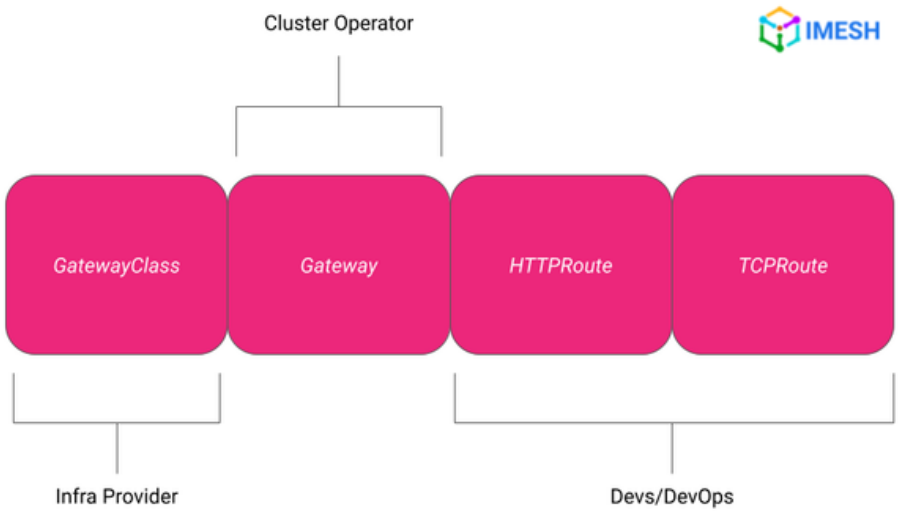
BEST PRACTICE DOCUMENT FOR ADOPTING KUBERNETES GATEWAY API

Table of Contents

01

Introduction to Kubernetes Gateway API

Kubernetes Gateway API is a collection of resources that helps to standardize the specifications for implementing Gateway/Ingress rules in Kubernetes. Gateway API provides resources, such as GatewayClass, Gateway, and *Route, which are also role-delineated (see Fig. A).



Note that Gateway API only standardizes the specifications; for implementation, DevOps and architects will need an Ingress controller or a service mesh, like Envoy Gateway or Istio.

The resources help DevOps and architects to configure basic to advanced networking rules in Kubernetes without relying on vendor-specific CRDs and annotations. There are various implementors and integrators of K8s Gateway API.

Why should you migrate from Ingress to Gateway API?

Kubernetes Ingress is officially frozen; new developments with respect to Ingress configuration will happen in Gateway API CRDs. Since there is no kind Ingress in the Gateway API, DevOps managers and architects will need to make the switch from Ingress, eventually.

Migrating from Ingress is justified even if we compare the capabilities of both Ingress and Gateway API CRDs. Gateway API far outweighs Ingress in the following aspects:

	Kubernetes Ingress	Kubernetes Gateway API
Multitenancy	Hard to implement	Multitenant design
Specifications	Annotation-heavy and tedious	Standard, controller-independent, and simplified
Advanced traffic management	Limited supported	Supported by default
Extensibility	Annotations	CRDs, custom filters, policies

- **Better RBAC and Multitenancy:** Unlike Ingress — which is a single resource without proper role separation — Gateway API CRDs have a role-oriented design which makes them a good fit for multitenant, shared Kubernetes clusters.

- **Standard, controller-independent specifications:** Ingress needs to be configured with annotations except for basic routing functionalities. The annotations are controller-dependent and are not portable across implementations. Gateway API standardizes the specifications for basic and advanced network management without annotations while providing seamless portability across a variety of Ingress controller and service mesh implementations.

- **Advanced traffic management:** Gateway API CRDs provide out-of-the-box support for advanced routing configurations, like header modification, canary and blue-green rollout, etc. Unlike Ingress, DevOps managers and architects can have multiple backends per route in Gateway API and configure network rules without writing and testing tedious custom annotations.

- **Extensibility:** Ingress is extensible with annotations, but the scope is limited since annotations are just strings of key-value pairs. Gateway API allows extensibility with custom filters, policies, and CRDs.

Check out the blog Kubernetes Gateway API vs Kubernetes Ingress to learn more about each of the above points with examples.

Kubernetes Gateway API resources

Kubernetes Gateway API provides the following resources that give freedom for IT teams to work on resources that fall under their respective roles:

GatewayClass (infra provider)

The resource specifies the controller that implements the Gateway API CRDs associated with the class in the cluster. The Gateway API implementation controller is specified using controllerName, which then manages the GatewayClass.

```
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: istio
spec:
  controllerName: istio.io/gateway-controller
```

In the above sample configuration, GatewayClass(es) with the *controllerName: istio.io/gateway-controller* will be managed by the Istio service mesh.

Gateway (Cluster admin/Architect)

A Gateway resource acts as a network endpoint that gets the traffic inside the cluster, like a cloud load balancer. DevOps or Infra team can add multiple listeners to the external traffic and apply filters, TLS, and traffic forwarding rules. Gateway is attached to a GatewayClass and is implemented using the respective controller defined in GatewayClass.

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
 name: k8s-gateway
 namespace: k8s-gw
spec:
 gatewayClassName: istio
 listeners:
   - name: default
       port: 80
       protocol: HTTP
       allowedRoutes:
         namespaces:
           from: All
```

In the above Gateway resource, *gatewayClassName* refers to the respective *GatewayClass* the resource is attached to, and listeners specify that the Gateway listens to HTTP traffic on port 80.

*Route (Devs/DevOps)

Route resources manage the traffic from the Gateway to the back-end services. Multiple Route resources, such as **HTTPRoute, TCPRoute, GRPCRoute, etc.,** are used to configure the routing rules of the respective traffic from a Gateway listener to a backend service. The app team can configure different path names and Header filters (under the rules section in the YAML) in the Route resources to handle the traffic to the backend. Each Route can be attached to a single or multiple Gateways as per the requirements, which can be specified under parentRefs.

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
 name: k8s-http-route-with-istio
 namespace: with-istio
spec:
 parentRefs:
   - name: k8s-gateway
     namespace: k8s-gw
 rules:
   - matches:
       - path:
           type: PathPrefix
           value: /with-istio
     backendRefs:
       - name: echoserver-service-with-istio
         port: 80
```

The rules field allows the set of advanced routing behaviors, such as header-based matching, traffic splitting, balancing, etc. The above configuration routes HTTP traffic with the request path /with-istio from the k8s-gateway — to echoserver-service-with-istio service on port 80, which is the destination.

Combining the above resources, the request flow in Gateway API would look like this: a request would first come to the Gateway, from which *Route applies the routing rules before the request finally ends up on the respective backend service (see Fig. B).

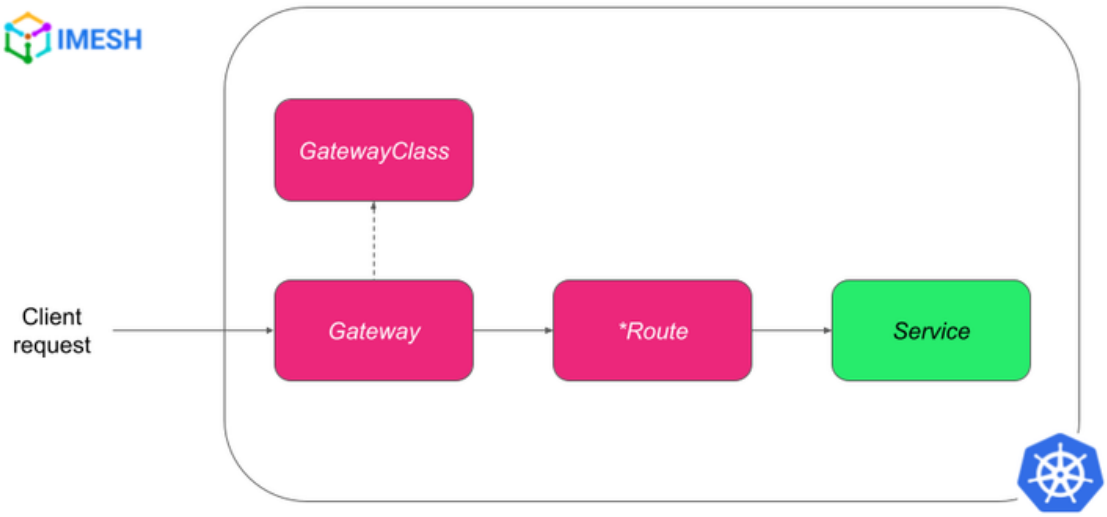
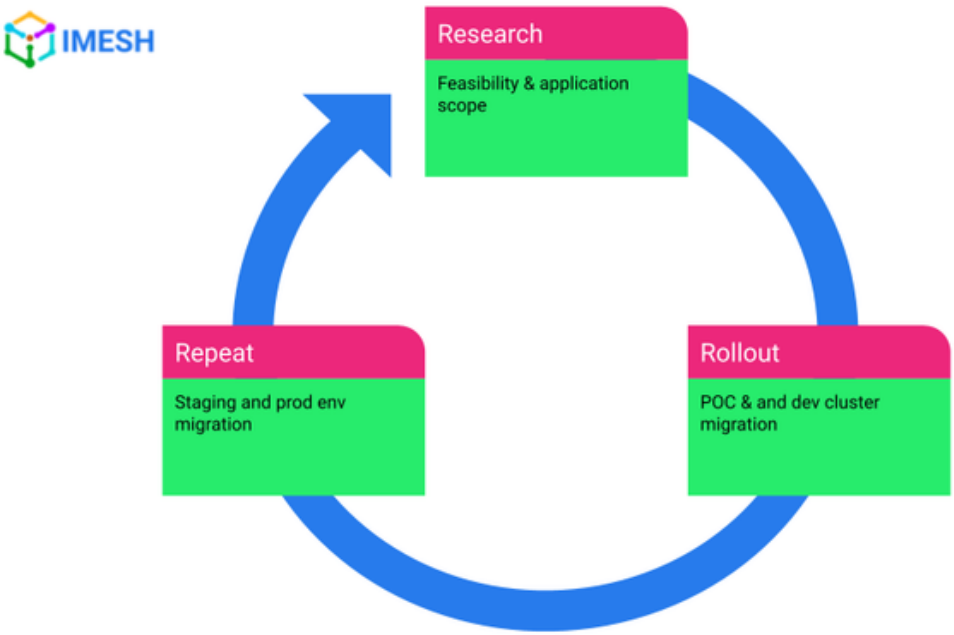


Fig. B - Request flow in Kubernetes Gateway API

How to migrate from Ingress to Gateway API — The 3 R's migration strategy

The migration strategy we are sharing here is not specific to any implementation. It focuses on a phased rollout approach for a safer switch from Ingress to Gateway API. Needless to say, DevOps and architects need a solid understanding of Ingress and the Gateway API CRDs to carry out a successful migration.



Step #1: Research

The research phase entails questions DevOps and architects need to ask regarding Gateway API feasibility and application scope, while planning the migration.

Gateway API feasibility

There can be 2 scenarios with respect to migration — either the DevOps team wants to adopt Gateway API CRDs with the current Ingress controller, or they want to implement the CRDs using a new controller.

In the latter case, it is important to research Ingress controller and service mesh implementations for Gateway API and finalize one. The team should also be clear about configuring Ingress rules with Gateway API CRDs.

So the questions to ask are,

- Which vendor to consider in case of migrating to a new Ingress controller?
- How to map current Ingress configurations to equivalent concepts in Gateway API?

gateway api

Application scope

DevOps and architects should be wary of changes that can happen to the application or infrastructure with Gateway API migration. For example, there can be controller-dependent features that need to be implemented with Gateway API.

At the time of writing this, RequestHeaderModifier and RequestRedirect HTTP filter types are in core supported features, and RequestMirror, ResponseHeaderModifier, and UrlRewrite are in extended support.

So, following are some questions to ask in this stage:

- How to migrate implementation-specific features?
- Should deployment strategies be changed?
- Is the migration going to change the current infra and request flow? If yes, how?
- How to ensure appropriate RBAC policies with Gateway API resources at different levels?

Asking these questions and gathering insights will save a lot of trouble during the actual migration. Once it is done, proceed to rollout.

Step #2: Rollout

This is where the actual migration happens. DevOps and architects can start the migration by making a POC.

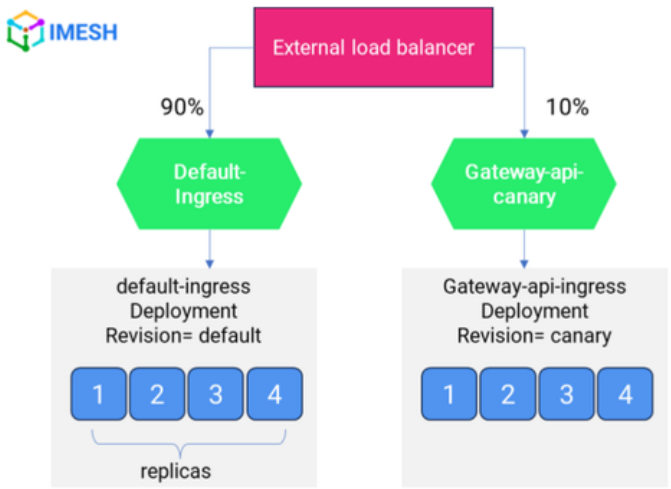
Create a proof of concept (POC) with non-critical workloads

Take a few non-critical workloads in the dev environment and rewrite their Ingress configuration with Gateway API resources. You can deploy a few services in a separate test environment for the POC so they don't interfere with your current workflows.

If you plan to migrate to a new controller, use it in this phase to implement Gateway API. Make sure that controller-dependent features can be implemented using the new controller. Check if the Ingress configurations are working well for the services and move ahead to the next step.

Start progressive delivery in the dev cluster using the canary rollout strategy

Migrate more workloads in the dev cluster to Gateway API. Shift traffic from Ingress to Gateway API using the canary deployment strategy. That is, let the Ingress and Gateway API resources coexist and use an external load balancer to route the traffic between the two (see the image below).



Route most of the traffic to the Ingress and a small portion to the Gateway API. You may start by routing 10% of the total traffic routed to your service, through Gateway API. Test any additional filters, matching rules, and extensions (if any) in the Gateway API routes.

Test and validate

Throughout the rollout process, DevOps and architects need to ensure that the controller-specific configurations and features achieved with Ingress can also be achieved with Gateway API. You can also test RBAC policies with Gateway API for further fine-grained security.

Continue testing the APIs after migration and if metrics and logs are fine, gradually increase the traffic percentage to Gateway API. Perform load testing and other benchmarking measurements to leave no room for errors. Based on the confidence in the performance and behavior of Gateway API, release it completely.

Step #3: Repeat

The stage involves repeating the steps taken in the rollout stage but in the staging and production environment. The configurations will mostly be the same, except for changes in the naming. DevOps and architects can first start with the staging environment and then move on to migrate production workloads.

You need to perform rigorous testing and performance benchmarking to prevent any downtime. The rollout and repeat stages follow a clean-up process, where the Ingress resources are decommissioned after Gateway API handles 100% of the traffic.

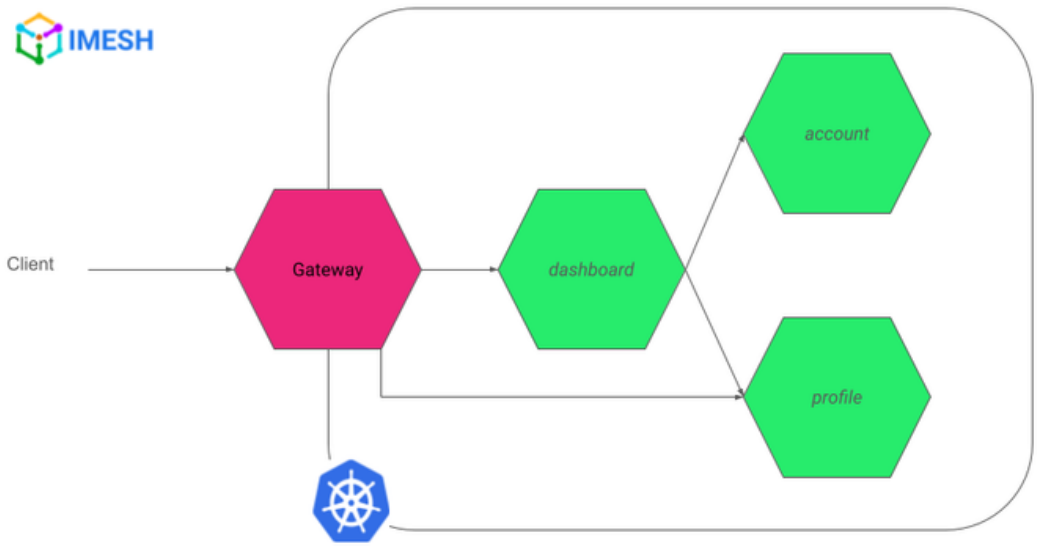
Gateway API implementation tutorial

I will follow the given 3-step process for the demo to implement Gateway API:

- Step #0: Demo overview and prerequisites
- Step #1: Deploy the banking application and Ingress
- Step #2: Deploy Gateway and HTTPRoute CRDs of Gateway API
- Step #3: Weighted traffic distribution between Ingress and Gateway API

Step #0: Demo overview and prerequisites

I will start by deploying the banking application, which has dashboard, account, and profile services (see Fig.A)



I will then expose the dashboard and profile services using Ingress. Later, you will see how to use K8s Gateway API CRDs (Gateway and HTTPRoute) to route traffic to the same services and then gradually shift the traffic from Ingress to Gateway API.

The only prerequisite for the demo is to have the Nginx controller and Istio Ingress configured in the cluster. Check out the blog, How to get started with Istio in Kubernetes in 5 steps, to get started with Istio.

I will use the Nginx controller for Ingress, and Istio Ingress will implement Gateway API CRDs. I will also add the name of the respective controller to the request header to identify the controller. This will be helpful in step #3.

All the YAMLs used in the demo are in the IMESH GitHub repo.

Step #1: Deploy the banking application and Ingress

Create a banking-app namespace and deploy the banking application. Use the following command to see if the services are up and running:

```
kubectl get all -n banking-app
```

```
loopaz@azmal-f15:~/Projects/webinar-and-videos/migrate-ingress-to-k8s-gateway-api$ k get all -n banking-app
NAME                             READY   STATUS    RESTARTS   AGE
pod/account-b85c96c7b-tqmhg      1/1     Running   0          12s
pod/dashboard-8559b7cb8-2pf5z    1/1     Running   0          10s
pod/profile-576f9fd6f7-6d62h     1/1     Running   0          9s

NAME                 TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
service/account      ClusterIP   10.0.89.12    <none>        8080/TCP   12s
service/dashboard    ClusterIP   10.0.78.110   <none>        3000/TCP   9s
service/profile      ClusterIP   10.0.107.16   <none>        8081/TCP   8s

NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/account      1/1     1            1           14s
deployment.apps/dashboard    1/1     1            1           12s
deployment.apps/profile      1/1     1            1           10s

NAME                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/account-b85c96c7b      1         1         1       13s
replicaset.apps/dashboard-8559b7cb8    1         1         1       12s
replicaset.apps/profile-576f9fd6f7     1         1         1       10s
```

First, I will expose the profile service using the following Ingress configuration:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: banking-app-profile-ingress
  namespace: banking-app
  annotations:
    service.beta.kubernetes.io/port_80_no_probe_rule: "true"
    service.beta.kubernetes.io/azure-load-balancer-health-probe-request-path: /
    nginx.ingress.kubernetes.io/server-snippet: |
      more_set_headers "controller: NGINX";
spec:
  ingressClassName: "nginx"
  rules:
    - http:
        paths:
          - path: /users
            pathType: Prefix
            backend:
              service:
                name: profile
                port:
                  number: 8081
```

You can see that Nginx is set as the Ingress controller from ingressClassName: “nginx”. The path field defines that whenever a request comes to /users path, it will be forwarded to the profile service.

Once you apply the Ingress configuration, you can verify it by trying to reach the service with a curl command. Use the controller’s IP address, which you can find using the following command, to curl the profile service:

```
kubectl get all -n your_ingress_nginx_namespace
```

```
loopaz@azmal-f15:~/Projects/webinar-and-videos/migrate-ingress-to-k8s-gateway-api$ k get all -n ingress-nginx
NAME                                            READY   STATUS    RESTARTS   AGE
pod/ingress-nginx-controller-856ff7bcc4-2655c   1/1     Running   0          6d12h

NAME                                         TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)                      AGE
service/ingress-nginx-controller             LoadBalancer   10.0.198.128   20.246.158.117   80:30668/TCP,443:31773/TCP   6d13h
service/ingress-nginx-controller-admission   ClusterIP      10.0.49.124    <none>           443/TCP                      6d13h
```

Curl profile service:

```
curl -v your_controller_ip/users
```

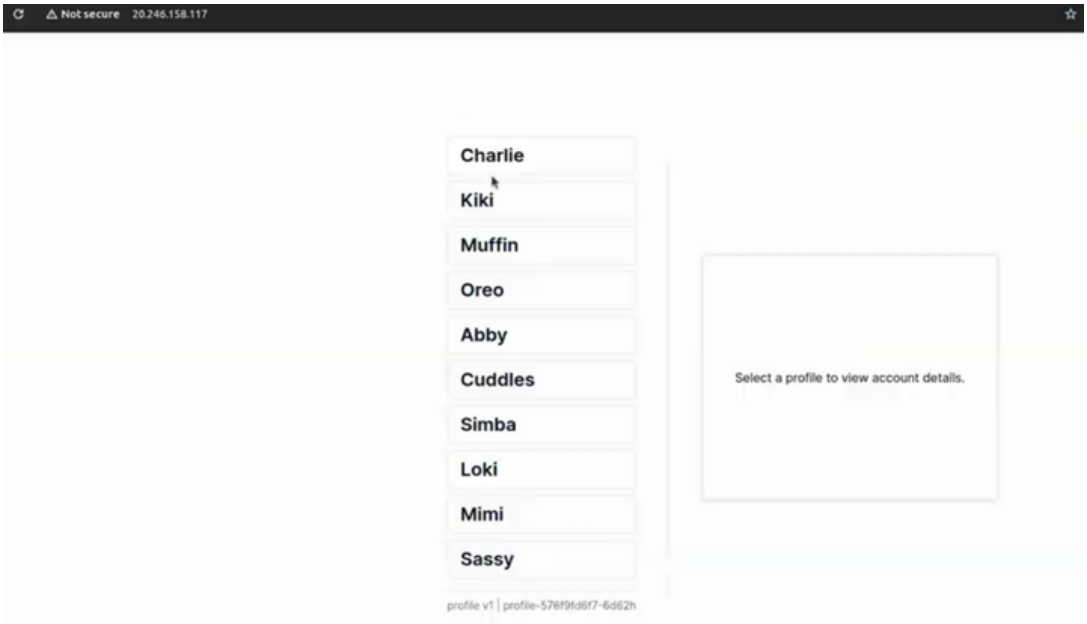
You will see a response as the following:

```
loopaz@aznal-f15:~/Projects/webinar-and-videos/migrate-ingress-to-k8s-gateway-api$ curl -v 20.246.158.117/users
*   Trying 20.246.158.117:80...
* Connected to 20.246.158.117 (20.246.158.117) port 80 (#0)
> GET /users HTTP/1.1
> Host: 20.246.158.117
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Thu, 25 Jan 2024 17:13:36 GMT
< Content-Type: application/json;charset=UTF-8
< Content-Length: 986
< Connection: keep-alive
< controller: NGINX
<
* Connection #0 to host 20.246.158.117 left intact
{"appVersion":"v1","data":[{"uid":"id1","name":"Charlie","avatar":"https://api.dicebear.com/6.x/lorelei/svg?seed=Charlie"},{"uid":"id2","name":"Kiki","ava
tar":"https://api.dicebear.com/6.x/lorelei/svg?seed=Kiki"},{"uid":"id3","name":"Muffin","avatar":"https://api.dicebear.com/6.x/lorelei/svg?seed=Muffin"},{
"uid":"id4","name":"Oreo","avatar":"https://api.dicebear.com/6.x/lorelei/svg?seed=Oreo"},{"uid":"id5","name":"Abby","avatar":"https://api.dicebear.com/6.x
/lorelei/svg?seed=Abby"},{"uid":"id6","name":"Cuddles","avatar":"https://api.dicebear.com/6.x/lorelei/svg?seed=Cuddles"},{"uid":"id7","name":"Simba","avat
ar":"https://api.dicebear.com/6.x/lorelei/svg?seed=Simba"},{"uid":"id8","name":"Loki","avatar":"https://api.dicebear.com/6.x/lorelei/svg?seed=Loki"},{"uid
":"id9","name":"Mimi","avatar":"https://api.dicebear.com/6.x/lorelei/svg?seed=Mimi"},{"uid":"id10","name":"Sassy","avatar":"https://api.dicebear.com/6.x/l
```

Similarly, you can apply the below Ingress configuration to expose the dashboard service:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: banking-app-dashboard-ingress
  namespace: banking-app
  annotations:
    service.beta.kubernetes.io/port_80_no_probe_rule: "true"
    service.beta.kubernetes.io/azure-load-balancer-health-probe-request-path: /
    nginx.ingress.kubernetes.io/server-snippet: |
      more_set_headers "controller: NGINX";
spec:
  ingressClassName: "nginx"
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: dashboard
                port:
                  number: 3000
```

You can verify it by entering the controller IP in the browser, which will open the dashboard:



So, we have successfully deployed the banking application, and exposed profile and dashboard services using Nginx Ingress. Let us now deploy and use Gateway API CRDs to expose the same services.

Step #2: Deploy Gateway and HTTPRoute CRDs of Gateway API

Use the following Gateway configuration and deploy it:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: banking-gateway
  namespace: banking-app
  annotations:
    service.beta.kubernetes.io/port_80_no_probe_rule: "true" # FOR AZURE
spec:
  gatewayClassName: istio
  listeners:
    - name: default
      port: 80
      protocol: HTTP
      allowedRoutes:
        namespaces:
          from: All
```

I'm specifying Istio Ingress as the controller here (gatewayClassName: istio). The configuration opens a listener at port 80, and all the routes from any namespace can communicate with it.

Note that we took the IP of the Ingress controller (nginx-ingress) to access the services exposed through Ingress in step #1. In Gateway API, the Gateway creates a new resource/pod with an external IP on its own.

You can use the following command to get the IP:

```
kubectl get all -n banking-app
```

```
loopaz@azmal-f15:~/Projects/webinar-and-videos/migrate-ingress-to-k8s-gateway-api$ k get all -n banking-app
NAME                                      READY   STATUS    RESTARTS   AGE
pod/account-b85c96c7b-tqmhg               1/1     Running   0          5m11s
pod/banking-gateway-istio-95465dd95-htqt9 1/1     Running   0          15s
pod/dashboard-8559b7cb8-2pf5z             1/1     Running   0          5m9s
pod/profile-576f9fd6f7-6d62h              1/1     Running   0          5m8s

NAME                                TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)                        AGE
service/account                     ClusterIP      10.0.89.12     <none>          8080/TCP                       5m12s
service/banking-gateway-istio       LoadBalancer   10.0.207.113   4.157.21.161    15021:30626/TCP,80:31353/TCP   16s
service/dashboard                   ClusterIP      10.0.78.110    <none>          3000/TCP                       5m9s
service/profile                     ClusterIP      10.0.107.16    <none>          8081/TCP                       5m8s
```

The Gateway resource only opens a listener and does not know where to route the traffic it receives. This is where you can use HTTPRoute resources and define routing logic for the traffic hitting the Gateway.

Use the following HTTPRoute configuration to route traffic to the profile service:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: profile-route
  namespace: banking-app
spec:
  parentRefs:
    - name: banking-gateway
```

```
rules:
  - matches:
      - path:
          type: PathPrefix
          value: /users
    filters:
      - type: ResponseHeaderModifier
        responseHeaderModifier:
          add:
            - name: Controller
              value: ISTIO
    backendRefs:
      - name: profile
        port: 8081
```

- The parentRefs field in the configuration specifies the Gateway to which the HTTPRoute should be attached. (An HTTPRoute can be attached to multiple Gateways.)
- The routing logic is the same as the Ingress one in step #1:
- any requests to the path /users will be routed to the profile service,
- and it uses Istio Ingress as the controller.

(Notice that in the Ingress configurations in step #1, I used annotations to add the controller name to the request header. Since the annotation is Nginx-specific, it is not portable and cannot be used with other controllers. However, in the Gateway API, the filters field — where I have specified the controller name to be added in the header — is a property of the HTTPRoute CRD. It is portable and can be used across a variety of Kubernetes Gateway API implementations.)

Similar to the route for profile service, apply the following HTTPRoute file to expose the dashboard service:

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: dashboard-route
  namespace: banking-app
spec:
  parentRefs:
    - name: banking-gateway
  rules:
    - filters:
        - type: ResponseHeaderModifier
          responseHeaderModifier:
            add:
              - name: Controller
                value: ISTIO
      backendRefs:
        - name: dashboard
          port: 3000
```

- You can use the Gateway resource's external IP to verify Gateway API CRD implementation:
- Use curl -v your_Gateway_pod_ip/users to check the profile service
- Enter the IP on a browser to access the banking dashboard.

So far, we have implemented both Ingress and Kubernetes Gateway API CRDs, and they co-exist in the cluster. Now, let us loadbalance between them and gradually shift the traffic from Ingress to Gateway API.

Step #3: Weighted traffic distribution between Ingress and Gateway API

To ensure a safe migration from Ingress to K8s Gateway API, start by shifting 10% of the traffic to Gateway API while Ingress serves 90% of the requests (see Fig.B). This step is helpful to check for errors and see if Gateway API can handle the load.

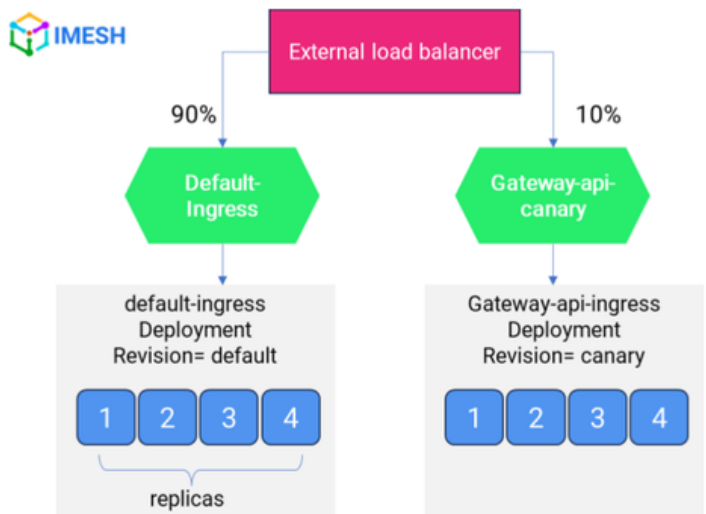


Fig.B - Weighted traffic distribution between Ingress (90) and Kubernetes Gateway API (10)

You can use any load balancer to implement weighted traffic distribution. I'm using HAProxy load balancer locally, for this demo. You can install it using any package manager.

First, modify the HAProxy configuration file to ensure you have the right configs. You can do that by entering the following command:

```
sudo vim /etc/haproxy/haproxy.cfg
```

```
listen stats
    bind :9000
    stats enable
    stats uri /stats
    stats refresh 10s

frontend my_frontend
    bind 192.168.29.223:80
    mode http
    default_backend my_backend

backend my_backend
    mode http
    balance roundrobin
    server server1 20.246.158.117:80 weight 90 check
    server server2 4.156.208.175:80 weight 10 check
```

- You can see that I have exposed a front end at port 80, which is using my WLAN adapter IP. The IP opens the banking dashboard in the browser.
- I have also specified 2 backends with respective traffic weights:
    a. server 1 represents Ingress and serves 90% of requests; server 2 represents Gateway API CRDs and they serve 10% of total traffic.
- Ensure that server 1 and server 2 have the IPs of the Ingress controller and Gateway pod, respectively.

After you save the configuration changes, reload HAProxy:

```
sudo systemctl reload haproxy
```

Now, the load balancer with weighted traffic distribution has been applied. You can see the banking dashboard by entering the front-end IP in a browser.

To verify the 90-10 traffic distribution between the Ingress controller and Gateway API, send 10 requests to the front end and watch for the header value using the following commands:

```
for i in $(seq 1 10) ; do curl sI http://your-lb-ip; done | grep -c NGINX
for i in $(seq 1 10) ; do curl sI http://your-lb-ip; done | grep -c ISTIO
```

See that the Nginx controller handles 9 requests and Istio Ingress receives only one. You can gradually increase the weight to Istio Ingress and implement K8s Gateway API completely.

Multicluster, multicloud Gateway implementation

We have seen how to implement K8s Gateway API and access the services that are in the same cloud network — using the Gateway and HTTPRoute resources.

Now, what if a service is in a different cloud environment and you have to access it from the Gateway i.e., multicluster, multicloud scenario?

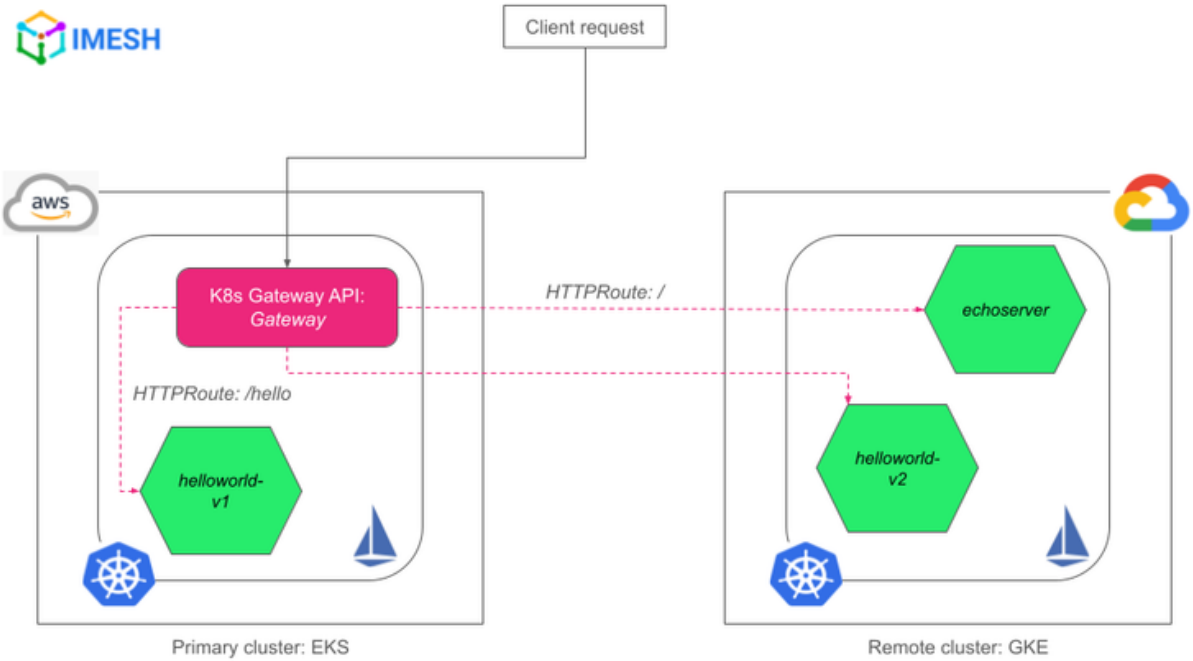
Multicluster Kubernetes Gateway demo overview

We have two clusters: one in EKS (primary) and the other in GKE (remote). I have deployed Istio in both the clusters and the setup is primary-remote Istio installation. Istio is used as the controller to implement the Gateway API resources.

Here's what I'm going to do:

- In the primary cluster/EKS, deploy the helloworld-v1 deployment, helloworld service, and echoserver service.
- In the remote cluster/GKE, deploy helloworld-v2 deployment, helloworld service, echoserver deployment, and echoserver service.
- Deploy the Kubernetes Gateway API resources — Gateway and HTTPRoutes — in the primary cluster.

After the deployments, we will verify that the Gateway in the primary cluster/EKS can access the services in the remote cluster/GKE, as shown in the image below:



Multicluster, multicloud Gateway with K8s Gateway API demo setup

Deploy the applications and services in clusters

Deploying helloworld-service in both the primary and remote clusters:

```
kubectl -f apply helloworld-service.yaml --context=eks-cluster
kubectl -f apply helloworld-service.yaml --context=gke-cluster
```

Deploying helloworld-deployment-v1 to the primary cluster/EKS and helloworld-deployment-v2 to the remote cluster/GKE:

```
kubectl -f apply helloworld-deployment-v1.yaml --context=eks-cluster
kubectl -f apply helloworld-deployment-v2.yaml --context=gke-cluster
```

Deploying echoserver-service in both the clusters and echoserver-deployment only in the remote cluster:

```
kubectl -f apply echoserver-service.yaml --context=eks-cluster
kubectl -f apply echoserver-service.yaml --context=gke-cluster
kubectl -f apply echoserver-deployment.yaml --context=gke-cluster
```

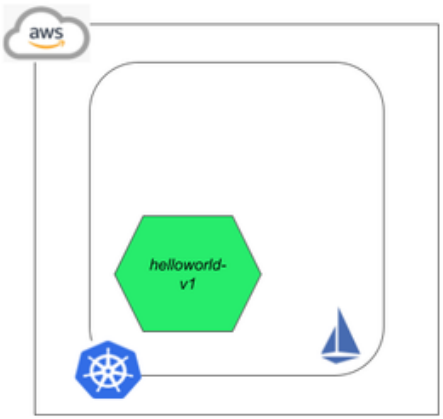
Note that service resources need to be deployed in both clusters for this to work. That is why I deployed the *echoserver-service* in the primary cluster/EKS although the deployment is only in the remote cluster/GKE.

Now, let us verify the deployments in both the primary and secondary clusters:

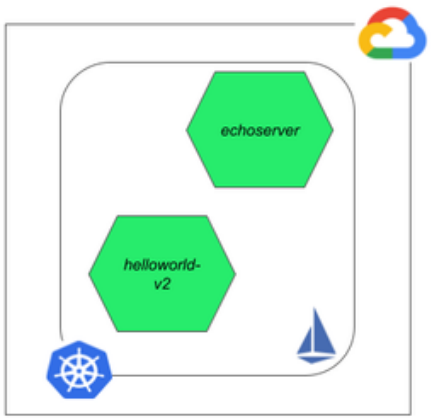
```
kubectl get svc -n demo --context=eks-cluster
kubectl get pods -n demo --context=eks-cluster
kubectl get svc -n demo --context=gke-cluster
kubectl get pods -n demo --context=gke-cluster
```

```
ravi@Ravis-iMac Getting started with Gateway API in multicluster Istio on EKS and GKE % kubectl get svc -n demo --context=eks-cluster
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
echoserver   ClusterIP   10.100.124.42    <none>        80/TCP    19s
helloworld   ClusterIP   10.100.200.111   <none>        80/TCP    3m10s
ravi@Ravis-iMac Getting started with Gateway API in multicluster Istio on EKS and GKE % kubectl get pods -n demo --context=eks-cluster
NAME                          READY   STATUS    RESTARTS   AGE
helloworld-v1-867747c89-9hhsj   2/2     Running   0          2m26s
ravi@Ravis-iMac Getting started with Gateway API in multicluster Istio on EKS and GKE % kubectl get svc -n demo --context=gke-cluster
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
echoserver   ClusterIP   10.76.14.37    <none>        80/TCP    98s
helloworld   ClusterIP   10.76.14.153   <none>        80/TCP    3m15s
ravi@Ravis-iMac Getting started with Gateway API in multicluster Istio on EKS and GKE % kubectl get pods -n demo --context=gke-cluster
NAME                          READY   STATUS    RESTARTS   AGE
echoserver-686bd46d8b-jmscg   2/2     Running   0          111s
helloworld-v2-79d5467d55-zfk8j   2/2     Running   0          2m36s
```

The primary cluster has the helloworld-v1 pod running, while the remote cluster has both *helloworld-v2* and *echoserver* pods running successfully:



Primary cluster: EKS



Remote cluster: GKE

Deploy K8s Gateway API resources and verify multicluster communication

Applying the gateway resource in the primary/EKS cluster:

```
kubectl apply -f gateway-api-gateway.yaml --context=eks-cluster
```

The Gateway uses Istio as the controller and is deployed in the istio-ingress namespace.

Deploying HTTPRoute in the primary cluster for helloworld application, which listens on path /hello:

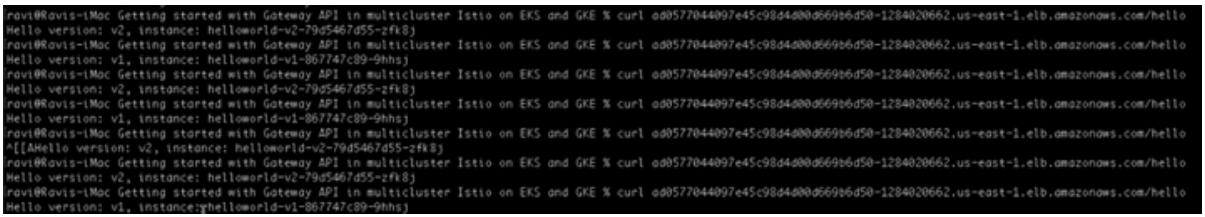
```
kubectl apply -f helloworld-httproute.yaml --context=eks-cluster
```

Now, let us verify multicluster communication by curling *helloworld* application, but first, we need to get the Gateway IP:

```
kubectl get svc -n istio-ingress --context=eks-cluster
```

Verifying multicluster communication:

```
curl your_gateway_external_ip/hello
```



You can see that the request is served by both the helloworld-v1 and helloworld-v2 that are deployed in the primary and secondary clusters, respectively.

Now, let us deploy the HTTPRoute for echoserver in the primary cluster, which listens on / :

```
kubectl apply -f echoserver-httproute.yaml --context=eks-cluster
```

Verifying if the Gateway is able to access *echoserver* deployed in the remote cluster:

```
curl your_gateway_external_ip
```

```
ravi@Ravis-iMac Getting started with Gateway API in multicluster Istio on EKS and GKE % curl ad0577044097e45c98d4d00d669b6d50-1284020662.us-east-1.elb.amazonaws.com

Hostname: echoserver-6866d46d8b-jmscg

Pod Information:
        -no pod information available-

Server values:
        server_version=nginx: 1.13.3 - lua: 10008

Request Information:
        client_address=::ffff:127.0.0.6
        method=GET
        real path=/
        query=
        request_version=1.1
        request_scheme=http
        request_uri=http://ad0577044097e45c98d4d00d669b6d50-1284020662.us-east-1.elb.amazonaws.com:80/

Request Headers:
        accept=*/*
        host=ad0577044097e45c98d4d00d669b6d50-1284020662.us-east-1.elb.amazonaws.com
        user-agent=curl/8.1.2
        x-b3-parentspanid=1544957c5f8e17e9
        x-b3-sampled=0
        x-b3-spanid=725f95f2e19f4c28
        x-b3-traceid=00cbd1e39db178741544957c5f8e17e9
        x-envoy-attempt-count=1
        x-envoy-internal=true
        x-forwarded-client-cert=By=spiffe://cluster.local/ns/demo/sa/default;Hash=6d4f74aec9d7f1633ed05e4c103ff0890c843240ab7e58761765986d499a511c;Subject=&quot;&quot;;URI=spiffe://c
luster.local/ns/demo/sa/gateway-istio
        x-forwarded-for=192.168.128.57
        x-forwarded-proto=http
        x-request-id=646e1481-9f59-4e2f-8a2e-abe7ece1ed8a

Request Body:
        -no body in request-
```

The Gateway is able to get response from *echoserver* deployed in the remote cluster successfully.

Benefits of K8s Gateway API

RBAC	Vendor Lock-in	Developer Experience
- Implement RBAC between cluster operators, DevOps, and app team is easy - No accidents in the ingress	- High standardization and less annotation - New controllers can be adopted quickly as per the requirement	- No more writing vendor-specific routing filters - Create custom resource as per the needs

Proper role delineation and RBAC

The cluster admin no longer needs to worry about developers accidentally introducing any wrong configuration to the Ingress resource, as they do not need to share the Gateway resource with anyone. Devs/DevOps can create *Routes and attach them to particular Gateways without distracting other teams' routes.

No vendor lock-in

The standardization brought by Gateway API makes it seamless to switch between vendors providing controllers. DevOps and architects can use the same API to configure networking with the new vendor by changing the gatewayClassName in the Gateway resource. Almost all popular Gateway controllers and service meshes support integration with K8s Gateway API.

Increased developer experience

Advanced traffic routing rules can now be configured in *Routes in Gateway API, which spares developers/DevOps from writing and rigorously testing vendor-specific CRDs and annotations. Besides, K8s Gateway API allows users to extend it by creating custom resources that suit their unique needs.

Integration scenarios with Istio service mesh

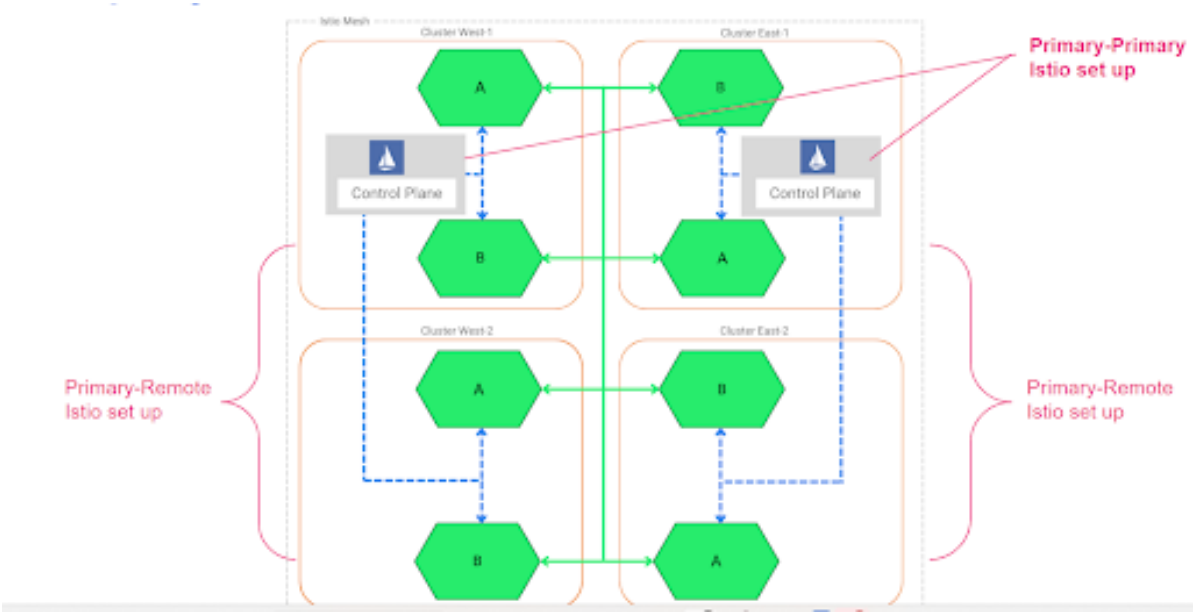
There can be three scenarios while implementing Gateway API:

1. Implementing only Gateway API and selecting Istio as a controller.
2. Already using Istio service mesh and want to migrate from Istio ingress to the Gateway API.
3. Planning to adopt Istio service mesh along with Gateway API.

Implementing Gateway API is straightforward and we discussed it in the above sections. However, integrating Gateway API with Istio service mesh can demand some more considerations.

Istio can be implemented in many ways based on the company drivers. If performance, isolation, and HA are the key imperatives for the IT team, then an architect would like to set up primary-remote or multi-primary Istio. In that case, you can have multiple Istio control planes governing the network and security of data in respective clusters.

In the below example, there are 4 clusters — 2 in the same network and 2 in different networks. Cluster West-1 and West-2 are in the same network and Istio is installed as the primary-remote type. Note that all the clusters have the same services, A & B, to serve the external traffic.



In this case, to achieve HA, one can set up 4 different Gateway API for each cluster. The configuration of the Gateway resource for cluster:West 2 will be controlled by the Istio control plane in Cluster:West1. Similarly the configuration of Gateway API for Cluster:East2 will be controlled by the Istio control plane of Cluster:East1.

However, since the external traffic will reach the same endpoint (say service A). We must set up a load balancer before the gateway API to receive the traffic and distribute it to all the clusters.

Conclusion

Kubernetes Ingress is already frozen. New features are being added to the Gateway API and thus, DevOps and architects will have to migrate from Ingress to Gateway API, eventually. As we have seen above, Gateway API is much more capable in terms of features and flexibility than the native Ingress. Feel free to get in touch with us if you need help implementing Gateway API in your unique cloud infrastructure.