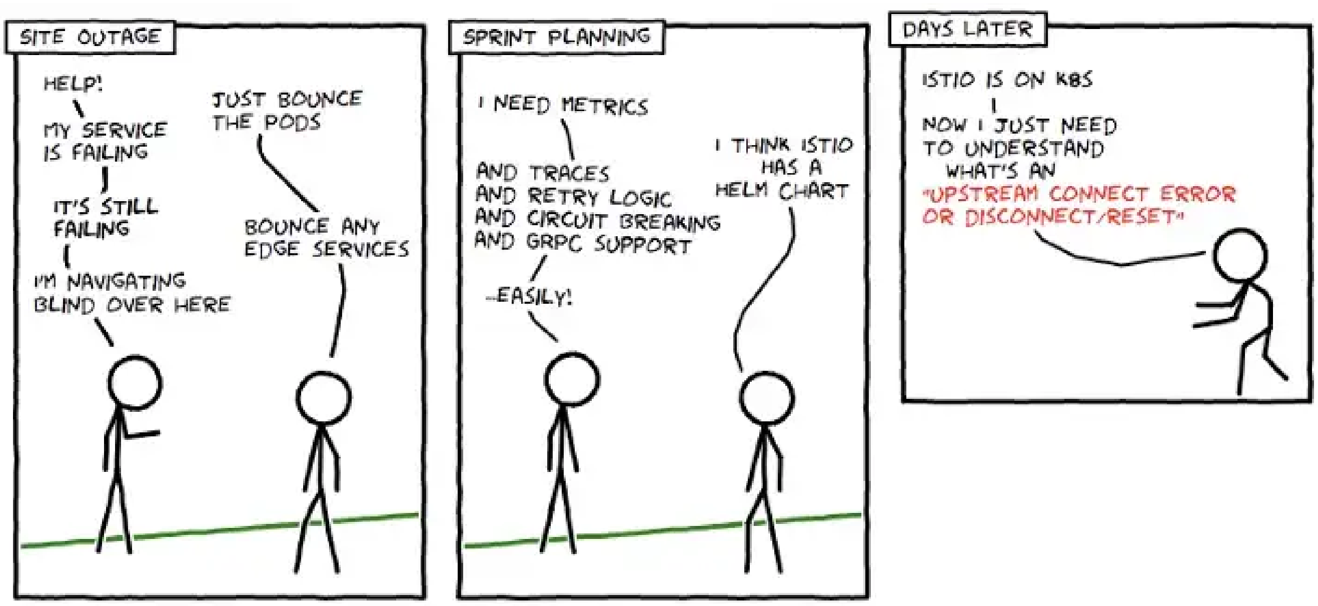
Published in Namely Labs

Shray Kumar Follow
Jan 17, 2019 · 10 min read · Listen

Save

A Crash Course For Running Istio


SITE OUTAGE
HELP! MY SERVICE IS FAILING
JUST BOUNCE THE PODS
IT'S STILL FAILING
BOUNCE ANY EDGE SERVICES
I'M NAVIGATING BLIND OVER HERE
SPRINT PLANNING
I NEED METRICS
AND TRACES AND RETRY LOGIC AND CIRCUIT BREAKING AND GRPC SUPPORT
I THINK ISTIO HAS A HELM CHART
-EASILY!
DAYS LATER
ISTIO IS ON K8S
NOW I JUST NEED TO UNDERSTAND WHAT'S AN "UPSTREAM CONNECT ERROR OR DISCONNECT/RESET"

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.

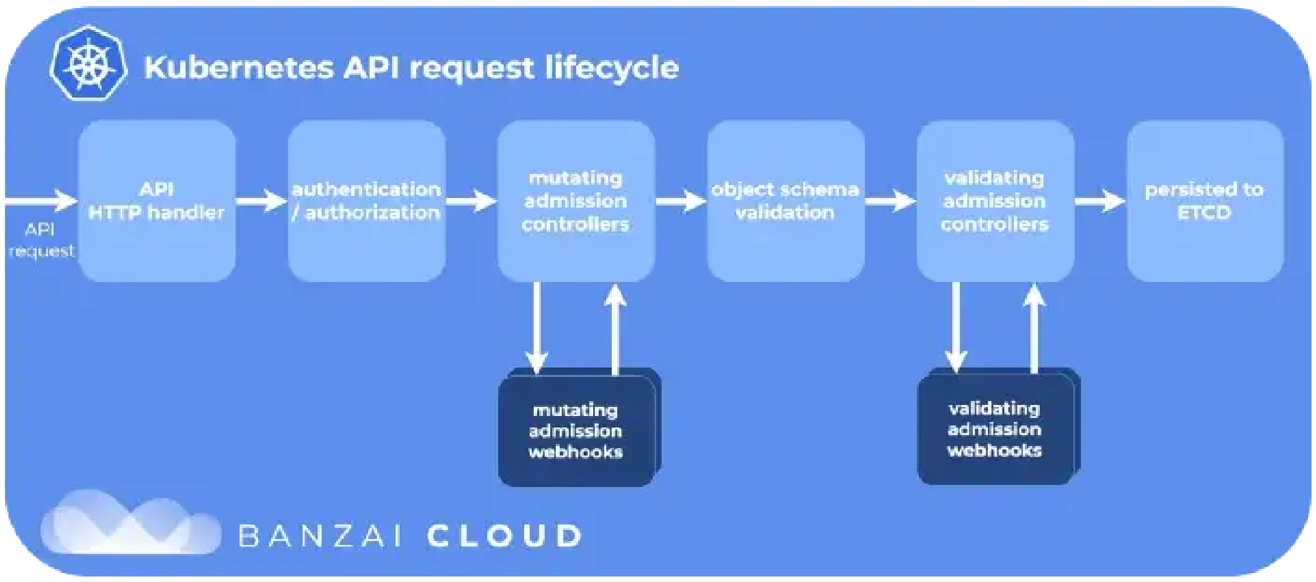
What is Istio?

Istio is a service mesh configuration engine. It reads the state of a Kubernetes cluster and performs updates to L7 (HTTP and gRPC) proxies that are initialized as "sidecars" to Kubernetes pods. These sidecars are Envoy containers 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:


Kubernetes API request lifecycle
API request
API HTTP handler
authentication / authorization
mutating admission controllers
object schema validation
validating admission controllers
persisted to ETCD
mutating admission webhooks
validating admission webhooks
BANZAI CLOUD

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 this same lifecycle, hitting mutating admission webhooks which modify the pod before it actually gets applied.

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, one temporarily for setting up the proxy rules, and another for the actual proxying. The sidecar injector service adds these two additional containers via a template; the template is found in the `istio-sidecar-injector` configmap. This process is otherwise known as "sidecar'ing".

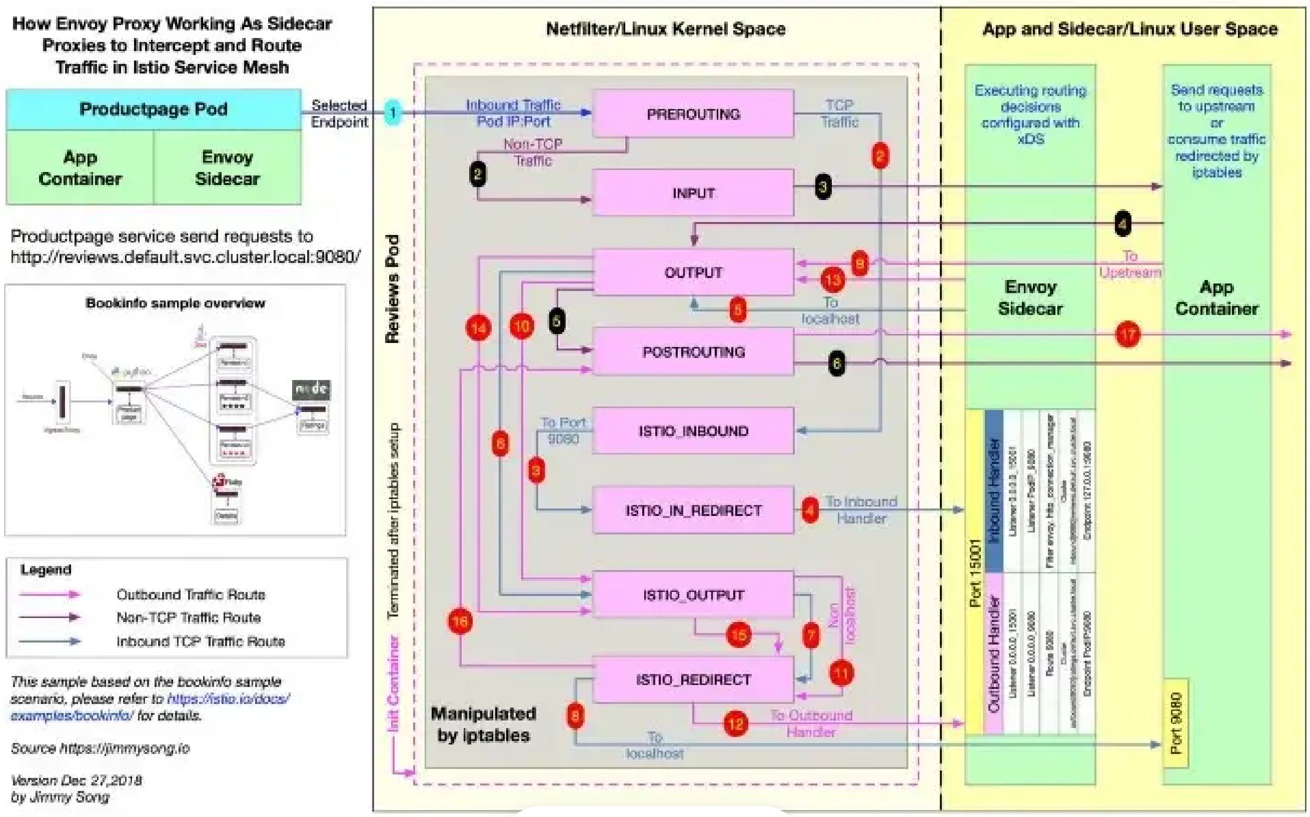
Sidecar'ed Pods

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

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:


How Envoy Proxy Working As Sidecar Proxies to Intercept and Route Traffic in Istio Service Mesh
Productpage Pod
App Container
Envoy Sidecar
Selected Endpoint
Productpage service send requests to http://reviews.default.svc.cluster.local:9080/
Bookinfo sample overview
Legend
Outbound Traffic Route
Non-TCP Traffic Route
Inbound TCP Traffic Route
This sample based on the bookinfo sample scenario, please refer to https://istio.io/docs/examples/bookinfo/ for details.
Source https://jimmysong.io
Version Dec 27,2018 by Jimmy Song
Netfilter/Linux Kernel Space
App and Sidecar/Linux User Space
Reviews Pod
Init Container
Terminated after iptables setup
Manipulated by iptables
Inbound Traffic Pod IP:Port
PREROUTING
TCP Traffic
Non-TCP Traffic
INPUT
OUTPUT
To localhost
POSTROUTING
ISTIO_INBOUND
To Port 9080
ISTIO_IN_REDIRECT
To Inbound Handler
ISTIO_OUTPUT
Non localhost
ISTIO_REDIRECT
To Outbound Handler
To localhost
Executing routing decisions configured with xDS
Send requests to upstream or consume traffic redirected by iptables
To Upstream
Envoy Sidecar
App Container
Port 15001
Inbound Handler
Outbound Handler
Port 9080

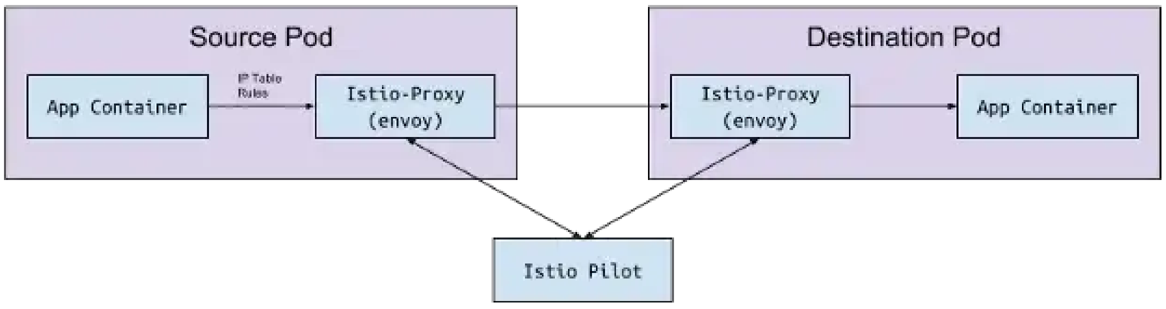
1.2K 9

Because Envoy receives all inbound and outbound traffic, that means all traffic now is "Envoy to Envoy", as shown in the graph above. The Istio Proxy is another container that is added to all pods that the Istio Sidecar Injector modifies. This container is what is running an Envoy process that is configured to receive all traffic for the pod (with some exceptions, such as traffic exiting your Kubernetes cluster).

The Envoy process is also configured to discover all of the routes via the Envoy v2 API that Istio implements.

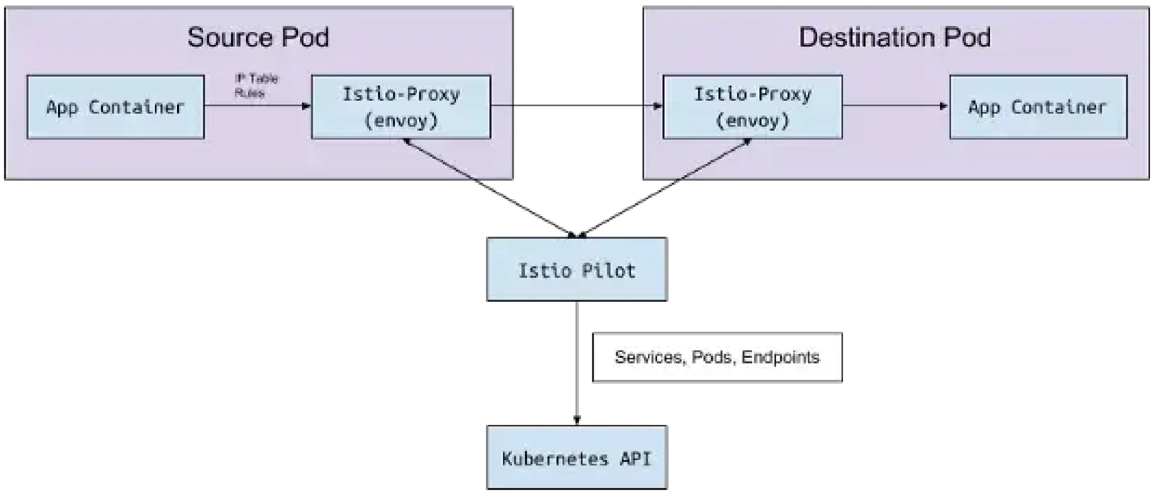
Envoy and Pilot

Envoy on its own doesn't contain the logic to discover pods and services in your cluster. Envoy acts as a *data plane* and wants a *control plane,* some kind of "puppet master", to handle the configuration behavior. Envoy has a configuration flag, asking for a host/port of a service to receive this configuration via a gRPC API. Istio, through its Pilot service, implements (read: fulfills the requirements) for this gRPC API. Envoy connects to this API based on the sidecar'ed configuration injected via the mutating webhook. This API provides all traffic rules that Envoy needs to discover and route traffic with the cluster. This is the 'service mesh'.



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:

```
kubectl get endpoints
```

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 proxy-config cluster taxparams-6777cf899c-wwhr7 -n
applications
SERVICE FQDN
PORT      SUBSET     DIRECTION     TYPE
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                                     TYPE          CLUSTER-IP
EXTERNAL-IP        PORT(S)
accounts-grpc-gw                         ClusterIP     10.3.0.91
<none>             80/TCP
accounts-grpc-public                     ClusterIP     10.3.0.202
<none>             50051/TCP
addressvalidator                         ClusterIP     10.3.0.56
<none>             50051/TCP
```

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-public's endpoints are implemented by Pilot as a cluster in Envoy with HTTP2 protocol options set, validating our assumptions:

```
$ 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 to a single listener on an address validator pod, listening on 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
      protocol: HTTP2
---
apiVersion: networking.istio.io/v1alpha3
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-rldb5 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
              max_grpc_timeout: 0s
            decorator:
              operation:
addressvalidator.applications.svc.cluster.local:50051/namely.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 install Istio?
We use Spinnaker for our deployments, but in general the process entails 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.

Thanks to Bobby Tables and Michael Hamrah for his contributions to this article.

Kubernetes Istio Service Mesh Microservices Envoy Proxy