Challenges caused by Microservices Architecture:

1. Service Discovery- How does a service discover the network address of other services?
2. Load Balancing- Given that each service is scaled horizontally, the problem of load-balancing was no longer an ingress-only problem.
3. Interservice Communication - How to deal with situations where calls to other services fail or take an inordinately long time? An increasing portion of developers' codebases had to be dedicated to handling the failures and dealing with long response times, by sprinkling in retries and network timeouts.
4. Resilience - Developers had to learn (perhaps the hard way) to build distributed applications that are resilient and that prevent cascading failures.
5. Security
6. Traffic Management- The ability to route requests flexibly to different services under different conditions started becoming a necessity.
7. E2E Testing- End-to-end testing became more difficult.
8. TroubleShooting- Stack traces no longer provided complete context for diagnosing an issue. Logs were now distributed. How does a developer diagnose issues that span multiple microservices?

In terms of implementation, Istio's main concerns are, therefore, solving the following problems:

1. Ensuring that each time a workload is deployed, an Envoy sidecar is deployed alongside it.
2. Ensuring traffic into and out of the application is transparently diverted through the proxy.
3. Assigning each workload a cryptographic identity as the basis for a more secure computing environment.
4. Configuring the proxies with all the information they need to handle incoming and outgoing traffic.

Side Car Injection Types:

1. Manual - Istio has a command-line interface (CLI) named **istioctl** with the subcommand **kube-inject**. The subcommand processes the original deployment manifest to produce a modified manifest with the sidecar container specification added to the pod (or pod template) specification. The modified output can then be applied to a Kubernetes cluster with the **kubectl apply -f** command.With manual injection, the process of altering the manifests is explicit.
2. Automatic- With automatic injection, the bundling of the sidecar is made transparent to the user.This process relies on a Kubernetes feature known as [Mutating Admission Webhooks](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#mutatingadmissionwebhook), a mechanism that allows for the registration of a webhook that can intercept the application of a deployment manifest and mutate it before the final, modified specification is applied to the Kubernetes cluster.The webhook is triggered according to a simple convention, where the application of the label **istio-injection=enabled** to a Kubernetes namespace governs whether the webhook should modify any deployment or pod resource applied to that namespace to include the sidecar.

**Routing Application Traffic Through the Sidecar**

With the sidecar deployed, the next problem is ensuring that the proxy transparently captures the traffic. The outbound traffic should be diverted from its original destination to the proxy, and inbound traffic should arrive at the proxy before the application has a chance to handle the incoming request.

**This is performed by applying**[**iptables**](https://en.wikipedia.org/wiki/Iptables)**rules.**In addition to the Envoy sidecar, the sidecar injection process injects a Kubernetes [init container](https://kubernetes.io/docs/concepts/workloads/pods/init-containers/" \t "_blank). This **init-container** is a process that applies these **iptables** rules before the Pod containers are started.

Today Istio provides two alternative mechanisms for configuring a Pod to allow Envoy to intercept requests. The first is the original **iptables** method, and the second uses a [Kubernetes CNI plugin](https://istio.io/latest/docs/setup/additional-setup/cni/).

**Installation Configuration Profiles**

Istio service mesh has numerous configuration settings that operators can update before installing Istio. To group the most common configuration settings into a higher-level abstraction, Istio uses the concept of configuration profiles.

The configuration profiles contain different configuration settings for the control plane as well as the data plane of Istio. The installation configuration profiles are expressed through the Istio Operator API and the IstioOperator resource.

Six configuration profiles are currently available, as shown in the list below.

1. Default - The default profile is meant for production deployments and deployments of primary clusters in multi-cluster scenarios. It deploys the control plane and ingress gateway.
2. Demo - The demo profile is intended for demonstration deployments. It deploys the control plane and ingress and egress gateways and has a high level of tracing and access logging enabled.
3. Minimal - The minimal profile is equivalent to the default profile but without the ingress gateway. It deploys the control plane.
4. External- The external profile is used for configuring remote clusters in a multi-cluster scenario. It does not deploy any components.
5. Empty - The empty profile is used as a base for custom configuration. It does not deploy any components.
6. Preview - The preview profile contains experimental features. It deploys the control plane and ingress gateway.

To install Istio using the Istio CLI, we can use the **--set** flag and specify the profile like this:

**istioctl install --set profile=demo**

**Using Istio Operator API**

The Istio Operator API and the [IstioOperator resource](https://istio.io/latest/docs/reference/config/istio.operator.v1alpha1/" \t "_blank) allow us to install and configure Istio on a Kubernetes cluster. At a high level, we can separate the configuration in the IstioOperator resource into the following sections:

1. **Global**  
   The global section allows us to configure the profile name, root Docker image path, image tags, namespace, revision, and so on.
2. **Mesh configuration** **(meshConfig)**  
   The **meshConfig** section includes the configuration of the control plane components. For example, in this section, we can configure access log format, log encoding, set up default proxy configuration, discovery selectors, trust domains, and more.
3. **Component configuration (components)**  
   The **components** section allows us to enable or disable individual components, install additional components (multiple ingress or egress gateways, for example), and configure Kubernetes resource settings for individual components. For example, for each component (e.g., pilot, ingress, or egress gateways), we can configure the CPU and memory requests and limits, annotations, labels, replica counts, and other settings in the Kubernetes resources.

Within the IstioOperator resource, we specify the desired state of Istio components. We can apply or deploy the resource to the Kubernetes cluster using the Istio CLI and the Code in a paragraph or file content: **install** command.

Once we have created the IstioOperator resource, we can install it on the cluster using the **install** command:

**istioctl install -f my-operator-resource.yaml**

**Using Helm**

[Helm](https://helm.sh/) is a Kubernetes package manager that helps install and upgrade complex applications on Kubernetes. A fundamental building block of Helm is a Helm Chart, a collection of YAML manifests.

When using Helm, there are three different Helm charts we need to be aware of, listed in the order we would install them:

1. **Base chart (istio/base)**  
   The **base** chart includes cluster-wide resources such as the validating webhook configuration resource, service accounts, cluster roles and bindings, and other resources to ensure backward compatibility.
2. **Istiod chart (istio/istiod)**  
   The **istiod** chart contains Istio’s control plane installation. It includes the **istiod** deployment and service, mutating webhook configuration (facilitates automatic sidecar injection into deployments), and other resources for the control plane.
3. **Gateway chart (istio/gateway)**  
   The **gateway** chart is used for deploying ingress and egress gateways to the cluster. It includes the service and deployment resources and other supporting resources.

Before installing the charts, we need to manually create the root namespace (i.e., **istio-system**) and use the **helm install** command to install the individual charts. Typically, we install the **base** and **istiod** charts to the **istio-system** namespace and **gateway** charts into separate namespaces.

Here is how we could install the **istiod** chart, for example:

**helm install istiod istio/istiod -n istio-system**

The first parameter in the above command is the release name, followed by the chart name.

To check on the installation progress, we can pass the release name (e.g. **istiod**) to the **status** command:

**helm status istiod -n istio-system**

The end-to-end request-response flow is known as a *trace*. Each component of a trace, such as a single call from one service to another, is called a *span*. Traces have unique IDs, and so do spans. All spans that are part of the same trace bear the same trace ID.

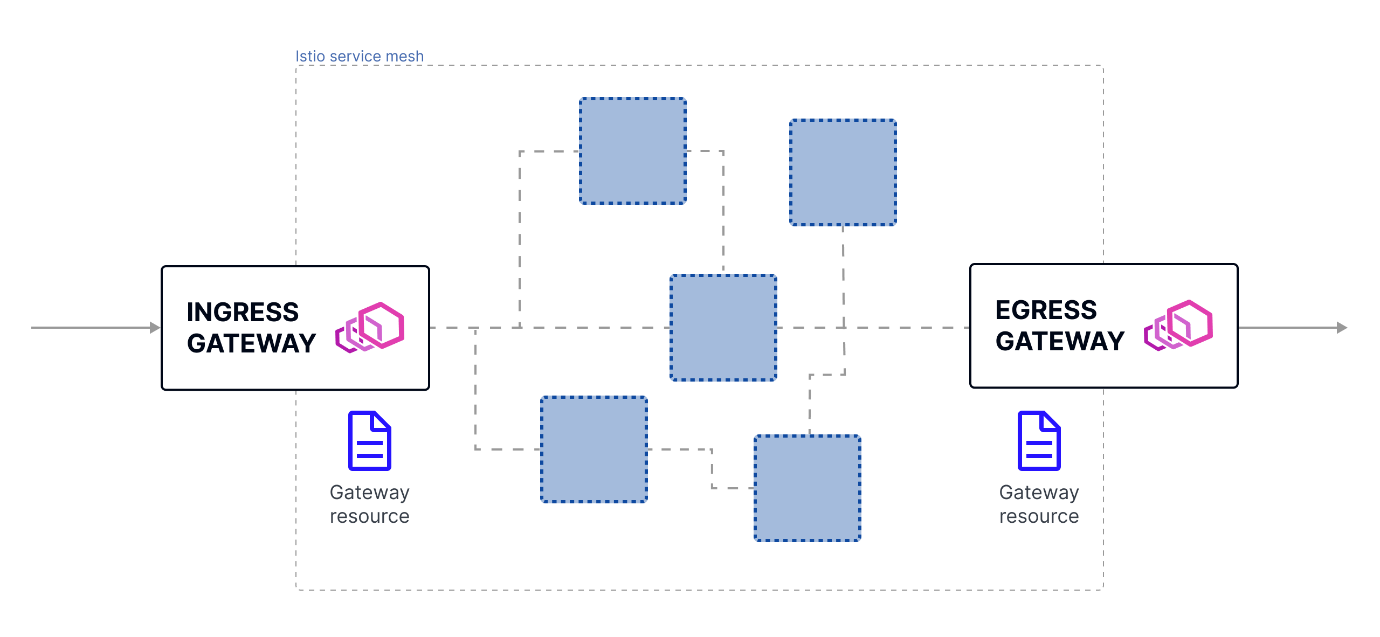
The IDs are propagated across the calls between services in HTTP headers whose names begin with **x-b3** and are known as *B3 trace headers* (see [B3 Propagation](https://github.com/openzipkin/b3-propagation)).

When Envoy sidecars receive the initial request that does not contain a B3 header and realize that this span represents the beginning of a new trace, they assign the request a new trace ID.

However, the propagation of these headers onto other services cannot be performed automatically by Envoy, and so developers must ensure that they propagate these headers in upstream calls to other services (see [Istio/FAQ](https://istio.io/latest/about/faq/#how-to-support-tracing)). This task is often easily accomplished by including a tracing client library as a dependency to the services.

Both gateways are Kubernetes deployments that run an instance of the Envoy proxy, and they operate as load balancers at the edge of the mesh. The ingress gateway receives inbound connections, while the egress gateway receives connections going out of the cluster.

Using the ingress gateway, we can apply route rules to the inbound traffic entering the cluster. As part of the ingress gateway, a Kubernetes service of type LoadBalancer is deployed, giving us an external IP address.



**Ingress and egress gateways are instances of Envoy, running at the edge of the cluster**

We can configure both gateways using a Gateway resource. The Gateway resource describes the exposed ports, protocols, SNI (Server Name Indication) configuration for the load balancer, etc.

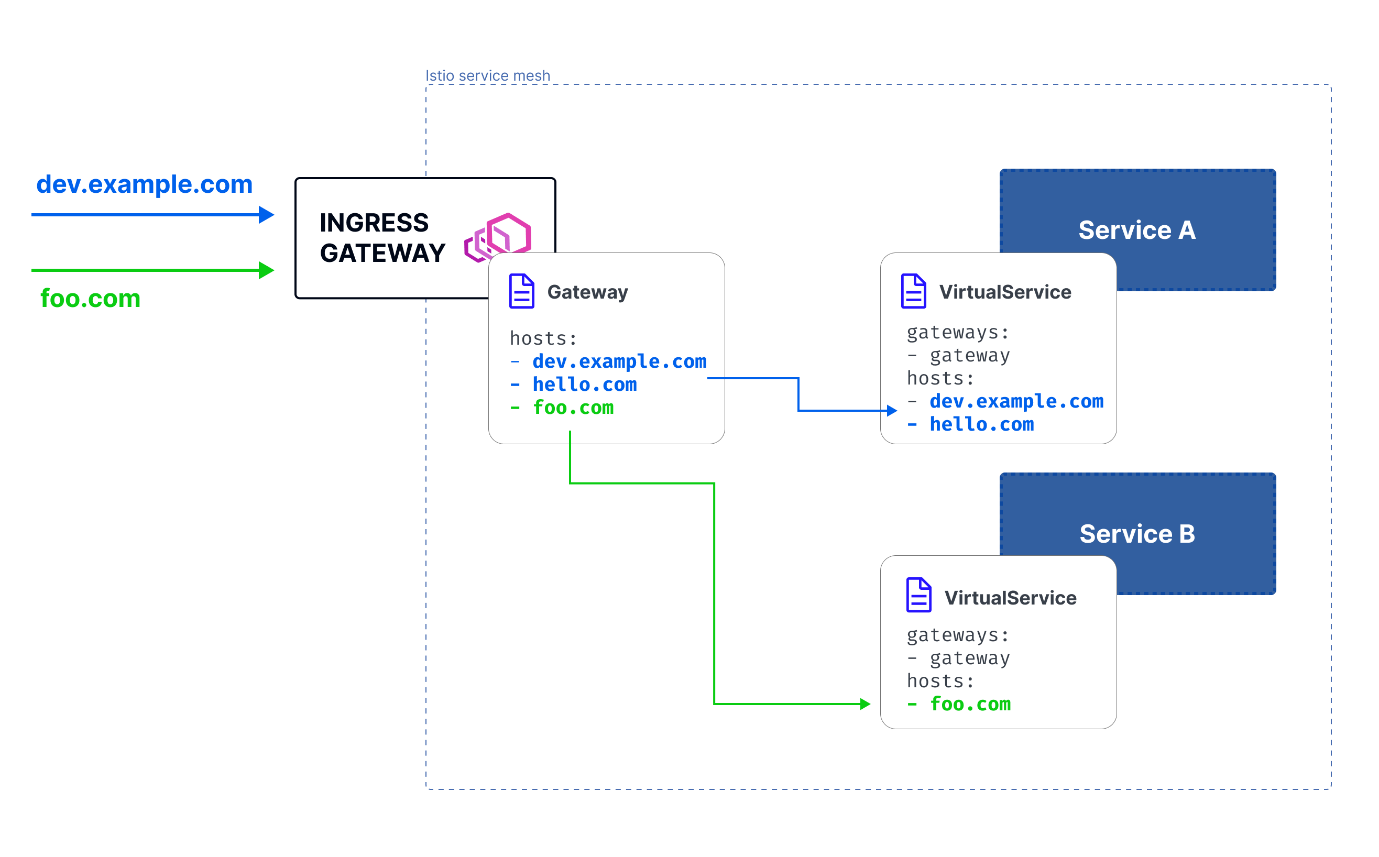
Under the covers, the Gateway resource controls how the Envoy proxy listens on the network interface and which certificates it presents.

Here's an example of a Gateway resource:

**apiVersion: networking.istio.io/v1beta1  
kind: Gateway  
metadata:  
  name: my-gateway  
  namespace: default  
spec:  
  selector:  
    istio: ingressgateway  
  servers:  
  - port:  
      number: 80  
      name: http  
      protocol: HTTP  
    hosts:  
    - dev.example.com  
    - test.example.com**

The above Gateway resource sets up the Envoy proxy as a load balancer exposing port 80 for ingress. The gateway configuration gets applied to the Istio ingress gateway proxy, which we deployed to the **istio-system** namespace and has the label **istio: ingressgateway** set. The **hosts** field acts as a filter and will let through only traffic destined for **dev.example.com** and **test.example.com**.

To control and forward the traffic to an actual Kubernetes service running inside the cluster, we have to configure a VirtualService with matching hostnames (**dev.example.com** and **test.example.com**, for example) and then attach the Gateway resource to it.

****

**Inbound traffic is matched by the hosts in the Gateway and VirtualService resources**

The Ingress gateway we deployed as part of the **demo** Istio installation created a Kubernetes service with the LoadBalancer type that gets an external IP assigned to it, for example:

**kubectl get svc -n istio-system**

**NAME                   TYPE           CLUSTER-IP     EXTERNAL-IP PORT(S)                                                                         AGE  
istio-egressgateway    ClusterIP      10.0.146.214   <none>           80/TCP,443/TCP,15443/TCP                                                     7m56s  
istio-ingressgateway   LoadBalancer   10.0.98.7      XX.XXX.XXX.XXX   15021:31395/TCP,80:32542/TCP,443:31347/TCP,31400:32663/TCP,15443:31525/TCP.  7m56s  
istiod                 ClusterIP      10.0.66.251    <none>           15010/TCP,15012/TCP,443/TCP,15014/TCP,853/TCP                                 8m6s**

***NOTE:****How the LoadBalancer Kubernetes service type works depends on how and where we run the Kubernetes cluster. For a cloud-managed cluster (GCP, AWS, Azure, etc.), a load balancer resource gets provisioned in your cloud account, and the Kubernetes LoadBalancer service will get an external IP address assigned to it. Suppose we are using Minikube or Docker Desktop. In that case, the external IP address will either be set to* **localhost** *(Docker Desktop) or, if we are using Minikube, it will remain pending, and we will have to use the* **minikube tunnel** *command to get an IP address.*

In addition to the ingress gateway, we can deploy an egress gateway to control and filter traffic leaving our mesh.

We can use the same Gateway resource to configure the egress gateway like we configured the ingress gateway. The egress gateway allows us to centralize all outgoing traffic, logging, and authorization.

**Traffic Routing in Istio**

Istio features a couple of resources we can use to configure how traffic is routed within the mesh. We have already mentioned the VirtualService and the Gateway resource in the Gateway section.

We can use the [VirtualService resource](https://istio.io/latest/docs/reference/config/networking/virtual-service/" \t "_blank) to configure routing rules for services within the Istio service mesh. For example, in the VirtualService resource, we match the incoming traffic based on the request properties and then route the traffic to one or more destinations. For example, once we match the traffic, we can split it by weight, inject failures and/or delays, mirror the traffic, and so on.

The [DestinationRule resource](https://istio.io/latest/docs/reference/config/networking/destination-rule/" \t "_blank) contains the rules applied after routing decisions (from the VirtualService) have already been made. With the DestinationRule, we can configure how to reach the target service. For example, we can configure outlier detection, load balancer settings, connection pool settings, and TLS settings for the destination service.

The last resource we should mention is the [ServiceEntry](https://istio.io/latest/docs/reference/config/networking/service-entry/" \t "_blank). This resource allows us to take an external service or an API and make it appear as part of the mesh. The resource adds the external service to the internal service registry, allowing us to use Istio features such as traffic routing, failure injection, and others against external services.

**Service Resiliency**

Resiliency is the ability to provide and maintain an acceptable level of service in the face of faults and challenges to regular operation. It's not about avoiding failures. It's responding to them, so there's no downtime or data loss. The goal of resiliency is to return the service to a fully functioning state after a failure occurs.

A crucial element in making services available is using **timeouts** and **retry policies** when making service requests. We can configure both in the VirtualService resource.

Using the **timeout** field, we can define a timeout for HTTP requests. If the request takes longer than the value specified in the **timeout** field, the Envoy proxy will drop the request and mark it as timed out (return an **HTTP 408** to the application). The connections remain open unless outlier detection is triggered.

Here's an example of setting a timeout for a route:

**...  
- route:  
  - destination:  
      host: customers.default.svc.cluster.local  
      subset: v1  
  timeout: 10s  
...**

In addition to timeouts, we can configure a more granular retry policy. We can control the number of retries for a given request, the timeout per try, and the specific conditions that should trigger a retry. Both retries and timeouts happen on the client side.

For example, we can only retry the requests if the upstream server returns any **5xx** response code, retry only on gateway errors (HTTP 502, 503, or 504), or even specify the retriable status codes in the request headers. When Envoy retries a failed request, the endpoint that initially failed and caused the retry is no longer included in the load balancing pool.

Let's say the Kubernetes service has three endpoints (Pods), and one of them fails with a retriable error code. When Envoy retries the request, it won't resend the request to the original endpoint anymore. Instead, it will send the request to one of the two endpoints that have not failed.

Here's an example of how to set a retry policy for a particular route:

**...  
- route:  
  - destination:  
      host: customers.default.svc.cluster.local  
      subset: v1  
  retries:  
    attempts: 10  
    perTryTimeout: 2s  
    retryOn: connect-failure,reset  
...**

The above retry policy will attempt to retry any request that fails with a connect timeout (**connect-failure**) or if the server does not respond at all (**reset**).

We set the per-try attempt timeout to 2 seconds and the number of attempts to 10. Note that if we set both retries and timeouts, the timeout value will be the most the request will wait. If we had a 10-second timeout specified in the above example, we would only ever wait 10 seconds maximum, even if there are still attempts left in the retry policy.

**Circuit Breaking with Outlier Detection**

Another pattern for creating resiliency applications is circuit breaking. It allows us to write services to limit the impact of failures, latency spikes, and other network issues.

Outlier detection is an implementation of a circuit breaker, and it’s a form of passive health checking. It’s called passive because Envoy isn’t actively sending any requests to determine the health of the endpoints. Instead, Envoy observes the performance of different pods to determine if they are healthy or not. If the pods are deemed unhealthy, they are removed or ejected from the healthy load balancing pool.

The pods’ health is assessed through consecutive failures, temporal success rate, latency, and so on.

Outlier detection in Istio is configured in the DestinationRule resource. Here’s a snippet that configures outlier detection:

**apiVersion: networking.istio.io/v1beta1  
kind: DestinationRule  
metadata:  
  name: customers  
spec:  
  host: customers  
  trafficPolicy:  
    connectionPool:  
      tcp:  
        maxConnections: 1  
      http:  
        http1MaxPendingRequests: 1  
        maxRequestsPerConnection: 1  
    outlierDetection:  
      consecutive5xxErrors: 1  
      interval: 1s  
      baseEjectionTime: 3m  
      maxEjectionPercent: 100**

The above snippet defines thresholds for TCP and HTTP connections in the **connectionPool** field. The circuit breaker trips if we exceed 1 TCP connection or one pending HTTP request, or more than one request per connection. When the circuit breaker trips, the service will start responding with HTTP 503 (service unavailable) responses.

In addition to the connection pool settings, we have also configured the outlier detection. When a pod is determined to be an outlier (i.e., it exceeds the configured threshold, for example, consecutive 5xx errors), Envoy checks whether it needs to be ejected from the healthy load balancing pool of pods. The **maxEjectionPercent** field is used here, and it specifies the maximum percentage of pods that can be ejected. So, when the thresholds in the connection pool are exceeded, and we get more than 1 consecutive 5xx error (the **consecutive5xxErrors** field) and the pod can be ejected, the outlier detection will eject a pod.

Each pod gets ejected for a predetermined amount of time. We can configure the ejection time using the **baseEjectionTime** value. This value is multiplied by the number of times the pod has been ejected in a row. If the pod continues to fail, it gets ejected for longer and longer periods.

Envoy checks the health of each pod at an **interval** specified in the interval field. For every check, the endpoint is healthy, the ejection multiplier gets decremented. After the ejection time passes, the pod returns to the healthy load balancing pool.

**Failure Injection**

Another feature to help us with service resiliency is **fault injection**. We can apply the fault injection policies on HTTP traffic and specify one or more faults to inject when forwarding the request to the destination.

There are two types of fault injection. We can **delay** the requests before forwarding and emulate a slow network or overloaded service, and we can **abort** the HTTP request and return a specific HTTP error code to the caller. With the abort, we can simulate a faulty upstream service.

Here's an example that aborts HTTP requests and returns HTTP 404, for 30% of incoming requests:

**- route:  
  - destination:  
      host: customers.default.svc.cluster.local  
      subset: v1  
  fault:  
    abort:  
      percentage:  
        value: 30  
      httpStatus: 404**

The Envoy proxy will abort all requests if we don't specify the percentage. Note that the fault injection affects services that use that VirtualService. It does not affect all consumers of the service. For example, if we configure a fault for a specific host name (e.g. **example.com**), then any request using the hostname **hello.com** will not be subjected to the fault injection.

Similarly, we can apply an optional delay to the requests using the **fixedDelay** field:

**- route:  
  - destination:  
      host: customers.default.svc.cluster.local  
      subset: v1  
  fault:  
    delay:  
      percentage:  
        value: 5  
      fixedDelay: 3s**

The above setting applies a 3-second delay to 5% of incoming requests.

Note that the fault injection will not trigger any retry policies we have set on the routes. For example, if we inject an HTTP 500 error, the retry policy configured to retry on the HTTP 500 will not be triggered.

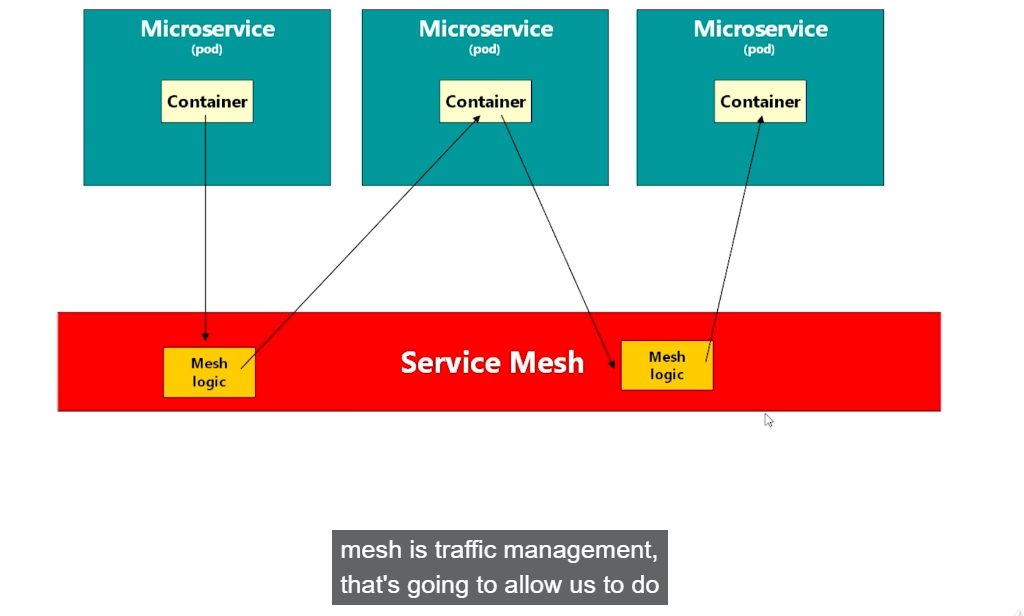
Istio is a service mesh.

CRD – Custom Resource Definitions. Extension to K8s api

Service mesh is a layer sitting underneath all of the pods in our system.

Telemetry : process of gathering metrics

Tracing: Tracking network calls



**Istio injects add its own container inside the pod named proxy**

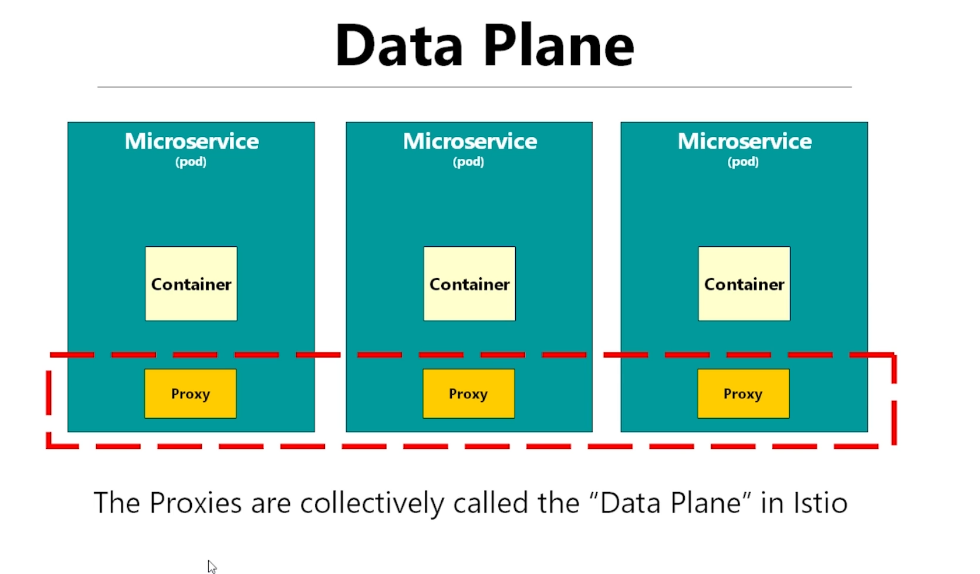
A diagram of a process

Description automatically generated

Istio-system: Istio pods running in their own namespaces

Istiod- Istio daemon. IT is earlier called pilot.

IStiod – performs telemetry



A screenshot of a computer

Description automatically generated

Istio Proxy – are sidecar container injected using istiod pod inside istio-system namespace in the application pod.

1. We need to setup a label on a namespace to automatically inject sidecar container inside the application pod.
2. Label name: istio-injection=enabled on default namespace

Envoy is a proxy for cluster based applications.

Kiali UI – Visualising Pods and how they are connected together