



Search Medium



To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy.



Published in Namely Labs



Shray Kumar

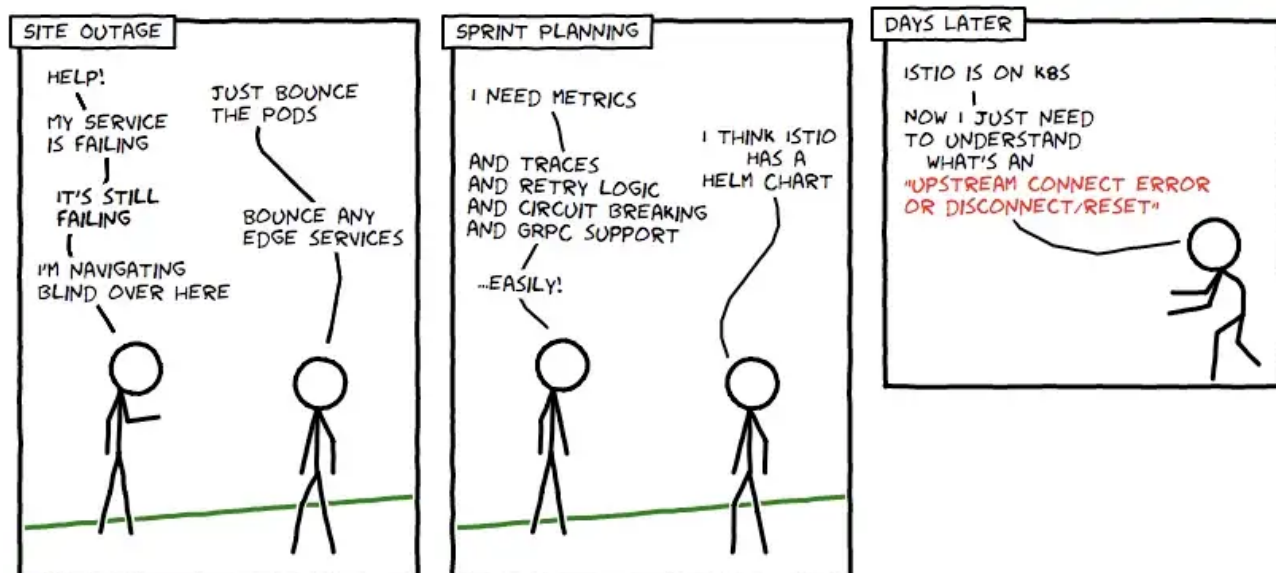
Follow

Jan 17, 2019 · 10 min read · [Listen](#)

Save



A Crash Course For Running Istio



Istio Service Mesh

At Namely we've been running with Istio for a year now. Yes, that's pretty much when it first came out. We had a major performance regression with a Kubernetes cluster, we wanted distributed tracing, and used Istio to bootstrap Jaeger to investigate. We immediately saw the potential of a service mesh as it relates to our infrastructure and decided to make an investment in the tool.

It hasn't always been the smoothest ride, but we have learned a *ton* about how it works and how to operate it. This post — the start of series — hopes to explain how Istio integrates with Kubernetes and some operational observations we've made along the way. We'll go into some technical details, but mostly keep it high level, and with additional posts to come.

What is Istio?

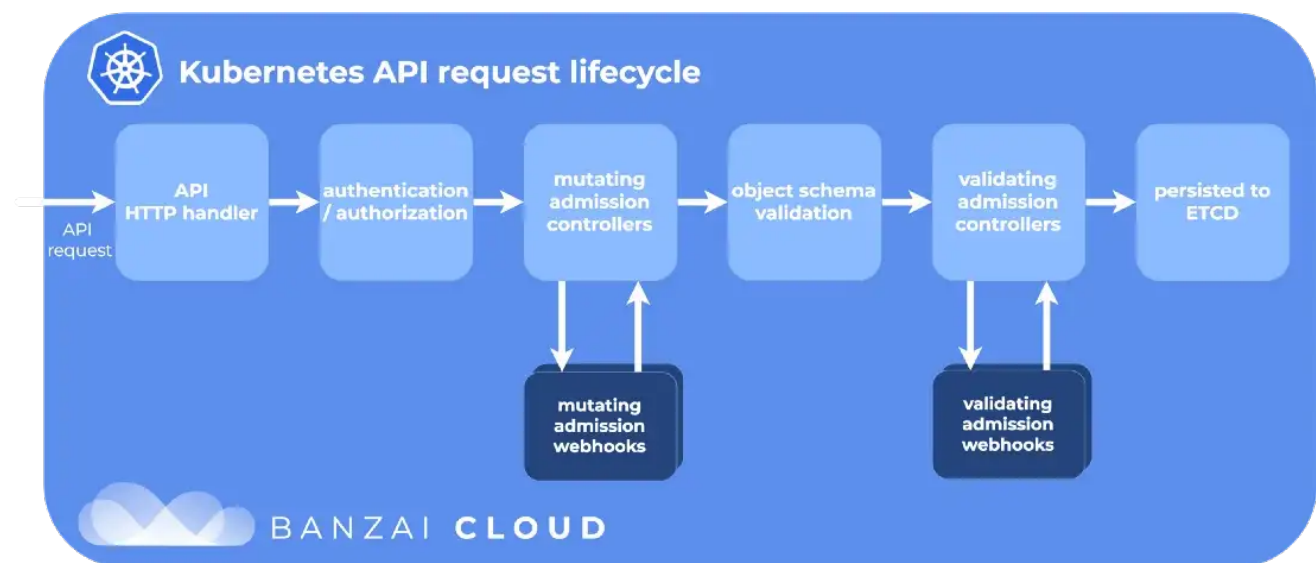
Istio is a service mesh that sits between your microservices and performs various functions. To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy.

“sidecars” that have been setup to read configuration from the Istio Pilot API (also a gRPC service) and then route traffic based on that configuration. The powerful L7 proxy under the hood allows us to leverage features such as metrics, tracing, retry logic, circuit breaking, load balancing and canary deployments.

Starting from the Beginning — Kubernetes

When using Kubernetes you create a Pod using something like a Deployment, StatefulSet, or just by creating a vanilla pod without a higher-level controller. Kubernetes then does its best to maintain a “desired state” by creating the pod(s) in your cluster on some node, making sure they start, and restart if they crash. When a pod is first created Kubernetes goes through its API lifecycle and ensures that each step succeeds before actually creating the pod in the cluster.

The API lifecycle involves the following phases:



Thanks Banzai Cloud for a great graphic.

The one piece of this graph to call out is the “Mutating Admission Webhooks”. These are a special piece of the Kubernetes lifecycle that allows customization of resources before they are committed to the etcd store, the ‘source of truth’ for Kubernetes configuration. This is where Istio gets a lot of its magic.

Mutating Admission Webhooks

When an individual Pod is created (either via `kubectl` or a Deployment resource), it goes through the pod before it can be modified. To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy.

During the process of installing Istio the `istio-sidecar-injector` is added as a mutating webhook configuration resource:

```
$ kubectl get mutatingwebhookconfiguration
NAME                                AGE
istio-sidecar-injector             87d
```

And viewing the configuration:

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  labels:
    app: istio-sidecar-injector
    chart: sidecarInjectorWebhook-1.0.4
    heritage: Tiller
    name: istio-sidecar-injector
webhooks:
- clientConfig:
    caBundle: redacted
    service:
      name: istio-sidecar-injector
      namespace: istio-system
      path: /inject
    failurePolicy: Fail
  name: sidecar-injector.istio.io
  namespaceSelector:
    matchLabels:
      istio-injection: enabled
  rules:
  - apiGroups:
    - ""
    apiVersions:
    - v1
    operations:
    - CREATE
    resources:
    - pods
```

This tells Kubernetes to send all Pod creation events to the `istio-sidecar-injector` service in the `istio-system` namespace if the namespace has the `istio-`

injection=enabled label The injector service then will modify the PodSpec to include two additional containers. To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy. and another for our [Privacy Policy](#), including cookie policy. o additional container. This container is used to log user data. This container is decorated with the injector configmap. This process is otherwise known as “sidecar’ing”.

Sidecar'ed Pods

Istio sidecars are components that are more or less “magical”. Istio does a good job of hiding these details from you once you’ve installed the project. But knowing what they are and how they work is helpful in situations where you need to debug network requests.

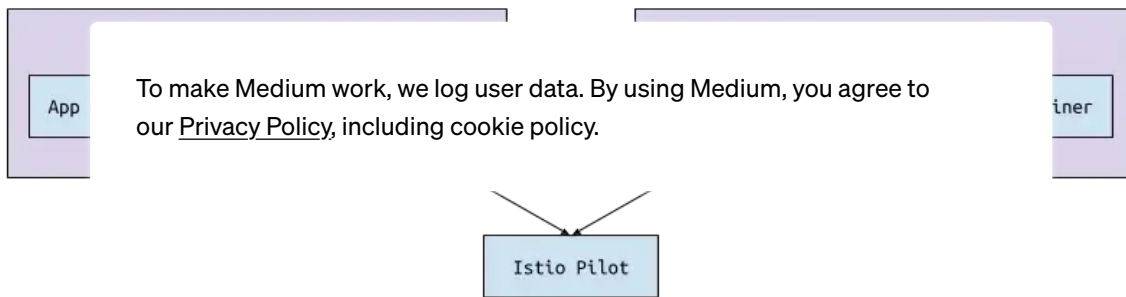
The Init and Proxy Containers

Kubernetes allows you to initialize temporary “one-off” containers before engaging your primary processes, called “init containers”. These are great to perform tasks like bundling assets, database migrations, or in Istio’s case: setting up network rules.

Istio uses Envoy for proxying all of the requests pods receive to the correct destination. To make this a reality, Istio creates `iptables` rules that sends outbound / inbound traffic directly to Envoy. Envoy is able to proxy traffic to its original destination as seamlessly as possible. You can think of this traffic as taking a small “detour” to allow things such as distributed tracing, request metrics, and policy enforcement. To see how Istio creates these iptable rules, you can [view this file](#) on the Istio repository.

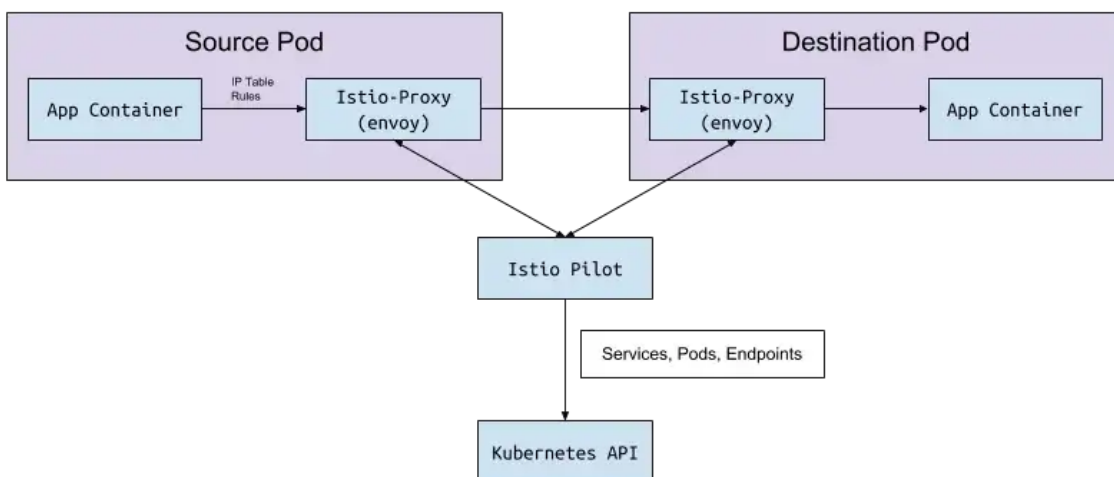
@jimmysongio has an excellent flowchart demonstrating the relationship between the iptables rules and the Envoy proxy:

Envoy to Envoy traffic



Pod <-> Pilot communication

Pilot is also hooked up to the Kubernetes cluster and reads the state of the cluster and listens for updates. It keeps track of Pods, Services, and Endpoints. By keeping track of these resources in your Kubernetes cluster, it can then provide the proper configuration to all of the Envoy sidecars that are connected to Pilot, acting as a bridge between Kubernetes state and Envoy.



Pilot to Kubernetes

When Pods, Services, or Endpoints are updated or created in Kubernetes, those changes are sent to Pilot, which will then send the proper configuration down to every connected Envoy instance.

What Configuration Is Being Sent?

So now that we know how Envoy / Pilot interact at a high level, let's talk about the type of configuration that Envoy receives from Istio's Pilot.

Out of the box, Kubernetes handles most of your networking needs with a type called `Service`, which manages a type called `Endpoint`. You can see a list of Endpoints by

running:

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy.

```
kubectl
```

This is a list of IPs and Ports that exist within the cluster and the target they reference (typically a Pod, created from a Deployment). They're also a very critical part of how Istio configures and sends routing information to Envoy.

Services, Listeners, and Routes!

When you create a `Service` resource in a Kubernetes cluster, you include a set of “match labels” that will select all pods that match those labels. When you send traffic to the Service IP, Kubernetes will then pick a Pod to send the traffic to. For example, performing:

```
curl my-service.default.svc.cluster.local:3000
```

Will first hit the virtual IP that has been assigned for the Service `my-service` in the `default` namespace, that IP will then forward the traffic to a pod that matches the service's label selector.

Istio and Envoy change this logic slightly. Istio configures Envoy based on Services and Endpoints in the Kubernetes cluster it is operating within to leverage Envoy's intelligent routing and load balancing, bypassing the Kubernetes Service. Instead of proxying through a single ip, Envoy knows and connects directly to pod IPs. **Istio maps Kubernetes configuration to Envoy's configuration to make this possible.**

The lingo between Kubernetes, Istio, and Envoy all vary slightly so it can be confusing to know which piece maps to what.

Services

A Service in Kubernetes maps to a **Cluster** in Envoy. An Envoy Cluster is something that contains a list of **Endpoints**, which are the IPs (or hostnames) of instances to handle requests. We can see a list of the Clusters that have been configured in an Istio sidecar'd pod by running `istioctl proxy-config cluster <pod name>`. Running this command shows this pod's current understanding of the “state of the world”. This is an example from one of our environments.

```
$ istioctl
application To make Medium work, we log user data. By using Medium, you agree to
SERVICE our Privacy Policy, including cookie policy.
PORT
BlackHoleCluster
- - - STATIC
accounts-grpc-gw.applications.svc.cluster.local
80 - outbound EDS
accounts-grpc-public.applications.svc.cluster.local
50051 - outbound EDS
addressvalidator.applications.svc.cluster.local
50051 - outbound EDS
```

Now, if we look, we'll see the same services exist in this namespace:

```
$ kubectl get services
```

NAME	EXTERNAL-IP	PORT(S)	TYPE	CLUSTER-IP
accounts-grpc-gw	<none>	80/TCP	ClusterIP	10.3.0.91
accounts-grpc-public	<none>	50051/TCP	ClusterIP	10.3.0.202
addressvalidator	<none>	50051/TCP	ClusterIP	10.3.0.56

If you're wondering how Istio determines what protocol your service uses, Istio configures protocols for service manifests by reading the `name` field on port entries.

```
$ kubectl get service accounts-grpc-public -o yaml
apiVersion: v1
kind: Service
metadata:
  name: accounts-grpc-public
spec:
  ports:
    - name: grpc
      port: 50051
      protocol: TCP
      targetPort: 50051
```

By setting your port to `grpc` or prefixing it with `grpc-`, Istio will configure the service specifically with the HTTP2 protocol. We learned the hard way on how Istio uses these port names when our proxy configs got screwed up because we didn't use `http` or `grpc` prefixes...

Using `kubectl` and port-forwarding the Envoy admin page, you can observe that

account-grpc To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy. envoy with

HTTP2 protocol

```
$ kubectl -n applications port-forward otherpod-dc56885ff-dqc6t
15000:15000 &
$ curl http://localhost:15000/config_dump | yq r -
```

```
...
- cluster:
  circuit_breakers:
    thresholds:
      - {}
  connect_timeout: 1s
  eds_cluster_config:
    eds_config:
      ads: {}
    service_name: outbound|50051||accounts-grpc-
public.applications.svc.cluster.local
  http2_protocol_options:
    max_concurrent_streams: 1073741824
    name: outbound|50051||accounts-grpc-
public.applications.svc.cluster.local
    type: EDS
...
```

Port 15000 is the [Envoy admin page](#) that is available in each sidecar.

Listeners

Similarly, Listeners incorporate Kubernetes endpoints in order to allow traffic into Pods, the address validator service has one endpoint here:

```
$ kubectl get ep addressvalidator -o yaml
apiVersion: v1
kind: Endpoints
metadata:
  name: addressvalidator
subsets:
- addresses:
  - ip: 10.2.26.243
    nodeName: ip-10-205-35-230.ec2.internal
    targetRef:
      kind: Pod
      name: addressvalidator-64885ccb76-87l4d
      namespace: applications
  ports:
  - name: grpc
    port: 50051
```

protocol: TCP

This translates our [Privacy Policy](#), including cookie policy. 1 port

50051:

```
$ kubectl -n applications port-forward addressvalidator-
64885ccb76-87l4d 15000:15000 &
$ curl http://localhost:15000/config_dump | yq r -
```

```
...
dynamic_active_listeners:
- version_info: 2019-01-13T18:39:43Z/651
  listener:
    name: 10.2.26.243_50051
    address:
      socket_address:
        address: 10.2.26.243
        port_value: 50051
    filter_chains:
      - filter_chain_match:
          transport_protocol: raw_buffer
...
```

Routes

The Istio project does not use the standard Kubernetes Ingress object, and instead opts for a more abstract and powerful custom resource known as the `VirtualService`. A `VirtualService` allows you to match routes to upstream clusters by attaching them to a gateway. This idea can be likened to using Kubernetes Ingress along with an Ingress Controller.

In Namely's case, we use the Istio's Ingress-Gateway to funnel all of our internal gRPC traffic:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: grpc-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - hosts:
        - '*'
      port:
        name: http2
```

```

      number: 80
    port:
---
  apiVersion: v1
  kind: VirtualService
  metadata:
    name: grpc-gateway
  spec:
    gateways:
    - grpc-gateway
    hosts:
    - '*'
    http:
    - match:
      - uri:
          prefix: /namely.address_validator.AddressValidator
        retries:
          attempts: 3
          perTryTimeout: 2s
      route:
      - destination:
          host: addressvalidator
          port:
            number: 50051

```

Upon first glance, the above example may be difficult to digest. What's occurring in the background is that there is an Istio-IngressGateway deployment that captures which endpoints to serve based upon the `istio: ingressgateway` selector. In this example, the IngressGateway serves traffic on all domains on port 80 on the HTTP2 protocol. A VirtualService is implemented that fulfills routes for this gateway, matches on the prefix `/namely.address_validator.AddressValidator` and passes them onto the upstream `addressvalidator` service on port 50051 with a 2 second retry rule.

Port forwarding the Istio-IngressGateway pod and viewing its Envoy configuration confirms the above VirtualService:

```

$ kubectl -n istio-system port-forward istio-ingressgateway-
7477597868-rlb5 15000

...
      - match:
        prefix: /namely.address_validator.AddressValidator
      route:
        cluster:
outbound|50051|addressvalidator.applications.svc.cluster.local
        timeout: 0s
        retry_policy:
          retry_on: 5xx,connect-failure,refused-stream
          num_retries: 3

```

```
per try timeout: 2s
```

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy.

```
address\
```

```
address_
```

```
validator .AddressValidator *
```

```
...
```

Random Things We Googled When Operating Istio

I'm receiving a 503 or 404 request!

There are a variety of reasons these errors can occur, but most commonly:

- Your application's sidecars cannot reach out to Pilot (confirm Pilot is running)
- The Kubernetes service manifest is not configured with the correct protocol
- Your VirtualService/Envoy Configuration may be capturing the route in the wrong upstream cluster. Start with an edge service where you expect ingress and study its Envoy logs, or use a tool such as Jaeger to confirm where failures are occurring.

What do the acronyms NR/UH/UF mean in Istio Proxy Logs?

- NR — No Route
- UH — Upstream Unhealthy
- UF — Upstream Failure

Read more on this topic on [Envoy's website](#).

What are some HA considerations for Istio?

- Adding NodeAffinity to critical Istio components to evenly distribute pods across different availability zones and increasing the minimum replica count.
- Run a newer version of Kubernetes with Horizontal Pod Autoscaling enabled. This will allow mission critical pods to scale up and down as load increases/decreases.

My Cronjob never ends, what could be the problem?

After the primary workload is finished, the sidecar container continues to stay up.

You can work around this by disabling sidecars on cronjobs by adding the

```
sidecar.istio.io/inject: "false" annotation to the PodSpec
```

How do you

To make Medium work, we log user data. By using Medium, you agree to our [Privacy Policy](#), including cookie policy.

Kubernetes

pulling

down the latest Helm charts, making our in-house modifications, using `helm template -f values.yml` and committing those files to Github to compare changes before we apply them via `kubectl apply -f -`. We take this approach in order to confirm there are no CRD or API changes across versions we may not have accounted for.

[About](#) [Help](#) [Terms](#) [Privacy](#)

Thanks to [Bobby Tables](#) and [Michael Hamrah](#) for his contributions to this article.

Get the Medium app

