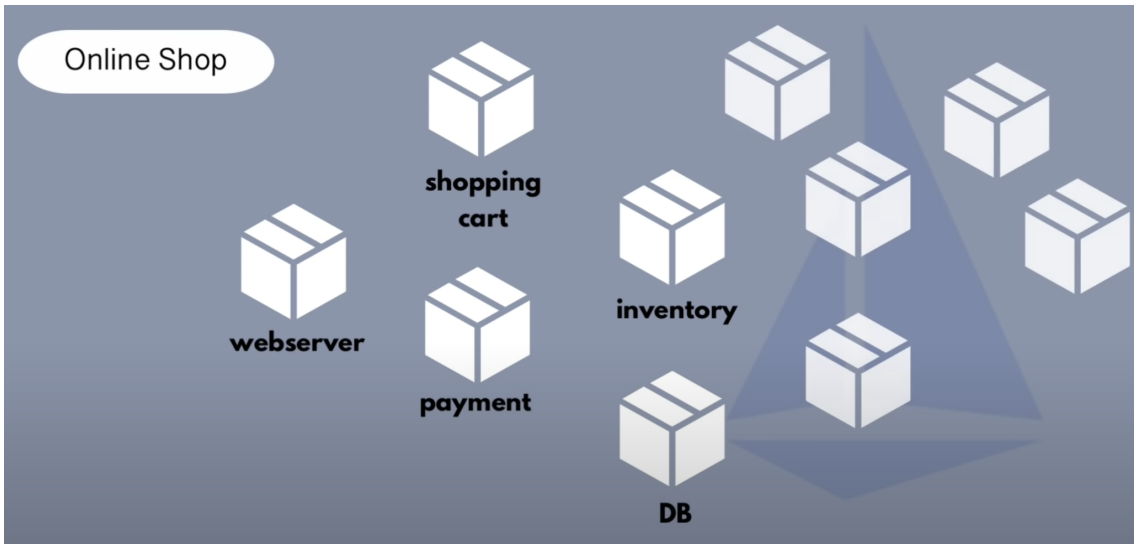
What is Istio? What is Service Mesh?

- Istio is a Service Mesh
- Service Mesh manages communication between microservices

What are the challenges that Istio takes care of?

- Challenges of a microservice architecture



  - Each microservice gets deployed in a single pod in kubernetes architecture
  - Each microservice needs the endpoint details configured so that they can communicate
  - When we add a new microservice, we need to add the endpoint of the new micro service to all other microservices. So we have that information in application deployment

Security:

  - We have firewall rules around kubernetes cluster or we can have proxy as entry point that gets the request first so that cluster can't be accessed directly. So we have security around the cluster.
  - However, once the request gets inside the cluster. The communication is insecure. The microservices talk to each other over http or some other insecure protocol.
  - Every service inside the cluster can talk to other service freely
  - From Security perspective, if a hacker gets inside the cluster. He can do anything as we don't have any additional security inside.
  - For small non critical application, it may be okay but in case of online banks or app that keeps personal user data, higher level of security is needed.
  - So in each service, additional configuration is needed to secure communication between services within the cluster.

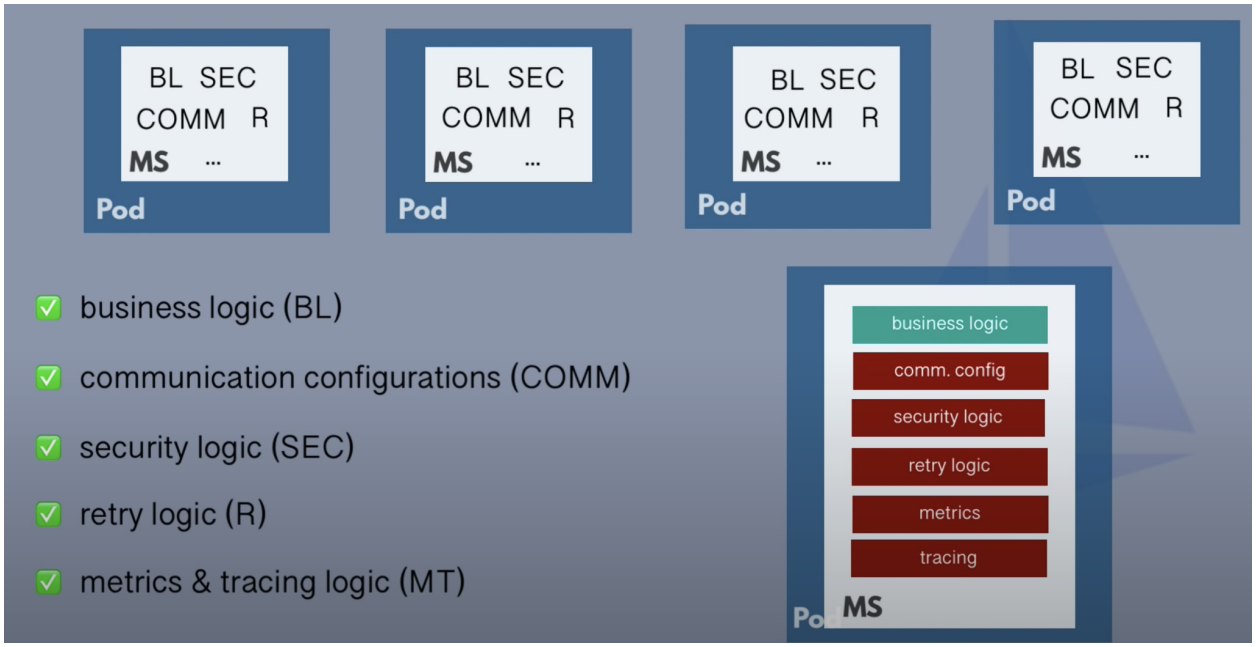
Retry Logic

  - We also need retry logic to make the application more robust

- If one microservice is unreachable or we loose connection a bit, we would retry the connection

Monitoring

- For each service, we would need to have metrics and tracing
- It will allow us to monitor how the application is performing. How much time each service is taking in fulfilling the request.
- We can add monitoring logic for prometheus using prometheus client library and collect tracing data using tracing library like zipkin for ex.



Conclusion:

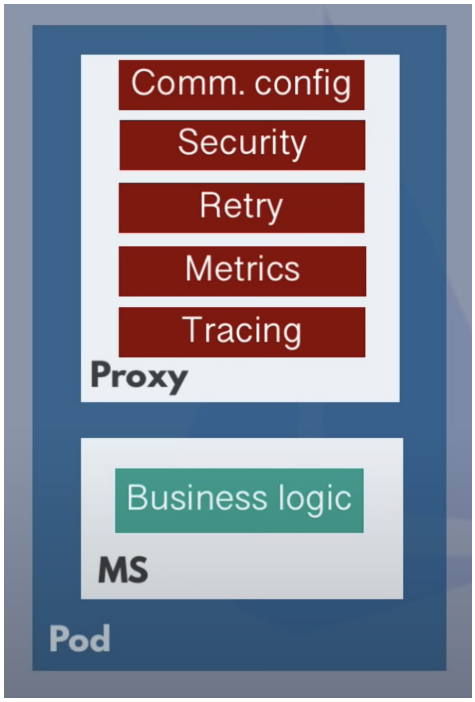
- These non business logic must be added to each service to address the challenges

Solution: Service mesh with Sidecar pattern

Sidecar Proxy:

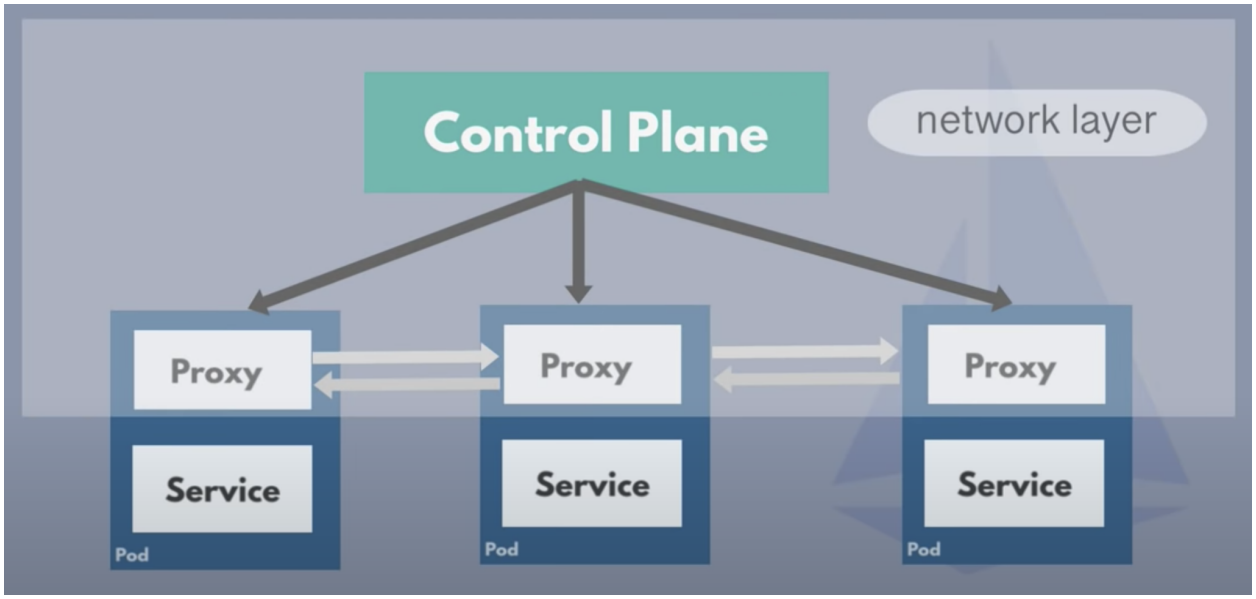
- Handles these networking logic
- Acts as a proxy
- Third party application
- Cluster operation can configure it easily

Note: We don't need to add Sidecar configuration in each microservice deployment file



Control Plane

- Control plane automatically injects the sidecar proxy in every microservice pod
- Now the microservices can talk to each other through those proxies
- The network layer for service to service communication consisting of control plane and the proxies is a service mesh.

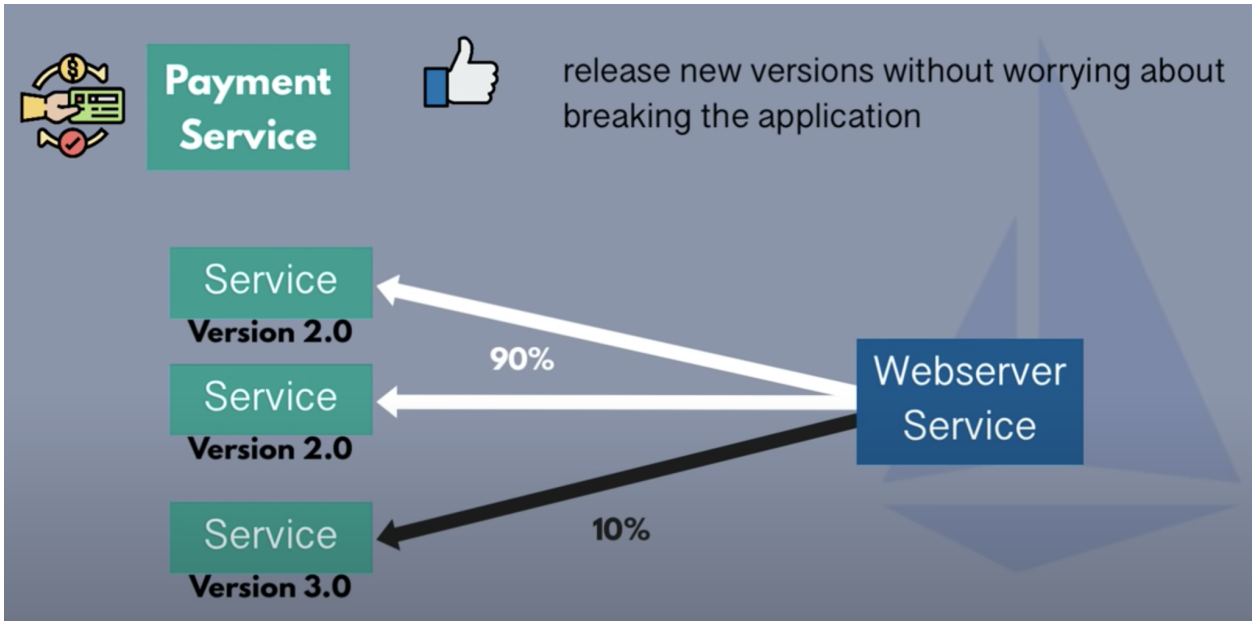


Service Mesh: Traffic Split

- This is one of the most important feature of Service Mesh

Traffic Split -

- When changes are made to any microservice, a new version is built, tested and deployed in the production environment.
- We can rely on the test to validate the new version. But the new version can have a bug that we cannot catch with the test.
- So instead of sending all the traffic to the new version. We can send only 1-10% of traffic to the new version and see if it really works.
- So using service mesh, we can easily route 90% of the traffic to old version and 10% of the traffic to the new version.



This is also known as Canary deployment.

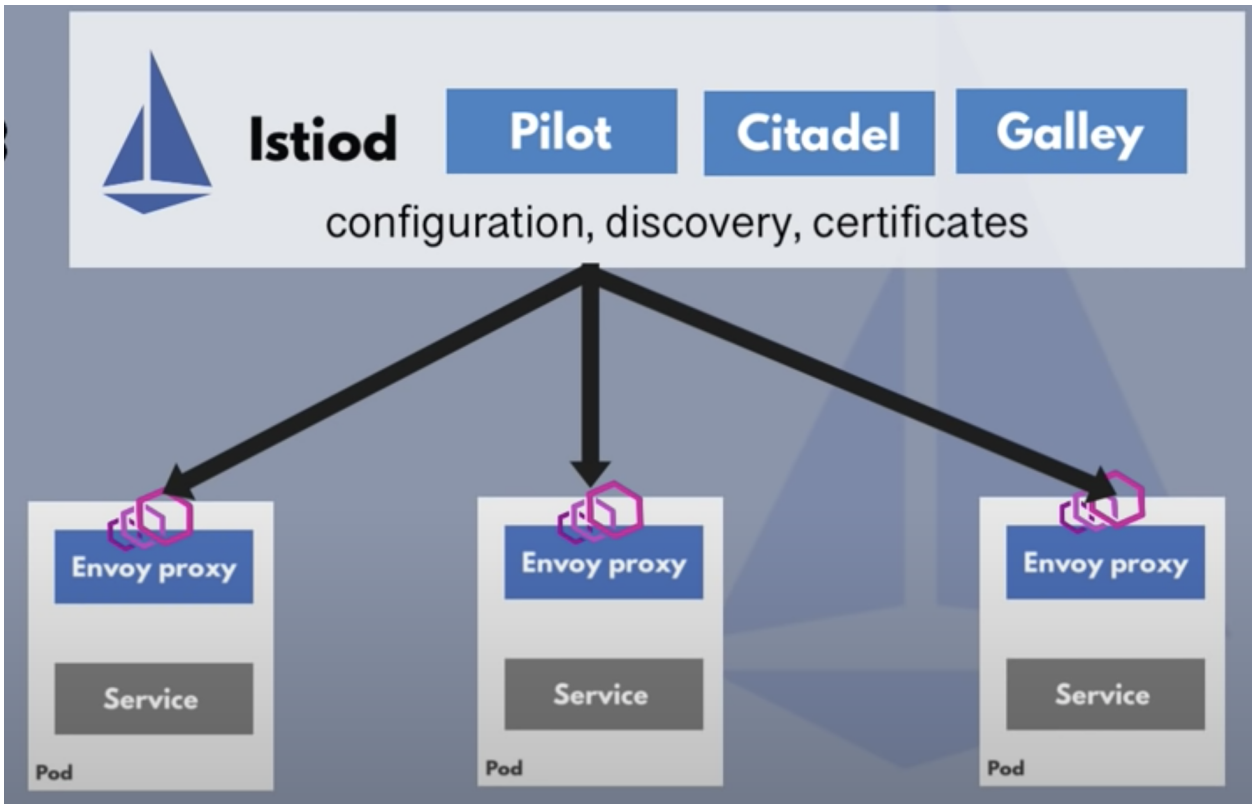
Note: Service mesh is a pattern or Paradigm and Istio is one of its implementation.

In the Istio architecture, the proxies are Envoy proxies which is an independent open source project that Istio as well as many other Service mesh implementations also use

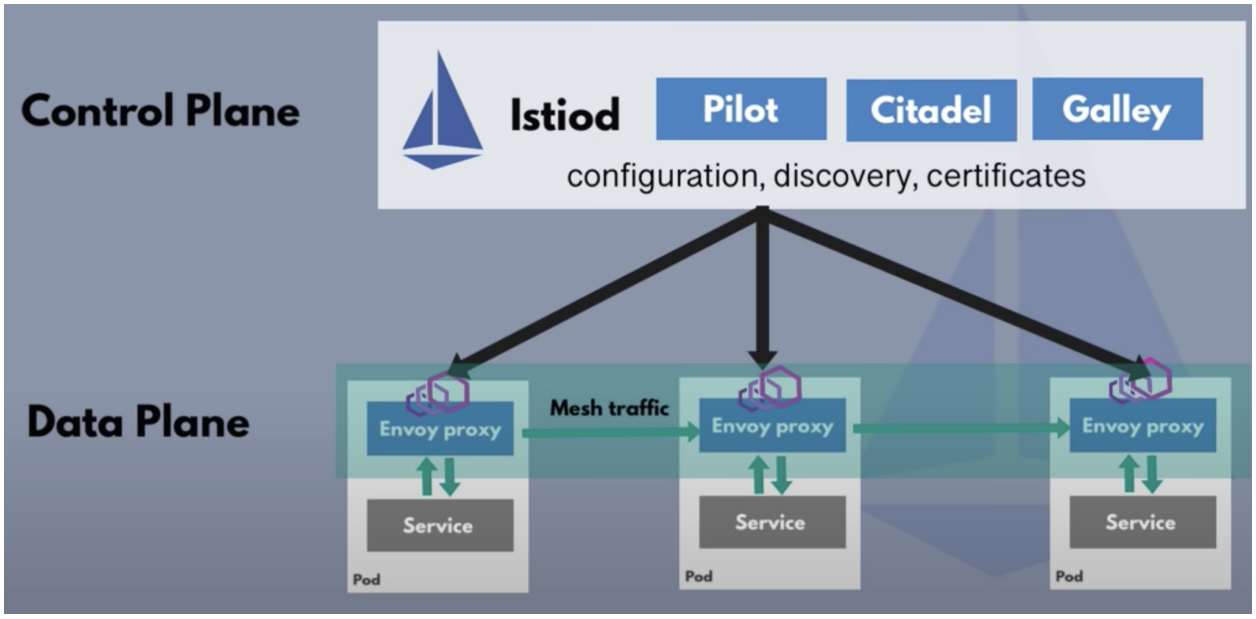
The control plane in Istio architecture is Istiod which manages and injects the Envoy proxies in each of the microservice pods

Before Istio V1.5, the control plane was a bundle of multiple components such as Citadel, Pilot, Galley, mixer and some other components.

However, In Istio v1.5, all the above components were combined into one single component called **Istiod.**



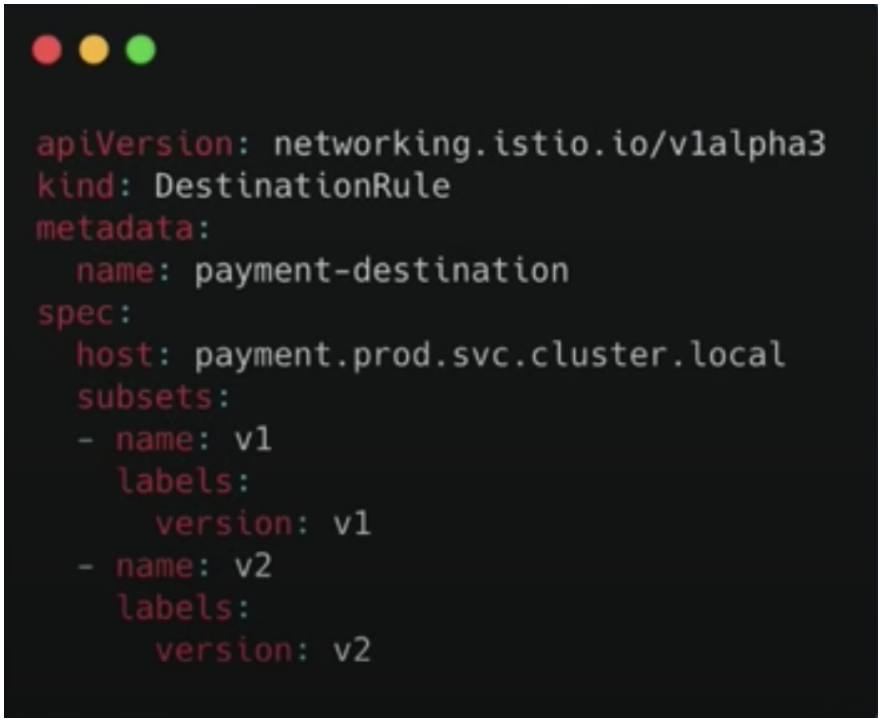
The Control Plane manages the Data Plane which is a group of Envoy Proxies.



How to configure Istio?

- We don't have to adjust Deployment and Services K8s YAML files
- Istio configuration is separate from application configuration
- Istio is configured with Kubernetes YAML files
    - Istio uses Kubernetes CustomResourceDefinition (CRD)
        - Extending the Kubernetes API
        - Custom k8s component/object for e.g third party technologies (like Istio, Prometheus, etc) using the same k8s yaml files and apply them using Kubectl withour having to learn a technology specific configuration language and adjusting that configuration inside the Istio for ex.
- So using few Istio CRDs, we can configure different traffic routing rules between our microservices like
    - which services can talk to each other
    - Traffic split
    - Retry rules
    - Other network configurations
    -
- There are two main CRDs for configuring Service to Service communication
    - Virtual Service - How to route traffic to a given destination
    - Destination Rule - Configure what happens to traffic for that destination

CRD:



```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: payment-destination
spec:
  host: payment.prod.svc.cluster.local
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

Stepwise:

- We create CRDs
- Istiod (control plane) converts these high level routing rules into envoy specific configuration
- Configuration is then propagated into Proxy Sidecars

Note - We don't configure Proxies, we configure Istiod (control plane). The control plane will then push that configuration out to each individual envoy proxies.
The proxies can now communicate with each other by applying this configuration that we define without having to go back to the Istio Control plane.
So they can independently talk to each other because they have all the logic and configuration they need without talking to the control plane.

Istio Features:

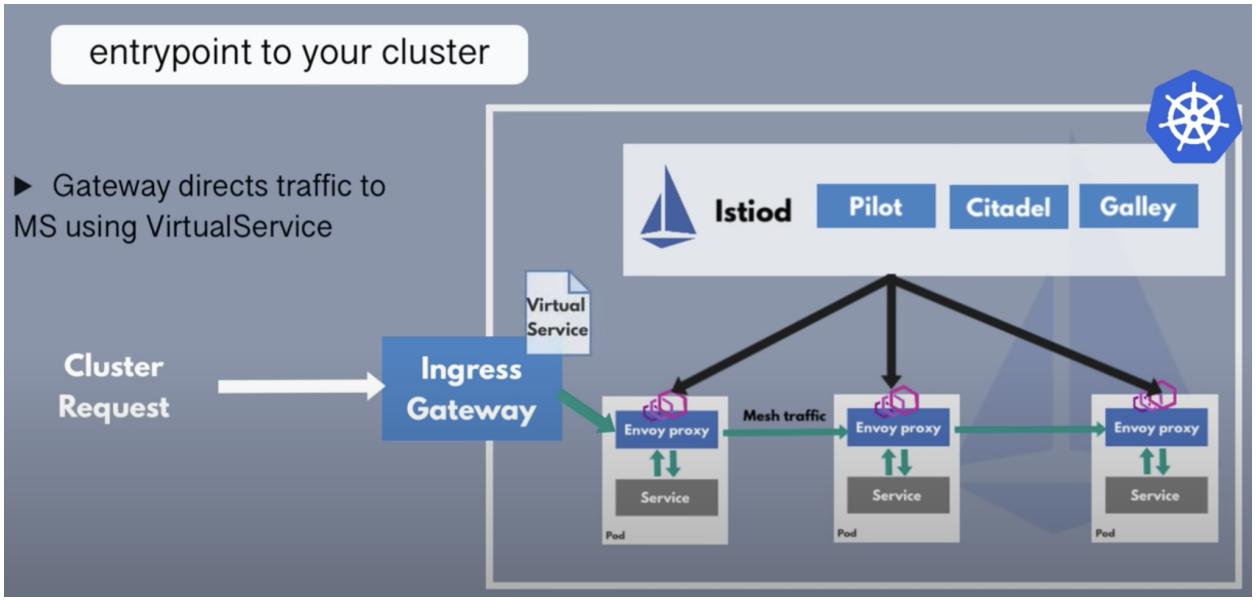
- ☐ Service Discovery - Internal registry for the Services and their endpoints
  - ☐ In addition to configuring proxies, Istiod also has a central registry for all the microservices. So instead of statically configuring the endpoints for each microservice.
  - ☐ When a new microservice is deployed, it will automatically gets registered in the service registry without the need of any additional configuration from our side. Istio automatically detects the services and their endpoints in the cluster.
  - ☐ Using this service registry, envoy proxy can now query the endpoints to send traffic to the relevant services

`

- Security
  - Istiod also acts as a CA (certificate authority) and generate certificate for all the microservices in the cluster to allow secured TLS communication between proxies of those microservices.
- Metrics and tracing
  - Istiod gets metrics and tracing data from the envoy proxies that it gathers that can be later consumed by monitoring server like prometheus or tracing server etc, to have out of the box metrics and tracing data for your whole microservice application.

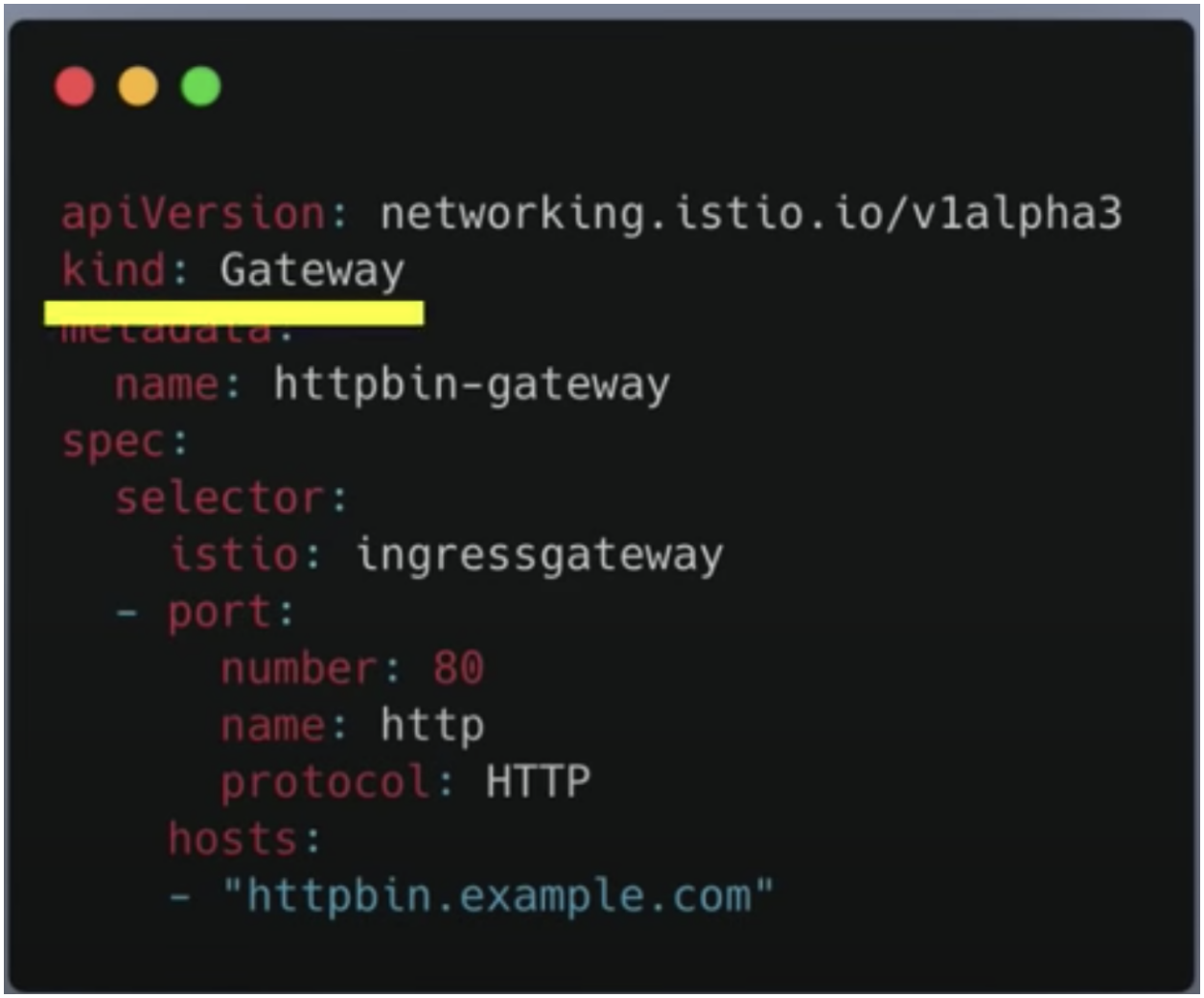
Istio Gateway:

- Istio Ingress Gateway - Its an entry point to our k8s cluster. Its an alternative to nginx ingress controller.
  - Istio Gateway runs as a pod in our cluster and acts as a load balancer by accepting incoming traffic in our cluster and gateway will then direct the traffic to one of our microservices inside the cluster using a virtual service component.

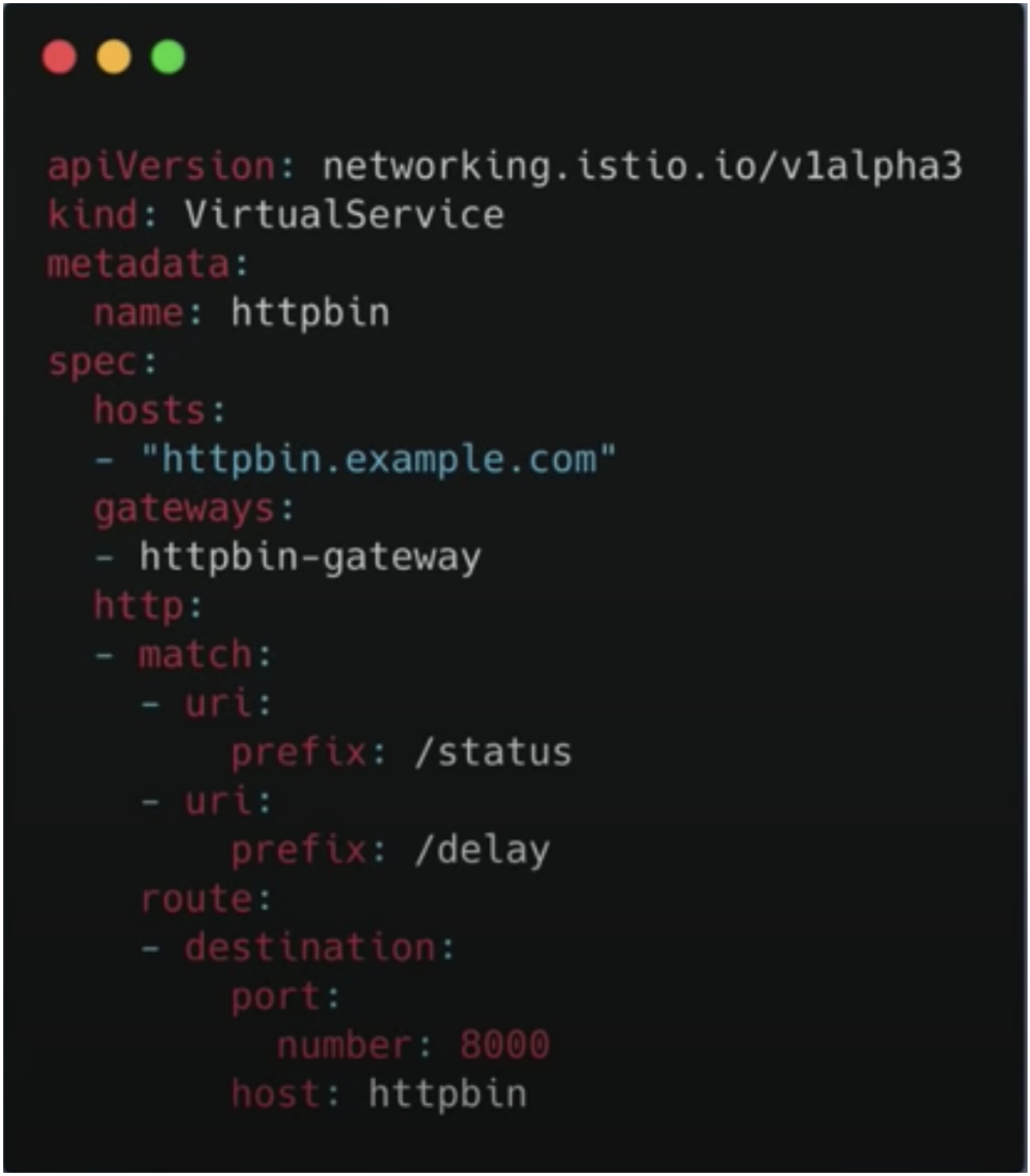


Note: We can configure the Istio gateway using a gateway CRD.

Creating an Istio Gateway:



Configure routes for traffic entering via the gateway:


apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: httpbin
spec:
hosts:
- "httpbin.example.com"
gateways:
- httpbin-gateway
http:
- match:
- uri:
prefix: /status
- uri:
prefix: /delay
route:
- destination:
port:
number: 8000
host: httpbin

Traffic Flow with Istio:

Traffic flow stepwise:

- The request will first hit the geteway because it is the entrypoint of the cluster
- Gateway then evaluates the virtual service rules about how to route the traffic and will send it to Web server microservice
- Finally that request will reach the envoy proxy inside our web server microservice.
- The envoy proxy then evaluates the request and then forward it to actually web server container within the same pod using localhost.
- The web server then initiates another request to some any other microservice for ex. Payment service
- The request will move from Web server container to web server proxy which will then by applying virtual service rule as well as destination rule and maybe some other configuration will communicate with the envoy proxy Payment microservice using mutual TLS.
- The same will for the communication between the payment service and database server and all the way back to the ui where response will be sent.
- During this all request flow, the proxies will gather all the metrics and tracing information about the request and send it back to the control plane, so we automatically have monitoring for our application.