ISOVALENT

# 🥇 Advanced Gateway API Use Cases

A Hands-on lab by Isovalent

Version 1.2.0

⚠️ **DISCLAIMER**

This document serves as a reference guide summarizing the content and steps of the hands-on lab. It is intended to help users recall the concepts and practices they learned when taking the lab.

Please note that the scripts, tools, and environment configurations required to recreate the lab are not included in this document.

For a complete hands-on experience, please refer to the live lab environment instead.

You can access the lab at https://isovalent.com/labs/cilium-gateway-api-advanced.

The World of Cilium Map at https://labs-map.isovalent.com also lets you access all labs and view your progress through the various learning paths.

# Challenges

## 💡 Cilium Gateway API

Your lab environment is currently being set up. Stay tuned!

This lab is a follow-up to the introductory Cilium Gateway API lab. We highly recommend you do the Cilium Gateway API lab first, if you haven't done it already.

In this one, you will learn about some additional specific use cases for Gateway API:

- HTTP request and response header rewrite
- HTTP mirror, rewrite and redirect
- gRPC routing
- Cross-namespace routing

Please click the arrow on the right to proceed.

## 💬 Resilient Connectivity

Service to service communication must be possible across boundaries such as clouds, clusters, and premises. Communication must be resilient and fault tolerant.

## 🤹 Embedded Envoy Proxy

Cilium already uses Envoy for L7 policy and observability for some protocols, and this same component is used as the sidecar proxy in many popular Service Mesh implementations.

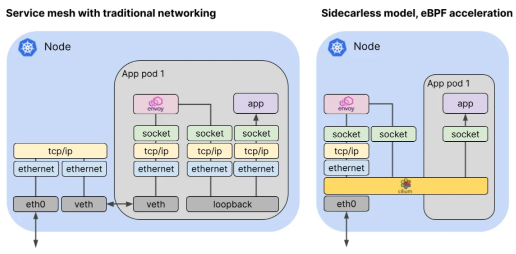
So it's a natural step to extend Cilium to offer more of the features commonly associated with Service Mesh — though contrary to other solutions, without the need for any pod sidecars.

Instead, this Envoy proxy is embedded with Cilium, which means that only one Envoy container is required per node.

## 🐝 eBPF acceleration

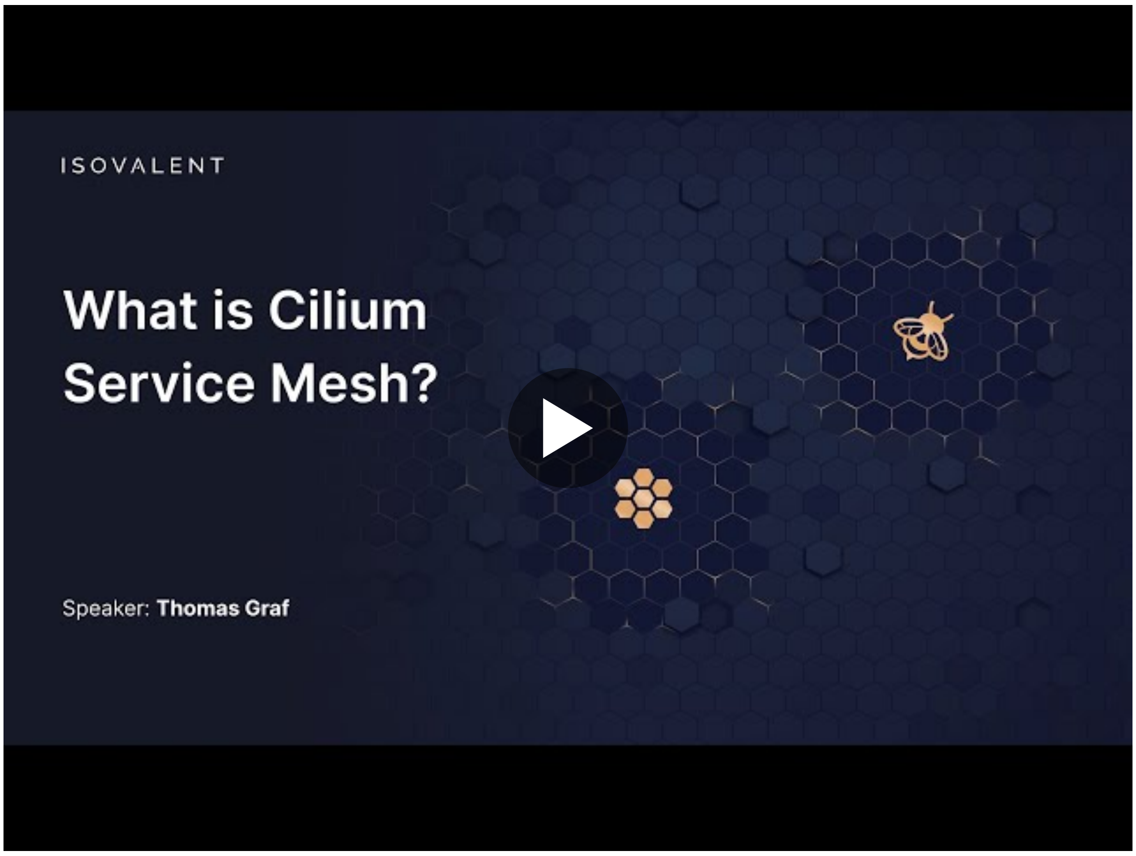
In a typical Service Mesh, all network packets need to pass through a sidecar proxy container on their path to or from the application container in a Pod.

This means each packet traverses the TCP/IP stack three times before even leaving the Pod.



In Cilium Service Mesh, we’re moving that proxy container onto the host and kernel so that sidecars for each application pod are no longer required.

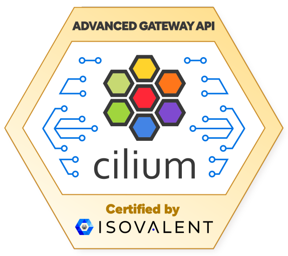
Because eBPF allows us to intercept packets at the socket as well as at the network interface, Cilium can dramatically shorten the overall path for each packet.



## 🏅 Get a Badge!

By completing this lab, you will be able to earn a badge.

Make sure to finish the lab in order to get your badge!



# 🏛️ The Lab Environment

## 🚦 Advanced Gateway API Use Cases

Before we can install Cilium with the Gateway API feature, there are a couple of important prerequisites to know:

- Cilium must be configured with `kubeProxyReplacement` set to **true**.
- CRD (Custom Resource Definition) from Gateway API `must` be installed beforehand.

As part of the lab deployment script, several CRDs were installed. Verify that they are available.

```
kubectl get crd \
  gatewayclasses.gateway.networking.k8s.io \
  gateways.gateway.networking.k8s.io \
  httproutes.gateway.networking.k8s.io \
  referencegrants.gateway.networking.k8s.io \
  tlsroutes.gateway.networking.k8s.io \
  grpcroutes.gateway.networking.k8s.io
```

During the lab deployment, Cilium was installed using the following flags:

```
--set kubeProxyReplacement=true \
--set gatewayAPI.enabled=true
```

Let's have a look at our lab environment and see if Cilium has been installed correctly. The following command will wait for Cilium to be up and running and report its status:

```
cilium status --wait
```

Verify that Cilium was enabled and deployed with the Gateway API features:

```
cilium config view | grep -w "enable-gateway-api "
```

More information about the Gateway API can be found in the previous **Cilium Gateway API** lab and in the Cilium Gateway API deep dive.

Press `Next` to continue.

## 🔛 We need a Load Balancer

The Cilium Service Mesh Gateway API Controller requires the ability to create `LoadBalancer` Kubernetes services.

Since we are using Kind on a Virtual Machine, we do not benefit from an underlying Cloud Provider's load balancer integration.

For this lab, we will use Cilium's own LoadBalancer capabilities to provide IP Address Management (IPAM) and Layer 2 announcement of IP addresses assigned to `LoadBalancer` services.

You can check the Cilium LoadBalancer IPAM and L2 Service Announcement lab to learn more about it.

# 🌐 Gateway API — Deploy Sample App

## 🚀 Deploy an application

First, let's deploy a sample **echo** application in the cluster. The application will reply to the client and, in the body of the reply, will include information about the original request header. We will use this information to illustrate how the Gateway can modify headers and other HTTP parameters.

```
kubectl apply -f echo-servers.yaml
```

Look at the YAML file with the command below. You'll see we are deploying a pod and associated service (`echo-1`).

```
yq echo-servers.yaml
```

Check that the application is properly deployed:

```
kubectl get pods
```

You should see the pod being deployed in the `default` namespace. Wait until it is `Running` (should take 10 to 15 seconds).

Have a quick look at the Service deployed:

```
kubectl get svc
```

Note this Service are only internal-facing (ClusterIP) and therefore there is no access from outside the cluster to these Services.

## 🚪Deploy the Gateway and HTTPRoute

Let's deploy the Gateway and the HTTPRoute with the following manifest:

```
kubectl apply -f gateway.yaml -f http-route.yaml
```

Let's have another look at the Services now that the Gateway has been deployed:

```
kubectl get svc
```

You will see a `LoadBalancer` service named `cilium-gateway-cilium-gw` which was created for the Gateway API.

We now have an IP address also associated with the Gateway:

```
kubectl get gateway
```

Let's retrieve this IP address:

```
GATEWAY=$(kubectl get gateway cilium-gw -o jsonpath='{.statu
s.addresses[0].value}')
echo $GATEWAY
```

Note that this IP address was assigned by Cilium's LB-IPAM (Load-Balancer IP Address Management) feature.

Let's now check that traffic based on the URL path is proxied by the Gateway API.

Check that you can make HTTP requests to that external address:

```
curl --fail -s http://$GATEWAY/echo
```

Note that you can see, in the reply, the headers from the original request. This will be useful in the next task.

Press `Check` to move on.

## 📜 HTTP Header Request Modifier

In this challenge, you will use Cilium Gateway API to modify HTTP headers of HTTP requests.

HTTP header modification is the process of adding, removing, or modifying HTTP headers in incoming requests. The Cilium Gateway API lets users easily customize incoming traffic to meet their specific needs.

# 🌐 HTTP Request Header Modifier

With this functionality, Cilium Gateway API lets us add, remove or edit HTTP Headers of incoming traffic.

This is best validated by trying without and with the functionality. We'll use the same echo servers.

## 🌐 Deploy the HTTPRoute

Let's deploy an `HTTPRoute` resource with the following manifest (we are using the same Gateway deployed in the previous task).

```
kubectl apply -f echo-header-http-route.yaml
```

Let's review it:

```
yq echo-header-http-route.yaml
```

Notice how the file has commented lines (we will uncomment them later).

Let's retrieve the Gateway IP address:

```
GATEWAY=$(kubectl get gateway cilium-gw -o jsonpath='{.statu
s.addresses[0].value}')
echo $GATEWAY
```

Make HTTP requests to that external address:

```
curl --fail -s http://$GATEWAY/cilium-add-a-request-header
```

In the reply, you should see the original request header. They should look similar to this:

```
Request Headers:
        accept=*/*
        host=172.18.255.200
        user-agent=curl/7.81.0
        x-forwarded-proto=http
        x-request-id=5711b7ce-a1b3-44b1-8c12-01d3c7678b97
```

Head to the **</> Editor** and uncomment the commented lines of `echo-header-http-route.yaml` (lines 14 to 19).

The lines you are uncommenting are pretty self-explanatory: you are using a filter of the type `RequestHeaderModifier` to add a specific header, with a name and a value.

Re-apply the HTTPRoute in the **>_ Terminal 1** tab:

```
kubectl apply -f echo-header-http-route.yaml
```

Let's now check that the header is modified by Cilium Gateway API:

Make a curl HTTP request to that address again:

```
curl --fail -s http://$GATEWAY/cilium-add-a-request-header
```

You should see, in the `Request Headers` section of the reply, that the header `my-cilium-header-name=my-cilium-header-value` has been added to the HTTP request.

Note that you can also remove or edit HTTP request headers sent from the client.

In the next task, we will use a feature to do the same, for HTTP response headers.

## 🔍 Observability (optional)

You can use the observability platform Hubble in `>_ Terminal 2` to observe the traffic, filtering on the specific HTTP path you adding with Gateway API:

```
hubble observe --http-path "/cilium-add-a-request-header"
```

You should see an output such as:

```
Oct 31 10:46:02.231: 172.18.0.1:52008 (ingress) -> default/
echo-1-78b66687b5-ttc4d:8080 (ID:7001) http-request
FORWARDED (HTTP/1.1 GET http://172.18.255.201/cilium-add-a-
request-header)
Oct 31 10:46:02.231: 172.18.0.1:52008 (ingress) <- default/
echo-1-78b66687b5-ttc4d:8080 (ID:7001) http-response
FORWARDED (HTTP/1.1 200 0ms (GET http://172.18.255.201/
cilium-add-a-request-header))
```

You can see how traffic was sent through the Cilium L7 Ingress (which implements Gateway API) and that you can use Hubble to observe traffic using Layer 7 filters such as HTTP Path.

## 📝 HTTP Response Header Rewrite

In this challenge, we will test HTTP response header rewrite using Cilium Gateway API.

Just like editing request headers can be useful, the same goes for response headers. For example, it allows teams to add/remove cookies for only a certain backend, which can help in identifying certain users that were redirected to that backend previously.

Another potential use case could be when you have a frontend that needs to know whether it's talking to a stable or a beta version of the backend server, in order to render different UI or adapt its response parsing accordingly.

# 🌐 HTTP Response Header Modifier

## 🌐 Response Header Modification

Let's deploy the HTTPRoute with the following manifest:

```
kubectl apply -f response-header-modifier-http-route.yaml
```

Let's review it in details:

```
yq response-header-modifier-http-route.yaml
```

Notice how this time, the header's response is modified, using the `type: ResponseHeaderModifier` filter.

We are going to add 3 headers in one go.

Let's retrieve the Gateway IP address:

```
GATEWAY=$(kubectl get gateway cilium-gw -o jsonpath='{.statu
s.addresses[0].value}')
echo $GATEWAY
```

Check that you can make HTTP requests to that external address:

```
curl --fail -s http://$GATEWAY/multiple
```

Note the body of the packet includes details about the original request.

If you run the following command:

```
curl --fail -s http://$GATEWAY/multiple | grep "Request
Headers" -A 10
```

You should only see this in the reply:

```
root@server:~# curl --fail -s http://$GATEWAY/multiple |
grep "Request Headers" -A 11
Request Headers:
        accept=*/*
        host=172.18.255.200
        user-agent=curl/7.81.0
        x-envoy-expected-rq-timeout-ms=15000
        x-envoy-internal=true
        x-forwarded-for=172.18.0.1
        x-forwarded-proto=http
        x-request-id=ef0ca229-ed5e-4fb6-a83d-217b27153655

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

To show the headers of the response, we can run `curl` in verbose mode:

```
curl -v --fail -s http://$GATEWAY/multiple
```

You should see, in the fields starting with <, the response header. You should see the headers added by the Gateway API to the response:

```
< x-header-add-1: header-add-1
< x-header-add-2: header-add-2
< x-header-add-3: header-add-3
```

Again, you can see how simple it is to use Cilium Gateway API to modify HTTP traffic - incoming requests or outgoing responses.

In the next task, we will look at how the Cilium Gateway API can mirror HTTP Traffic.

## 📝 HTTP Traffic Mirroring

In this challenge, we will mirror HTTP traffic using Cilium Gateway API.

You can mirror traffic destined for a backend to another backend.

This is useful when you want to introduce a v2 of a service or simply for troubleshooting and analytics purposes.

# 🌐 Gateway API — HTTP Mirroring

In this task, we will use the Gateway to mirror traffic destined for a backend to another backend.

This is useful when you want to introduce a v2 of a service or simply for troubleshooting and analytics purposes.

We will be using a different demo app. This demo app will deploy some Pods and Services - `infra-backend-v1` and `infra-backend-v2`. We will mirror the traffic bound for `infra-backend-v1` to `infra-backend-v2`.

Verify that the demo app has been properly deployed:

```
kubectl get -f demo-app.yaml
```

We have prepared an HTTPRoute manifest to mirror HTTP requests to a different backend. Mirroring traffic to a different backend can be useful for troubleshooting, analysis and observability. Note that, while we may mirror traffic to another backend, we will ignore the responses from said backend.

Deploy the HTTPRoute:

```
kubectl apply -f http-mirror-route.yaml
```

Inspect the spec for the route:

```
yq .spec http-mirror-route.yaml
```

Note that the `filters` block is currently commented.

Retrieve the Gateway IP address:

```
GATEWAY=$(kubectl get gateway cilium-gw -o jsonpath='{.status.addresses[0].value}')
echo $GATEWAY
```

Make a request to the gateway:

```
curl -s http://$GATEWAY/mirror | jq
```

Expect an output such as:

```
{
 "path": "/mirror",
 "host": "172.18.255.201",
 "method": "GET",
 "proto": "HTTP/1.1",
 "headers": {
  "Accept": [
   "*/*"
  ],
  "User-Agent": [
   "curl/7.81.0"
  ],
  "X-Envoy-Expected-Rq-Timeout-Ms": [
   "15000"
  ],
  "X-Envoy-Internal": [
   "true"
  ],
  "X-Forwarded-For": [
   "172.18.0.1"
  ],
  "X-Forwarded-Proto": [
   "http"
  ],
  "X-Request-Id": [
   "e6f4ed16-ba71-448a-85af-3e2539326c87"
  ]
 },
 "namespace": "default",
 "ingress": "",
 "service": "",
 "pod": "infra-backend-v1-6b94cf477-fvl2v"
```

Check the **>_ 📜 Backend Logs** tab. This tab watches the access logs for both `infra-backend-v1` and `infra-backend-v2` pods.

You will only see logs in the left panel.

## 🪞 Deploy the mirrored route

Using the **</> Editor**, edit the `http-mirror-route.yaml` manifest and uncomment the `filters` section (lines 14-19) in the manifest, then apply it in the **>_ Terminal**:

```
kubectl apply -f http-mirror-route.yaml
```

Make a new request to the gateway:

```
curl -s http://$GATEWAY/mirror | jq
```

Has the mirroring actually happened?

Check the **>_ 📜 Backend Logs** tab again.

You will see logs on both sides of the split screen, showing that the traffic was indeed mirrored.

Press `Check` to move on to the next task, where you will be rewriting the HTTP URL.

## 📝 HTTP URL Rewrite

In this challenge, we will rewrite the URL used in HTTP traffic using Cilium Gateway API.

Rewrites modify the URL used of a client request before proxying it.

# 🌐 Gateway API — HTTP Rewrite

In this task, we will use the Gateway to rewrite the path used in the HTTP requests.

Let's start by, again, retrieving the Gateway IP address:

```
GATEWAY=$(kubectl get gateway cilium-gw -o jsonpath='{.statu
s.addresses[0].value}')
echo $GATEWAY
```

We have prepared a HTTPRoute to rewrite the URL in HTTP requests.

Let's apply it.

```
kubectl apply -f http-rewrite-route.yaml
```

## 🌐 HTTP Path Rewrite

Let's review the HTTPRoute.

```
yq http-rewrite-route.yaml
```

Let's start with the first rule.

```
- matches:
    - path:
        type: PathPrefix
        value: /prefix/one
    filters:
    - type: URLRewrite
      urlRewrite:
        path:
          type: ReplacePrefixMatch
          replacePrefixMatch: /one
    backendRefs:
    - name: infra-backend-v1
      port: 8080
```

With this manifest, we will replace the `/prefix/one` in the request URL to `/one`.

Let's now check that traffic based on the URL path is proxied and altered by the Gateway API:

Make HTTP requests to that external address and path:

```
curl -s http://$GATEWAY/prefix/one | jq
```

Expect to see this reply. The request is received by an echo server that copies the original request and sends the reply back in the body of the packet.

```
{
 "path": "/one",
 "host": "172.18.255.200",
 "method": "GET",
 "proto": "HTTP/1.1",
 "headers": {
  "Accept": [
   "*/*"
  ],
  "User-Agent": [
   "curl/7.81.0"
  ],
  "X-Envoy-Internal": [
   "true"
  ],
  "X-Envoy-Original-Path": [
   "/prefix/one"
  ],
  "X-Forwarded-For": [
   "172.18.0.1"
  ],
  "X-Forwarded-Proto": [
   "http"
  ],
  "X-Request-Id": [
   "c0798e09-49b1-4ea8-83c0-161cbe497884"
  ]
 },
 "namespace": "default",
 "ingress": "",
 "service": "",
 "pod": "infra-backend-v1-6fdcb7fdd6-8fcqk"
```

What does it tell us? The Gateway changed the original request from `"/prefix/one"` to `"/one"` (see `"path"` in the output above).

Note that, as we use Envoy for L7 traffic processing, that Envoy also adds the information about the original path in the packet (see `"X-Envoy-Original-Path"`).

Note that rewriting the URL in HTTP Requests can also be combined with some features we explored in a previous lab. For example, you can add custom HTTP headers while also rewriting the URL path.

Press `Check` to move on to the next task.

## 📝 HTTP Traffic Redirect

In this challenge, we will redirect clients to a different URL using Cilium Gateway API.

You can customize the path, hostname and the HTTP redirection code (such as 301 or 302) in your redirection messages.

This is useful during a temporary or permanent migration of an application.

# 🌐 Gateway API — HTTP Redirect

In this task, we will redirect HTTP traffic!

Let's deploy the HTTPRoute - we will review it, section by section, throughout this task.

```
kubectl apply -f redirect-route.yaml
```

Let's retrieve the Gateway IP address:

```
GATEWAY=$(kubectl get gateway cilium-gw -o jsonpath='{.statu
s.addresses[0].value}')
echo $GATEWAY
```

### 🔀 HTTP Path Redirect

Let's now check that traffic based on the URL path is proxied and altered by the HTTPRoute created above:

```
yq '.spec.rules[0]' redirect-route.yaml
```

which will show you:

```
matches:
  - path:
      type: PathPrefix
      value: /original-prefix
  filters:
  - type: RequestRedirect
    requestRedirect:
      path:
        type: ReplacePrefixMatch
        replacePrefixMatch: /replacement-prefix
```

Make HTTP requests to that external address and path:

```
curl -l -v http://$GATEWAY/original-prefix
```

Notice we use `-l` in the curl request to follow the redirects (by default, curl will not follow redirects). Notice we use the verbose option of curl to see the response headers.

Notice that, in the reply, you get the name of the pod that received the query. For example:

```
< HTTP/1.1 302 Found
< location: http://172.18.255.200:80/replacement-prefix
< date: Tue, 05 Sep 2023 12:03:41 GMT
< server: envoy
< content-length: 0
<
* Connection #0 to host 172.18.255.200 left intact
```

The `location` is used in Redirect messages to tell the client where to go. As you can see, the client is redirected to `http://172.18.255.200:80/replacement-prefix`.

Only the path prefix was modified.

### ✨ Redirect to new hostname and new prefix

You can also direct the client to a different host. Check the second rule:

```
yq '.spec.rules[1]' redirect-route.yaml
```

which will show you this specification:

```
matches:
- path:
    type: PathPrefix
    value: /path-and-host
filters:
- type: RequestRedirect
  requestRedirect:
    hostname: example.org
    path:
      type: ReplacePrefixMatch
      replacePrefixMatch: /replacement-prefix
```

Make HTTP requests to that external address and path:

```
curl -l -v http://$GATEWAY/path-and-host
```

Expect a reply such as this one:

```
< HTTP/1.1 302 Found
< location: http://example.org:80/replacement-prefix
< date: Tue, 05 Sep 2023 12:05:30 GMT
< server: envoy
< content-length: 0
<
* Connection #0 to host 172.18.255.200 left intact
```

As you can see, the client is redirected to `http://example.org:80/replacement-prefix`.

Both the hostname and the path prefix were modified.

## 🔢 Redirect - new status code and new prefix

Next, you can also modify the status code. By default, as you can see, the redirect status code is `302`. It means that the resources have been moved temporarily.

To indicate that the resources the client is trying to access have moved permanently, you can use the status code `301`. You can also combine it with the prefix replacement.

Check the third rule:

```
yq '.spec.rules[2]' redirect-route.yaml
```

which is:

```
matches:
- path:
    type: PathPrefix
    value: /path-and-status
filters:
- type: RequestRedirect
  requestRedirect:
    path:
      type: ReplacePrefixMatch
      replacePrefixMatch: /replacement-prefix
    statusCode: 301
```

Make HTTP requests to that external address and path:

```
curl -l -v http://$GATEWAY/path-and-status
```

Expect a reply such as this one:

```
< HTTP/1.1 301 Moved Permanently
< location: http://172.18.255.200:80/replacement-prefix
< date: Tue, 05 Sep 2023 12:07:02 GMT
< server: envoy
< content-length: 0
<
* Connection #0 to host 172.18.255.200 left intact
```

As you can see, the status code returned is `301 Moved Permanently` and the client is redirected to `http://172.18.255.200:80/replacement-prefix`.

## 🔒 Redirect - from HTTP to HTTPS and new prefix

Finally, we can also change the scheme and tell the client to use HTTPS instead of HTTP for example.

You can achieve that with the fourth rule:

```
yq '.spec.rules[3]' redirect-route.yaml
```

which returns:

```
matches:
- path:
    type: PathPrefix
    value: /scheme-and-host
filters:
- type: RequestRedirect
  requestRedirect:
    hostname: example.org
    scheme: "https"
```

Make HTTP requests to that external address and path:

```
curl -l -v http://$GATEWAY/scheme-and-host
```

Expect a reply such as this one:

```
< HTTP/1.1 302 Found
< location: https://example.org:443/scheme-and-host
< date: Tue, 05 Sep 2023 12:13:31 GMT
< server: envoy
< content-length: 0
<
* Connection #0 to host 172.18.255.200 left intact
```

As you can see, the client initally tried to connect via HTTP and is redirected to `https://example.org:443/scheme-and-host`.

Press `Check` to move on to the next task.

## 🔀 Cross Namespace Support

The Gateway API has core support for cross Namespace routing. This is useful when more than one user or team is sharing the underlying networking infrastructure, yet control and configuration must be segmented to minimize access and fault domains.

Gateways and Routes can be deployed into different Namespaces and Routes can attach to Gateways across Namespace boundaries. This allows user access control to be applied differently across Namespaces for Routes and Gateways, effectively segmenting access and control to different parts of the cluster-wide routing configuration.

The ability for Routes to attach to Gateways across Namespace boundaries are governed by Route attachment. Route attachment is explored in this lab and demonstrates how independent teams can safely share the same Gateway.

## 📙 Route Attachment

Route attachment is an important concept that dictates how Routes attach to Gateways and program their routing rules. It is especially relevant when there are Routes across Namespaces that share one or more Gateways.

Gateway and Route attachment is bidirectional - attachment can only succeed if the Gateway owner and Route owner both agree to the relationship.

Gateways support attachment constraints which are fields on Gateway listeners that restrict which Routes can be attached.

Gateways support Namespaces and Route types as attachment constraints. Any Routes that do not meet the attachment constraints are not able to attach to that Gateway. Similarly, Routes explicitly reference Gateways that they want to attach to through the Route's `parentRef` field.

Together these create a handshake between the infra owners and application owners that enables them to independently define how applications are exposed through Gateways.

This is effectively a policy that reduces administrative overhead. App owners can specify which Gateways their apps should use and infra owners can constrain the Namespaces and types of Routes that a Gateway accepts.

# 🌐 Cross-namespace & Shared Gateway API

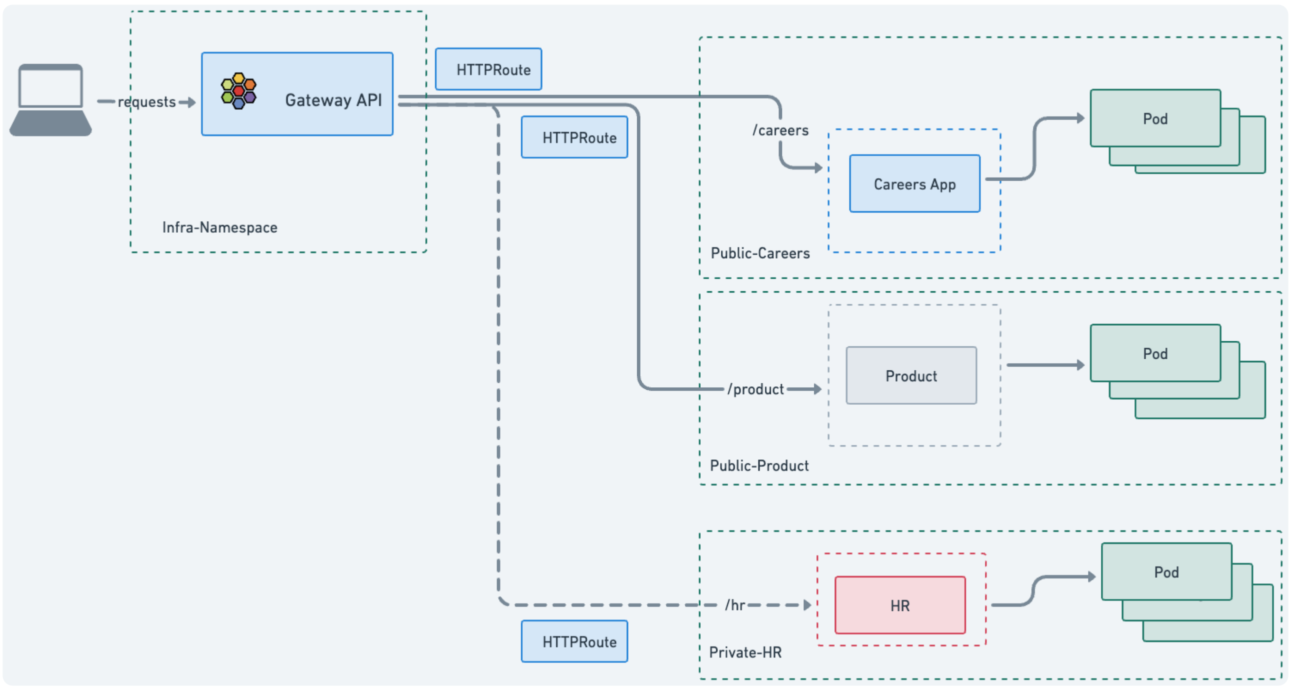
## 🚪Cross-Namespaces at ACME

In this task, we will consider a fictional ACME company and three different business units within ACME. Each of them has its own environment, application and namespace.

- The recruiting team has a public-facing `careers` app where applicants can submit their CV.
- The product team has a public-facing `product` app where prospective customers can find out more the ACME product.
- The HR team has an internal-facing `hr` app storing private employee details.

Each app is deployed in its own Namespace. Because `careers` and `product` are both public-facing apps, the Security team approved the use of a shared Gateway API. A benefit of a shared Gateway API is that platform and security teams could control centrally the Gateway API, including its certificate management. In the public cloud, it would also reduce the cost (a Gateway API per app would require a public IP and a cloud load balancer, which are not free resources).

However, the Security team does not want the HR details to be exposed and accessible from outside the cluster and therefore does not approve a HTTPRoute attachment from the `hr` namespace to the Gateway.



## 🚪Cross-Namespaces Gateway

When this task was initialized, four namespaces were created: a shared `infra-ns` namespace and namespaces for each of the three business units.

Verify with:

```
kubectl get ns --show-labels \
  infra-ns careers product hr
```

Notice that `product` and `careers` both have the `shared-gateway-access=true` label, but `hr` does not.

Let's deploy the Gateway and the HTTPRoutes with the following manifest:

```
kubectl apply -f cross-namespace.yaml
```

Review it:

```
yq cross-namespace.yaml
```

By now, you should be familiar with the vast majority of the manifest. Here are some of the differences. First, in the Gateway definition, notice it has been deployed in the `infra-ns` namespace:

```
metadata:
  name: shared-gateway
  namespace: infra-ns
```

This section might also look unfamiliar:

```
    allowedRoutes:
      namespaces:
        from: Selector
        selector:
          matchLabels:
            shared-gateway-access: "true"
```

This Gateway uses a Namespace selector to define which HTTPRoutes are allowed to attach. This allows the infrastructure team to constrain who —or which apps— can use this Gateway by allowlisting a set of Namespaces.

Only Namespaces which are labelled `shared-gateway-access: "true"` will be able to attach their Routes to the shared Gateway.

In the `HTTPRoute` definitions, notice how we refer to the `shared-gateway` in the `parentRefs`. We specify the Gateway we want to attach to and the Namespace it is in.

Let's test the HTTPRoutes. First, let's fetch the Gateway IP:

```
GATEWAY=$(kubectl get gateway shared-gateway -n infra-ns -o
jsonpath='{.status.addresses[0].value}')
echo $GATEWAY
```

Now, let's connect to the `product` and `careers` Services:

```
curl -s -o /dev/null -w "%{http_code}\n" http://$GATEWAY/
product
```

This command should return a `200` status code.

```
curl -s -o /dev/null -w "%{http_code}\n" http://$GATEWAY/
careers
```

This command should also return a `200` status code.

Let's try to connect to the `hr` Service:

```
curl -s -o /dev/null -w "%{http_code}\n" http://$GATEWAY/hr
```

It should return a `404`. Why?

The HTTPRoute in the `hr` Namespace with a parentRef for `infra-ns/shared-gateway` would be ignored by the Gateway because the attachment constraint (Namespace label) was not met.

Verify with the following commands by checking the status of the HTTPRoutes:

```
echo "Product HTTPRoute Status"
kubectl get httproutes.gateway.networking.k8s.io -n product
-o jsonpath='{.items[0].status.parents[0].conditions[0]}' |
jq
echo "Careers HTTPRoute Status"
kubectl get httproutes.gateway.networking.k8s.io -n careers
-o jsonpath='{.items[0].status.parents[0].conditions[0]}' |
jq
echo "HR HTTPRoute Status"
kubectl get httproutes.gateway.networking.k8s.io -n hr -o
jsonpath='{.items[0].status.parents[0].conditions[0]}' | jq
```

The first two should be "Accepted HTTPRoute" while the last one should have been rejected (its status should be `False` and the message should start with `HTTPRoute is not allowed to attach to this Gateway`).

This feature provides engineers with multiple options: either have a dedicated Gateway API per Namespace or per app if required or alternatively use shared Gateway API for centralized management and to reduce potential costs.

In the next challenge, you will see how Gateway API can be used to route gRPC traffic!

## 🌐 Deploying a gRPC Route

While HTTP is still the king of protocols in the web, gRPC is increasingly used, in particular for its low latency and high throughput capabilities.

Let's see how we can deploy a Kubernetes gRPC Route with Cilium Gateway API for a gRPC application using Cilium!

## 🚀 Deploy a gRPC Application

In this challenge, we will deploy a sample gRPC application, which consists of multiple services such as:

- 📧 `email`
- 🛒 `checkout and cart`
- 💡 `recommendation`
- 👨‍💻 `frontend`
- 💳 `payment`
- 🚚 `shipping`
- 💱 `currency`
- 📦 `productcatalog`

In this challenge, we will set up a `gRPCRoute` with two path prefixes:

- `/hipstershop.ProductCatalogService` pointing to the `productcatalog` service
- `/hipstershop.CurrencyService` pointing to the `currency` service

# 🌐 Gateway API — gRPC Example

## 🚀 Deploy a gRPC-based application

For this demo we will use GCP's microservices demo app.

Install the app with the following command.

```
kubectl apply -f /opt/gcp-microservices-demo.yml
```

Since gRPC is binary-encoded, you also need the proto definitions for the gRPC services in order to make gRPC requests. Download this for the demo app:

```
curl -o demo.proto https://raw.githubusercontent.com/
GoogleCloudPlatform/microservices-demo/main/protos/demo.proto
```

## 🚪Deploy the gRPCRoute

You'll find the gRPC definition in `grpc-route.yaml`:

```
yq grpc-route.yaml
```

This defines paths for requests to be routed to the `productcatalogservice` and `currencyservice` microservices.

Let's deploy it:

```
kubectl apply -f grpc-route.yaml
```

Let's retrieve the load balancer's IP address:

```
GATEWAY=$(kubectl get gateway cilium-gw -o jsonpath='{.statu
s.addresses[0].value}')
echo $GATEWAY
```

## 🌐 Make gRPC requests to backend services

Before verifying gRPC routing with the Cilium Gateway API, let's verify that the app is ready:

```
kubectl rollout status deploy/emailservice
kubectl rollout status deploy/checkoutservice
kubectl rollout status deploy/recommendationservice
kubectl rollout status deploy/frontend
kubectl rollout status deploy/paymentservice
kubectl rollout status deploy/productcatalogservice
kubectl rollout status deploy/cartservice
kubectl rollout status deploy/loadgenerator
kubectl rollout status deploy/currencyservice
kubectl rollout status deploy/shippingservice
kubectl rollout status deploy/redis-cart
kubectl rollout status deploy/adservice
```

Let's try to access the currency service of the application, which lists the currencies the shopping app supports:

```
grpcurl -plaintext -proto ./demo.proto $GATEWAY:80
hipstershop.CurrencyService/GetSupportedCurrencies | jq
```

Also try accessing the product catalog service with:

```
grpcurl -plaintext -proto ./demo.proto $GATEWAY:80
hipstershop.ProductCatalogService/ListProducts | jq
```

You should see, in the output, a collection of products in JSON, including a candle holder, a hairdryer and sunglasses!

In the next task, you will see how we can use Gateway API not just for Ingress use cases but for Layer 7 traffic management within the cluster.

## Internal Layer 7 Traffic Management

In Kubernetes, the `Service` resource type lets you load-balance traffic inside the cluster (East-West). However, the load-balancing options are very limited: only L3/L4, optionally with topology hints.

Achieving layer 7 load-balancing and advanced routing typically requires deploying a service mesh solution to the cluster.

This usually means using non-standard resource types specific to the service mesh solution. Could there be a way to achieve this result without an additional component, and using standard resources?

## ɣ The GAMMA initiative

The GAMMA initiative is a dedicated workstream within the Gateway API subproject.

GAMMA stands for Gateway API for Mesh Management and Administration and its goal is to define how Gateway API can be used to configure a service mesh, with the intention of making minimal changes to Gateway API and always preserving the role-oriented nature of Gateway API.

## ɣ The GAMMA initiative

In Gateway API v1.0, GAMMA supports adding extra HTTP routing to Services by binding a HTTPRoute to a Service as a parent (as opposed to the north/south Gateway API usage of binding a HTTPRoute to a Gateway as a parent, as you've seen so far throughout this lab).

GAMMA provides a standard API for Layer 7 traffic management capabilities within the cluster.

Let's find out more in this challenge!

# 🌐 Gateway API — Gamma Example

## 🚀 Deploy a sample application

For this demo we will use a very simple echo application, which we will deploy in the `gamma` namespace

Install the app with the following command.

```
kubectl apply -f gamma-manifest.yaml
```

Wait until all pods are running:

```
kubectl -n gamma get pods,svc
```

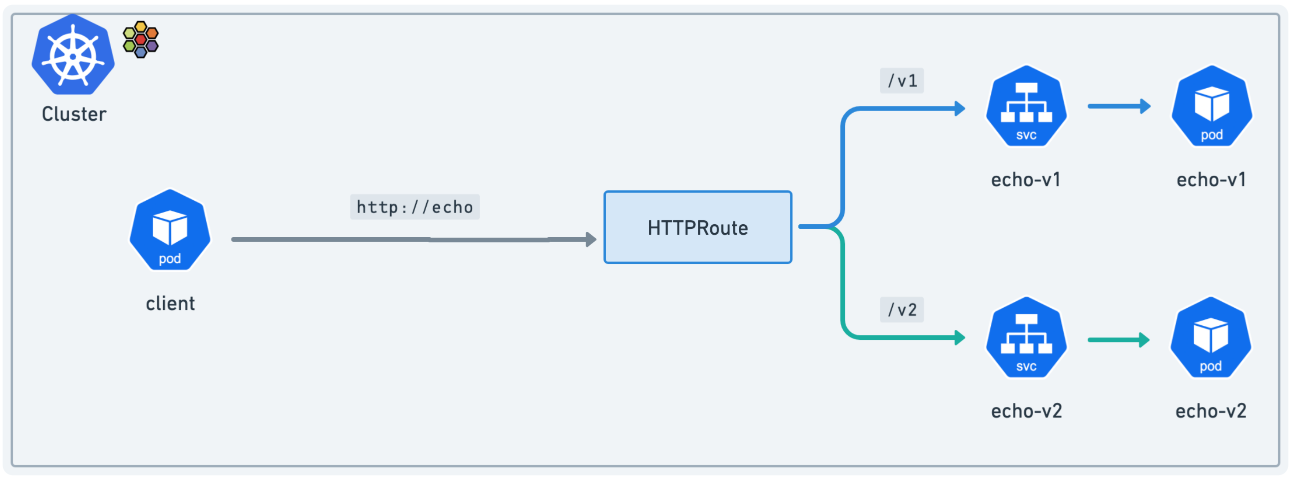
## 🚪 Deploy the East/West HTTPRoute

Let's deploy a `HTTPRoute` in the `gamma` namespace.

Check its definition in `gamma-route.yaml`:

```
yq gamma-route.yaml
```

You will notice that, instead of attaching a route to a (North/South) Gateway like we did in previous challenges, we are attaching the route to a parent Service, called `echo`, using the `parentRefs` field.



Traffic bound to this parent service will be intercepted by Cilium and routed through the per-node Envoy proxy.

Note how we will forward traffic to the `/v1` path to the `echo-v1` service and the same for v2. This is how we can, for example, do a/b or green/blue canary testing for internal apps.

Let's deploy it:

```
kubectl apply -f gamma-route.yaml
```

## 🌐 Verifying East-West L7 traffic Management

Unlike the previous tasks where, from outside the cluster, we accessed a service inside the cluster through the North/South Gateway, this time we will make the request from a client inside the cluster to a service also living in the cluster (East-West) traffic.

Let's verify our cluster client is ready:

```
kubectl get -n gamma pods client
```

Let's try to access the `http://echo/v1` from our client. The echo Pod for `echo-v1` will reply with information, including its own hostname.

```
kubectl -n gamma exec -it client -- curl http://echo/v1
```

Expect the last line in the reply to follow this format:

```
Hostname=echo-v1-*********-*****
```

Let's now access the `http://echo/v2` from our client. This time, the traffic will be forward to the echo Pod serving the `echo-v2` Service. Let's verify that traffic was received by the `echo-v2` pod by filtering with `grep`:

```
kubectl -n gamma exec -it client -- curl http://echo/v2
```

As you can see, using the same API and logic as with the Gateway API, we're able to do path-based routing for east-west traffic within the cluster.

Let's explore another use case.

## 🚪 Load-Balance East-West

We explored this use case in the first Gateway API lab where we did some traffic splitting across 2 services. Again, using the same API, we can now do it within the cluster for east-west traffic.

Let's update the gamma HTTPRoute with the following manifest:

```
kubectl apply -f load-balancing-http-route.yaml
```

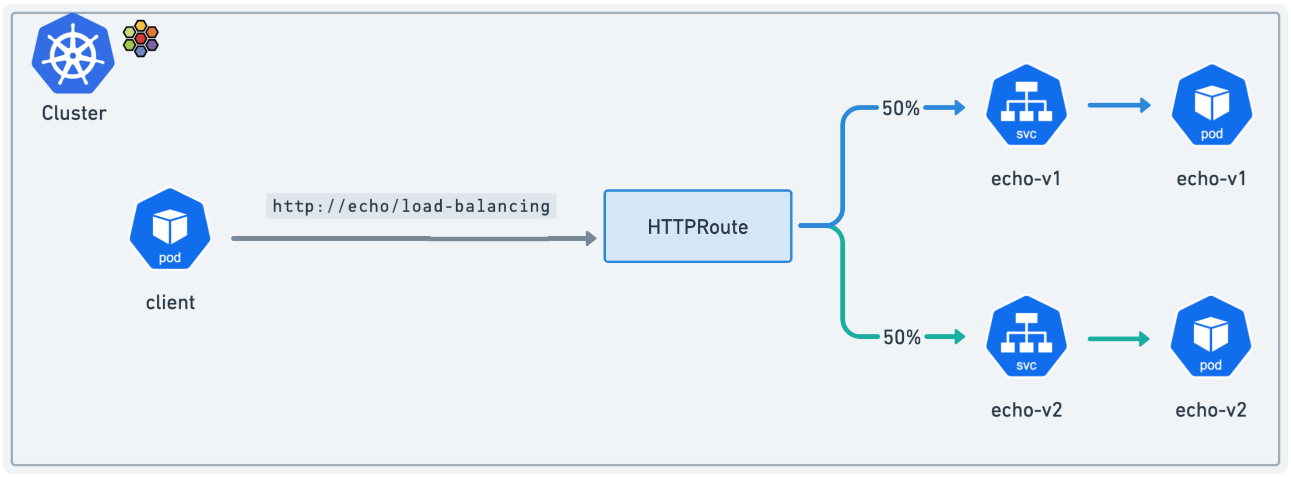
Let's review the `HTTPRoute` manifest.

```
yq load-balancing-http-route.yaml
```

This manifest adds a rule with a simple L7 proxy route: for HTTP traffic with a path starting with `/load-balancing`, forward the traffic over to the `echo-v1` and `echo-v2` Services.

```
backendRefs:
  - kind: Service
    name: echo-v1
    port: 80
    weight: 50
  - kind: Service
    name: echo-v2
    port: 80
    weight: 50
```

Notice the even 50/50 weighing.



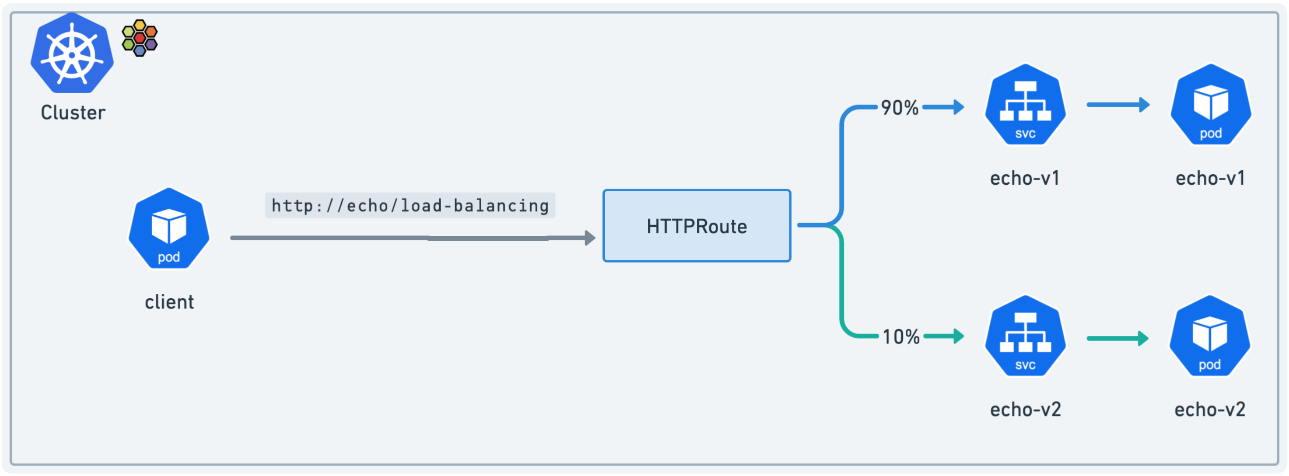
Let's double check that traffic is evenly split across the two services by running a loop and counting the requests. Run the following script.

```
kubectl -n gamma exec -it client -- bash -c '
for _ in {1..500}; do
  curl -s -k "http://echo/load-balancing" >>
curlresponses.txt;
done
grep -o "Hostname=echo-v1" curlresponses.txt | sort | uniq -
c
grep -o "Hostname=echo-v2" curlresponses.txt | sort | uniq -
c
'
```

Verify that the responses have been (more or less) evenly spread.

## 🔢 90/10 Traffic Split

This time, we will be applying a different weight.



Using the **</> Editor** tab, edit the `load-balancing-http-route.yaml` file. Replace the weights from 50 for both `echo-v1` and `echo-v2` to 90 for `echo-v1` and 10 for `echo-v2`.

Then go back to the **>_ Terminal** tab and apply the manifest again:

```
kubectl apply -f load-balancing-http-route.yaml
```

Let's run the test script again and count the responses:

```
kubectl -n gamma exec -it client -- bash -c '
for _ in {1..500}; do
  curl -s -k "http://echo/load-balancing" >>
curlresponses9010.txt;
done
grep -o "Hostname=echo-v1" curlresponses9010.txt | sort |
uniq -c
grep -o "Hostname=echo-v2" curlresponses9010.txt | sort |
uniq -c
'
```

Verify that the responses are spread with about 90% of them to `echo-1` and about 10% of them to `echo-2`.

## ⏳ Timeouts

HTTPRoutes support timeouts as an experimental feature. Let's apply that to the `/v1` path of the `gamma-route` we deployed earlier.

First, check the response headers of the service:

```
kubectl -n gamma exec -it client -- curl http://echo/v1
```

There is no header mentioning timeouts at this point.

Let's add a timeout of 10 ms to the route. Using the **</> Editor** tab, edit the `load-balancing-http-route.yaml` file. Edit the first rule, which matches `/v1` to add a `timeouts` section with `request: 10ms`.

The first matches section should now look like this:

```
    - matches:
      - path:
          type: Exact
          value: /v1
      backendRefs:
      - name: echo-v1
        port: 80
      timeouts:
        request: 10ms
```

Then go back to the **>_ Terminal** tab and apply the manifest again:

```
kubectl apply -f load-balancing-http-route.yaml
```

Now, check the service again:

```
kubectl -n gamma exec -it client -- curl http://echo/v1
```

You should see a new Envoy header for the timeout setting:

```
RequestHeader=X-Envoy-Expected-Rq-Timeout-Ms:10
```

Edit the file again in the **</> Editor**, set the timeout to `1ms` and apply the manifest again in the **>_ Terminal**:

```
kubectl apply -f load-balancing-http-route.yaml
```

Finally, check the service once more:

```
kubectl -n gamma exec -it client -- curl http://echo/v1
```

Given the very low threshold, you should now get a timeout most of the time (try multiple times if you don't):

```
upstream request timeout
```

Press Check to move on to the next task.

In the next task, you will take a short quiz before attempting the final exam challenge!

# ❓ Final Quiz

**Select all answers that apply.**

**References:**

- **Gateway API cross-namespace specs**
- **Blog Post on Gateway API**

With the Gateway API, you can modify HTTP request headers but not HTTP response headers.

Cross-namespace attachment is bi-directional.

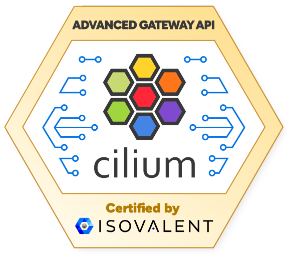
To do traffic splitting into your cluster, you can adjust the weights set to the Services.

With the Gateway API, you can add a HTTP header to your request but you cannot remove or edit them.

## 🏆 Final Exam Challenge

This last challenge is an exam that will allow you to win a badge.

After you complete the exam, you will receive an email from Credly with a link to accept your badge. Make sure to finish the lab!



# 🥋 Exam Challenge

## 📝 Exam Instructions

For this final challenge, you will need to use two of the Gateway API features explored in this lab.

The task requires you to set the value of the `x-request-id` header to the value `exam-header-value`. This should only apply to HTTP Requests bound to an `exam` namespace that can only be reached via the shared Gateway created earlier (on the `exam` path).

- A namespace `exam` was created in the background.
- An `echoserver-exam` Deployment and an `echo-exam` Service have also been deployed in the background.
- A template `HTTPRoute` has been pre-created in the background (`exam-httproute.yaml`). Feel free to use the `</> Editor` to modify it.
- You will need to update the XXXX fields with the correct values.
- Make sure you apply the manifests.
- The final exam script will check for the value of `curl --fail -s http://$GATEWAY/exam | jq -r '.request.headers."x-request-id"'`, with $GATEWAY the IP Address assigned to the Gateway,. If the value returned is `exam-header-value`, you will have successfully completed the lab.

**Notes and tips:**

- You may find this Gateway API documentation useful.
- Feel free to watch the Gateway API Demo Playlist on YouTube if you're stuck.
- The previous Gateway configurations are still available in the `/root` directory .

Good luck!

## 🎓 You've done it!

On completion of this lab, you will receive a badge. Feel free to share your achievement on social media!

We hope that this lab gave you a deeper understanding of the various Cilium Gateway API use cases. Press Next to go and take your next lab.