has been reached, the circuit breaker “trips” and stops further connections to that host. Using a circuit breaker pattern enables fast failure rather than clients trying to connect to an overloaded or failing host.

As circuit breaking applies to “real” mesh destinations in a load balancing pool, you configure circuit breaker thresholds in destination rules, with the settings applying to each individual host in the service. This example limits the number of concurrent connections for the reviews service workloads of the v1 subset to 100.

Also, you can configure outlier detection settings to detect and evict unhealthy hosts from the load balancing pool. In this example, we can see that if the reviews service returns three consecutive 500 errors in a 1-second interval, the service will be ejected for 2 minutes. Up to 50% of all services can be evicted in this example.



Inject faults to test application resiliency

Use delays and aborts in your VirtualService to test how your application reacts:

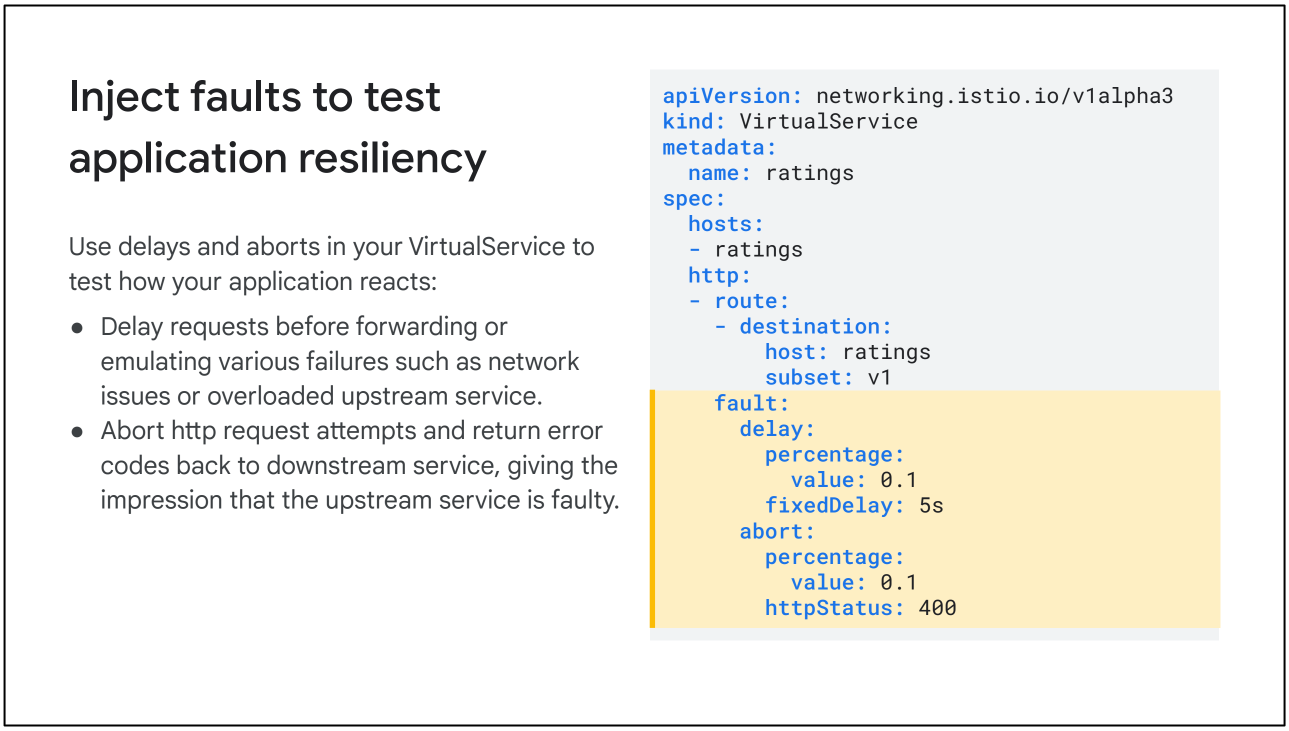
- Delay requests before forwarding or emulating various failures such as network issues or overloaded upstream service.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
    fault:
      delay:
        percentage:
          value: 0.1
        fixedDelay: 5s
      abort:
        percentage:
          value: 0.1
        httpStatus: 400
```

Inject faults to test the resiliency of your application without instrumentation.

Delay requests are applied before forwarding the request from the calling Envoy, emulating various failures such as network issues, overloaded upstream service, etc.

This example introduces a 5-second delay in one out of every 1000 requests. The fixedDelay field is used to indicate the amount of delay in seconds. The optional percentage field can be used to only delay a certain percentage of requests. If left unspecified, all request will be delayed.



Inject faults to test application resiliency

Use delays and aborts in your VirtualService to test how your application reacts:

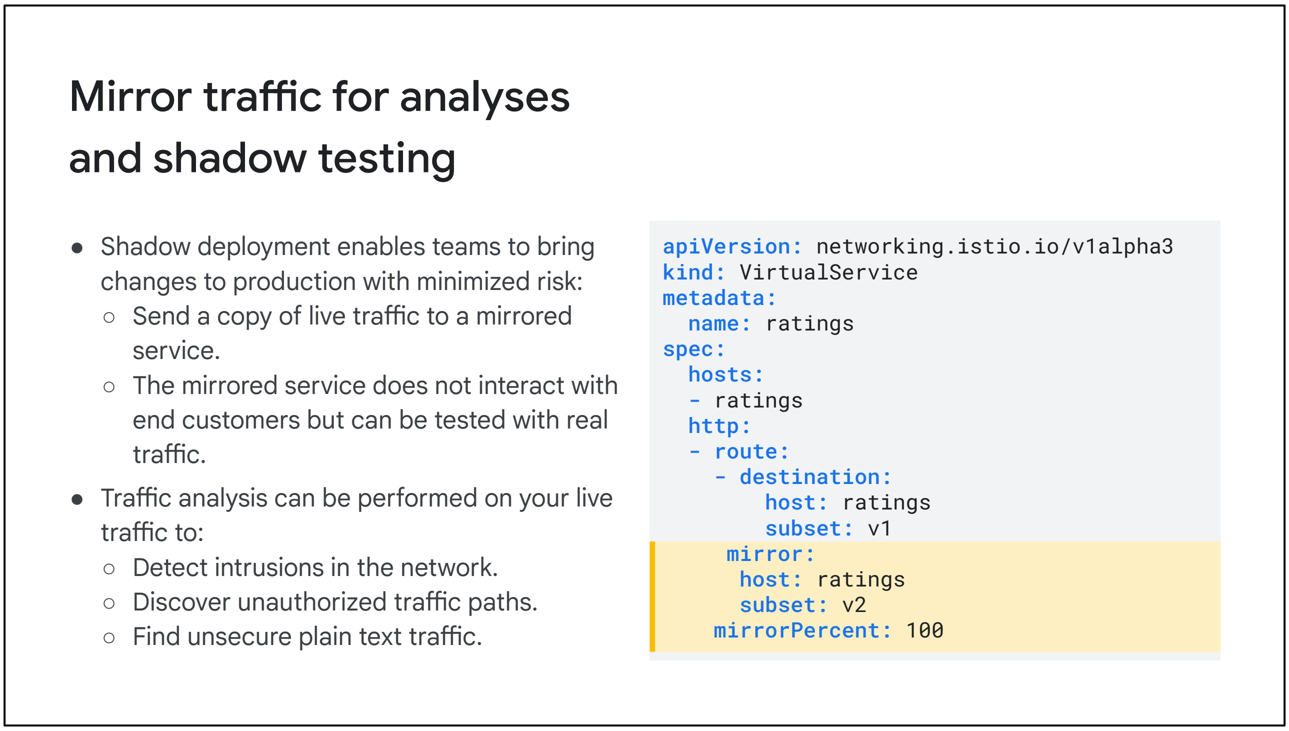
- Delay requests before forwarding or emulating various failures such as network issues or overloaded upstream service.
- Abort http request attempts and return error codes back to downstream service, giving the impression that the upstream service is faulty.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
    fault:
      delay:
        percentage:
          value: 0.1
        fixedDelay: 5s
      abort:
        percentage:
          value: 0.1
        httpStatus: 400
```

Abort http request attempts and return error codes back to downstream service, giving the impression that the upstream service is faulty.

This example introduces a 400 HTTP status code to return to the caller. The optional percentage field can be used to only abort a certain percentage of requests. If not specified, all requests are aborted.

Delay and abort faults are independent of one another, even if both are specified simultaneously.



Mirror traffic for analyses and shadow testing

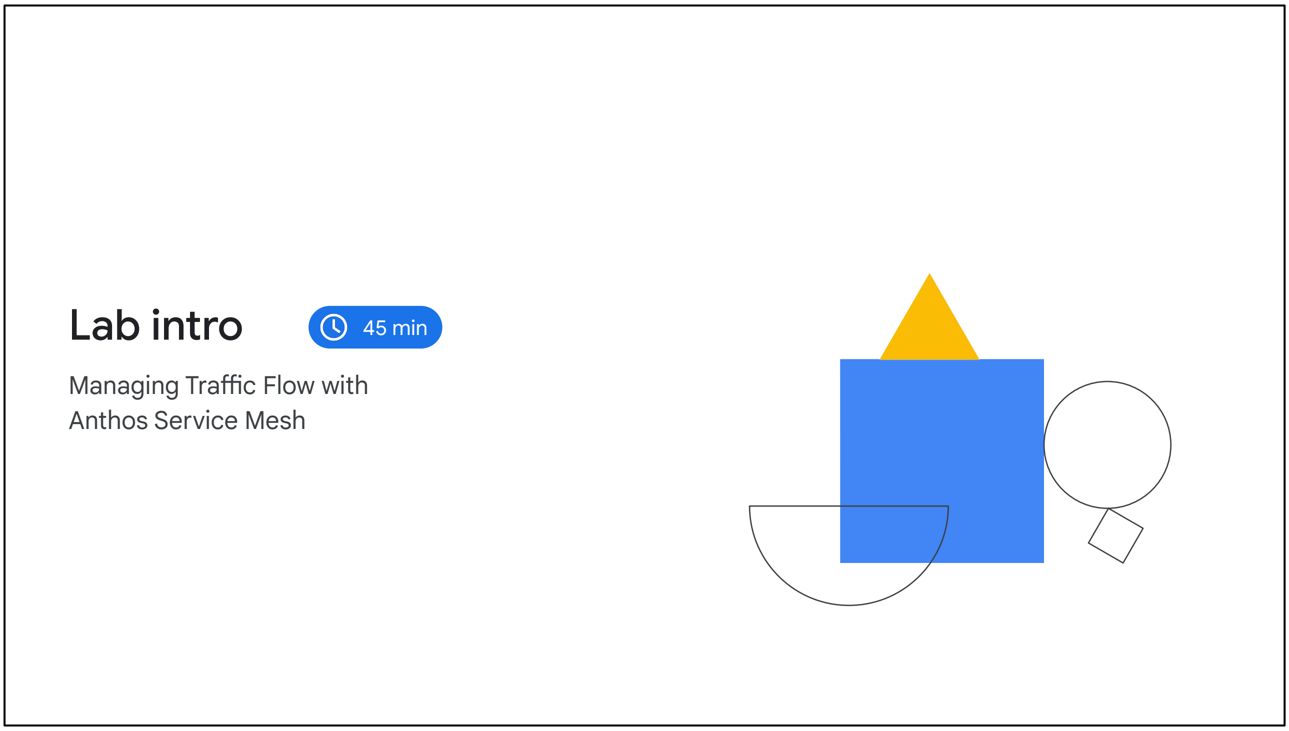
- Shadow deployment enables teams to bring changes to production with minimized risk:
  - Send a copy of live traffic to a mirrored service.
  - The mirrored service does not interact with end customers but can be tested with real traffic.
- Traffic analysis can be performed on your live traffic to:
  - Detect intrusions in the network.
  - Discover unauthorized traffic paths.
  - Find unsecure plain text traffic.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
     mirror:
      host: ratings
      subset: v2
    mirrorPercent: 100
```

Shadow deployments is a powerful concept that allows feature teams to bring changes to production with as little risk as possible. Mirroring sends a copy of live traffic to a mirrored service. The mirrored traffic happens out of band of the critical request path for the primary service.

Traffic analysis is another interesting use case. Send a copy of your live traffic to an analytics service to detect intrusions in the network, discover unauthorized traffic paths or find unsecure plain text traffic.

In this example, we are mirroring 100% of the traffic going to v1 to v2.



Lab intro

45 min

Managing Traffic Flow with Anthos Service Mesh

In this lab, you learn how to manage service discovery, traffic routing, and load balancing for your services without having to update code in your services.

You will perform the following tasks:

- Configure and use Istio Gateways
- Apply default destination rules, for all available versions
- Apply virtual services to route by default to only one version
- Route to a specific version of a service based on user identity
- Shift traffic gradually from one version of a microservice to another
- Use the Anthos Service Mesh dashboard to view routing to multiple versions
- Setup networking best practices such as retries, circuit breakers and timeouts