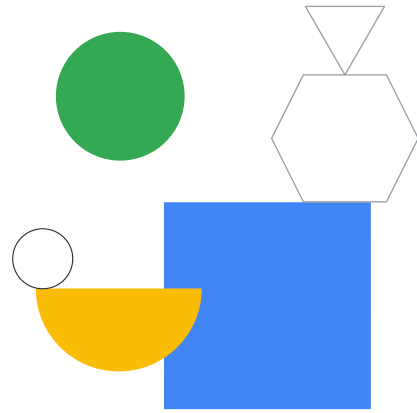



Multi-cluster Concepts on Anthos



Welcome to Multi-cluster Concepts on Anthos.




Today's agenda


- 01 Anthos fleets
- 02 Fleet networking
- 03 Multi-cluster Services
- 04 Multi-cluster Gateway

Google Cloud

Here is our agenda for the module.



Today's agenda



- 01 [Anthos fleets](#)
- 02 Fleet networking
- 03 Multi-cluster Services
- 04 Multi-cluster Gateway

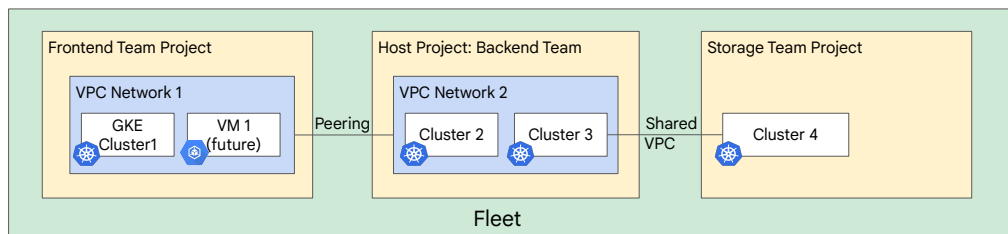
Google Cloud

Let's start by getting a clear understanding of Anthos fleets.

Introducing fleets

Fleets are used to logically group clusters and other resources, which makes multi-cluster administration easier.

- Are created within a single project (Host Project).
- Work across Google Cloud VPC networks and Google Cloud Projects.
- Work with Anthos Clusters with upcoming support for Compute Engine VMs.



Google Cloud

Typically, as organizations embrace cloud-native technologies like containers, container orchestration, and service meshes, they reach a point where running a single cluster is no longer sufficient. There are different reasons for running multiple clusters, such as distributing workloads in multiple regions or clouds or for organization purposes such as separating development from production workloads.

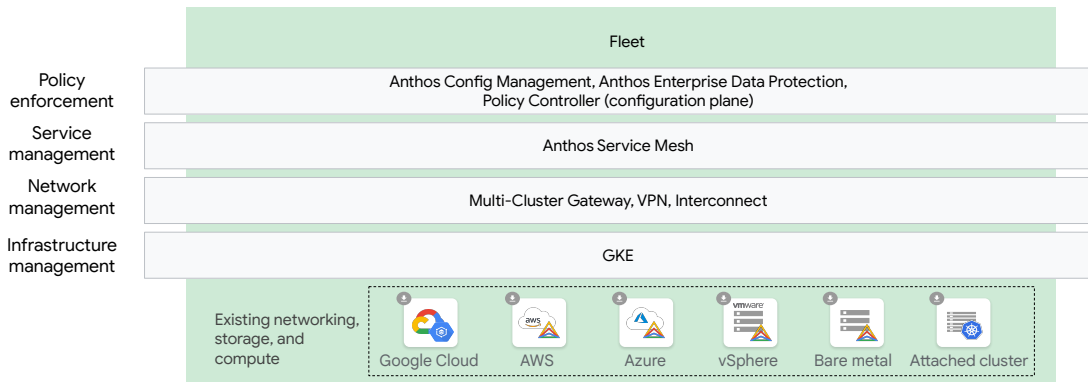
As the number of clusters grows, providing management and governance over these clusters and the resources inside them becomes increasingly difficult. Often at this point, organizations resort to building custom tooling and operational policies to obtain the level of control that they require.

Google Cloud provides the fleet concept to help administrators manage multiple clusters. A fleet provides a way to logically group and normalize clusters, making administration of infrastructure easier.

- Fleets are created within a single project, called the Host Project.
- They work across Google Cloud VPC networks and Google Cloud Projects.
- They work with Anthos Clusters, Compute Engine VMs, and a selection of fleet-enabled Anthos components.

Use fleet-enabled components

Connect the following services to a fleet and share them across clusters:



Google Cloud

Both Anthos clusters and other fleet-enabled components leverage fleet concepts such as namespace and identity sameness to provide a simplified way to work with your clusters and services. Examples of fleet enabled-components include:

- Workload identity pools for Anthos and GKE clusters, so that services can be easily authenticated and authorized within a service mesh and to external services.
- Multi-cluster Gateways to define the set of clusters and service endpoints that traffic can be load balanced over, enabling low-latency and high-availability services.
- Anthos Service Mesh to help you monitor and manage a reliable [service mesh](#) on Google Cloud, on-premises, or on other supported cloud providers.
- Anthos Config Management to deploy, monitor, and enforce declarative policy and configuration changes for your system stored in a central Git repository.

While not explicitly prevented, Google recommends that *fleet-aware resources in the same project be added to the same fleet*.

Fleet considerations

- Grouping infrastructure
 - Group fleet by environment (prod), by line of business (LOB), or when high cross-service communication or deployment happens together.
- Sameness
 - Clusters in a fleet have the same namespaces, services, workload identities, and mesh identity.
- Exclusivity
 - Fleet-aware resources can only be members of a single fleet at any given time.
- High trust
 - Lack of a strong isolation boundary for policy and governance because multiple clusters work as one.

Google Cloud

In order to decide what components go into the same fleet, it's important to take into account the following concepts:

- In terms of grouping infrastructure, we should think about resource relations and administration. In terms of relations, we might want to place together resources that have a high cross-service communication with one another. In terms of administration, it makes sense to place services that are part of the same environment, same line of business, or that are going to be deployed together.
- To make management easy, fleets promote normalization of configuration and resources across clusters, so that all namespaces and services exist across all clusters in the same fleet, and there is a unique way to authenticate and authorize users and services across clusters and service meshes.
- Fleet-aware resources can only be members of a single fleet at any given time. This restriction ensures that there is only one source of truth governing a cluster. Without exclusivity, even the most simple components would become complex to use, requiring your organization to reason about and configure how multiple components from multiple fleets would interact.
- By applying the principle of sameness on namespaces, services, and identity across clusters, we make the cluster boundary less important and expand that boundary to that of the fleet. We still have the benefits of high availability and

- resiliency from running multiple clusters, but we remove the strong isolation boundary for policy and governance from clusters and apply that high trust to the fleet instead.

Fleet example

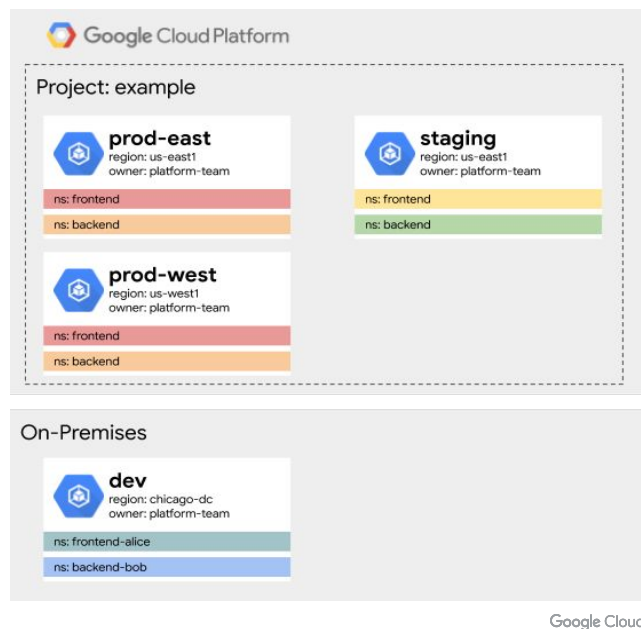
Four clusters:

- Two clusters for prod (in two regions for redundancy)
- One cluster for staging
- One cluster for development

Administered by a Platform Team.

Two services:

- Frontend
- Backend



To land these concepts, let's discuss an example with different solutions.

In this example, there are four clusters. Two clusters are for production (in two regions for redundancy), one is for staging and testing, and the final one is for development. All of the clusters are owned and administered centrally by a platform team. Also, there are two services: **frontend** and **backend**. In more complex scenarios, there might be a greater number of both services and clusters.

Cluster are administered by a centralized platform team who is responsible for managing the mesh and establishing trust between the frontend and backend teams. If this trust cannot be achieved, a possible solution could be to connect two separate meshes, but we would have an additional management overhead.

Let's look at three valid implementations. We discuss *isolation* versus *consistency* (and ease of management); in other words, how much *isolation* is needed between different resource types versus how much *consistency* is needed across them. More consistency is easier to achieve with fewer fleets. The third approach is offered as a possible compromise, keeping production completely isolated while giving developers the ability to work against staged services.

Fleet Solution 1

Use separate fleets for production, staging, and development resources.

Advantage: There is very strong isolation between each of the fleets.

Drawback: It is harder to achieve consistency between production, staging, and development.



One possible approach to leveraging fleets is to create separate fleets for production, staging, and development resources.

To do this, we create three separate fleet host projects and either place resources in those projects, or in the case of our on-premises development cluster, register the cluster to the `example-dev` project. We did not have to address many of the namespace sameness and service sameness concerns due to the granularity in this example, but we did ensure that the `prod-east` and `prod-west` clusters' namespaces were well-normalized.

The advantage of this approach is that we have very strong isolation between each of the fleets. The main drawback of this approach is that we need to administer three different fleets, which makes it harder to achieve consistency between production, staging, and development. For development teams, it is also harder to develop against staged services.

Fleet Solution 2

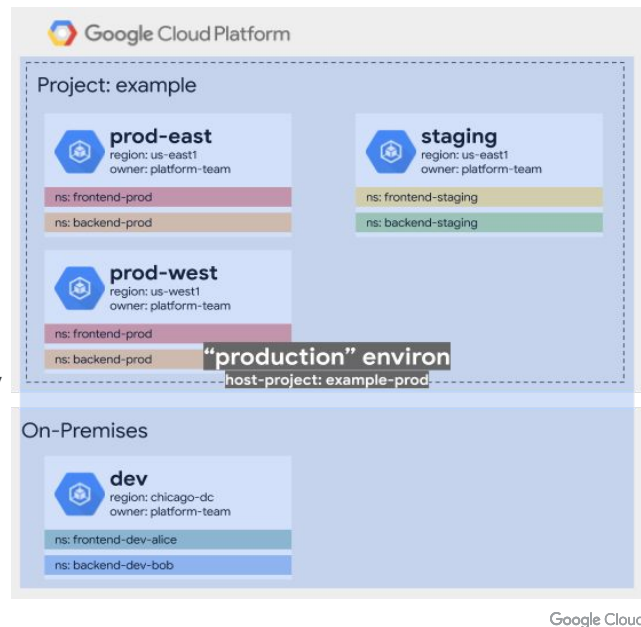
Use a single fleet in the host project.

Other clusters in the same project or separate projects are connected with a Shared VPC.

Use separate environments with namespaces.

Advantage: It is easy to achieve consistency across prod, dev, and staging.

Drawback: It relies on service mesh authorization to provide isolation, which is more work and can lead to misconfigurations.



A second solution involves leaving the resources in the `example` project, and creating the fleet in that project. We could have separated our production and staging resources by placing them in other fleet host projects and leveraging [Shared VPC](#), but we chose not to for simplicity in this example.

With this approach, we need to ensure that our namespaces and services are normalized throughout the fleet. For example, we rename our generic `frontend` to `frontend-prod` and `frontend-staging` in the production and staging clusters, respectively. Finally, while we could keep the original names for our development namespaces, we provide clearer names (like `frontend-dev-alice`) to indicate that they are development namespaces.

With this approach, we're trading off isolation for ease of management. We're relying on service mesh authorization to prevent unwanted service-to-service communication, but we can easily administer the overall system with the one fleet. This arrangement enables us to apply policies across all resources, which can give us confidence that development looks and feels very close to production.

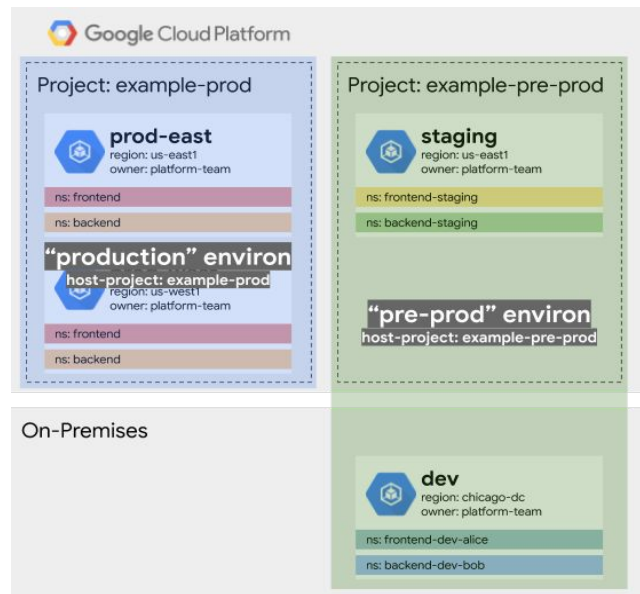
Fleet Solution 3

Use two different fleets for prod and non-prod workloads.

Environments in pre-prod with namespaces must be normalized.

Advantage: Prod is well isolated from pre-prod.

Drawback: environments differ from each other.




Google Cloud


A third solution represents taking the middle ground that combines the staging and development resources together into a non-production fleet while placing production in a separate fleet.

To do this, we create two fleet host projects, one for production and one for non-production. We also place our resources directly into those projects, with the `dev` cluster on-premises registered into our non-production fleet. We need to normalize the namespaces and services between our staging and development resources to provide clarity; for example, we rename `frontend` to `frontend-staging` in the staging cluster.

The advantage here is that production is well-isolated from non-production. For example, we can enable development services to talk to staging services, so developer Alice's `frontend` can talk to a staged `backend` while she's developing her service.



Today's agenda



- 01 Anthos fleet
- 02 [Fleet networking](#)
- 03 Multi-cluster Services
- 04 Multi-cluster Gateway

Google Cloud

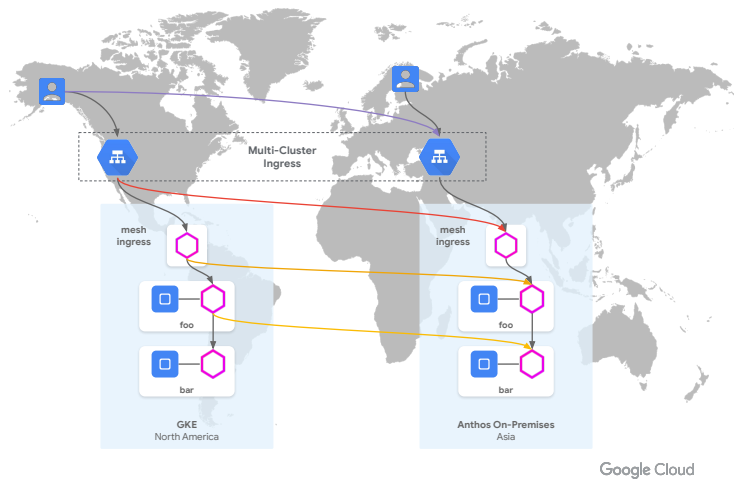
Next, let's talk about fleet networking.

Fleet networking

There are two ways of communicating between Anthos clusters:

- **North-south routing:** communication from a load balancer into the clusters
- **East-west routing:** communication between clusters

In this module, we talk about north-south routing. We discuss east-west routing together with the Anthos Service Mesh.



There are two ways of communicating between Anthos clusters, and in most cases, you will be using both approaches.

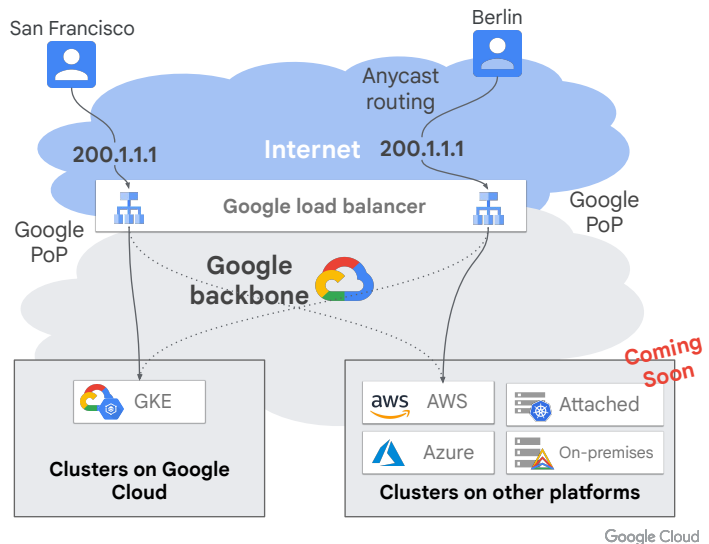
- North-South routing handles the communication from a load balancer into our clusters. That load balancer can be receiving traffic from the internet or from some internal applications.
- East-West routing handles the communication between clusters. This is important for use cases, such as having dependencies in other clusters, implementing fall-back strategies, or performing blue-green deployments.

In this module, we talk about North-South routing. We discuss East-West routing together with the Anthos Service Mesh in a different module.

Ingressing Anthos fleets

Google provides support for ingress clusters through Google-managed load balancers.

- Currently, Anthos only supports ingress through clusters located in Google Cloud.
- In the near future, routing to Anthos clusters in other platforms will also be supported.



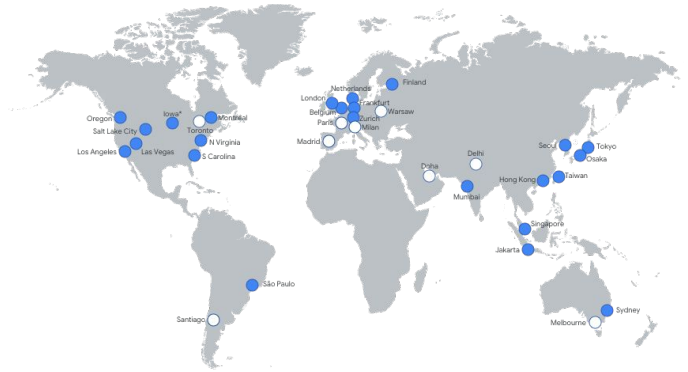
Google provides support for ingress clusters through Google-managed Cloud Load Balancers.

Currently, that support is limited to GKE clusters located in Google Cloud. In the near future, routing to Anthos clusters in other platforms will also be supported. For now, for ingress through Anthos clusters on platforms outside of Google Cloud, you can use Network Endpoint Groups - which we will talk about later in this lecture.

Multi-cluster use cases in Google Cloud

Reasons for running multiple clusters:

- Location
 - Latency
 - Availability
 - Jurisdiction
 - Data gravity
- Isolation
 - Environment
 - Workload tiering
 - Reduce impact from failure
 - Upgrades
 - Security/regulatory separation
 - Tenant separation



Google Cloud

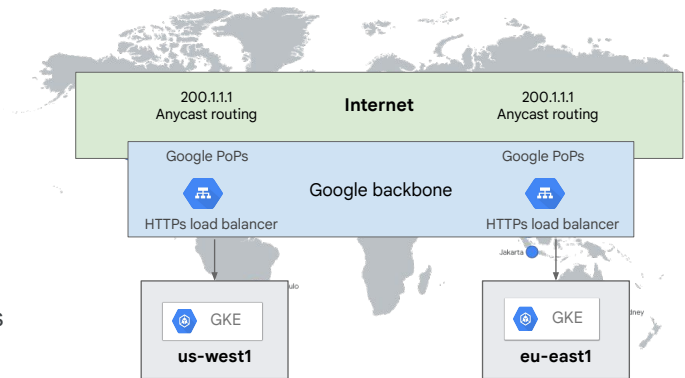
There are multiple reasons why companies choose to run multiple clusters in Google Cloud. The reasons are classified based on location and isolation needs. For example:

- Companies choose to have clusters in different locations to...
 - Be close to their customers so that they can serve traffic with minimum latencies.
 - Increase availability by continuing to provide support after a region is unavailable.
 - Be in the same jurisdiction as their customers to fulfill compliance requirements.
 - Have the compute close to the data, so that they can minimize data transfer costs and increase computation speed.
- In addition, companies choose to have multiple clusters to guarantee isolation when they want to...
 - Have separate environments such as development and production workloads in different clusters.
 - Split workloads into different categories.
 - Reduce impact from failure.
 - Provide upgrades without reducing availability.
 - Provide compliance and regulatory separation of clients, data, applications, etc.

Multi-cluster networking in Google Cloud

Technologies available:

- A single Anycast IP address can be deployed to multiple regions and routed based on proximity.
- Global HTTP(S) load balancers use Anycast IP addresses to receive traffic.
- Google's Premium Network offers "cold-potato" routing to Google PoPs and "hot-potato" routing inside Google's network.



Google Cloud

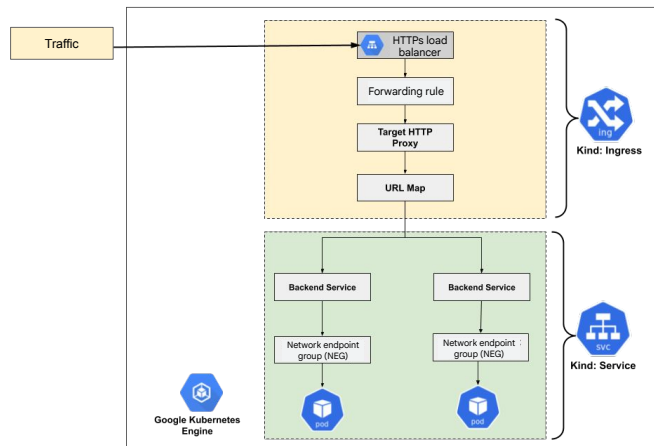
Multi-cluster networking is available in Google Cloud by combining different technologies:

- A single Anycast IP address can be deployed to multiple regions and routed based on proximity to the user.
- Global HTTPs Load Balancers use Anycast IPs to receive traffic and route the request to the nearest cluster.
- Google's Premium Network offers cold potato routing to Google PoPs, hot potato routing inside Google's network.

Network Endpoint Groups enhance flexibility

The global HTTP(S) load balancer uses network endpoint groups (NEGs) to send traffic to dynamically configured endpoints. NEGs can send traffic to:

- Serverless offerings like Cloud Functions or Cloud Run.
- Primary IP addresses like Compute Engine VMs.
- Secondary IP addresses like Pods on Kubernetes for container-native load balancing.



Once we are in Google's Network, The Global HTTPs Load Balancer uses Network Endpoint Groups, or NEGs, to send traffic to dynamically configured endpoints. NEGs can send traffic to:

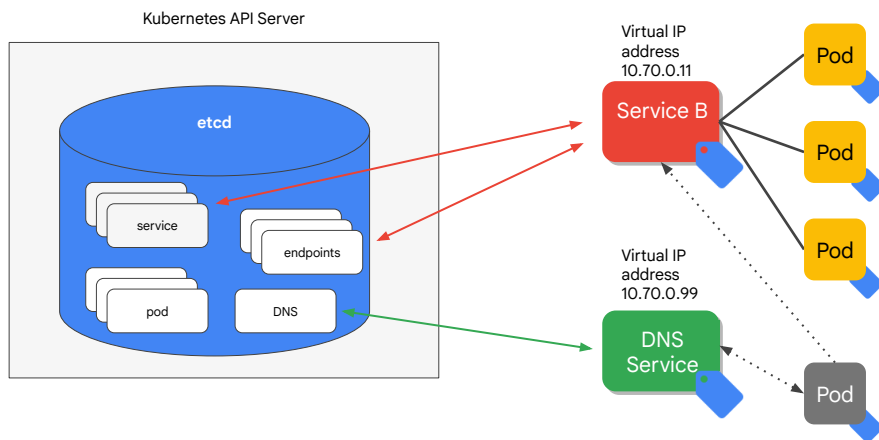
- Serverless offerings like Cloud Functions or Cloud Run.
- Primary IPs like Compute Engine VMs.
- Secondary IPs like pods on Kubernetes for container-native load balancing.

How do you make **Pods discoverable** in Kubernetes?

Google Cloud

How do we make pods discoverable in Kubernetes?


Review: Kubernetes service discovery




Google Cloud

1. Service B is defined with a service object in etcd.
2. The Service B definition includes a selector, and based on that selector the top three pods are identified as endpoints.
3. The endpoints object in etcd contains the IP addresses for the pods that sit behind Service B.
4. The kubelet agent on each node uses the endpoints information to update IPTables so that outgoing connections to 10.70.0.11 get rewritten and sent to one of the pod IP addresses.
5. Most Kubernetes clusters will have a DNS server workload running in the control plane, exposed through a DNS service.
6. If the bottom pod wants to establish a connection with a Service B pod, it will first perform a DNS lookup on the service name (e.g., serviceb.default.cluster-domain.local).
7. Once it gets the VIP for Service B, it attempts to connect to that service IP, and IPtables will rewrite the packet and send to a Service B pod IP.
8. Service discovery, and ultimately request routing, is handled entirely WITHIN the cluster.

What if you want a Service that runs on multiple clusters, and you want pods within any of the clusters to have requests to multi-cluster Services to be load balanced across clusters?



Today's agenda



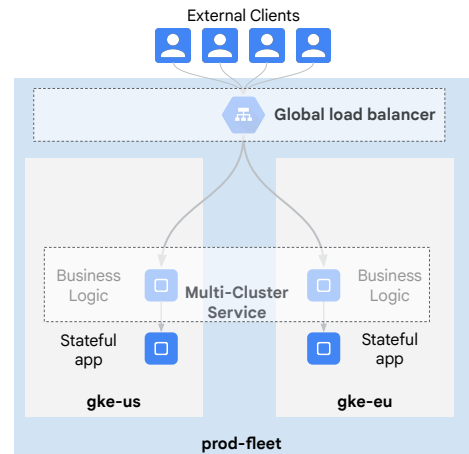
- 01 Anthos fleet
- 02 Fleet networking
- 03 [Multi-cluster Service](#)
- 04 Multi-cluster Gateway

Google Cloud

To easily run a Service on multiple clusters, you can use multi-cluster Services. Let's take a look at this Anthos component.

Multi-cluster Service works cross-cluster

- Multi-cluster Service (MCS) is a cross-cluster discovery and invocation mechanism for GKE clusters.
 - Enabled services act like ClusterIP services, but are discoverable across clusters.
 - MCS integrates with fleets to define the set of clusters they operate on.
 - MCS configures Cloud DNS zones and records for each exported Service in your fleet's clusters.



Google Cloud

Multi-cluster Service, or MCS, is a cross-cluster discovery and invocation mechanism for GKE clusters.

Enabled services act like ClusterIP services, but are discoverable and accessible across clusters via a virtual IP.

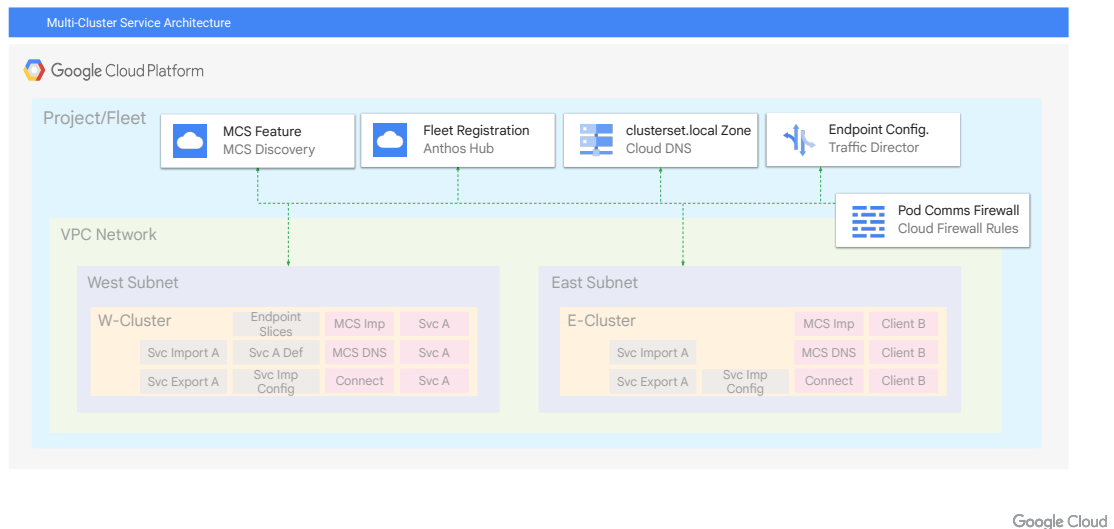
MCS integrates with fleets to define the set of clusters they operate on.

MCS configures Cloud DNS zones and records for each exported Service in your fleet's clusters, letting you connect to Services that are running in other clusters. These zones and records are created, read, updated, and deleted based on the Services that you choose to export across clusters.

Anthos licensing is NOT required for the clusters that are participating in the multi-cluster Service setup. Also, this functionality is entirely separate from Istio or Anthos Service Mesh.

Multi-cluster Service is supported only on GKE on Google Cloud clusters. GKE clusters must be using VPC-native networking.

Multi-cluster Service extends reach



In the diagram, we see two clusters in the same VPC network deployed in different regions, East and West. Normally, a client deployment in the West cluster would not be able to connect to a ClusterIP service running on the East cluster. Multi-cluster Services, however, make that possible.


To configure Multi-cluster Services, follow these steps:

1. Enable the **multiclusterservicediscovery** API and the **multi-cluster-services** feature of Anthos Hub.
2. Register all clusters that access multi-cluster Services to the fleet. This is done with an Anthos Hub registration.
3. Participating clusters automatically run the multi-cluster importer and a multi-cluster DNS deployment.
4. On the cluster that hosts the service to be shared (in this case the NA cluster), the operator creates a ServiceExport resource.
5. MCS configures Cloud DNS zones and records for the exported service; clients can access the service by FQDN.
6. MCS configures firewall rules that let pods on each cluster communicate with each other.
7. MCS configures Traffic Director resources to enable health checks and endpoint information to each cluster.
8. The client pod connects to the service using an FQDN of this format: `SERVICE_EXPORT_NAME_NAMESPACE.svc.cluster.local`

1. the node, and ultimately gets sent to an IP corresponding to one of the service pods on the host cluster.

Multi-cluster Service extends reach

```
kind: ServiceExport
apiVersion: net.gke.io/v1
metadata:
  namespace: foo-namespace
  name: foo-service
```



```
kind: ServiceImport
apiVersion: net.gke.io/v1
metadata:
  namespace: foo-namespace
  name: foo-service
  annotations:
    net.gke.io/derived-service: gke-foo
  labels:
    app.kubernetes.io/managed-by: gke-mcs-importer
spec:
  ips:
    - 10.8.2.197
  ports:
    - port: 80
      protocol: TCP
  sessionAffinity: None
  type: ClusterSetIP
...
```

Google Cloud

On the cluster that hosts the service to be shared, the operator creates a ServiceExport resource.

Note that the service cannot be hosted in the default or kube-system namespace, since services from those namespaces are not exported.

The MCS service then generates a ServiceImport resource and does other supporting configuration on the other clusters in the fleet.

The importing clusters must have a namespace that matches the source service's namespace.

MultiClusterService (MCS)

- MCS is a logical representation of a Service across multiple clusters that exists only in the config cluster.
- MCS selectors apply to labels and clusters.
- MCS generates derived Services in target clusters.
- The derived Service creates a NEG that tracks POD endpoints in every target cluster.

```
kind: MultiClusterService
apiVersion: networking.gke.io/v1alpha1
metadata:
  name: foo
spec:
  template:
    spec:
      selector:
        app: foo
      ports:
        - name: web
          port: 80
```

```
kind: Service
metadata:
  annotations:
    cloud.google.com/neg: '{{..}}'
    cloud.google.com/neg-status: '{{..}}'
    networking.gke.io/mc-parent: '{{..}}'
  name: foo
spec:
  ...
```

Google Cloud

A MultiClusterService, or MCS, is a custom resource used by multi-cluster Gateways that is a logical representation of a Service across multiple clusters. An MCS is similar to, but substantially different from, the core Service type. An MCS exists only in the config cluster and generates derived Services in the target clusters. An MCS does not route anything like a ClusterIP, LoadBalancer, or NodePort Service does. It simply allows the multi-cluster Gateway to refer to a singular distributed resource.

Like a Service, an MCS is a selector for pods but it is also capable of selecting labels *and* clusters. The pool of clusters that it selects across are called member clusters, and these are all the clusters registered to the fleet. This MCS deploys a derived Service in all member clusters with the selector app: foo. If app: foo pods exist in that cluster, then those pod IPs will be added as backends for the multi-cluster Gateway.

The following service is a derived Service that the MCS generated in one of the target clusters. This Service creates a Network Endpoint Group, or NEG, which tracks pod endpoints for all pods that match the specified label selector in this cluster. A derived Service and NEG will exist in every target cluster for every MCS (unless using cluster selectors). If no matching pods exist in a target cluster, then the Service and NEG will be empty. The derived Services are managed fully by the MCS and are not managed by users directly.

A few notes about the derived Service:

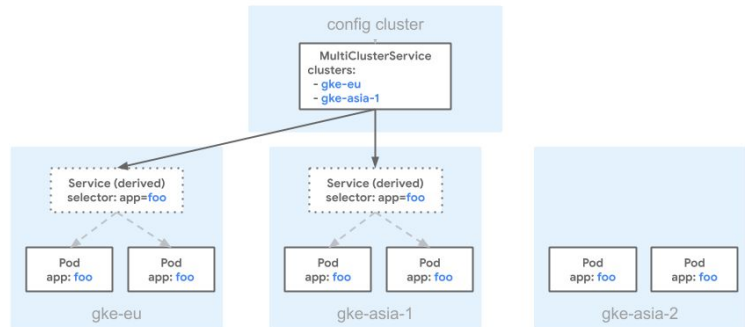
- Its function is as a logical grouping of endpoints. Those endpoints can serve

- as backends for multi-cluster Gateways, a cross-cluster ingress resource that we will cover in a couple of slides.
- It manages the lifecycle of the NEG for a given cluster and application.
- It's created as a headless Service. Note that only the Selector and Ports fields are carried over from the MCS spec to the derived service spec.
- The MCS controller manages its lifecycle.

MultiClusterService (MCS)

- Reasons to use specific clusters:
 - Blue-green deployments
 - App migration
 - Routing to clusters where the application is deployed

```
kind: MultiClusterService
apiVersion: networking.gke.io/v1alpha1
metadata:
  name: foo
...
clusters:
- link: "europe-west1-c/gke-eu"
- link: "asia-northwest1-a/gke-asia-1"
```



Google Cloud


MCS resources have the capability to explicitly select across clusters. By default, an MCS schedules derived Services on every target cluster. Cluster selection defines an explicit list of clusters for a given MCS where derived Services should be scheduled and all other target clusters will be ignored.

There are many use cases where you may want to apply ingress rules to specific clusters:


- Isolating the config cluster to prevent MCSs from selecting across them.
- Controlling traffic between clusters in a blue-green fashion for app migration.
- Routing to application backends that only exist in a subset of clusters.
- Using a single L7 VIP for host/path routing to backends that live on different clusters.

Cluster selection is done via the `clusters` field in the MCS. Clusters are explicitly referenced by `<region | zone>/<name>`. Member clusters within the same fleet and region should have unique names so that there are no naming collisions.

In the example on this slide, the `foo` MCS has a `clusters` field that references `europe-west1-c/gke-eu` and `asia-northeast1-a/gke-asia`. As a result, pods with the matching labels in the `gke-asia` and `gke-eu` clusters can be included as backends for a given multi-cluster Gateway. This will exclude the `gke-us` cluster from Ingress even if it has pods with the `app: foo` label. This can be useful for onboarding or migrating to new clusters and controlling traffic independently of pod deployment.



Today's agenda



- 01 Anthos fleet
- 02 Fleet networking
- 03 Multi-cluster Services
- 04 [Multi-cluster Gateway](#)

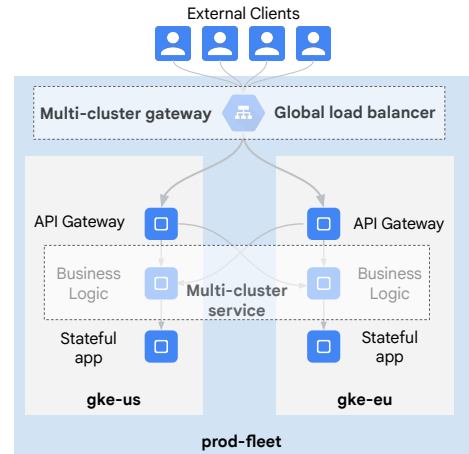
Google Cloud

To control network ingress from outside the cluster, we use multi-cluster Gateways. Let's take a look at this component.

Multi-cluster Gateway provides cross-cluster HTTPs Load Balancing

Multi-cluster Gateway is a cloud-hosted controller for GKE clusters:

- Provisions Google Cloud HTTP(S) load balancers to route traffic across clusters and regions.
- Routing capabilities include traffic splitting, traffic mirroring, and health-based failover.
- Multi-cluster Gateways make managing application networking across many clusters and teams easy, secure, and scalable for infrastructure administrators.



Google Cloud

Multi-cluster Gateway, or MCG, is a cloud-hosted controller for GKE clusters.

It provisions Google Cloud HTTPs Load Balancers to route traffic across clusters and regions.

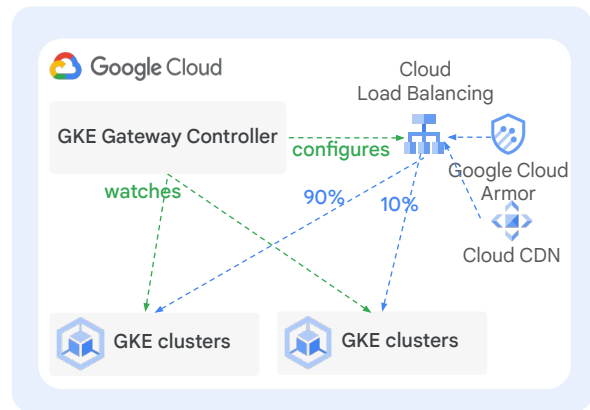
Routing capabilities include traffic splitting, traffic mirroring, health-based failover, and more.

MCG makes managing application networking across many clusters and teams easy, secure, and scalable for infrastructure administrators.

Multi-cluster Gateway Controller configures advanced networking and Google Cloud services

Supports a range of features:

- Internal and External HTTP(S) Load Balancing
- Host, path, header-based routing
- HTTP header manipulation
- Weight-based traffic splitting
- Traffic capacity-based load balancing
- Traffic mirroring
- Geographic-based load balancing
- HTTP, HTTPS, HTTP/2
- Support for Google Cloud Armor, Identity-Aware Proxy (IAP), and Cloud CDN



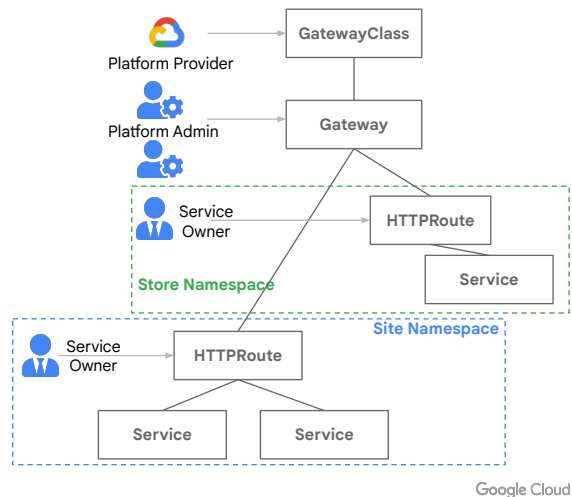
Google Cloud

Multi-cluster Gateway Controller configures advanced networking and Google Cloud services such as:

- Internal and external HTTPs Load Balancers.
- L7 routing based on the host, path, and header in the request to handle requests with different backends.
- Weight-based traffic splitting to enable deployment strategies such as Canary Deployments or A/B testing.
- Traffic mirroring to perform shadow deployments and test your applications with real data before serving production traffic or analyzing requests for security purposes.
- Geographic-based load balancing to make sure user data stays in the country of origin.
- Advanced security features with Cloud Armor to prevent denial of service attacks, Identity-Aware Proxy to make corporate application authentication easier for Cloud CDN to distribute static content to the edges and, therefore, reduce latencies.

Multi-cluster Gateway is based on the Kubernetes Gateway API

- A modern interface for L4/L7 routing in Kubernetes.
- Designed to be role-oriented, portable, expressive, and extensible.
- **GatewayClass** formalizes types of load balancing implementations; Google Cloud implements the Multi-cluster Gateway controller.
- **Gateways** can be shared and deployed cross-namespace by the Platform Admin, thus centralizing policies such as TLS.
- **Routes** run in their own namespace so that Service Owners in different teams can independently control their routing logic.



Multi-cluster Gateway is based on the [Kubernetes Gateway API](#).

It's a modern interface for L4/L7 routing in Kubernetes.

Gateway API is designed to be role-oriented, portable, expressive, and extensible.

GatewayClass formalizes types of load balancing implementations, letting providers build products on top of this interface. Google Cloud takes the GatewayClass definition and implements the multi-cluster Gateway Controller.

There are two components that can be configured, namely Gateways and Routes, and they are independent resources allowing different personas to configure them:

- Gateways can be shared and deployed cross-namespace by the Platform Admin, centralizing policies such as TLS.
- Routes run in their own namespace so that Service Owners in different teams can independently control their routing logic.

Use the Gateway CRD to configure the Google load balancer

- Choose the Gateway controller with the Gateway Class Name:
 - Single Cluster
 - Internal: gke-l7-rilb
 - External: gke-l7-gxlb
 - Multi-Cluster
 - Internal: gke-l7-rilb-mc
 - External: gke-l7-gxlb-mc
- Bind the Gateway and the HTTPRoute using a selector.
- Reference the TLS credentials to accept HTTP(S) requests from your clients.

```
kind: Gateway
apiVersion: networking.x-k8s.io/v1alpha1
metadata:
  name: external-http
  namespace: store
spec:
  gatewayClassName: gke-l7-gxlb-mc
  listeners:
  - protocol: HTTPS
    port: 443
    routes:
      kind: HTTPRoute
      selector:
        matchLabels:
          gateway: external-http
  tls:
    mode: Terminate
    options:
      .../pre-shared-certs: cert
```

Google Cloud

To configure Google Cloud Load Balancer, first you need to use the Gateway CRD.

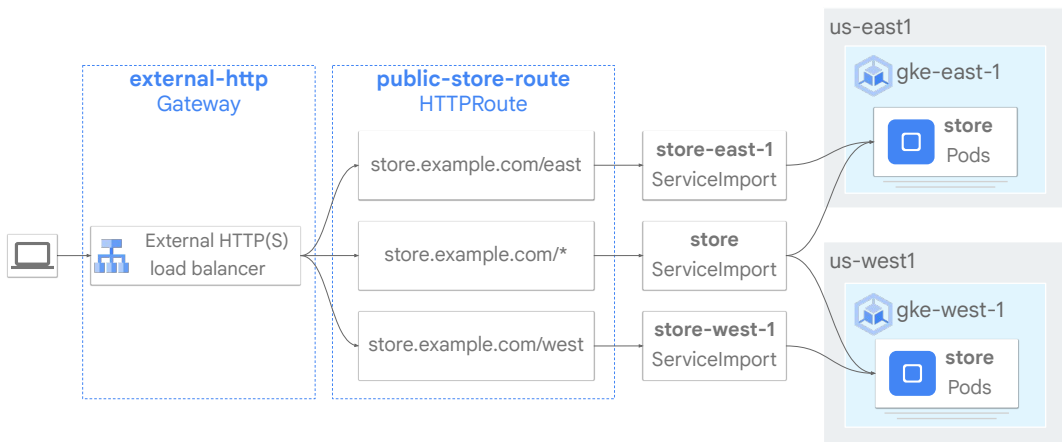
In the metadata, specify the name of the resource and the namespace where you want to launch it.

In the spec, use the gatewayClassName to specify which controller you want to use. To deploy an internal load balancer, use RILB, while to deploy an external, use GXLB instead. If you want them to work across clusters, make sure you include the dash MC at the end of the name.

Routes can be created inside the Gateway or in a separate resource. In this case, we specify the HTTPRoute inside the Gateway and then link it with the Gateway using a selector.

Notice that we are specifying the file for the credentials that will be used to establish TLS encryption on the requests.

Use HTTPRoute and ServiceImport to route to services across clusters



Google Cloud

If we deployed a multi-cluster Gateway, the HTTPRoute leverages ServiceImports as logical identifiers for a Service that exists in another cluster or that stretches across multiple clusters.

Use HTTPRoute and ServiceImport to route to services across clusters

```
kind: HTTPRoute
apiVersion: networking.gke.io/v1alpha1
metadata:
  name: store-route
  namespace: store
  labels:
    gateway: external-http
rules:
- forwardTo: external-http
  - backendRef:
      group: net.gke.io
      kind: ServiceImport
      name: store
  ...
```

```
- matches:
  - path:
      type: Prefix
      value: /west
    forwardTo:
      - backendRef:
          kind: ServiceImport
          name: store-west-1
          port: 8080
  - matches:
    - path:
        type: Prefix
        value: /east
      forwardTo:
        - backendRef:
            kind: ServiceImport
            name: store-east-1
            port: 8080
```

Google Cloud

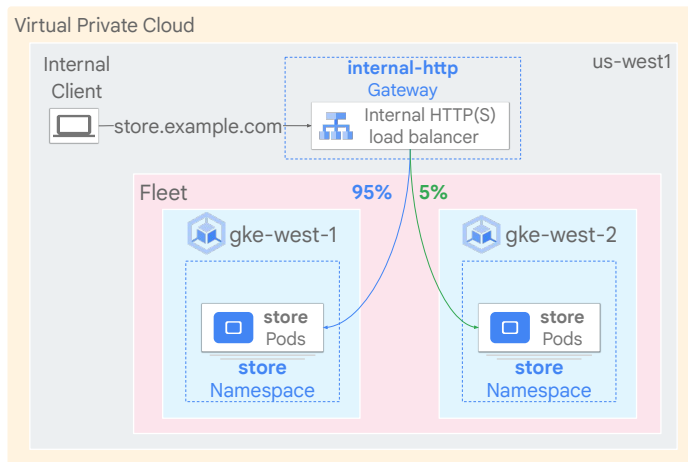
In a single-cluster Gateway, you reference the service from the HTTPRoute. However, since we are referencing a service that exists in another cluster or that stretches across multiple clusters, we use the ServiceImports as logical identifiers for a Service instead.

On this slide, you see an example of the HTTPRoute CRD yaml file.

The left part of the yaml file explains the connection with the Gateway and sets the “store” as the default route.

The right part of the yaml file explains the paths to /east and /west would go to multi-cluster services called store-west-1 and store-east-1.

Use traffic splits to implement deployment strategies such as Canary or Blue-Green



Google Cloud

HTTPRoutes are very flexible and support features such as traffic splits to divert requests based on weights to services in the same or different clusters.

You can use traffic splits to implement deployment strategies such as:

- Canary deployments, where a small percentage of your traffic goes to the newer version to test that there are no issues, and that way you can continue to increase the percentage of traffic going to the new version at a pace that you feel comfortable to avoid major customer disruption.
- Blue-green deployments, where you can fully deploy a new version of your app and do a cutover of the traffic, to make sure that customers only see one version of your software at the same time while there is no downtime.
- A/B testing, where you can deploy multiple versions of your application, divert the same percentage of traffic to each version, and test conversion rates to see which version is performing better.

Use traffic splits to implement deployment strategies such as Canary or Blue-Green

Use weights to specify the percentage of traffic split across clusters and services.

```
kind: HTTPRoute
apiVersion: networking.gke.io/v1alpha1
metadata:
  name: store-route
  namespace: store
  labels:
    gateway: internal-http
  rules:
  ...
```

```
- forwardTo:
  - backendRef:
      group: net.gke.io
      kind: ServiceImport
      name: store-west-1
      port: 8080
      weight: 90
  - backendRef:
      group: net.gke.io
      kind: ServiceImport
      name: store-east-1
      port: 8080
      weight: 10
```

Google Cloud

In order to implement traffic splits, you have to add a weight field on the backends that the multi-cluster Gateway service is acting as a load balancer for. If the weight component is not added, a round-robin strategy is performed and all services get a similar percentage of the traffic.

Enhance your Gateway with Google Cloud features

- The GatewayClass supports the BackendConfig resource to customize backend settings on a per-Service level.
- Configure features such as Cloud CDN, connection draining, health checks, Google Cloud Armor security policies, Cloud Logging, and IAP.
 - This requires the `cloud.google.com/backend-config` Service annotation to reference the BackendConfig resource.

```
apiVersion: cloud.google.com/v1
kind: BackendConfig
metadata:
  name: my-backendconfig
spec:
  cdn:
    enabled: true
    cachePolicy:
      ...
  connectionDraining:
    drainingTimeoutSec: 60
  healthCheck:
    checkIntervalSec: 10
    timeoutSec: 5
    healthyThreshold: 3
    unhealthyThreshold: 3
    requestPath: /healthz
  securityPolicy:
    name: "cloud-armor-security-policy"
  logging:
    enable: true
  iap:
    enabled: true
```

Google Cloud

You can benefit from additional Google Cloud features when using the multi-cluster Gateway.

That's possible because the GatewayClass supports the BackendConfig resource, where you can customize backend settings on a per-Service level. In the BackendConfig yaml on the right, you can find some of the features available:

- CDN enables you to cache your static data closer to your users on the 100+ edge locations available across the globe. You can configure settings such as whether different hosts should be cached separately, query strings should be included, or whether requests containing a specific query should be cached or not.
- ConnectionDraining enables existing connections to complete when a backend is removed. By default the value is 0.
- HealthChecks test that the backend services are available and respond within a specified time. For instance, in this example, the backend with path `"/healthz"` is checked every 10 seconds. If the backend responds with an error or takes longer than 5 seconds to respond, the backend will be put in an unhealthy state. If within the next three tries the backend is not healthy again, the backend will be removed.
- Google Cloud Armor Security Policies help you protect your load-balanced applications from web-based attacks such as denial of service, cross-scripting injection, or sql-injection.
- Enabling Logging can log all HTTP requests from clients to Cloud Logging at a

- sampling rate of your choice.
- IAP enables you to authenticate and authorize employees to use internal applications.

In addition to creating the BackendConfig, you need to link it a Kubernetes Service. To do so, you have to add the `cloud.google.com/backend-config` annotation referencing the BackendConfig resource name.

Lab intro

🕒 30 min

AHYBRID022 North-south
routing with Multi-Cluster
Gateways

