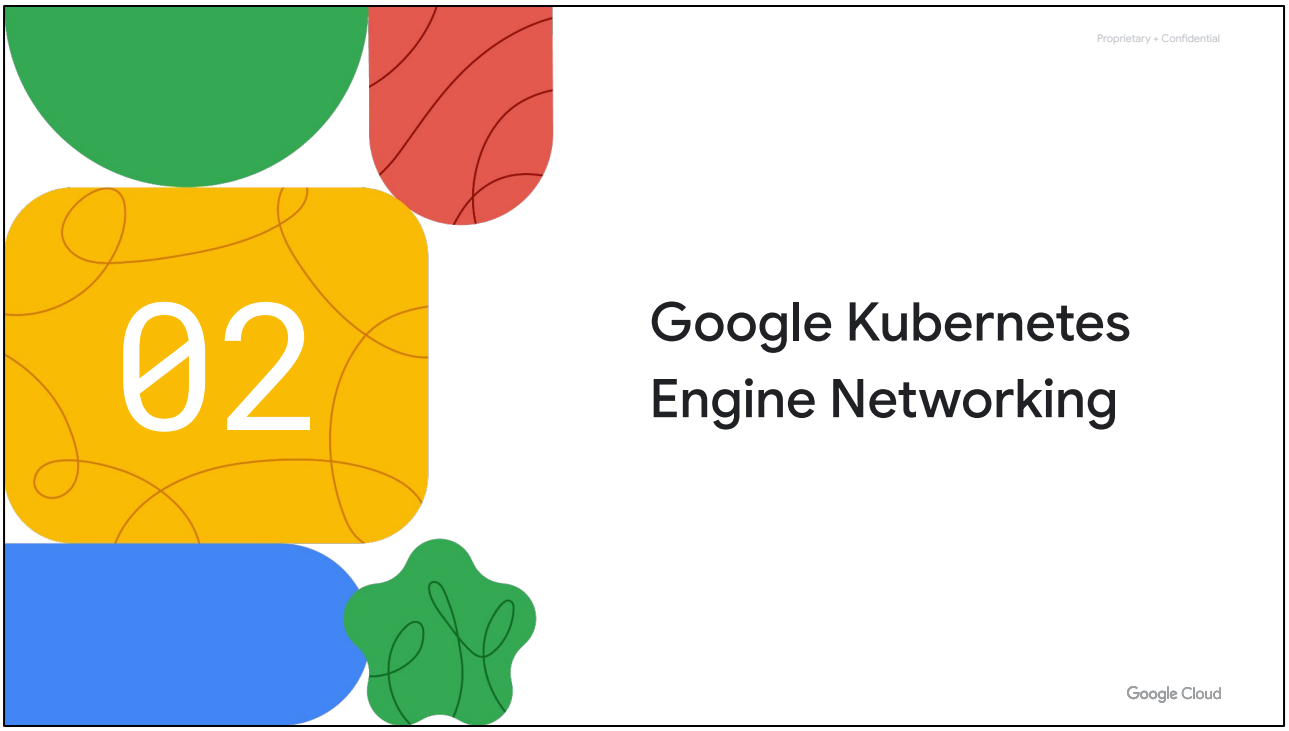


When you design an application, it's important to understand what *other* types of applications it will talk to and where they can be located.

This means it's also important to consider how to expose them to the outside world for consumption, and how to balance incoming traffic to cope with varying workload demands.

Proprietary + Confidential

Google Kubernetes Engine Networking

01 Kubernetes networking

02 Kubernetes Services

03 Ingress

04 Container-native load balancing

05 Network policies in GKE

Google Cloud

In this section of the course titled, “Google Kubernetes Engine Networking”, you’ll:

- Explore Kubernetes networking, including Pod, and cluster networking.
- Create Services to expose to applications running within Pods. This includes configuring load balancers to expose services to external clients.
- Explore the Ingress resource.
- Explain the purpose of network policies in GKE
- And configure Google Kubernetes Engine Networking.

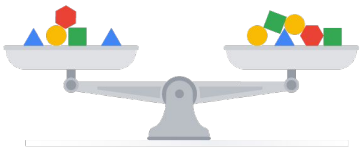
Proprietary + Confidential

Google Kubernetes Engine Networking

Proprietary + Confidential

Pod networking

Pods, containers, and nodes all communicate using IP addresses and ports.



Kubernetes provides different types of load balancing to direct traffic to the correct Pods.

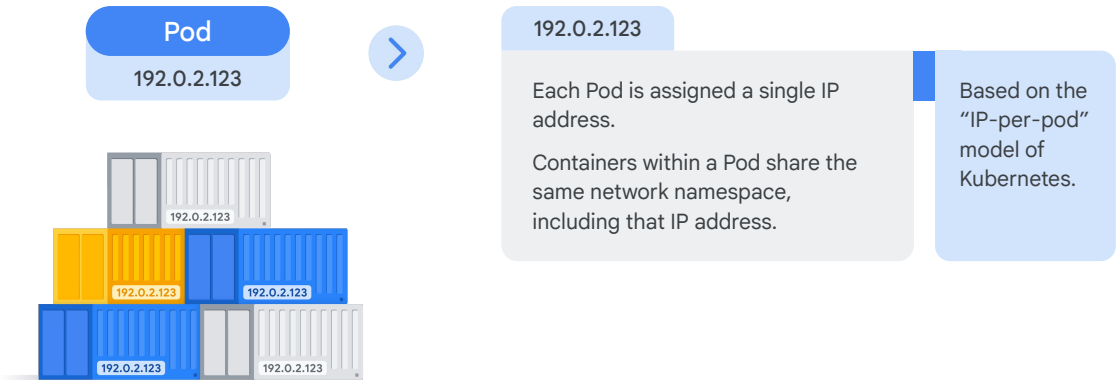
Google Cloud

The Kubernetes networking model relies heavily on IP addresses, because services, Pods, containers, and nodes all communicate using IP addresses and ports.

And Kubernetes provides different types of load balancing to direct traffic to the correct Pods. Let's review some of the basics of Pod networking.

Proprietary + Confidential

Pods share storage and networking

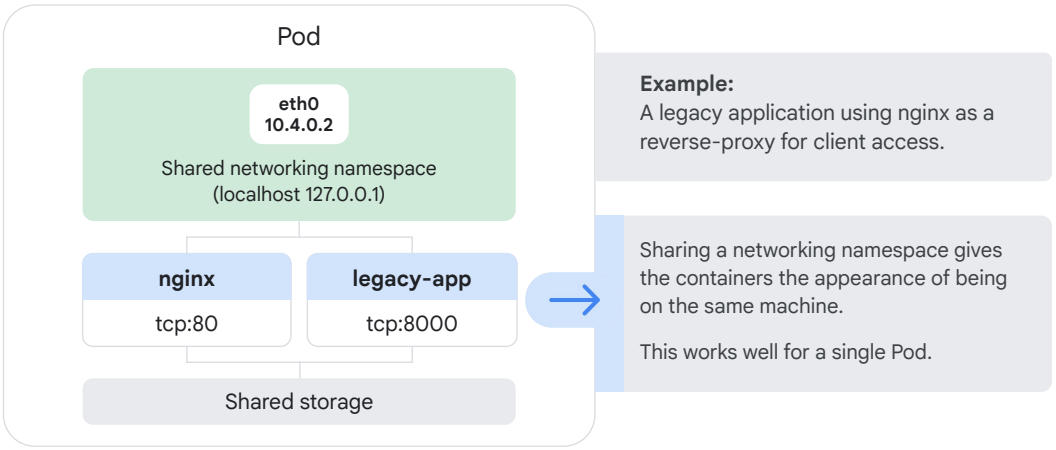


Google Cloud

A Pod is a group of containers that shares storage and networking. This is based on the "IP-per-pod" model of Kubernetes, where each Pod is assigned a single IP address, and the containers within a Pod share the same network namespace, including that IP address.

Proprietary + Confidential

Namespaces isolate resources within a cluster



Google Cloud

In Kubernetes, **namespaces** provide a mechanism for isolating groups of resources within a single cluster.

For example, let's say a legacy application uses nginx as a reverse-proxy for client access. If the nginx container runs on TCP port 80, and the legacy application runs on TCP port 8000, the two containers appear as though they're installed on the same machine because both containers share the same networking namespace. The nginx container will contact the legacy application by establishing a connection to "localhost" on TCP port 8000.

This works well for a single Pod, but workloads can run in multiple Pods and can be composed of many different applications that need to talk to each other.

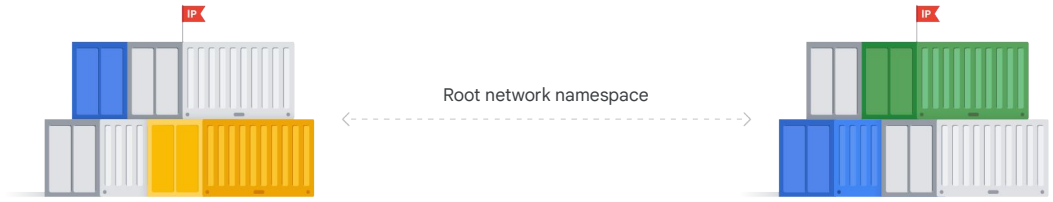
Proprietary + Confidential

How Pods talk to other Pods

Each Pod has a unique IP address, just like a host on the network.

On a node, Pods are connected to each other through the node's root network namespace.

The root network namespace is connected to the node's primary network interface card (NIC).



Google Cloud

So, how do Pods talk to other Pods?

Each Pod has a unique IP address, just like a host on the network.

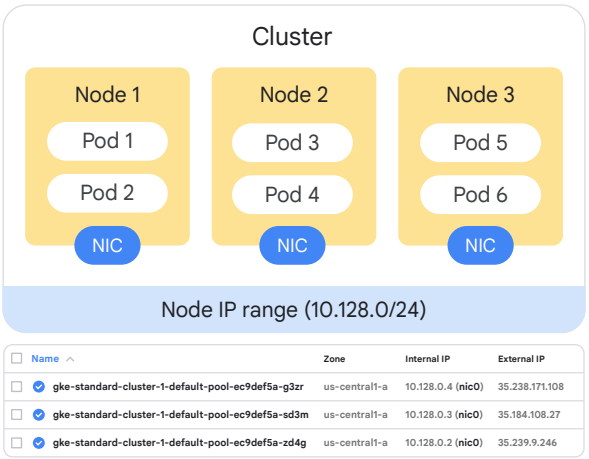
On a node, Pods are connected to each other through the node's root network namespace, which ensures that Pods can find and reach each other on that virtual machine.

The root network namespace is connected to the Node's primary NIC. Using the node's VM NIC, the root network namespace is able to forward traffic out of the node.

This means that the IP addresses on the Pods must be routable on the network that the node is connected to.

Proprietary + Confidential

Nodes source Pod IPs from VPC address ranges



VPCs are logically isolated networks that provide connectivity for resources deployed within Google Cloud.

A VPC can be composed of many different IP subnets in regions around the world.

When you deploy a GKE cluster, you can select a VPC along with a region or zone.

Google Cloud

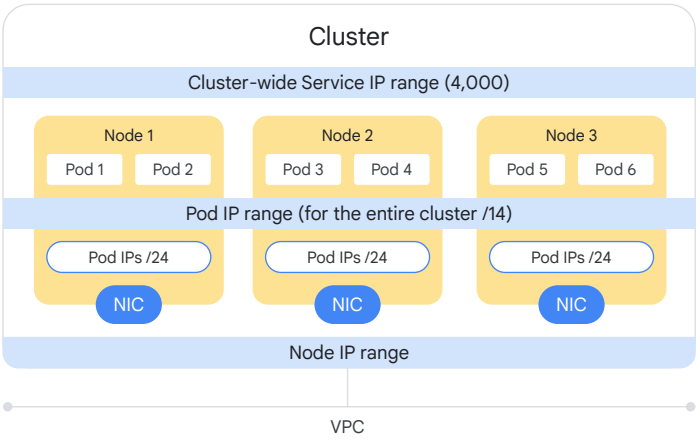
In GKE, nodes get their Pod IP addresses from address ranges assigned to the Virtual Private Cloud, or VPC, being used.

VPCs are logically isolated networks that provide connectivity for resources deployed within Google Cloud, such as Kubernetes Clusters, and Compute Engine instances. A VPC can be composed of many different IP subnets in regions around the world.

When you deploy a GKE cluster, you can select a VPC along with a region or zone. By default, a VPC has an IP subnet pre-allocated for each Google Cloud region in the world. The IP addresses in this subnet are then allocated to the compute instances that you deploy in the region.

Proprietary + Confidential

GKE cluster nodes: Managed compute instances



- Alias IPs can configure additional secondary IP addresses or IP ranges on Compute Engine VM instances.
- VPC-Native GKE clusters automatically create an alias IP range to reserve approximately 4,000 IP addresses for cluster-wide services.
- VPC-Native GKE clusters also create a separate alias IP range for your Pods.

Google Cloud

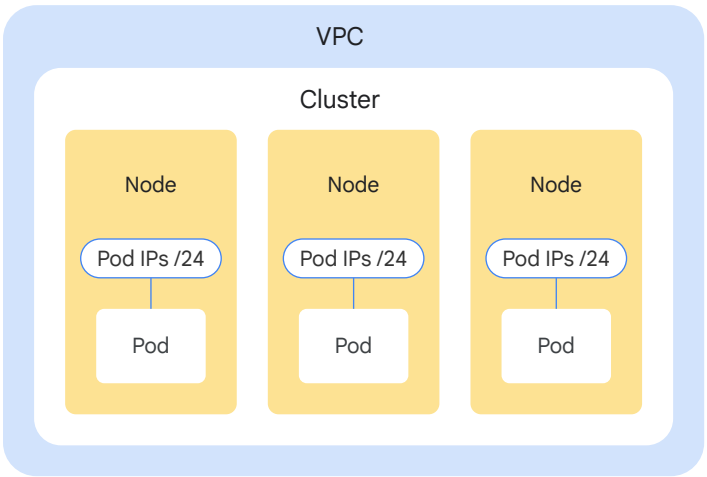
Then there are GKE cluster nodes, which are compute instances that GKE customizes and manages *for you*. These machines are assigned IP addresses from the VPC subnet that they reside in.

Google Cloud offers something called an alias IPs, which can configure additional secondary IP addresses or IP ranges on Compute Engine VM instances. VPC-Native GKE clusters automatically create an alias IP range to reserve approximately 4,000 IP addresses for cluster-wide services that you might want to create later.

VPC-Native GKE clusters also create a separate alias IP range for your Pods. You'll recall that each Pod must have a unique address, so this address space will be large. By default the address range uses a /14 block, which contains over 250,000 IP addresses.

Proprietary + Confidential

IP ranges allow GKE to divide IP space



- Large default Pod IP range (/14) enables flexible IP distribution across nodes.
- Each node receives a smaller IP block (/24) for Pod allocation.
- Design supports scaling both nodes and Pods per node.

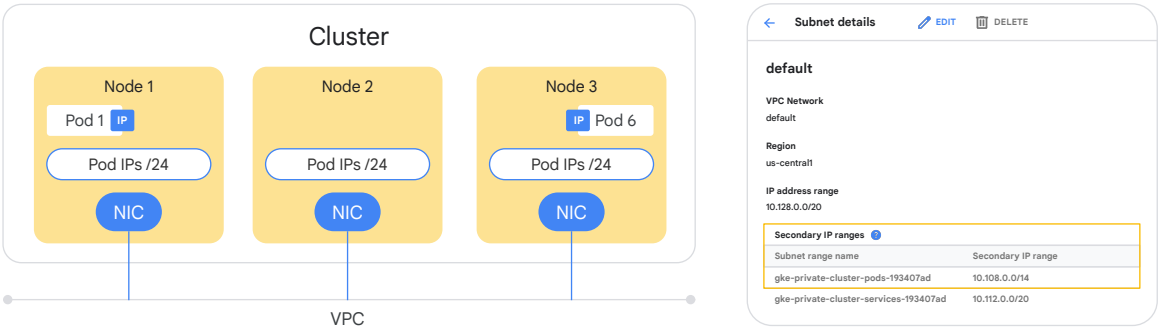
Google Cloud

Google Cloud quota limits will prevent you from running 250,000 Pods in a single cluster, but the large IP address range allows GKE to divide the IP space among the nodes. Using this large Pod IP range, GKE allocates a much smaller /24 block to each node, which contains about 250 IP addresses. This allows for 1000 nodes, with over running 100 pods each, by default.

This functionality allows you to configure both the number of nodes you expect to use and a maximum number of pods per node.

Proprietary + Confidential

Nodes assign unique IP addresses to Pods



GKE automatically configures your VPC to recognize this range of IP addresses as an authorized secondary subnet of IP addresses

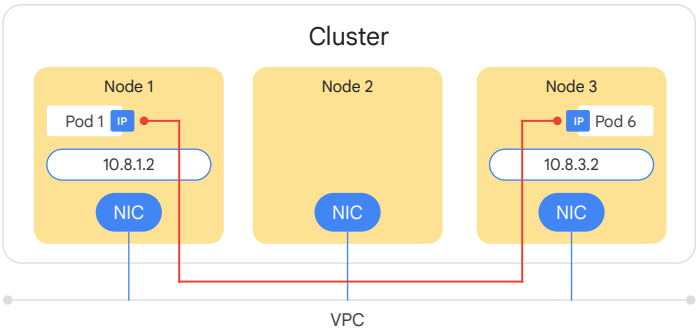
Google Cloud

The nodes allocate a unique IP address from their assigned range to each Pod as it starts.

Because the Pods' IP addresses are a part of the alias IP, GKE automatically configures your VPC to recognize this range of IP addresses as an authorized secondary subnet of IP addresses. This ensures the Pod’s traffic is permitted to pass the anti-spoofing filters on the network.

Proprietary + Confidential

Nodes maintain separate Pod IP address spaces



- Pods can directly connect to each other by using their native IP addresses.
- Pod IP addresses are natively routable by VPC Network Peering.
- Traffic from clusters is routed or peered inside Google Cloud, but becomes NAT translated at the node IP address if it has to exit Google Cloud.

Google Cloud

Each node maintains a separate IP address space for its Pods, which means the nodes don't need to perform network address translation on the Pod IP addresses. Pods can directly connect to each other by using their native IP addresses. Pod IP addresses are natively routable within the cluster's VPC network and other VPC networks connected to it by VPC Network Peering.

The traffic from clusters is routed or peered inside Google Cloud but becomes NAT translated at the node IP address if it has to exit Google Cloud.

Proprietary + Confidential

Google Kubernetes Engine Networking

01 Kubernetes networking

02 Kubernetes Services

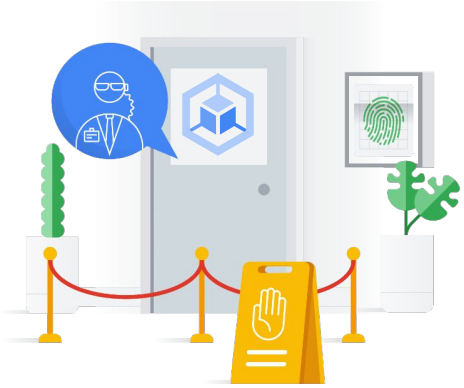
03 Ingress

04 Container-native load balancing

05 Network policies in GKE

Proprietary + Confidential

What is a Service?



- It's a logical abstraction that defines a set of Pods and a single IP address for accessing them.
- It's how the outside world accesses the cluster. Think of it like a GKE doorman or bouncer, keeping out unwanted visitors.
- It's used to provide a stable IP address and name for a Pod, because these can change frequently.

Google Cloud

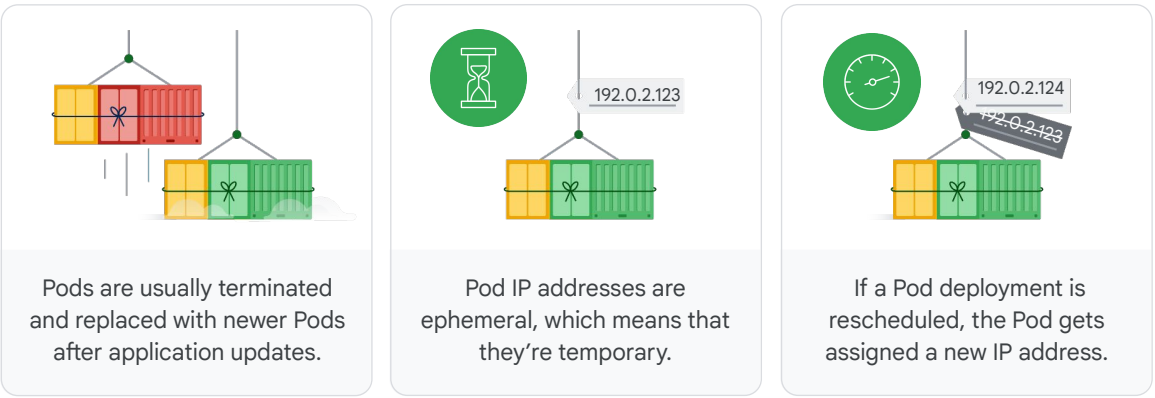
So, what is a Service in Kubernetes?

A Service is a logical abstraction that defines a set of Pods and a single IP address for accessing them. A Service is how the outside world accesses the cluster. Think of it like a GKE doorman or bouncer, keeping out unwanted visitors!

Services are used to provide a stable IP address and name for a Pod, because these can change frequently. With Services, these details can remain the same through updates, upgrades, scalability changes, and Pod failures. This stability allows applications to connect to each other and be found by external clients, even when Pods are updated or replaced.

Proprietary + Confidential

Pods and virtual machines have different life cycles



Google Cloud

Pods have a different life cycle compared to virtual machines.

While VMs are typically designed to be durable and persist through application updates and upgrades, Pods are usually terminated and replaced with newer Pods after application updates.

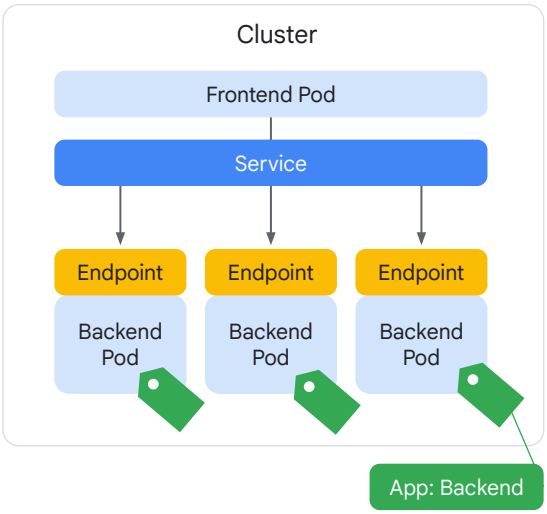
Pod IP addresses are ephemeral, which means that they're temporary, and as a result of the new Pod deployment, the updated version of the application gets a new IP address.

Also, if a Pod deployment is rescheduled for any reason, then the Pod gets assigned a new IP address. Please note, if a Pod's IP address changes unexpectedly, it can cause significant Service disruptions.

All of this is to say that locating an application running in your cluster by IP address can be difficult, so this is where a **Service** comes in.

Proprietary + Confidential

Endpoints



- Kubernetes services create dynamic IP address collections called **endpoints**.
- Endpoints link to Pods matching the Service's labels.
- Services receive a fixed virtual IP address upon creation.
- Service virtual IPs are durable, unlike Pod IPs.

Google Cloud

The dynamic collection of IP addresses created by a Kubernetes Service is called an endpoint. Endpoints belong to Pods that match the Service's label selector.

When a Service is created, it's issued a static virtual IP address from the pool of IP addresses that the cluster reserves for Services.

Unlike the Pod's IP address, the Virtual IP is durable. It's published to all nodes in the cluster and it doesn't change, even if all of the Pods behind it change.

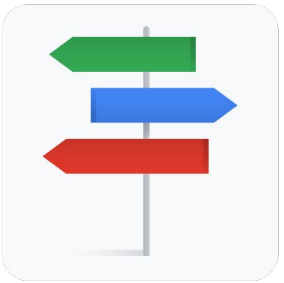
In GKE, this range is automatically managed, and by default contains over 4,000 addresses per cluster.

Proprietary + Confidential

Ways to search for and locate a Service in GKE

By environment variables

By DNS



By Service type

Google Cloud

There are multiple ways to search for and locate a Service in GKE, including searching by environment variables, DNS, or by Service type.

Let's explore what each means, starting with environment variables.

Proprietary + Confidential

Locate a Service by environment variables

By environment variables

- Kubelet adds a set of environment variables for each active Service.
- The Pod can access the Service by using the environment variables.
- Not the most robust mechanism for discovery.

  Changes made to a Service *after* Pods have been started *will not* be visible to the Pods that are already running.

Google Cloud

When a new Pod starts running on a node, kubelet adds a set of environment variables for each active service in the same namespace as the Pod. This allows the Pod to access the Service by using the environment variables.

However, this is not the most robust mechanism for discovery, as changes made to a Service after Pods have been started will not be visible to the Pods that are already running.

Proprietary + Confidential

An example of environment variables

```
DEMO_SERVICE_HOST=10.70.0.11
DEMO_SERVICE_PORT=6379
DEMO_PORT=TCP://10.70/0/11:6379
DEMO_PORT_6379_TCP=tcp://10.70.0.11:6379
DEMO_PORT_6379_TCP_PROTO=tcp
DEMO_PORT_6379_TCP_PORT=6379
DEMO_PORT_6379_TCP_ADDR=10.70.0.11
```

Example of environment variables for a Service named **demo.**

```
demo.my-project
```

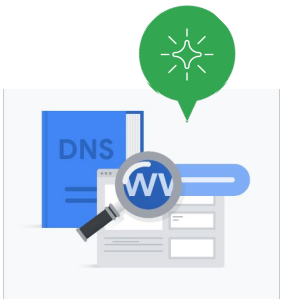
```
_http._tcp.demo.my-project
```

Google Cloud

Here's an example of the environment variables for a Service named "demo" showing information such as the host IP and port address.

Proprietary + Confidential

Locate a Service using a DNS server



By DNS

- GKE includes pre-installed DNS.
- DNS monitors the API server for new Services.
- Kube-dns auto-generates DNS records for new Services.
- Client pods' DNS search includes their namespace and the cluster's default domain.

Google Cloud

A better practice to locate a Service in GKE is using a DNS server.

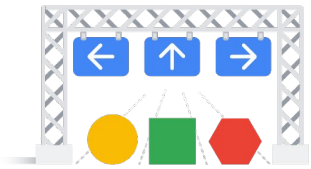
DNS comes pre-installed in Google Kubernetes Engine, and the DNS server watches the API server to identify when new Services get created. When a new Service is created, kube-dns, which is a lightweight DNS server, automatically creates a set of DNS records. This allows all the Pods in the cluster to resolve Kubernetes Service names automatically.

By default, a client Pod's DNS search list will include the Pod's own namespace and the cluster's default domain.

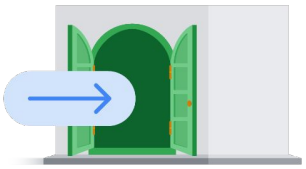
The final way to find a Service in GKE is by changing the service type, which we'll explore next.

Proprietary + Confidential

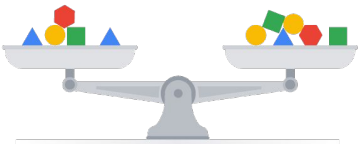
Find a Service by changing the Service type



ClusterIP Service



NodePort



LoadBalancer

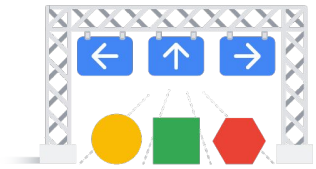
Google Cloud

In addition to using environmental variables and DNS to find a Service in GKE, you can also find a Service by changing the Service type.

There are three principal types of Services: ClusterIP, NodePort, and LoadBalancer.

Proprietary + Confidential

ClusterIP Service



ClusterIP Service

- Has a static IP address.
- Operates as a traffic distributor within the cluster.
- Not accessible by resources outside the cluster.
- Other Pods use it to communicate with the Service.

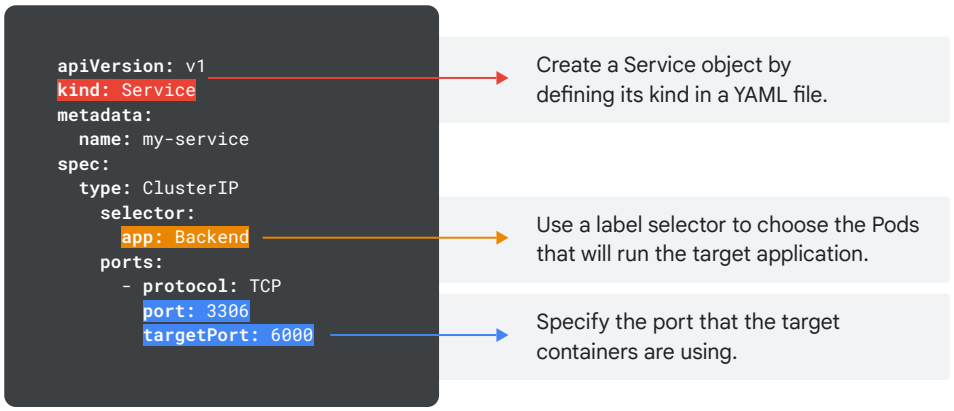
Google Cloud

A Kubernetes **ClusterIP Service** has a static IP address and operates as a traffic distributor within the cluster. A ClusterIP Service isn't accessible by resources outside the cluster.

Other Pods will use the ClusterIP as their destination IP address when they communicate with the Service.

Proprietary + Confidential

Creating a ClusterIP Service



Google Cloud

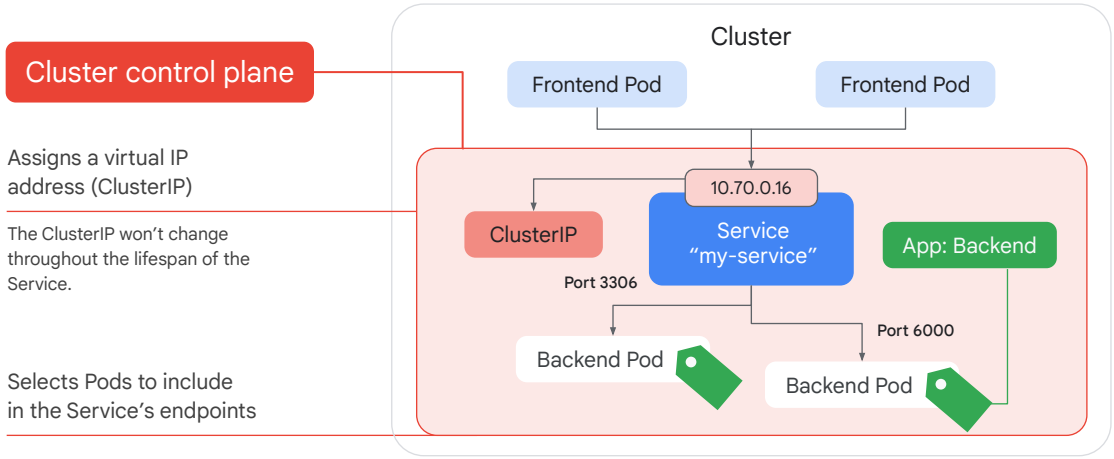
To create a ClusterIP Service, you need to start by creating a Service object by defining its kind in a YAML file. If a Service type isn't specified during the creation of the Service, ClusterIP will be the default Service type.

Next, a label selector can be used to choose the Pods that will run the target application. In this case, the Pods with a label of "app: Backend" are selected and included as endpoints for this Service.

From there, you need to specify the port that the target containers are using, for example, TCP port 6000. This means that this Service will receive traffic on port 3306 and then remap it to 6000 as it delivers it to the target Pods. By creating, loading or applying this manifest, ClusterIP Service named "my-service" will be created.

Proprietary + Confidential

The cluster control plane



Google Cloud

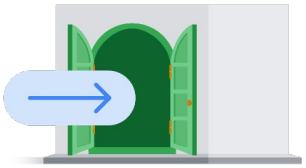
When the Service is created, the cluster control plane, which is responsible for managing the cluster's nodes and workloads, assigns a virtual IP address—also known as ClusterIP—from a reserved pool of Alias IP addresses in the cluster's virtual private cloud. This IP address won't change throughout the lifespan of the Service.

The cluster control plane selects Pods to include in the Service's endpoints based on the label selector on the Service and the labels on the Pods.

Proprietary + Confidential

NodePort Service

- ClusterIP Services are for internal communication, while NodePort services are used for external communication.
- A NodePort Service includes an underlying ClusterIP service.
- A ClusterIP Service is automatically created within a NodePort service for internal traffic distribution.



NodePort

Google Cloud

Alright, so we've established that the ClusterIP Service is useful for internal communication within a cluster, but what about the external communication?

This is where we look at the NodePort Service.

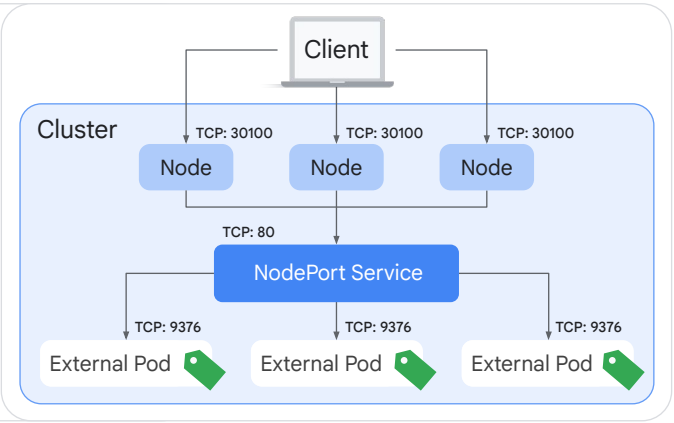
A NodePort Service is built on top of a ClusterIP Service. This means that when you create a NodePort Service, a ClusterIP Service is automatically created in the process to distribute the inbound traffic internally across a set of Pods.

Proprietary + Confidential

NodePort Service

Example

- There's a Service that can be reached from outside of the cluster by using the IP address of any node and the corresponding NodePort number.
- For this to work, incoming traffic is directed to a Service on port 80, then forwarded to a target Pod on port 9376.



Google Cloud

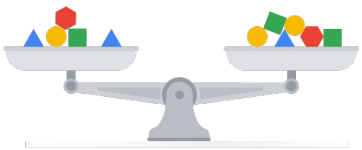
So, let's say that there's a Service that can be reached from outside of the cluster by using the IP address of any node and the corresponding NodePort number.

For this to work, we'll say that traffic through this port would need to be directed to a Service on Port 80 and further redirected to one of the Pods on port 9376.

NodePort Service can be useful to expose a Service through an external load balancer that you set up and manage yourself. Using this approach, you would have to deal with node management, making sure there are no port collisions.

Proprietary + Confidential

LoadBalancer Service



LoadBalancer

- Builds on the ClusterIP Service.
- Can be used to expose a Service to resources outside the cluster.
- Implemented using Google Cloud's passthrough Network Load Balancer.

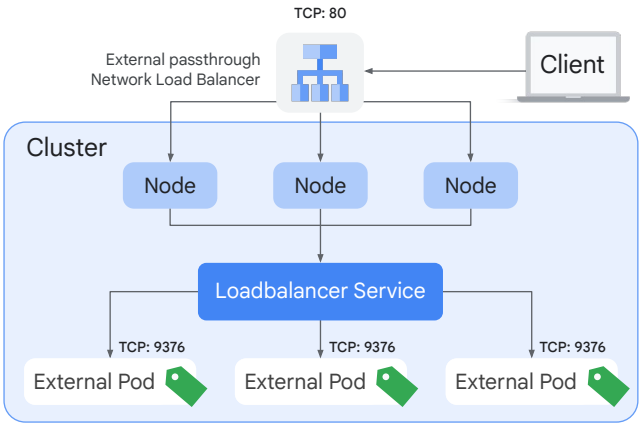
Google Cloud

And finally, there is the LoadBalancer Service type, which also builds on the ClusterIP Service and can be used to expose a Service to resources outside the cluster.

With GKE, the LoadBalancer Service is implemented using Google Cloud's passthrough Network Load Balancer.

Proprietary + Confidential

How does a LoadBalancer Service work?



GKE automatically creates an external passthrough Network Load Balancer when a LoadBalancer Service is deployed.

Client traffic is sent to the load balancer's external IP, which forwards it to the service nodes.

Nodes forward traffic to the internal LoadBalancer Service, which distributes it to a Pod.

Google Cloud

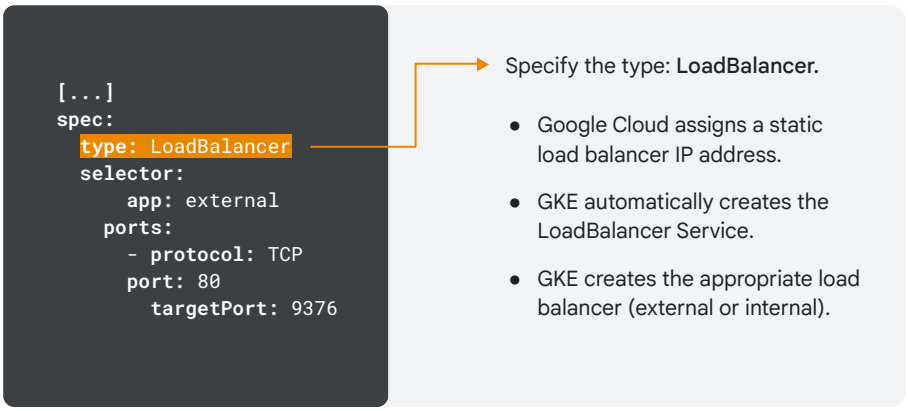
When a LoadBalancer Service is created, GKE automatically provisions an external passthrough Network Load Balancer for inbound access to the Services from outside the cluster.

Client traffic will be directed to the external IP address of the external passthrough Network Load Balancer, and the load balancer will forward the traffic on to the nodes for this Service.

The nodes will forward that traffic to the internal LoadBalancer Service, and the LoadBalancer Service will then forward the request to one of the Pods.

Proprietary + Confidential

Creating a LoadBalancer Service



Google Cloud

To create a LoadBalancer Service, you need to start by specifying the type: **`LoadBalancer`**. Google Cloud will then assign a static load balancer IP address that is accessible from outside your project.

When you specify kind: **`Service`** with type: **LoadBalancer** in the resource manifest, GKE creates a LoadBalancer Service.

GKE makes appropriate Google Cloud API calls to create either an external passthrough Network Load Balancer or an internal passthrough Network Load Balancer.

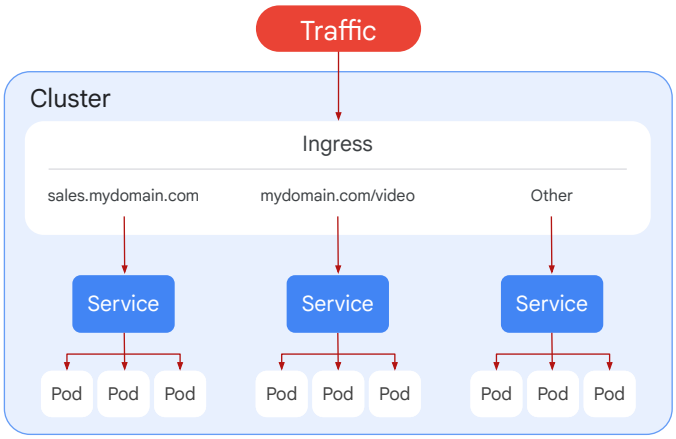
GKE creates an internal passthrough Network Load Balancer when you add the networking.gke.io/load-balancer-type: Internal annotation. Otherwise, GKE creates an external passthrough Network Load Balancer.

Proprietary + Confidential

Google Kubernetes Engine Networking

Proprietary + Confidential

Ingress resource



The Ingress resource operates one layer higher than Services.

Think of it as **a Service for Services.**

It's a collection of rules that direct external inbound connections to a set of Services within the cluster.

Google Cloud

The Ingress resource operates one layer higher than Services. You can think of it as a Service for Services.

That being said, Ingress is not a Service, or even a type of Service. It's a collection of rules that direct external inbound connections to a set of Services within the cluster.

Proprietary + Confidential

Kubernetes Ingress uses Cloud Load Balancing

HTTP/HTTPS

Application

When an Ingress resource is created in the cluster, GKE creates an Application Load Balancer and configures it to route traffic to the application.

Ingress can deliver traffic to either **NodePort Services** or **LoadBalancer Services**.

Google Cloud

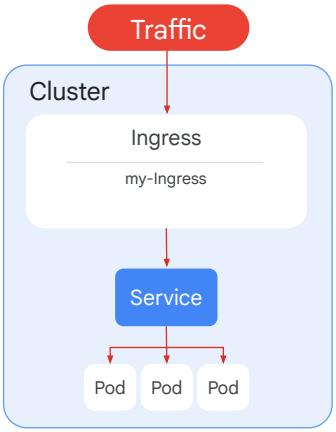
With GKE, Kubernetes Ingress resources are implemented using Cloud Load Balancing. When an Ingress resource is created in the cluster, GKE creates an Application Load Balancer and configures it to route traffic to the application.

Ingress can deliver traffic to either NodePort Services or LoadBalancer Services.

Proprietary + Confidential

Creating an Ingress

Example 1




```
apiVersion:
networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-Ingress
spec:
  backend:
    serviceName: demo
    servicePort: 80
```

The Ingress controller creates Application Load Balancer.

The backend Service is selected by name and port.

- Routes traffic to specified service/port.

Google Cloud

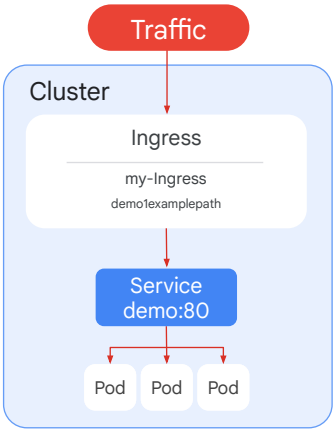
Within a simple Ingress resource, the Ingress controller creates an Application Load Balancer using the Ingress object specification.

In the object, a backend Service can be selected by specifying the Service name “demo” and the Service port “80.” This configuration tells the Application Load Balancer to route all client traffic to the Service named “demo” on port 80.

Proprietary + Confidential

Creating an Ingress

Example 2




```
apiVersion:
networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-Ingress
spec:
  Rules:
  - host: demo1.example.com
    backend:
      serviceName: demo
      servicePort: 80
```

Inside this Ingress, the specifications are rules.

- GKE only supports HTTP rules, and each rule takes the same name as the host.
- The host name can be further filtered based on the path which will have a Service backend that defines the Service's name and port.

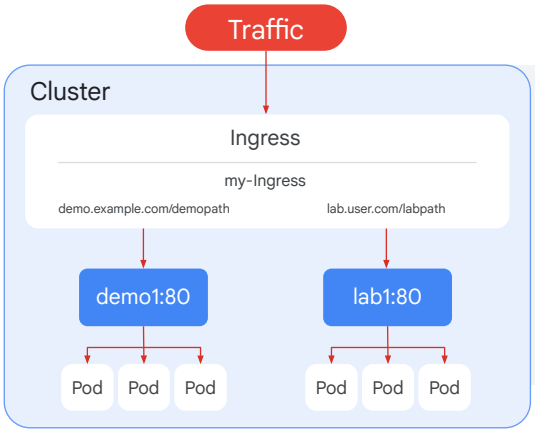
Google Cloud

Let's shift to a different example where there is an Ingress manifest and inside the specifications are rules. Currently, GKE only supports HTTP rules, and each rule takes the same name as the host. The host name can be further filtered based on the path, and a path will have a Service backend that defines the Service's name and port.

Proprietary + Confidential

Creating an Ingress

Example 3



The traffic will be redirected from the Application Load Balancer, based on the host names, to their respective backend Services.

For example, the load balancer will route traffic for demo.example.com/demopath to the Service named demo1 on port 80.

Google Cloud

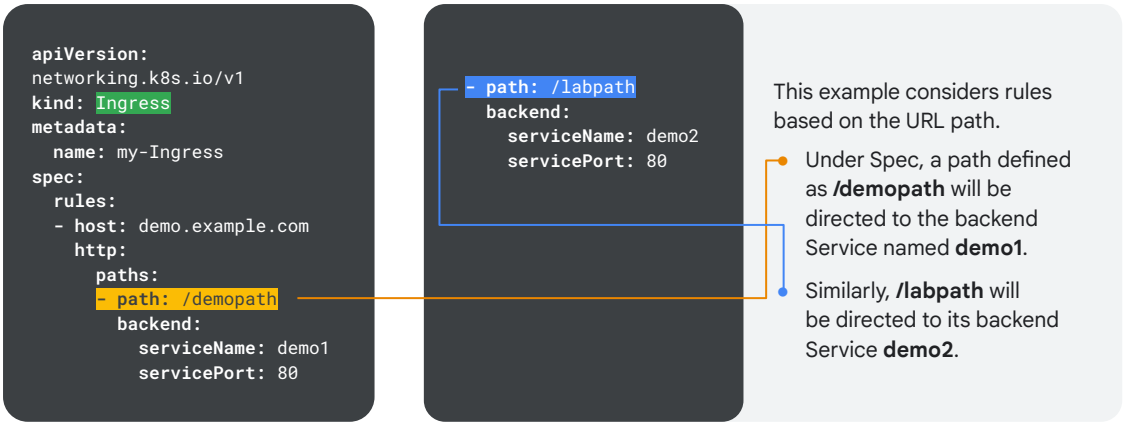
Ingress supports multiple host names for the same IP address, for example demo.example.com and lab.user.com.

The traffic will be redirected from the Application Load Balancer, based on the host names, to their respective backend Services. For example, the load balancer will route traffic for demo.example.com to the Service named demo1 on port 80.

Proprietary + Confidential

Creating an Ingress

Example 3 (continued)



Google Cloud

This example considers rules based on the URL path.

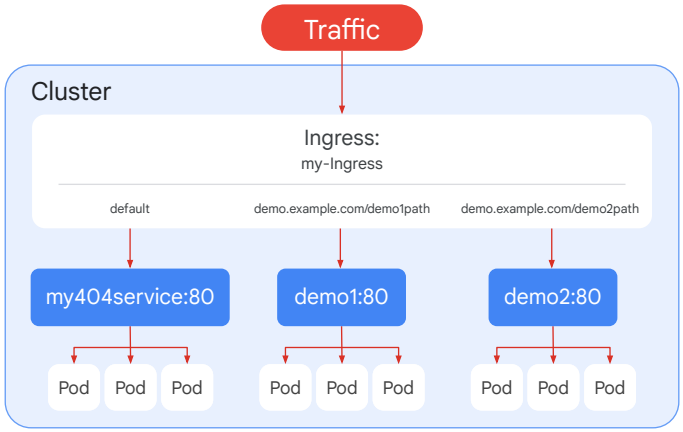
Under Spec, a path defined as /demopath will be directed to the backend Service named demo1.

Similarly, /labpath will be directed to its backend Service demo2.

Proprietary + Confidential

What happens to the traffic that doesn't match any rules?

- Traffic with no matching rules is sent to the default backend.
- Specify a default backend in the Ingress manifest to handle unmatched traffic.
- If no default backend is specified, GKE provides one that returns a 404 error.



Google Cloud

So what happens to the traffic that doesn't match any of these host-based or path-based rules? The traffic with no matching rules is simply sent to the default backend, which is a service that acts as the destination for traffic that doesn't match any of the specified paths in an Ingress manifest.

A default backend can be specified by providing a backend field in your Ingress manifest. If a default backend isn't specified, GKE will supply one that replies with error code 404.

Proprietary + Confidential

Updating an Ingress

`$ kubectl edit ingress [NAME]`	Updates the Ingress manifest.
`$ kubectl replace [FILE]`	Replaces the Ingress object manifest file entirely.

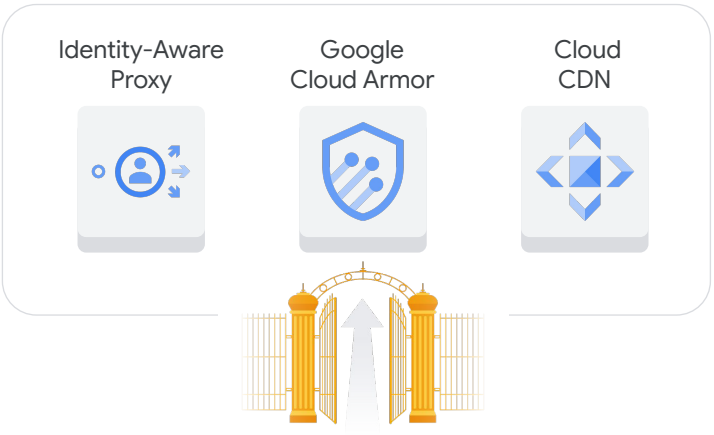
Google Cloud

Ingress can be updated with a **`kubectl edit`** command. When the Ingress resource has been updated, the API server will tell the Ingress controller to reconfigure the Application Load Balancer according to the changes that have been made.

Ingress can also be updated by using the **`kubectl replace`** command, which replaces the Ingress object manifest file entirely.

Proprietary + Confidential

Native Ingress support for Google Cloud services

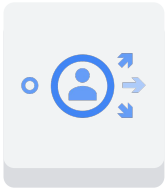


Google Cloud

Ingress for an external Application Load Balancer natively supports many Google Cloud services.

Proprietary + Confidential

Identity-Aware Proxy



Identity-Aware Proxy

Provides granular access control at the application level.

Authenticated users can have HTTPS access to the applications within a cluster without any VPN setup.

Google Cloud

Using Identity-Aware Proxy provides granular access control at the application level. With this, authenticated users can have HTTPS access to the applications within a cluster without any VPN setup.

Proprietary + Confidential

Google Cloud Armor

Google Cloud Armor

- Protects against DDoS and web attacks.
- Allows IP allow/deny lists and predefined rules.
- Customizable security rules for varied threats.

Google Cloud

Google Cloud Armor provides built-in protection against distributed denial of service (DDOS) and web attacks for clusters using an Application Load Balancer.

Security rules can be set up to allow list or deny list IP addresses or ranges. Predefined rules can also be used to defend against cross-site scripting (XSS) and SQL injection (SQLi) application-aware attacks.

Security rules can be customized to mitigate multivector attacks and restrict access using geolocation.

Proprietary + Confidential

Cloud CDN

Cloud CDN

Allows an application's content to be brought closer to its users by using more than 100 Edge points of presence.

Settings can be configured using **BackendConfig**, a custom resource used by the Ingress controller to define configuration for these Services.

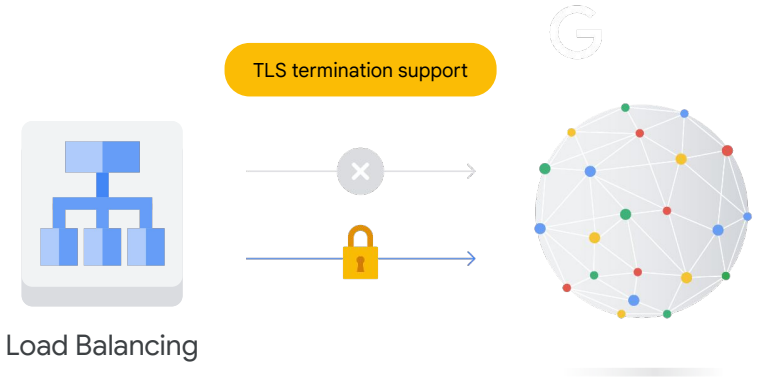
Google Cloud

Cloud CDN allows an application's content to be brought closer to its users. It does so by using more than 100 Edge points of presence.

These settings can be configured using BackendConfig. BackendConfig is a custom resource used by the Ingress controller to define configuration for all these Services.

Proprietary + Confidential

Ingress gains security features from underlying Google Cloud resources



Google Cloud

Ingress gains many security features from the underlying Google Cloud resources it relies on.

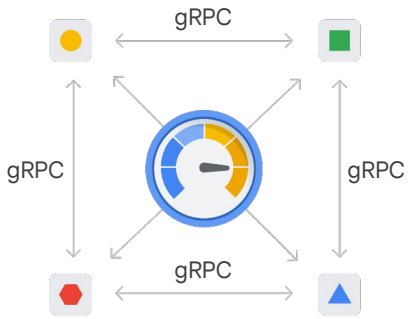
For example, Ingress provides TLS termination support at the loadbalancer at the edge of the network.

This allows the load balancer to create another connection to the destination. Although this second connection is unsecured by default, it can be secured.

This allows SSL certificates to be managed in one place. Ingress can serve multiple SSL certificates.

Proprietary + Confidential

Ingress has support for HTTP/2, HTTP/1.0, and HTTP/1.1



Microservices must communicate efficiently using a high-performance remote procedure call system.

gRPC can be used along with HTTP/2 to create performant, low-latency, scalable microservices within the cluster.

Google Cloud

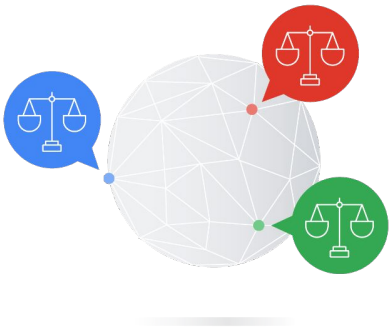
Ingress also supports the HTTP/2 standard in addition to HTTP/1.0 and HTTP/1.1.

So, if a microservices-based application is being developed, it's necessary to ensure that each microservice's communication with all the others uses a high-performance, low-overhead remote procedure call system.

gRPC is a common way to solve this, and it needs HTTP/2. So gRPC can be used along with HTTP/2 to create performant, low-latency, scalable microservices within the cluster.

Proprietary + Confidential

Global load balancing with Ingress



A single standard Ingress resource can be used to load balance traffic globally to multiple clusters across multiple regions.

This also supports location-based load balancing, called **geo-balancing** across multiple regions, which improves the availability of the cluster.

Google Cloud

Finally, with multi-cluster and multi-region support, a single standard Ingress resource can be used to load balance traffic globally to multiple clusters across multiple regions.

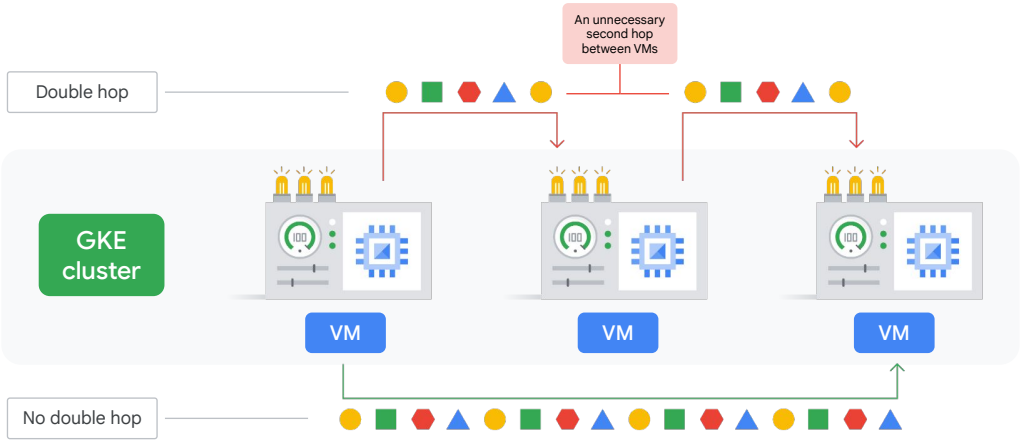
This also supports location-based load balancing, called **geo-balancing** across multiple regions, which improves the availability of the cluster.

Proprietary + Confidential

Google Kubernetes Engine Networking

Proprietary + Confidential

The double-hop dilemma



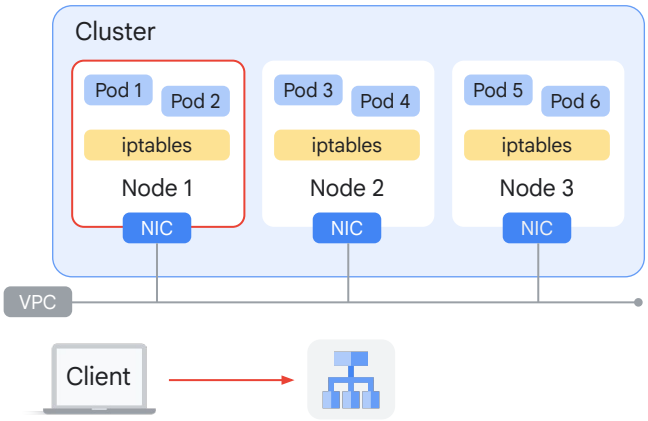
Google Cloud

With the combination of an Application Load Balancer and the Ingress object, it's possible to encounter something called the **double-hop dilemma**.

The double-hop dilemma describes when traffic makes an unnecessary second hop between VMs running containers in a GKE cluster.

Proprietary + Confidential

Traffic distribution without a container-native load balancer



The Network Load Balancer chooses a random node in the cluster and forwards the traffic to it.

In this example, there are three possible Nodes to choose from and **Node 1** is chosen.

Google Cloud

Let's explore a scenario where traffic is distributed without a container-native load balancer.

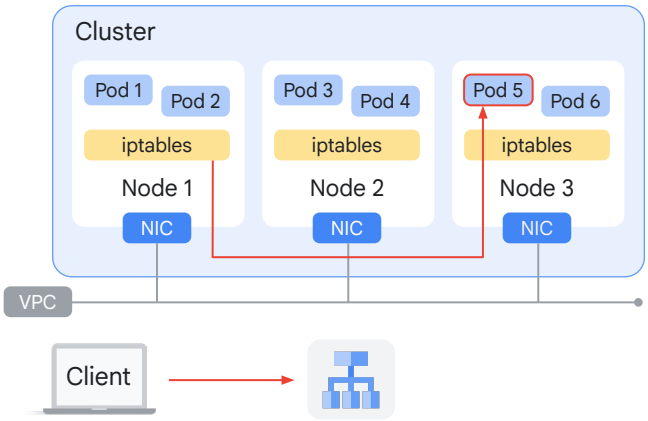
The responsibility of a regular Application Load Balancer is to distribute traffic to all nodes of an instance group, regardless of whether the traffic was intended for the Pods within that node. By default, a load balancer can route traffic to any node within an instance group.

When the client sends traffic, it is directed through the Network Load Balancer.

The Network Load Balancer chooses a random node in the cluster and forwards the traffic to it. In this example, there are three possible Nodes to choose from. Node 1 is chosen.

Proprietary + Confidential

Traffic distribution without a container-native load balancer



The initial node will use kube-proxy to select a Pod at random to handle the incoming traffic.

Node 1 chooses **Pod 5**, which isn't on this node.

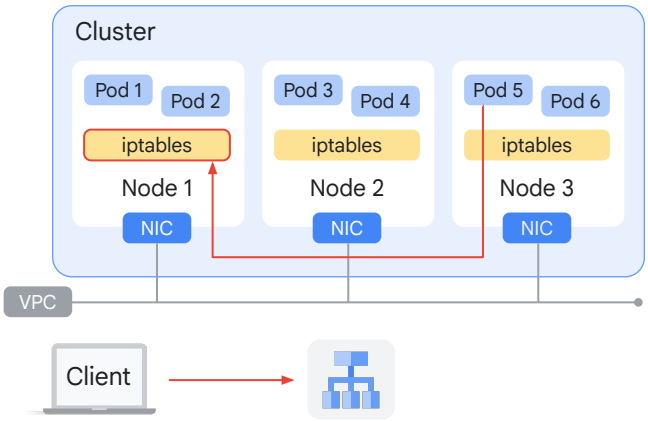
This means that **Node 1** will forward the traffic to **Pod 5** on **Node 3**.

Google Cloud

Next, to keep the Pod use as even as possible, the initial node will use **`kube-proxy`** to select a Pod at random to handle the incoming traffic. The selected Pod might be on this node or on another node in the cluster. For this example, let's say that, Node 1 chooses Pod 5, which isn't on this node. This means that Node 1 will forward the traffic to Pod 5 on Node 3.

Proprietary + Confidential

Traffic distribution without a container-native load balancer



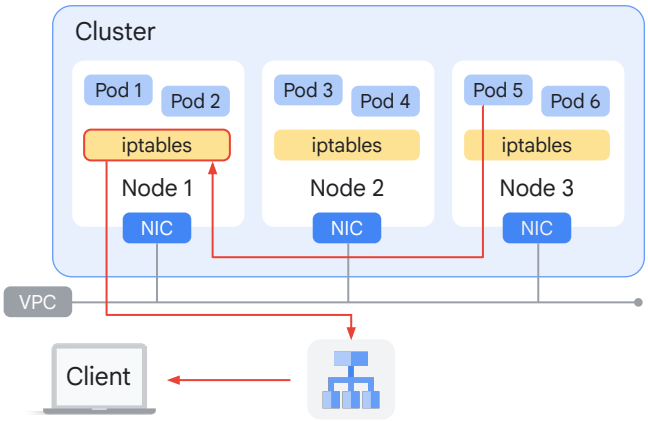
Pod 5 then directs its responses back through **Node 1**, which is when the double-hop happens.

Google Cloud

Pod 5 then directs its responses back through Node 1, which is when the double-hop happens.

Proprietary + Confidential

Traffic distribution without a container-native load balancer



Node 1 then forwards the traffic back to the Network Load Balancer, which sends it back to the client.

A double-hop is not optimal for load balancing.

Google Cloud

Node 1 then forwards the traffic back to the Network Load Balancer, which sends it back to the client.

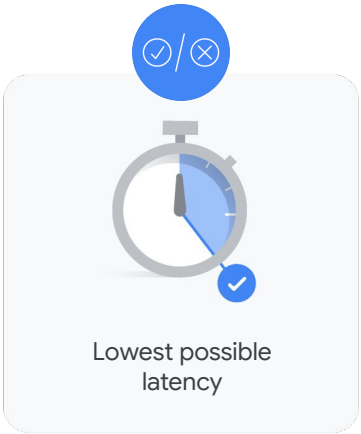
This method has two levels of load balancing, one by the load balancer, and the other by kube-proxy. This results in multiple network hops.

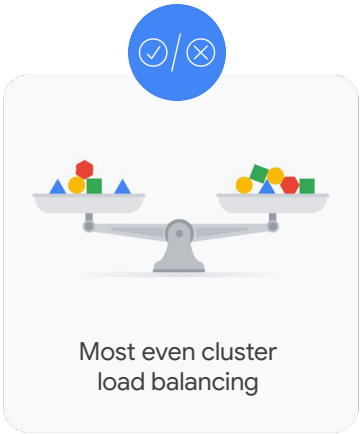
The response traffic also follows the same path.

As the name double-hop **dilemma** indicates, this method is not optimal for load balancing. This process only keeps the Pod use even at the expense of increased latency and extra network traffic–which is not ideal.

Proprietary + Confidential

What's more important?





Google Cloud

When using traditional Kubernetes networking, you'll need to decide what is more important: having the lowest possible latency or the most even cluster load balancing.

Proprietary + Confidential

Prioritizing low latency

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: LoadBalancer
    externalTrafficPolicy: Local
  selector:
     app: external
  ports:
     - protocol: TCP
       port: 80
       targetPort: 9376
```

Configure the **LoadBalancer Service** to force the kube-proxy to choose a Pod local to the node that received the client traffic.

Set the **externalTrafficPolicy** field to "Local."

This eliminates the double-hop to another node as the kube-proxy will always choose a Pod on the receiving node.

Google Cloud

If low latency is the most important, then the LoadBalancer Service can be configured to force the kube-proxy to choose a Pod local to the node that received the client traffic.

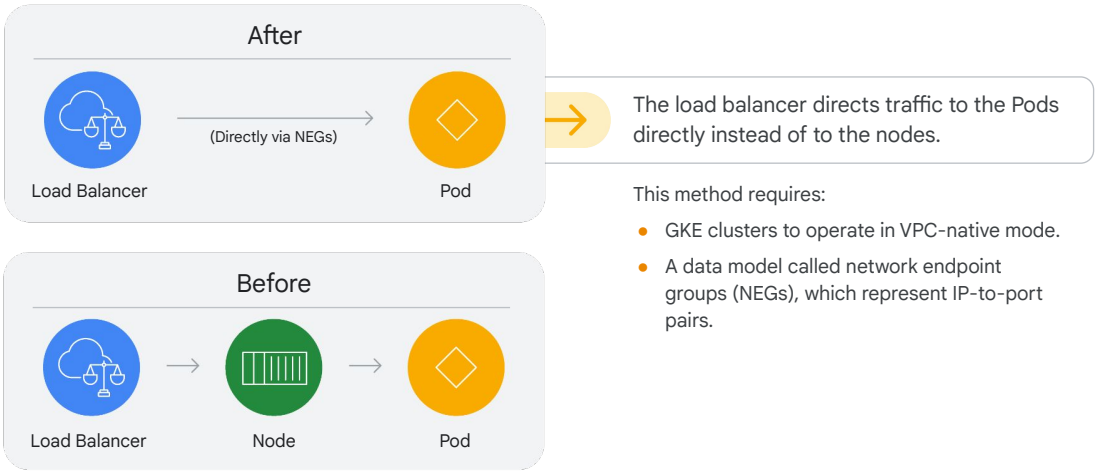
To do this, set the externalTrafficPolicy field to "Local" in the Service manifest.

This choice eliminates the double-hop to another node, as the kube-proxy will always choose a Pod on the receiving node.

When packets are forwarded from node to node, the source client IP address is preserved and directly visible to the destination Pod. Although this preserves the source IP address, it introduces a risk of creating an imbalance in cluster load.

Proprietary + Confidential

Container-native Load Balancing



Google Cloud

And if getting the most even cluster load balancing is more important, then the container-native load balancing configuration might be a better choice. With this option, the powerful Google Cloud Application Load Balancer is still used however, the load balancer will direct the traffic to the Pods directly instead of to the nodes.

This method requires GKE clusters to operate in VPC-native mode, and it relies on a data model called network endpoint groups, or NEGs.

Network endpoint groups represent IP-to-port pairs, which means that Pods can simply be just another endpoint within that group, equal in standing to compute instance VMs.

Every connection is made directly between the load balancer and the intended Pod. For example, traffic intended for Pod 3 will be routed directly from the load balancer to the IP address of Pod 3 using a network endpoint group.

Proprietary + Confidential

What's the best choice? It depends.

Lowest possible latency

Most even cluster load balancing

"Local" external-traffic policy may cause other issues because:

- It imposes constraints on the mechanisms that balance Pod traffic internally.
- The Application Load Balancer forwards traffic via nodes with no awareness of the state of the Pods themselves.

Google Cloud

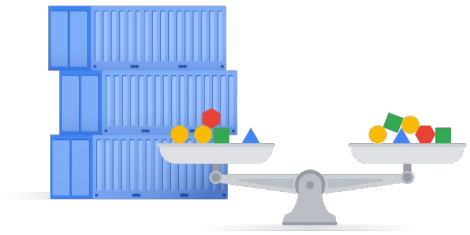
So, what's the best choice? The answer to that question depends on the application.

Both configurations can be profiled and the one that provides the best overall application performance can be selected.

However, the "Local" external-traffic policy may cause other issues as it imposes constraints on the mechanisms that balance Pod traffic internally. It may also cause issues externally as the Application Load Balancer forwards traffic via nodes with no awareness of the state of the Pods themselves.

Proprietary + Confidential

Container-native load balancing benefits



- Pods can be specified directly as endpoints for Google Cloud load balancers.
- Features, such as traffic shaping and advanced algorithms, are supported.
- Direct visibility to the Pods and more accurate health checks.
- Time it takes traffic to travel from the client to the load balancer can be measured.
- Fewer network hops in the path, which optimizes the data path.
- Support for Google Cloud networking services.

Google Cloud

There are many benefits to using container-native load balancing and Network Endpoint Groups. Let's explore a few.

One benefit is that Pods can be specified directly as endpoints for Google Cloud load balancers. This means that the traffic will be directed to the intended Pod, eliminating extra network hops.

Another benefit is that load balancer features, such as traffic shaping and advanced algorithms, are supported The load balancer can accurately distribute the traffic, since it has a direct connection to the Pod.

The next benefit is that container-native load balancing allows direct visibility to the Pods and more accurate health checks.

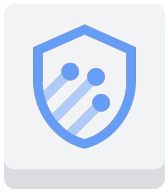
Since the source IP is preserved, the round trip time it takes traffic to travel from the client to the load balancer can be measured, which can be useful when troubleshooting issues.

This visibility can be easily extended using Google Cloud Observability.

And then there is the fact that there are fewer network hops in the path, which optimizes the data path. This improves latency and throughput, providing a better network performance overall.

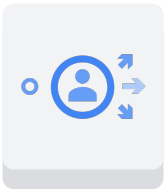
Proprietary + Confidential

Support for Google Cloud networking services



Google Cloud Armor

Cloud CDN



Identity-Aware Proxy

Google Cloud

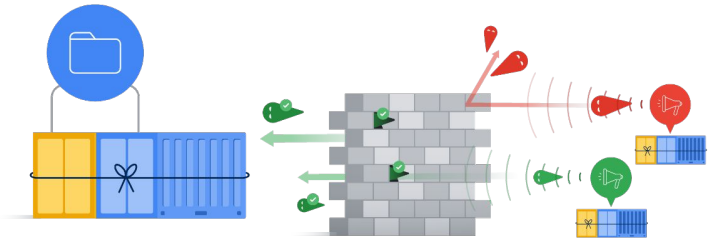
And finally, it's worth noting that there is support for Google Cloud networking services, like Google Cloud Armor, Cloud CDN, and Identity-Aware Proxy.

Proprietary + Confidential

Google Kubernetes Engine Networking

Proprietary + Confidential

Restricting access to Pods with network policies



A **network policy** is a set of firewall rules applied at the Pod level that restrict access to other Pods and Services inside the cluster.

Google Cloud

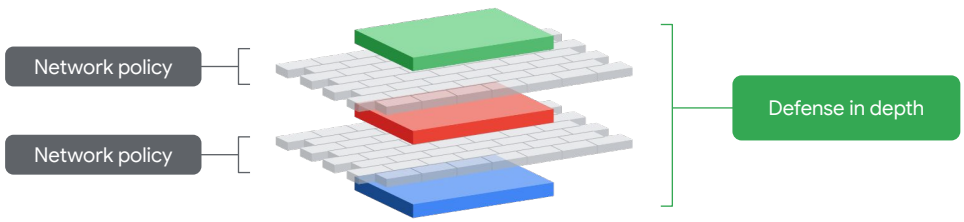
By default, all Pods can communicate with one another. However, what if access to certain Pods must be restricted?

The solution is to implement a network policy.

A network policy is a set of firewall rules applied at the Pod level that restrict access to other Pods and Services inside the cluster.

Proprietary + Confidential

Network policies can be used to restrict access at each stack level



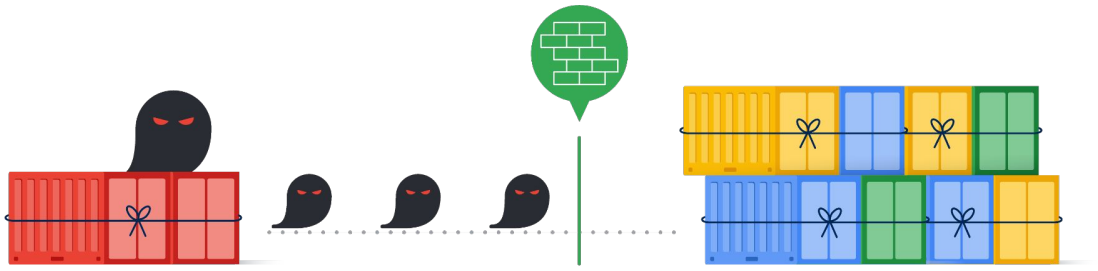
Google Cloud

For example, in a multi-layered application, network policies can be used to restrict access at each stack level.

A web layer could only be accessed from a certain Service, and an application layer beneath this could only be accessed from the web layer. This effectively promotes defense in depth.

Proprietary + Confidential

Why consider network policies?



Imagine an attacker compromised one of your Pods by exploiting a security vulnerability in it.

Network policies let you lock down network traffic to only the pathways you want traffic to travel.

Google Cloud

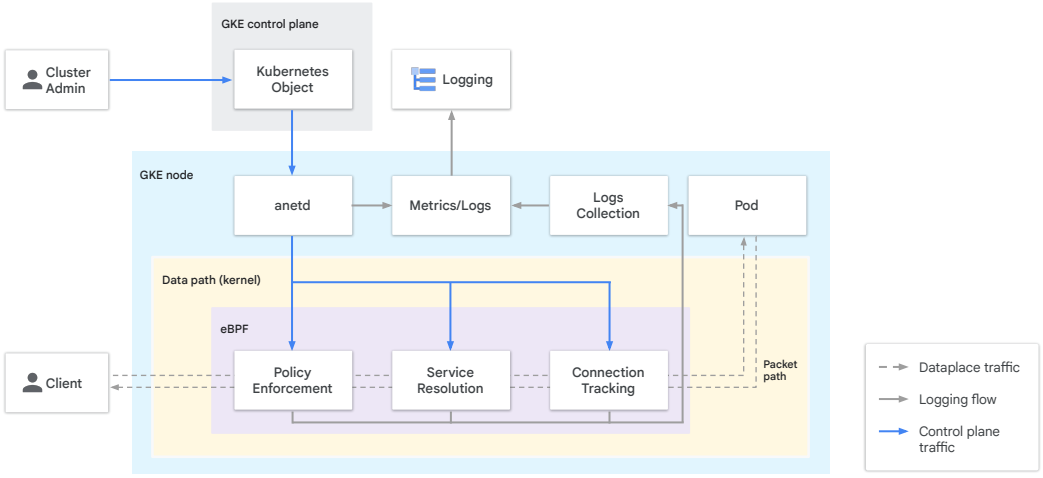
So, why should you consider using network policies?

Imagine an attacker compromised one of your Pods by exploiting a security vulnerability in it. Just by gaining access, an attacker would have the power to probe outwards from the compromised Pod to all other Pods in your cluster. And they might even find other vulnerabilities. As a result, it would be wise to limit the attack surface of your cluster.

Network policies let you lock down network traffic to only the pathways you want traffic to travel.

Proprietary + Confidential

Enabling network policies with GKE Dataplane V2



Google Cloud

While network policies can be complex to manage, GKE Dataplane V2 simplifies the process with built-in policy enforcement, eliminating extra overhead.

Autopilot clusters in GKE come with Dataplane V2 enabled by default, simplifying network policy management. For standard clusters, you can manually enable Dataplane V2 and take advantage of its streamlined network policies, provided your cluster meets the minimum requirements.

GKE Dataplane V2 leverages eBPF, a powerful technology that operates within the Linux kernel**.** As packets arrive at a GKE node, GKE Dataplane V2 decides how to route and process the packets.

Crucially, Dataplane V2 taps into Kubernetes-specific metadata embedded in these packets. This enables efficient packet processing and provides detailed, annotated logs for comprehensive network visibility.

Proprietary + Confidential

Define the actual network policy

Defining the network policy is necessary after enabling network policy enforcement.

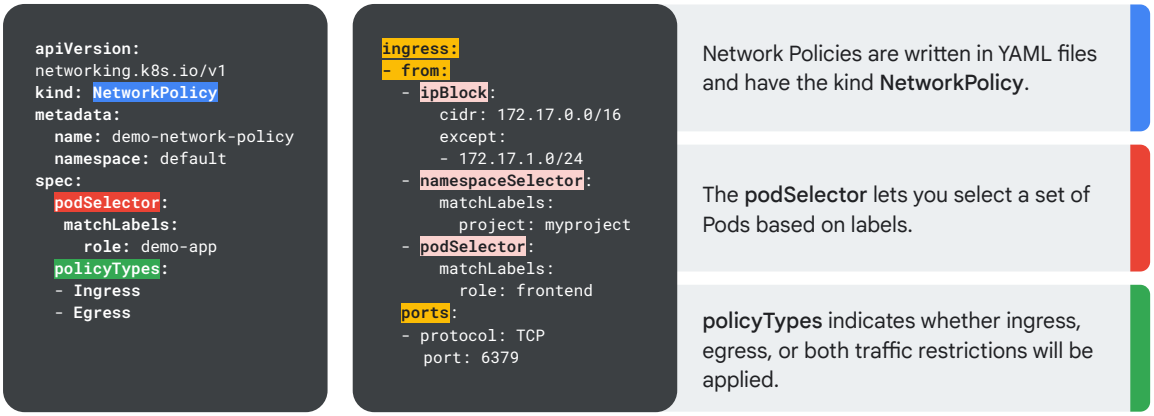
Enabling network policy enforcement may require **increasing cluster size** due to increased resource consumption.

Google Cloud

After you enable network policy enforcement for your cluster, you need to define the actual network policy. Enabling network policy enforcement consumes additional resources in nodes. This means that you might need to increase your cluster's size to keep your scheduled workloads running.

Proprietary + Confidential

NetworkPolicy, podSelector, and policyTypes



Google Cloud

Network Policies are written in YAML files and have the kind **NetworkPolicy**.

The **podSelector** lets you select a set of Pods based on labels.

If “podSelector” isn’t provided, or is empty, the networking policy will be applied to all Pods in the namespace.

policyTypes indicates whether ingress, egress, or both traffic restrictions will be applied.

If policyTypes is left empty, a default ingress policy will apply automatically, and an egress policy won’t be specified.

Proprietary + Confidential

The Ingress section of the policy

```
apiVersion:
networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: demo-network-policy
  namespace: default
spec:
  podSelector:
   matchLabels:
     role: demo-app
  policyTypes:
  - Ingress
  - Egress
```

```
ingress:
- from:
  - ipBlock:
      cidr: 172.17.0.0/16
      except:
      - 172.17.1.0/24
  - namespaceSelector:
      matchLabels:
        project: myproject
  - podSelector:
      matchLabels:
        role: frontend
ports:
- protocol: TCP
  port: 6379
```

In the Ingress section of the policy, there are two main sections: from and ports.

- The from section can be from three sources:
  - ipBlock
  - namespaceSelector
  - podSelector
- The ports section states what ports ingress will be accepted from.

Google Cloud

In the Ingress section of the policy, there are two main sections: “**from**” and “**ports**.”

The “**from**” section can be from three sources:

- **ipBlock**
- **namespaceSelector**
- **podSelector**

The “**ports**” section states what ports ingress will be accepted from.

The combination of these 2 elements defines where traffic is allowed to enter the network.

Proprietary + Confidential

The Egress section of the policy

```
apiVersion:
networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: demo-network-policy
  namespace: default
spec:
  podSelector:
   matchLabels:
     role: demo-app
  policyTypes:
  - Ingress
  - Egress
```

```
egress:
- to:
  - ipBlock:
      cidr: 10.0.0.0/24
  ports:
  - protocol: TCP
    port: 5978
```

In the **Egress** section of the policy, there are two main sections: **to** and **ports**.

In this example, traffic destined for network 10.0.0.0/24 on TCP port 5978 will be permitted to egress from the demo-app Pods.

Google Cloud

In the Egress section of the policy, there are two main sections: “**to**” and “**ports**.”

In the example, traffic destined for network 10.0.0.0/24 on TCP port 5978 will be permitted to egress from the demo-app Pods.

Remember, applying network policies does nothing if you haven’t enabled network policy on your cluster.

Proprietary + Confidential

Disabling a network policy

Disable a network policy for a cluster

```
gcloud container clusters update [NAME] \
--no-enable-network-policy
```

Google Cloud console

Step 1: Disable the network policy for Nodes.

Step 2: Disable the network policy for the control plane.

If no network policies exist, all traffic between Pods in the namespace is allowed.

Google Cloud

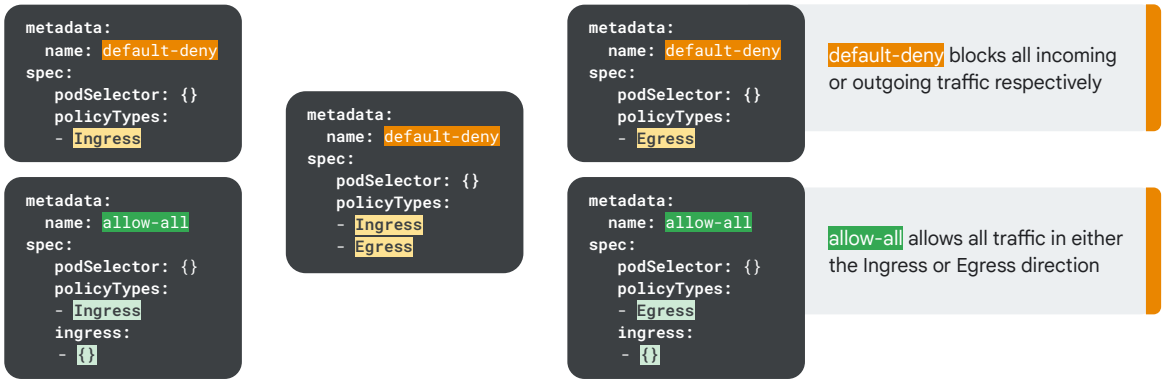
You can use the “gcloud container clusters update” command to disable a network policy. In the Google Cloud console, disabling a network policy is a two-step process.

You first disable the network policy for nodes, and then you disable the network policy for the control plane.

If no network policies exist, all ingress and egress traffic will be allowed between the Pods in the namespace regardless of whether network policy enforcement is enabled or disabled.

Proprietary + Confidential

Default policies for Ingress and Egress



Google Cloud

Default-deny policy for ingress, or **Default-deny** policy for Egress block all incoming or outgoing traffic respectively. An "**allow-all**" policy exists for both ingress and egress. This allows all traffic in that direction.

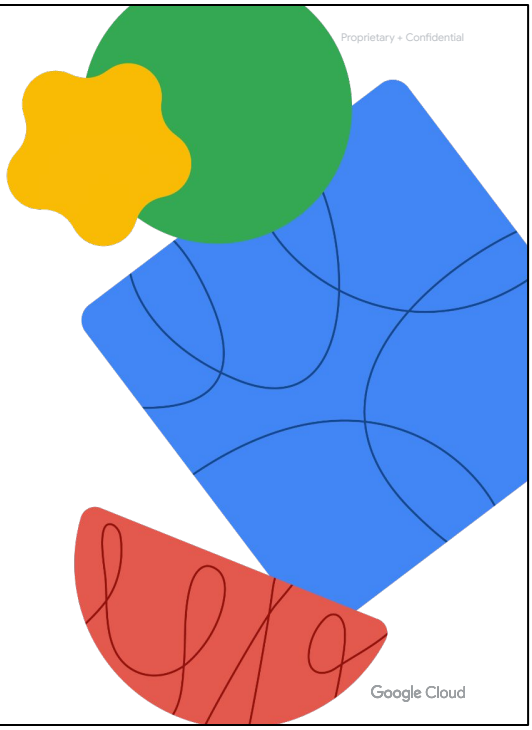
Note the difference in wording between this and the "**default-deny**" keyword.

For the default policies, a **policyType: Ingress** or **policyType: Egress** is required depending on the direction of the traffic.

Proprietary + Confidential

Quiz questions

Let's pause for a quick check in.



Proprietary + Confidential

Quiz | Question 1

Question

Your Pod has been rescheduled, and the original IP address that was assigned to the Pod is no longer accessible. What is the reason for this?

A. The new Pod IP address is blocked by a firewall.

B. The old Pod IP address is blocked by a firewall.

C. The Pod IP range for the cluster is exhausted.

D. The new Pod has received a different IP address.

Proprietary + Confidential

Quiz | Question 1

Answer

Your Pod has been rescheduled, and the original IP address that was assigned to the Pod is no longer accessible. What is the reason for this?

A. The new Pod IP address is blocked by a firewall.

B. The old Pod IP address is blocked by a firewall.

C. The Pod IP range for the cluster is exhausted.

D. The new Pod has received a different IP address.

Google Cloud

Your Pod has been rescheduled, and the original IP address that was assigned to the Pod is no longer accessible. What is the reason for this?

A. The new Pod IP address is blocked by a firewall.: This is the **incorrect answer** because firewall rules can block traffic, they wouldn't typically target specific Pod IPs in a way that causes this issue.
B. The old Pod IP address is blocked by a firewall.: This is the **incorrect answer** because firewall rules can block traffic, they wouldn't typically target specific Pod IPs in a way that causes this issue.
C. The Pod IP range for the cluster is exhausted.: This is the **incorrect answer** because GKE clusters are designed with large IP ranges for Pods, making exhaustion unlikely unless there's a misconfiguration.
D. **The new Pod has received a different IP address.**: This is the **correct answer** because Pod IP addresses are temporary. If a Pod deployment is rescheduled, the Pod gets assigned a new IP address. .

Proprietary + Confidential

Quiz | Question 2

Question

You have updated your application and deployed a new Pod. How can you ensure consistent network access to the Pod throughout its lifecycle?

A. Deploy a Kubernetes Service with a selector that locates the application's Pods.

B. Register the fully qualified domain name of the application's Pod in DNS.

C. Add the fully qualified domain name of the application's Pod to your local hostfile.

D. Add metadata annotations to the Pod manifest that define a persistent DNS name.

Proprietary + Confidential

Quiz | Question 2

Answer

You have updated your application and deployed a new Pod. How can you ensure consistent network access to the Pod throughout its lifecycle?

A. Deploy a Kubernetes Service with a selector that locates the application's Pods. 

B. Register the fully qualified domain name of the application's Pod in DNS.

C. Add the fully qualified domain name of the application's Pod to your local hostfile.

D. Add metadata annotations to the Pod manifest that define a persistent DNS name.

Google Cloud

You have updated your application and deployed a new Pod. How can you ensure consistent network access to the Pod throughout its lifecycle?

A. **Deploy a Kubernetes Service with a selector that locates the application's Pods.**:
This is the **correct answer** because Kubernetes Services provide a stable endpoint
B. Register the fully qualified domain name of the application's Pod in DNS.: This is the **incorrect answer** because Pod IP addresses are ephemeral. This makes fully qualified domain name of the application's Pod unreliable for maintaining consistent network access.
C. Add the fully qualified domain name of the application's Pod to your local hostfile.: This is the **incorrect answer** because Pod IP addresses are ephemeral. This makes fully qualified domain name of the application's Pod unreliable for maintaining consistent network access.
D. Add metadata annotations to the Pod manifest that define a persistent DNS name.: This is the **incorrect answer** because Pod IP addresses are ephemeral. This makes DNS unreliable for maintaining consistent network access.

Proprietary + Confidential

Quiz | Question 3

Question

You are designing a GKE solution. One of your requirements is that network traffic load balancing should be directed to Pods instead of balanced across nodes. How can you enable this for your environment?

A. Configure or migrate your cluster to VPC-Native Mode and deploy a container-native load balancer.

B. Set the externalTrafficPolicy field to local in the YAML manifest for your external services.

C. Configure all external access for your application using Ingress resources rather than services.

D. Configure affinity and anti-affinity rules that ensure your application's Pods are distributed across nodes.

Proprietary + Confidential

Quiz | Question 3

Answer

You are designing a GKE solution. One of your requirements is that network traffic load balancing should be directed to Pods instead of balanced across nodes. How can you enable this for your environment?

A. Configure or migrate your cluster to VPC-Native Mode and deploy a container-native load balancer.

B. Set the externalTrafficPolicy field to local in the YAML manifest for your external services.

C. Configure all external access for your application using Ingress resources rather than services.

D. Configure affinity and anti-affinity rules that ensure your application's Pods are distributed across nodes.

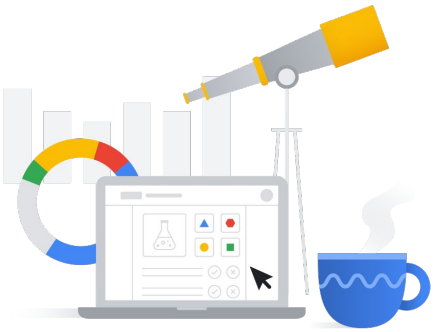
Google Cloud

You are designing a GKE solution. One of your requirements is that network traffic load balancing should be directed to Pods instead of balanced across nodes. How can you enable this for your environment?

A. Configure or migrate your cluster to VPC-Native Mode and deploy a container-native load balancer.: This is the **correct answer** because **Container-native Load Balancing** allows a load balancer to direct traffic straight to pods, bypassing nodes. It requires VPC-native mode and Network Endpoint Groups.
B. Set the externalTrafficPolicy field to local in the YAML manifest for your external services.: This is the **incorrect answer** because **externalTrafficPolicy** field to local: This prioritizes low latency by keeping traffic within a node, but it doesn't directly balance traffic to Pods..
C. Configure all external access for your application using Ingress resources rather than services.: This is the **incorrect answer** because **Ingress** manages external access rules, but doesn't inherently enable pod-level load balancing..
D. Configure affinity and anti-affinity rules that ensure your application's Pods are distributed across nodes.: This is the **incorrect answer** because these rules control pod placement on nodes, not the load balancing method itself..

Proprietary + Confidential

Lab: Configuring Google Kubernetes Engine Networking



01 Configure a cluster for authorized network control plane access.

02 Configure a Cluster network policy.

Google Cloud

It's time for some hands-on practice with GKE.

In the lab titled "Configuring Google Kubernetes Engine Networking," you'll:

- Configure a cluster for authorized network control plane access
- Configure a cluster network policy to restrict communication between the Pods.