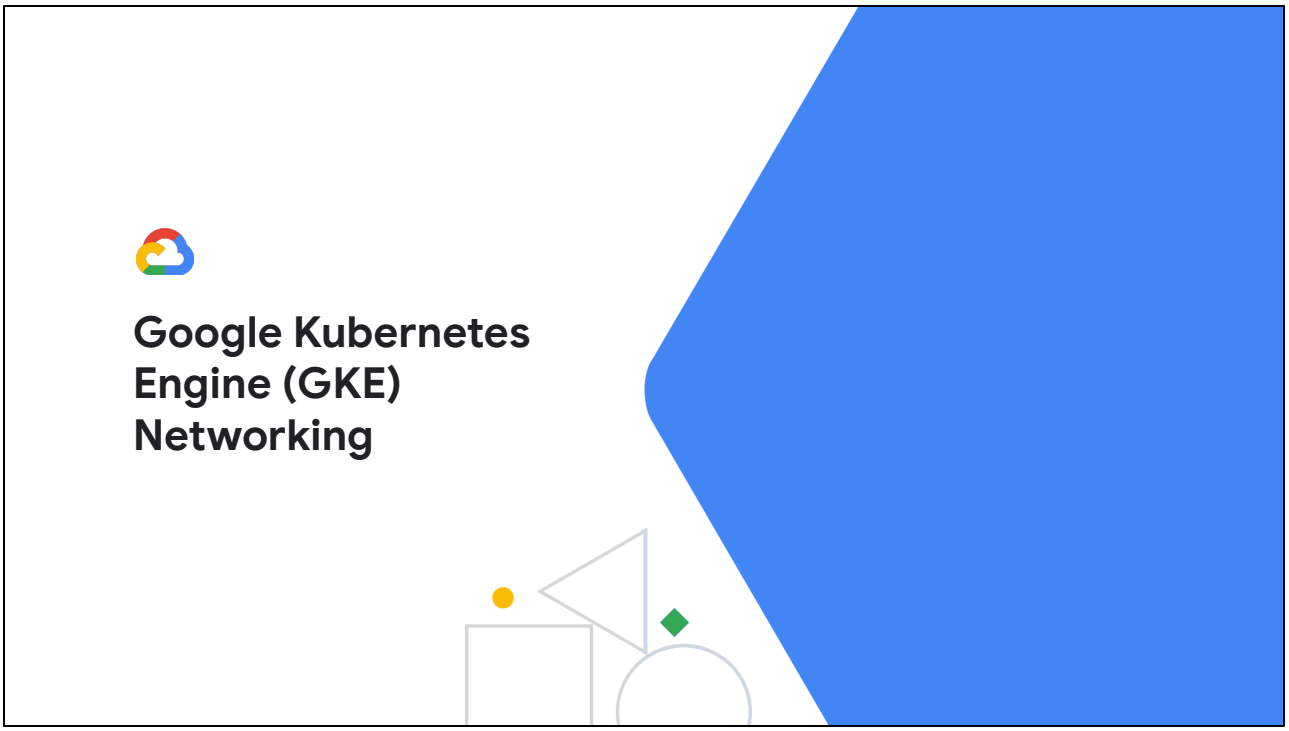


Welcome to Google Kubernetes Engine Networking. When designing your application, you need to understand what other types of applications it will talk to and where they may be located. You also need to consider how to expose them to the outside world for consumption, and how to balance incoming traffic to cope with varying workload demands. That's the theme of this module.

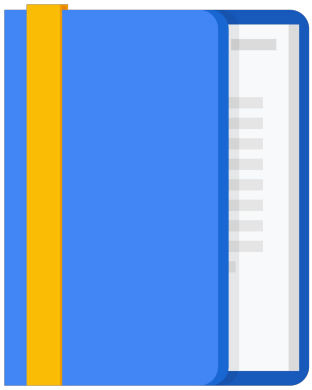
Learn how to ...

Create Services to expose applications that are running within Pods.

Use load balancers to expose Services to external clients.

Create Ingress resources for HTTP(S) load balancing.

Leverage container-native load balancing to improve Pod load balancing.



Google Cloud

In this module, you'll learn how to create Services to expose applications running within Pods, allowing them to communicate with each other, and the outside world, use load balancers to expose Services to external clients and spread incoming traffic across your cluster as evenly as possible, create Ingress resources for HTTP or HTTPS load balancing and leverage container-native load balancing to allow you to directly configure Pods as network endpoints with Cloud Load Balancing.

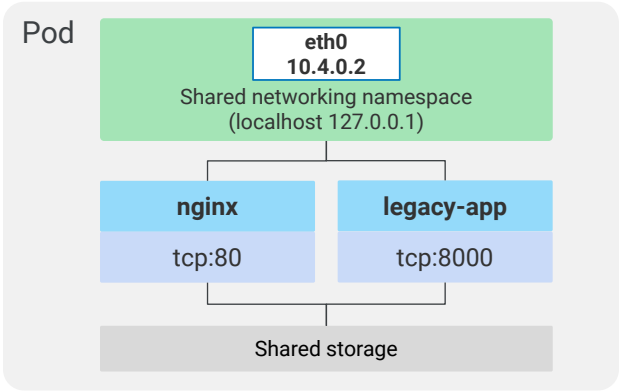
Agenda

- Pod Networking
- Services
- Service Types and Load Balancers
- Ingress Resource
- Container-Native Load Balancing
- Network Security
- Lab: Configuring Google Kubernetes Engine (GKE) Networking
- Lab: Creating Services and Ingress Resources
- Summary

Google Cloud

The Kubernetes networking model relies heavily on IP addresses. Services, Pods, containers, and nodes communicate using IP addresses and ports. Kubernetes provides different types of load balancing to direct traffic to the correct Pods. So, let's start by reviewing basic Pod networking.

A Pod is a group of containers with shared storage and networking

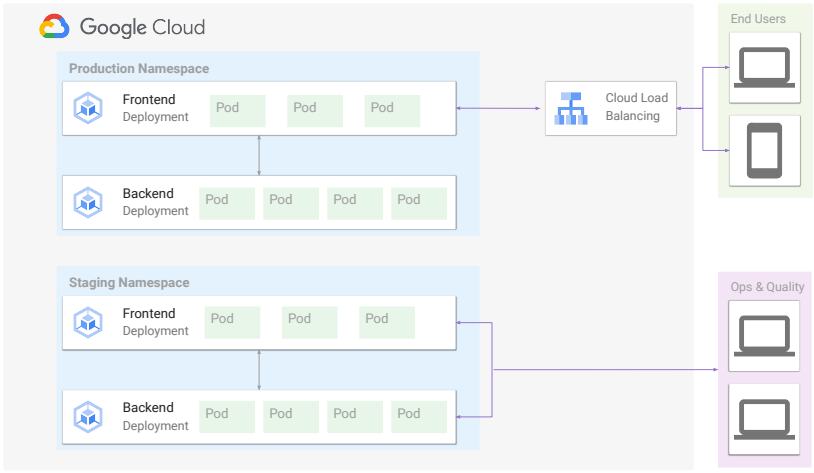


Google Cloud

Remember, a Pod is a group of containers with shared storage and networking. This is based on the “IP-per-pod” model of Kubernetes. With this model, each Pod is assigned a single IP address, and the containers within a Pod share the same network namespace, including that IP address.

For example, you might have a legacy application that uses nginx as a reverse-proxy for client access. The nginx container runs on TCP port 80, and the legacy application runs on TCP port 8000. Because both containers share the same networking namespace, the two containers appear as though they’re installed on the same machine. The nginx container will contact the legacy application by establishing a connection to “localhost” on TCP port 8000.

Your workload doesn't run in a single Pod



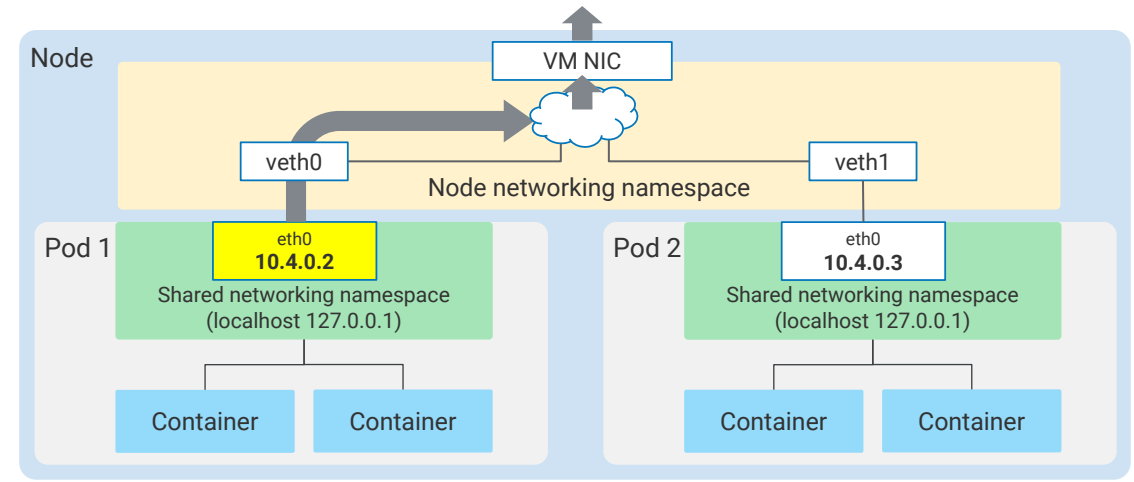
Google Cloud

This works well for a single Pod, but your workload doesn't run in a single Pod.

Your workload is composed of many different applications that need to talk to each other.

So how do Pods talk to other Pods?

Pod-to-Pod communication on the same node



Google Cloud

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

On a node, the 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 VM.

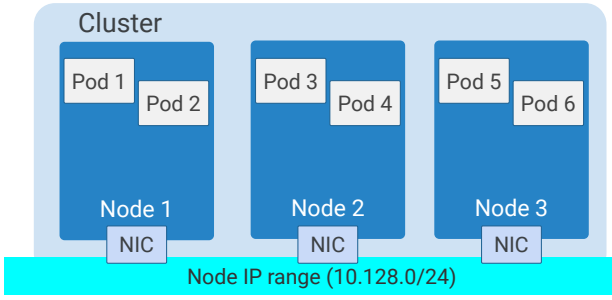
This allows the two Pods to communicate on the same node.

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.

Nodes get Pod IP addresses from address ranges assigned to your Virtual Private Cloud



Name	Zone	Internal IP	External IP
gke-standard-cluster-1-default-pool-ec9def5a-g3zr	us-central1-a	10.128.0.4 (nic0)	35.238.171.108
gke-standard-cluster-1-default-pool-ec9def5a-sd3m	us-central1-a	10.128.0.3 (nic0)	35.184.108.27
gke-standard-cluster-1-default-pool-ec9def5a-zd4g	us-central1-a	10.128.0.2 (nic0)	35.239.9.246

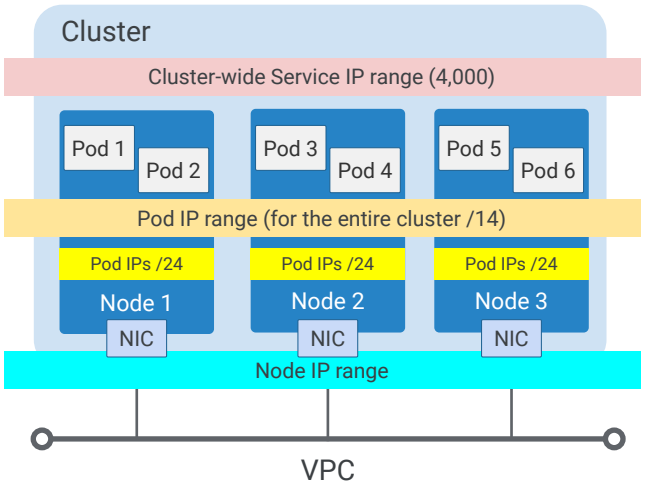
Google Cloud

In GKE, the nodes will get the Pod IP addresses from address ranges assigned to your Virtual Private Cloud, or *VPC*.

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

When you deploy GKE, 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.

Addressing the Pods



Google Cloud

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

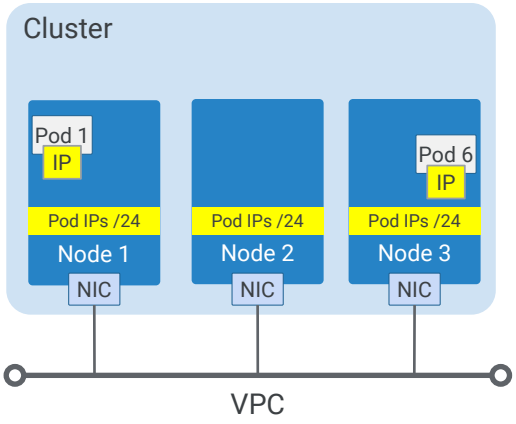
On Google Cloud, Alias IPs allow you to configure additional secondary IP addresses or IP ranges on your Compute Engine VM instances. VPC-Native GKE clusters automatically create an Alias IP range to reserve approximately 4,000 IP addresses for cluster-wide Services that you may create later. This mitigates the problem of unexpectedly running out of service IP addresses, which as you'll learn later, your applications use to talk to one another.

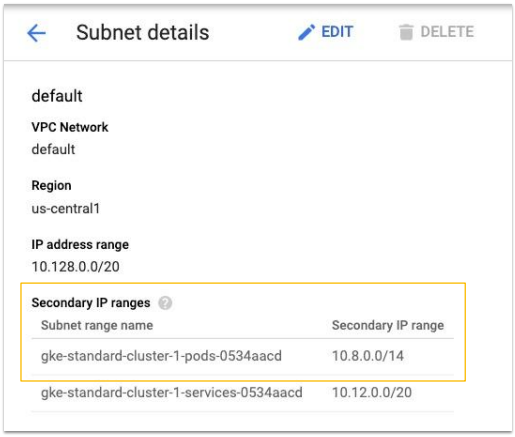
VPC-Native GKE clusters also create a separate Alias IP range for your Pods. Remember, each Pod must have a unique address, so this address space will be large. By default the address range uses a /14 block, which contains over 250,000 IP addresses. That's a lot of Pods.

In reality, Google doesn't expect you to run 250,000 Pods in a single cluster. Instead, that massive 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.

The number of nodes you expect to use and the maximum number of pods per node are configurable, so you don't have to reserve a whole /14 for this.

Pod-to-Pod communication





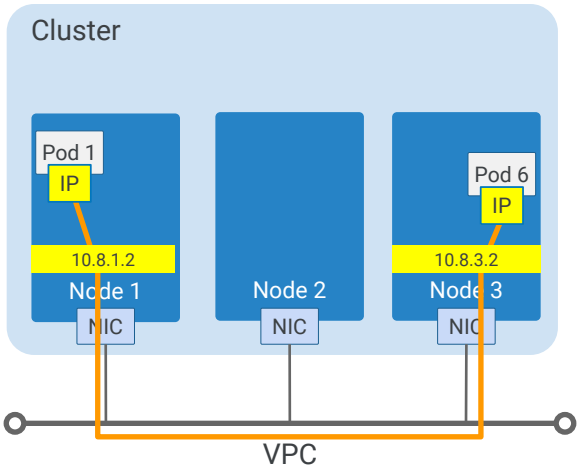
Google Cloud

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

The Pods' IP addresses are part of an address range called an Alias IP. GKE automatically configures your VPC to recognize this range of IP addresses as an authorized secondary subnet of IP addresses.

As a result, the Pod's traffic is permitted to pass the anti-spoofing filters on the network.

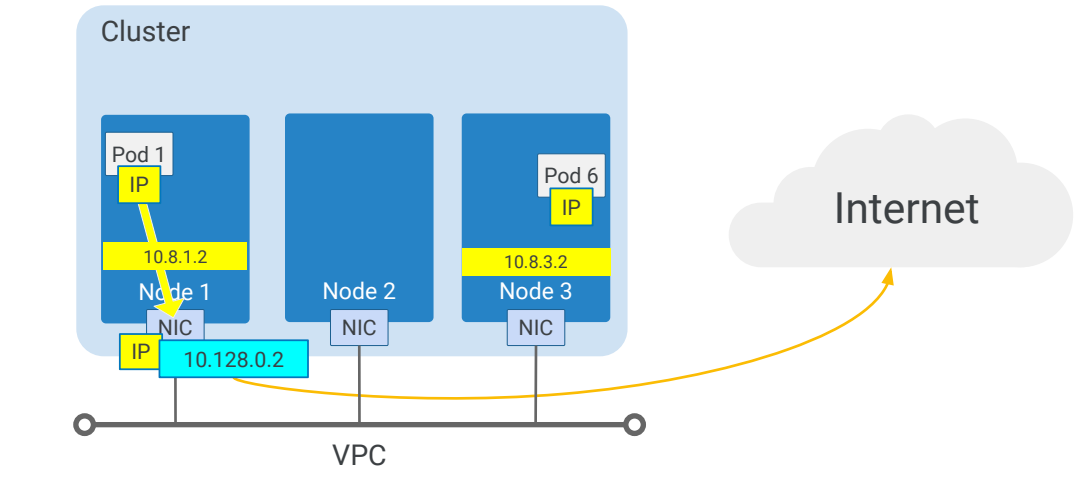
Pods can connect directly using native IP addresses



Google Cloud

Because each node maintains a separate IP address space for its Pods, the nodes don't need to perform Network Address Translation on the Pod IP addresses. That means that the Pods can directly connect to each other using their native IP addresses.

Communicating outside Google Cloud



Google Cloud

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

Agenda

Pod Networking

Services

Service Types and Load Balancers

Ingress Resource

Container-Native Load Balancing

Network Security

Lab: Configuring Google Kubernetes Engine (GKE) Networking

Lab: Creating Services and Ingress Resources

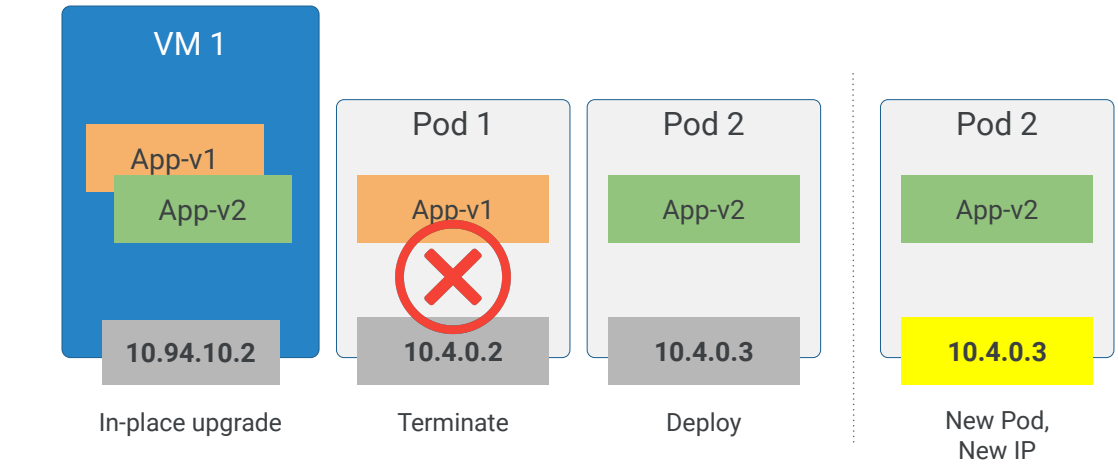
Summary

Google Cloud

In an ever-changing container environment, Services give Pods a stable IP address and name that remains the same through updates, upgrades, scalability changes, and even Pod failures. Instead of connecting to a specific Pod, applications in Kubernetes rely on Services to locate suitable Pods and forward the traffic via those Services rather than directly to Pods.

In this lesson, you'll learn about the different ways to find Services in GKE.

Pods versus VMs: Durability



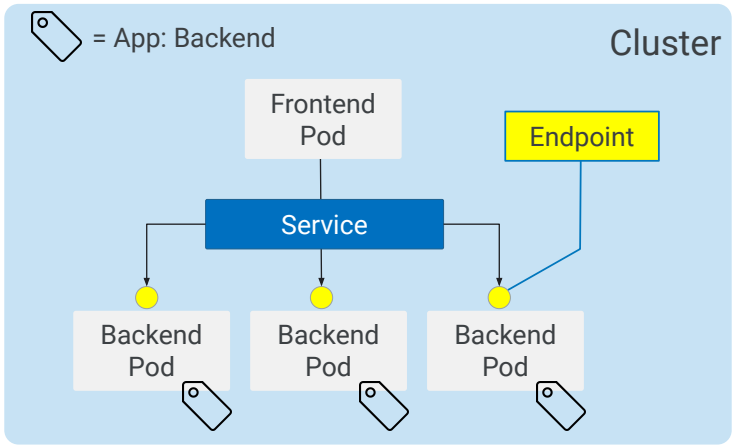
Google Cloud

Virtual machines and Pods have very different life cycles.

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

As a result of the new Pod deployment, the updated containerized version of the application gets a new IP address. Also, if a Pod is rescheduled for any reason, then the Pod gets a new IP address. This unexpected change of addresses could cause significant service disruptions in large, quickly changing environments. Pod IP addresses are ephemeral. Therefore, you need a more dependable way to locate the applications running in your cluster. Fortunately, Kubernetes has an answer: Services.

A Kubernetes Service creates endpoints

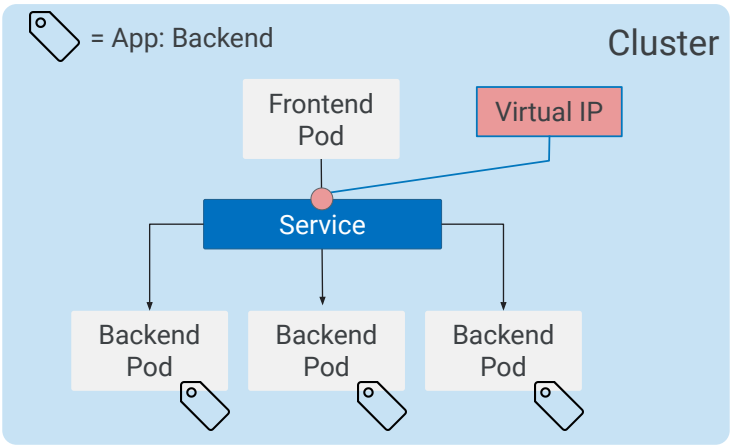


Google Cloud

A Kubernetes Service is an object that creates a dynamic collection of IP addresses, called endpoints, that belong to Pods matching the Service's label selector.

When you create a Service, that Service is issued a static Virtual IP address from the pool of IP addresses that the cluster reserves for Services.

A Service is issued a static Virtual IP address



Google Cloud

The Virtual IP is durable. It is published to all nodes in the cluster. It doesn't change, even if all of the pods behind it change.

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

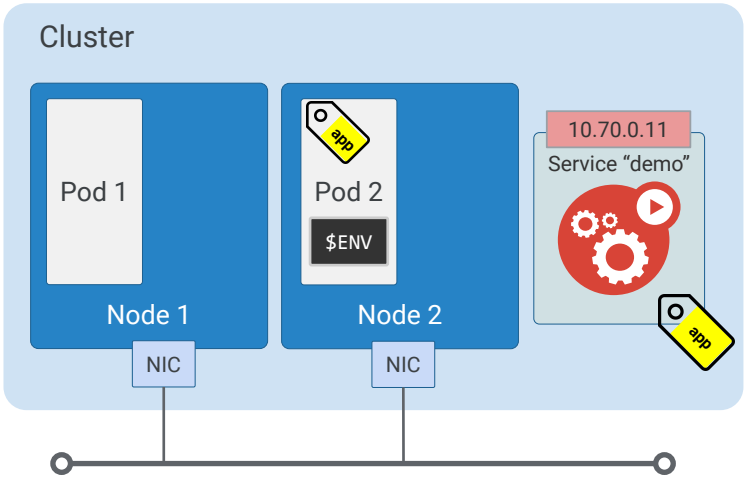
There are several ways to find a Service in GKE

- **Environment Variables**
- Kubernetes DNS
- Istio

Google Cloud

There are several ways to find a Service in GKE. Let's start by looking at the use of environment variables for Service discovery. This is enabled by default but it is not the most robust mechanism for discovery.

Environment variables for Service discovery (1/2)

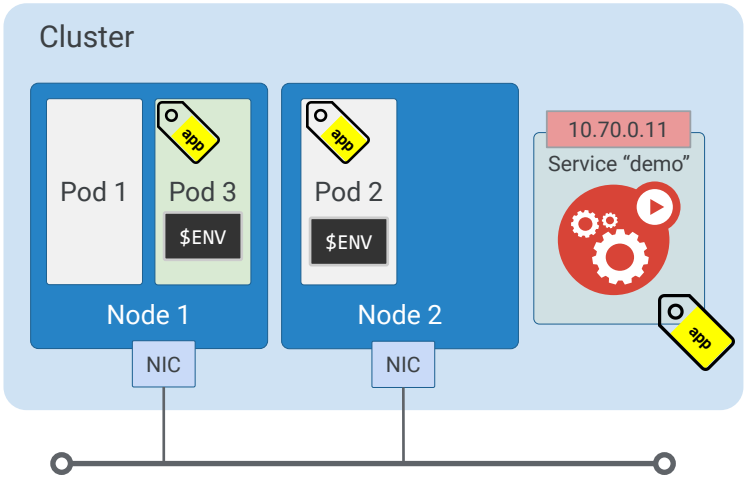


Google Cloud

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.

In our example, we have a Service that matches Pod 2; therefore if Pod 2 is started after the demo Service has been created it has a set of environment variables for that Service.

Environment variables for Service discovery (2/2)



Google Cloud

When Pod 3 starts running, which also matches the Service, Node 1 adds the environment variables for the Service to Pod 3.

The problem with this method can be seen if you consider Pod 1. Pod 1 was already running when the Service was created and it will not have the environment variables for the Service set. Similarly if changes are made to a Service after Pods have been started those changes will not be visible to Pods that are already running. Pods will only see the changes that were made up to the point where they are started because the environment variables are defined inside the Pods when the Pods are started.

Example of environment variables for a Service

```
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
```

```
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.

Generally speaking, it's a better practice to use DNS for Service discovery than to rely on environment variables. DNS names are more discoverable than environment variables. And DNS changes can be visible to Pods during their lifetimes. That's different from environment variables that Pods inherit from Kubernetes, because the initial value that Pods get when they are started remains the same throughout their lifetimes. So if you need Pods to see the effect of a change you make, you must stop them and let Kubernetes restart them.

There are several ways to find a Service in GKE

Environment Variables

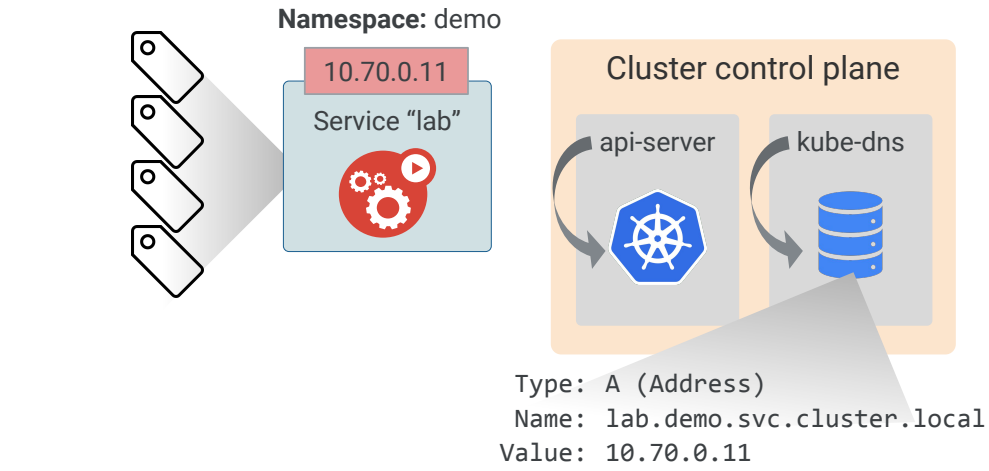
Kubernetes DNS

Istio

Google Cloud

Since using environment variables for Service discovery has drawbacks, let's look at how Kubernetes DNS can help you with Service discovery.

Service discovery through Kubernetes DNS



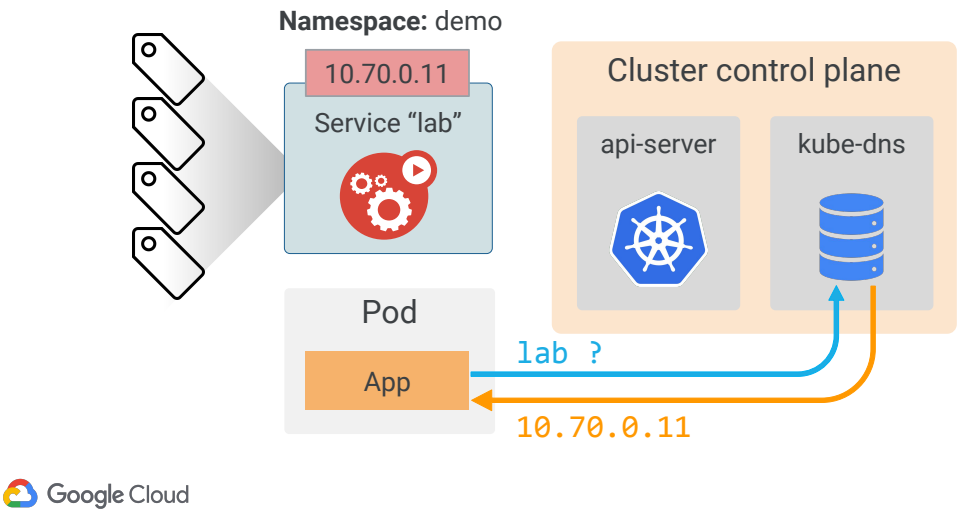
Google Cloud

In Kubernetes, DNS is an optional addon. However, DNS is pre-installed in Google Kubernetes Engine.

The Kubernetes DNS server watches the API server for the creation of new Services.

When a new Service is created, kube-dns automatically creates a set of DNS records for it.

Service discovery through Kubernetes DNS



Kubernetes is configured to use the kube-dns server's IP to resolve DNS names for all Pods. With this, all the Pods in the cluster can 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.

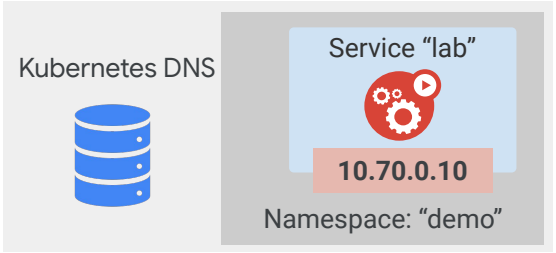
For example, our Pod can now resolve the IP address of the demo Service by querying DNS using its short name, lab, provided the Service is in the same namespace as the Pod. A Pod in any other namespace can resolve the IP address of the Service using the FQDN, lab.demo.svc.cluster.local, or just the part of the FQDN that includes the namespace, lab.demo.

This resolves to the Service’s ClusterIP, which the DNS server returns so that the Pod can make its connection to the Service.

Kubernetes currently includes a Pod-based DNS solution called kube-dns to facilitate Service discovery within the Pods.

Kube-dns maintains the DNS records of Pods and Services. To maintain high performance for Service discovery, GKE autoscales kube-dns based on the number of nodes in the cluster.

Services are assigned one A (Address) record and one SRV (Service) record

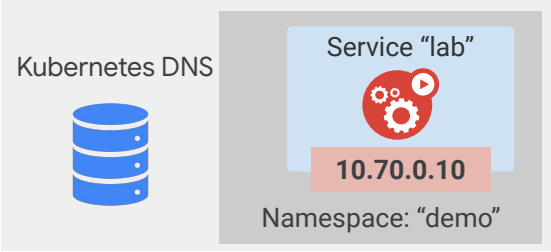



```
record type:    A (Address)
   hostname:    lab
  namespace:    demo
       FQDN:    lab.demo.svc.cluster.local
 IP address:    10.70.0.10
```

Google Cloud

Every Service defined in the cluster is assigned a DNS A record. In this example, there is a Service named “lab” in the “demo” namespace. The name of the namespace and the label “SVC” are added to the DNS name, resulting in a fully qualified domain name of lab.demo.svc.cluster.local.

Services are assigned one A (Address) record and one SRV (Service) record




```
record type:    SRV (Service)
   hostname:    lab
  namespace:    demo
       FQDN:    _http._tcp.lab.demo.svc.cluster.local
      Value:    80
```

Google Cloud

Kubernetes DNS also supports SRV records for named ports. For example, with a port named http with protocol TCP, you can do a DNS SRV Query for _http._tcp.lab.demo.svc.local. Notice the underscores around the port name. The whole fully qualified domain name resolves to the port number and the domain name: lab.demo.

Headless Kubernetes Services, those that are defined without a ClusterIP, also have DNS A and and SRV records defined but those resolve to the set of IP addresses of the pods selected by the Service.

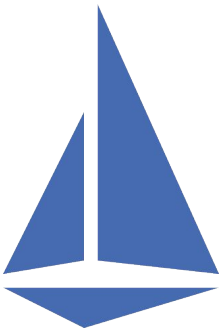
There are several ways to find a Service in GKE

- Environment Variables
- Kubernetes DNS
- **Istio**

Google Cloud

GKE makes using DNS for Service discovery a much simpler task, but there are other solutions, such as the open source service mesh, Istio.

Service discovery through Istio



Istio

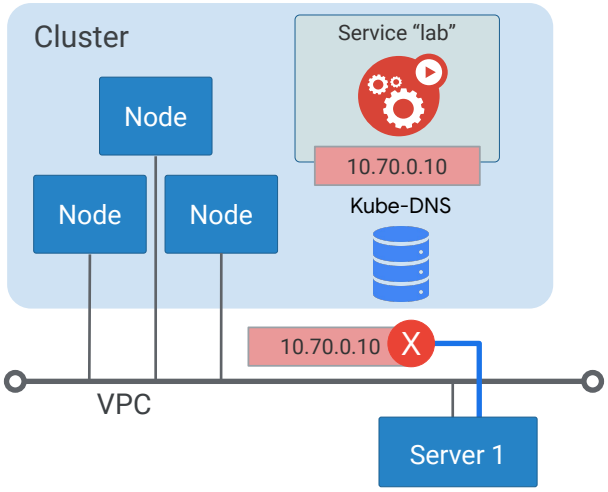
- Open, platform-independent service mesh.
- Adds visibility and control to a microservices architecture.
- Available as a plug-in in GKE[beta].

Google Cloud

A service mesh provides an infrastructure layer that is configurable for microservices applications. Istio is a service mesh to aid in service discovery, control, and visibility in your microservices deployments. Istio is an open-source project started by teams at Google, IBM, and the Envoy team from Lyft. And Istio is available as an add-on for GKE.

It lets you quickly create a cluster with all the components you need to create and run an Istio service mesh in a single step, and once it's installed, your Istio control plane components are automatically kept up to date. When this course was being developed, the Istio plug-in for GKE is in beta.

Service discovery outside the cluster



Google Cloud

Now that you have a Service configured, and it’s in the cluster’s DNS, can your compute instances in the VPC reach this Service?

No. You have one more step. You need to change the Service type.

Agenda

Pod Networking

Services

Service Types and Load Balancers

Ingress Resource

Container-Native Load Balancing

Network Security

Lab: Configuring Google Kubernetes Engine (GKE) Networking

Lab: Creating Services and Ingress Resources

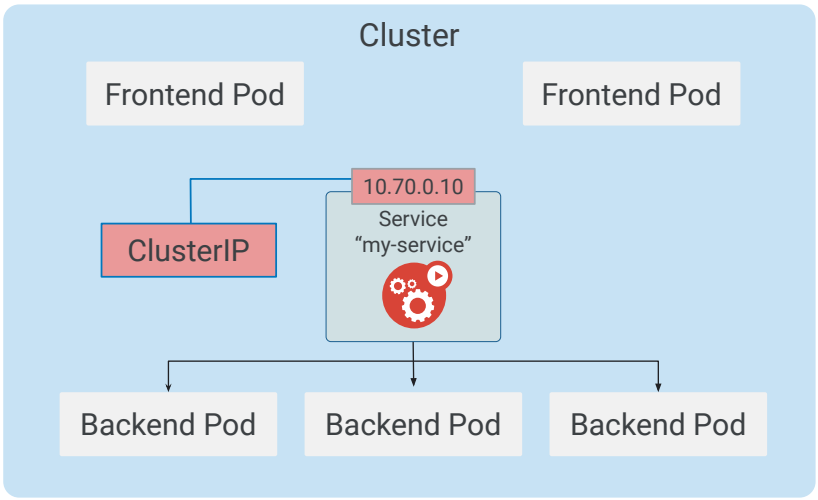
Summary

Google Cloud

In this lesson we'll identify the different types of Kubernetes Services as well as discuss Load Balancers.

There are three principal types of Services: ClusterIP, NodePort, and LoadBalancer. These Services build conceptually on one another, adding functionality with each step. We'll start with the basis of all the Kubernetes Services, the ClusterIP Service.

A ClusterIP Service has a static IP address



Google Cloud

A Kubernetes ClusterIP Service has a static IP address and operates as a traffic distributor within the cluster, but ClusterIP Services aren’t accessible by resources outside the cluster.

Other Pods will use this ClusterIP as their destination IP address when communicating with the Service.

Setting up a ClusterIP Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: ClusterIP
  selector:
    app: Backend
  ports:
    - protocol: TCP
      port: 3306
      targetPort: 6000
```

Google Cloud

Let's consider the step-by-step process of setting up a ClusterIP Service type. Here, you create a Service object by defining its kind. If you don't specify a Service type during the creation of the service, it will default to a Service type of ClusterIP.

Next, you use a label selector to select 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.

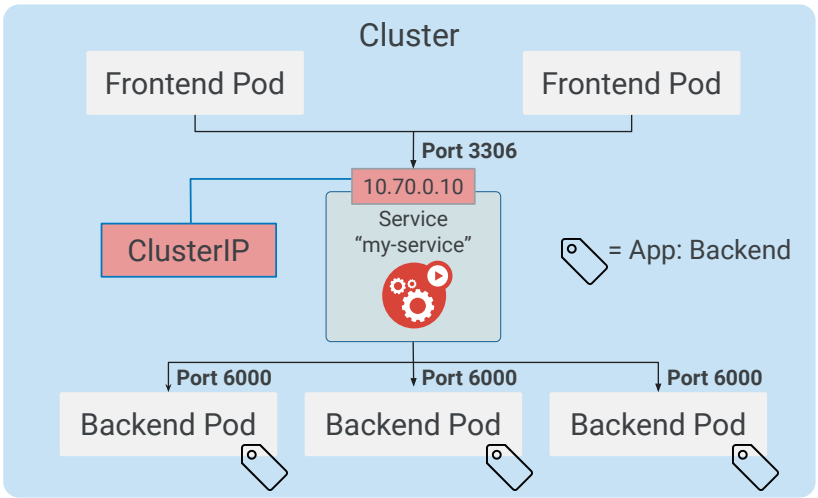
If you create a Service before its corresponding backend workloads, such as Deployments or StatefulSets, the Pods that make up that Service get a nice bonus: they get the hostname and IP address of the Service in an environment variable.

But remember that relying on environment variables for service discovery is not as flexible as using DNS, so this practice isn't mandatory.

Next, you specify the port that the target containers are using, which in this case is TCP port 6000. This Service will receive traffic on port 3306 and then remap it to 6000 as it delivers it to the target Pods.

Creating, loading or applying this manifest will create the ClusterIP Service named "my-service". But how does this Service work?

How does the ClusterIP Service work?



Google Cloud

In this example, you have frontend Pods that must be able to locate the backend Pods.

When you create the Service, the cluster control plane assigns a virtual IP address—also known as ClusterIP—from a reserved pool of Alias IP addresses in the cluster's VPC. 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.

The ip-addresses of these backend pods are mapped to the target port, TCP port 6000 in this example, to create the endpoint resources that the service forwards requests to. The ClusterIP Service will answer requests, on TCP port 3306 in this example, and forward the requests to the backend pods using their ip-addresses and target port as the endpoint resources.

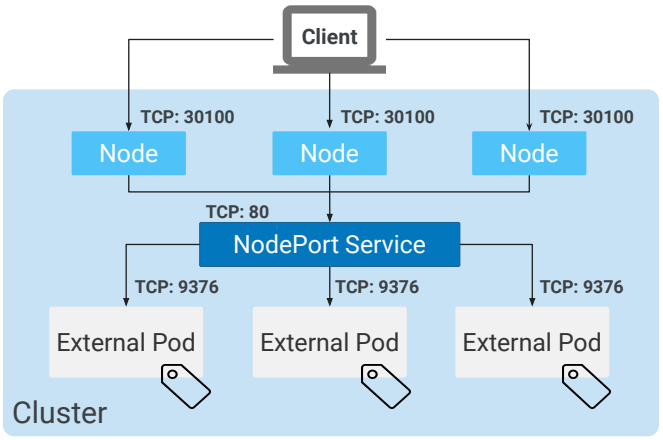
Creating a NodePort Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: external
  ports:
    - protocol: TCP
      nodePort: 30100
      port: 80
      targetPort: 9376
```

Google Cloud

In addition to the setup of the internal ClusterIP Service, a specific port is exposed for external traffic on every node. This port—also known as nodePort —is automatically allocated from the range 30,000 to 32767. In some cases users may want to manually specify it, which is allowed as long as the value also falls within that range, but this is not usually necessary.

The NodePort service enables external communication



default NodePort Range (30000-32767)

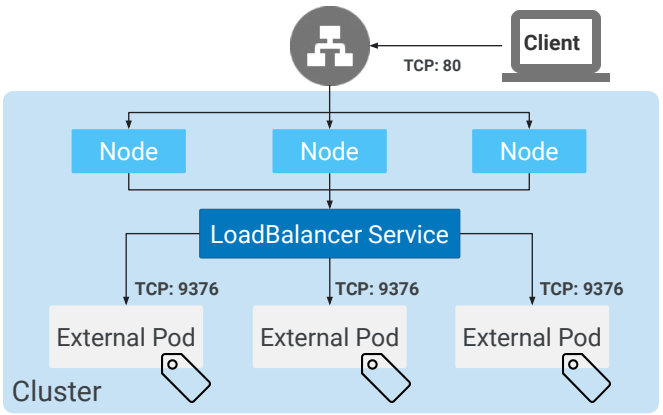
Google Cloud

ClusterIP is useful for internal communication within a cluster, but what about the external communication?

NodePort enables this. NodePort is built on top of ClusterIP Service; therefore, 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.

In this example we now have a Service that can be reached from outside of the cluster using the IP address of any node and the corresponding NodePort number. Traffic through this port is directed to a Service on Port 80 in this example 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.

The LoadBalancer Service can be used to expose a Service to resources outside the cluster



Google Cloud

Conceptually, the LoadBalancer Service type 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 network load balancer.

When you create a LoadBalancer Service, GKE automatically provisions a Google Cloud network load balancer for inbound access to the Services from outside the cluster. Traffic will be directed to the IP address of the network load balancer, and the load balancer forwards the traffic on to the nodes for this Service.

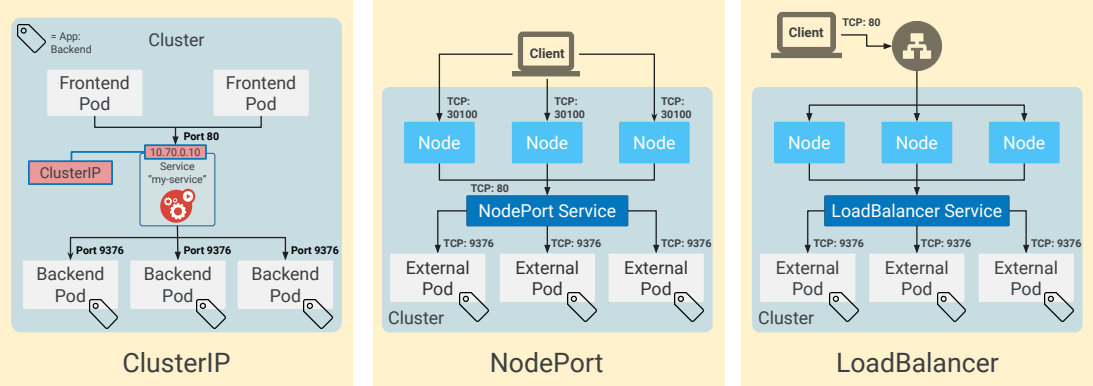
Creating a LoadBalancer Service

```
[...]
spec:
  type: LoadBalancer
  selector:
      app: external
    ports:
      - protocol: TCP
      port: 80
        targetPort: 9376
```

Google Cloud

Here's how you create a LoadBalancer Service. You only need to specify the type: LoadBalancer. Google Cloud will assign a static load balancer IP address that is accessible from outside your project.

Service types summary



Google Cloud

You've seen how ClusterIP Services can be used within the cluster to provide a stable endpoint that allows Pods to connect to other Pods without the risk of the IP address changing.

You learned how NodePort Services extend the ClusterIP Services concept to expose and then bind a port number on each of the nodes in the cluster to the Service. This allows you to access resources within the cluster from external resources such as Google Cloud compute instances.

Finally, you saw how the LoadBalancer Service type implementation in GKE improved on the NodePort Service by using the cloud provider API to create a Google Cloud network load balancer for traffic distribution across the nodes.

Now let's look at a way to group Services together using the Ingress resource.

Agenda

Pod Networking

Services

Service Types and Load Balancers

Ingress Resource

Container-Native Load Balancing

Network Security

Lab: Configuring Google Kubernetes Engine (GKE) Networking

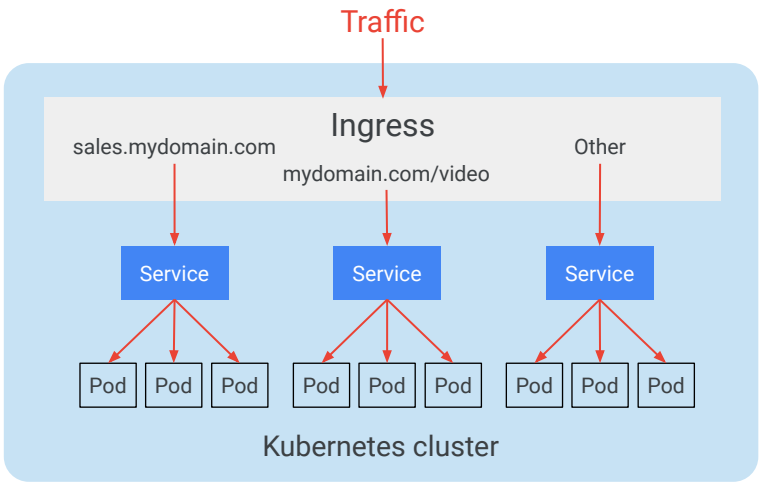
Lab: Creating Services and Ingress Resources

Summary

Google Cloud

Now we'll discuss one of the most powerful tools to direct traffic into your cluster: the Ingress resource.

The Ingress resource, a service for Services



Google Cloud

The Ingress resource operates one layer higher than Services. In fact, it operates a bit like a service for Services.

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. In GKE, an Ingress resource exposes these Services using a single public IP address bound to an HTTP or HTTPS load balancer provisioned within Google Cloud.

In GKE an Ingress combines a Google Cloud load balancer with Kubernetes Services

Google Kubernetes Engine specifics

GKE deploys a Google Cloud HTTP(S) load balancer.

Service considerations

Ingress can direct traffic to:

- NodePort Services
- LoadBalancer Services

LoadBalancer Services still have the double-hop problem.

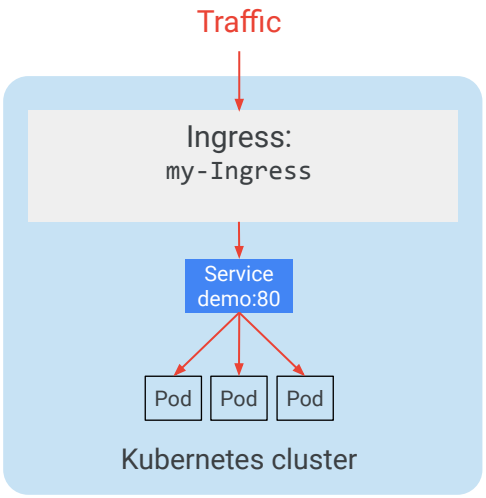
Google Cloud

On GKE, Kubernetes Ingress resources are implemented using Cloud Load Balancing. When you create an Ingress resource in your cluster, GKE creates an HTTP or HTTPS load balancer and configures it to route traffic to your application.

Ingress builds on the prior Service constructs and can deliver traffic to either NodePort Services or LoadBalancer Services. Even with Ingress, LoadBalancer Services can still suffer from the double-hop problem, but as before, this problem can be mitigated by using the “Local” external-traffic policy in the Service manifest.

Creating an Ingress

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-Ingress
spec:
     backend:
         serviceName: demo
         servicePort: 80
```



Google Cloud

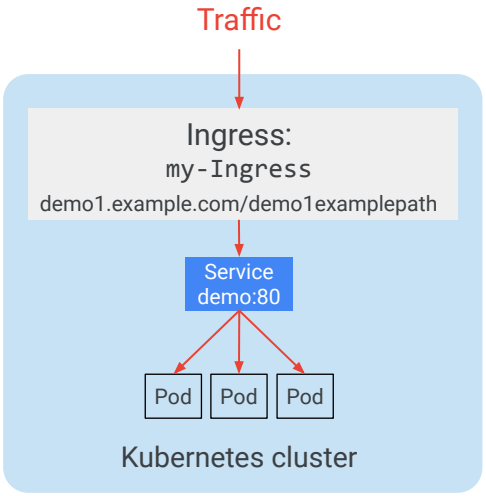
Here's an example of a simple Ingress resource. In this first example, the Ingress controller creates an HTTP or HTTPS load balancer using the Ingress object specification here.

In the object, you've selected a backend Service by specifying the Service name "demo" and the Service port "80." This configuration tells the HTTP or HTTPS load balancer to route all client traffic to the Service named "demo" on port 80.

It also exposes port 443 if the manifest has included the TLS field, or if you populate the annotation "ingress.kubernetes.io.pre-shared-cert" with a valid SSL certificate name.

Ingress manifest example 2

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-Ingress
spec:
  rules:
  - host: demo1.example.com
    http:
      paths:
      - path: /demo1examplepath
        backend:
          serviceName: demo
          servicePort: 80
```



Google Cloud

Here's another Ingress manifest. Inside the specifications, there are rules. Only HTTP rules are currently supported, and each rule is named with the host name. The host name can be further filtered based on the path. A path will have a Service backend defining the Service's name and port. You can set up multiple hosts and paths. This will become clear through some of the examples that follow.

Ingress manifest example 3

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-Ingress
spec:
  rules:
  - host: demo.example.com
    http:
      paths:
      - path: /demopath
        backend:
          serviceName: demo1
          servicePort: 80
```

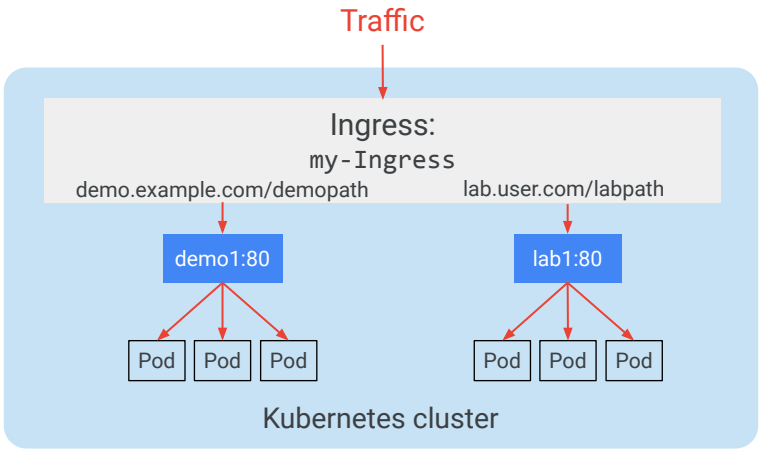
```
  - host: lab.user.com
    http:
      paths:
      - path: /labpath
        backend:
          serviceName: lab1
          servicePort: 80
```

Google Cloud

Ingress supports multiple host names for the same IP address. Here, the code is split for readability.

As you see, there are two host names: demo.example.com and lab.user.com.

Ingress manifest example 3



Google Cloud

The traffic will be redirected from the HTTP or HTTPS 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.

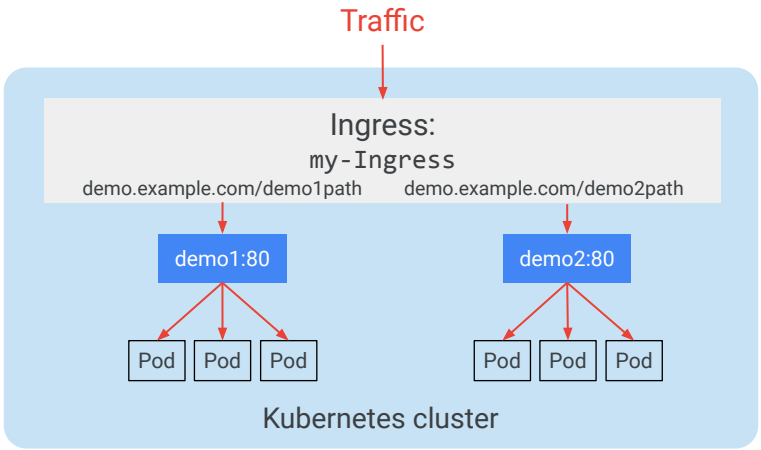
Ingress manifest example 4

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-Ingress
spec:
  rules:
  - host: demo.example.com
    http:
      paths:
      - path: /demo1path
        backend:
          serviceName: demo1
          servicePort: 80
      - path: /demo2path
        backend:
          serviceName: demo2
          servicePort: 80
```

Google Cloud

This example considers rules based on the URL path.

Ingress manifest example 4



Google Cloud

Here, the traffic from demo.example.com/demo1path will be directed to the backend Service named demo1. Similarly, demo.example.com/demo2path will be directed to its backend Service demo2.

Specifying a default backend

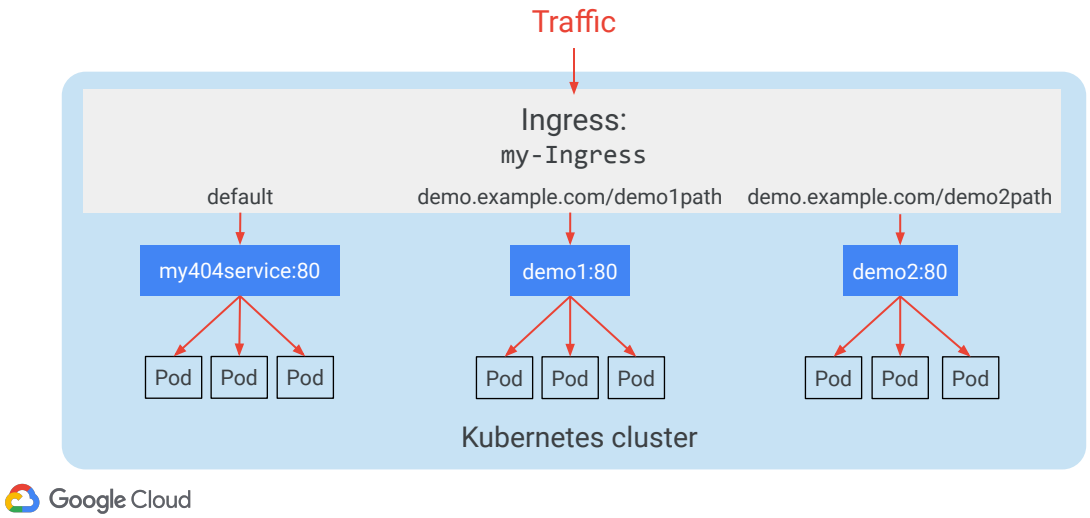
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-Ingress
spec:
  backend:
     serviceName: my404service
     servicePort: 80
```

```
rules:
- host: demo.example.com
  http:
    paths:
    - path: /demo1path
      backend:
        serviceName: demo1
        servicePort: 80
    - path: /demo2path
      backend:
        serviceName: demo2
        servicePort: 80
```

Google Cloud

You can specify a default backend by providing a backend field in your ingress manifest.

Specifying a default backend



Google Cloud

So what happens to the traffic that doesn't match any of these host-based or path-based rules? Well, the traffic with no matching rules is simply sent to the default backend.

If you do not specify a default backend, GKE will supply one that replies with error code 404.

Updating an Ingress

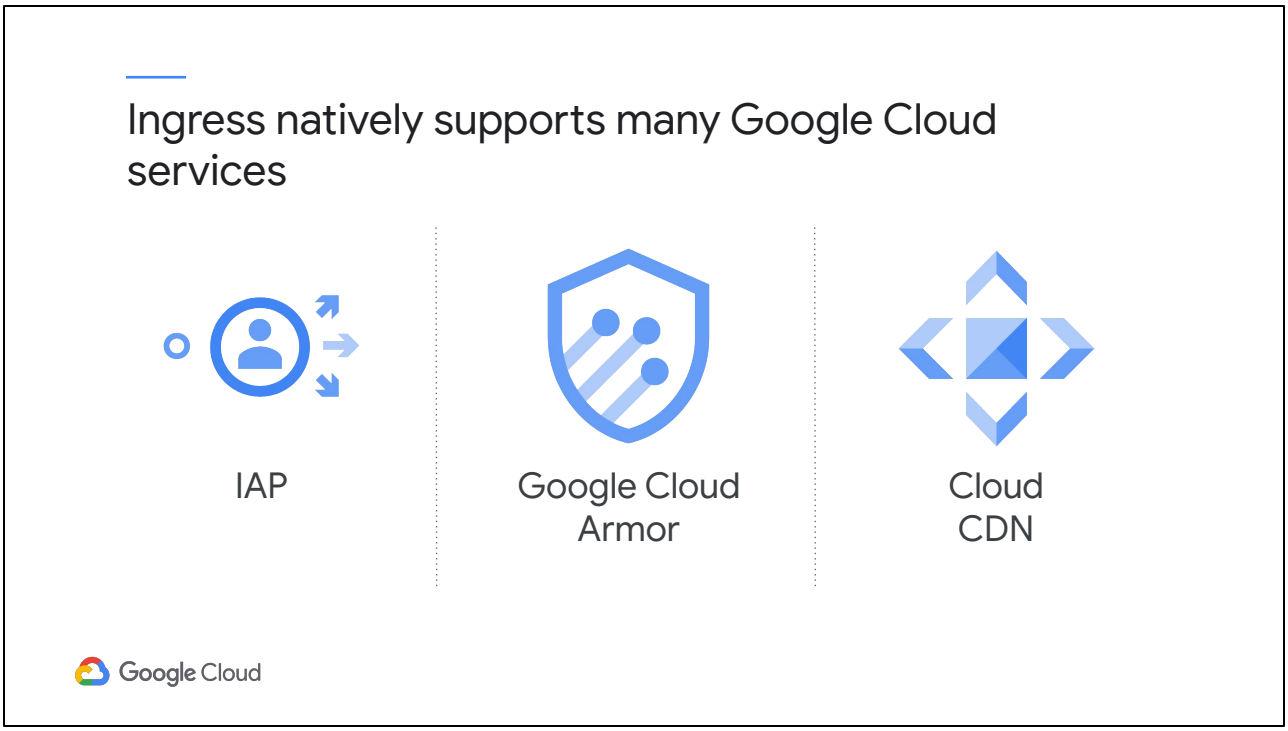
```
$ kubectl edit ingress [NAME]
```

```
$ kubectl replace -f [FILE]
```

Google Cloud

Ingress can be updated by a simple kubectl edit command. When the ingress resource has been update, the API server will tell the Ingress controller to reconfigure the HTTP or HTTPS load balancer according to the changes you made.

You can also update Ingress by using the kubectl replace command, which replaces the Ingress object manifest file entirely.



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

Google Cloud Armor provides built-in protection against distributed denial of service (DDOS) and web attacks for your cluster using a HTTP or HTTPS load balancer. You can set up security rules to allow list or deny list IP addresses or ranges. You can also use predefined rules to defend against cross-site scripting (XSS) and SQL injection (SQLi) application-aware attacks. You can customize security rules to mitigate multivector attacks and restrict access using geolocation.

Cloud CDN allows you to bring your application's content closer to your users. It does so by using more than 100 Edge points of presence.

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

Additional Ingress features

- TLS termination.
- Multiple SSL certificates.
- HTTP/2 and gRPC.
- Multi-cluster and multi-region support.

Google Cloud

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

Ingress provides TLS termination support at the loadbalancer at the edge of the network. From there the load balancer creates another connection to the destination. Although this second connection is unsecured by default, it can be secured. This allows you to manage all your SSL certificates in one place. So you will be happy to know that Ingress can serve multiple SSL certificates.

It also supports the HTTP/2 standard in addition to HTTP/1.0 and HTTP/1.1. Why should you care about that? If you are developing a microservices-based application, you need to ensure that each microservice's communication with all the others uses a high-performance, low-overhead remote procedure call system. gRPC is an increasingly popular way to solve this problem, and it needs HTTP/2. So you can use gRPC along with HTTP/2 to create performant, low-latency, scalable microservices within your cluster.

Lastly, with multi-cluster and multi-region support, you can use a single standard Ingress resource to load balance your traffic globally to multiple clusters across multiple regions. This also supports geo-balancing (that is, location-based load balancing) across multiple regions, which improves the availability of the cluster.

Agenda

Pod Networking

Services

Service Types and Load Balancers

Ingress Resource

Container-Native Load Balancing

Network Security

Lab: Configuring Google Kubernetes Engine (GKE) Networking

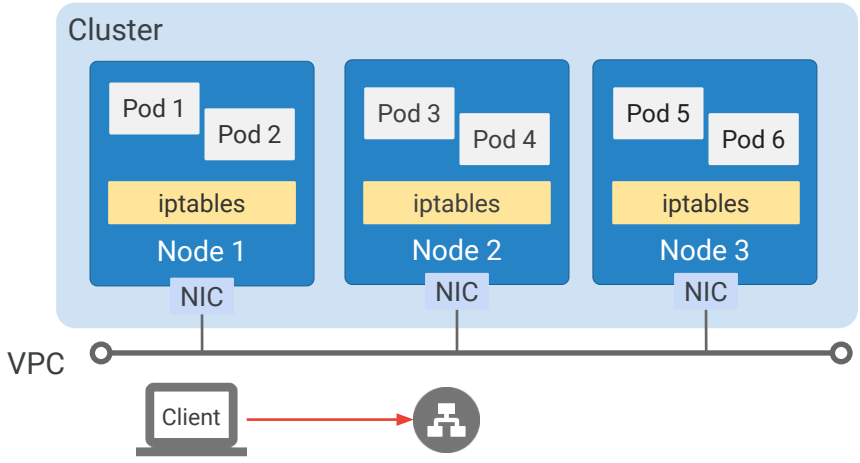
Lab: Creating Services and Ingress Resources

Summary

Google Cloud

Now that you understand the HTTPS load balancer and the Ingress object, let's look in detail at container-native load balancing.

The double-hop dilemma

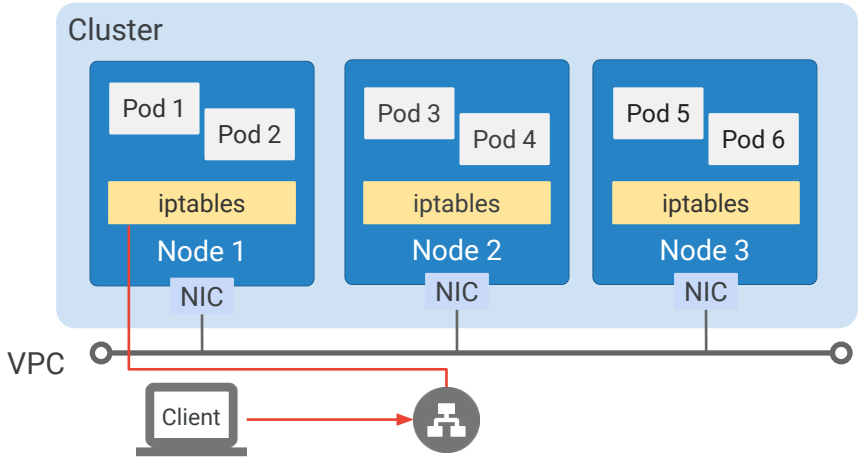


Google Cloud

First, let's take another look at the scenario without a container-native load balancer. A regular HTTPS load balancer distributes 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 routes traffic to any node within an instance group.

As usual, we start with the client, whose traffic is directed through the network load balancer.

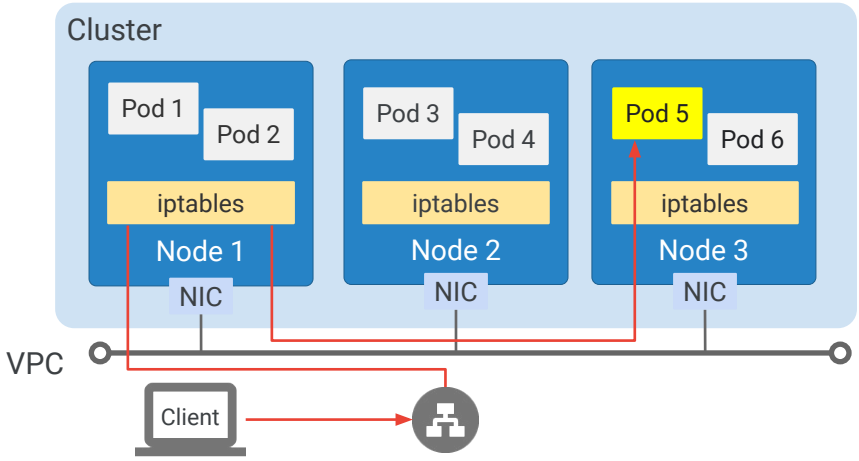
The double-hop dilemma



Google Cloud

The network load balancer chooses a random node in the cluster and forwards the traffic to it. Here we have chosen Node 1.

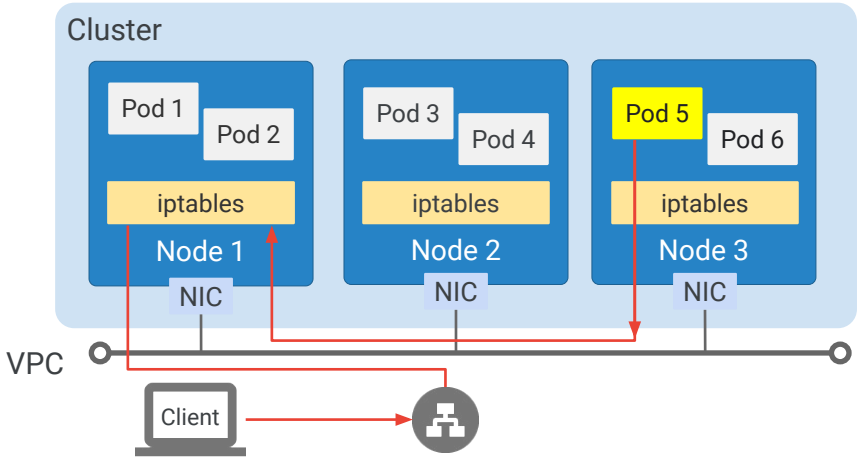
The double-hop dilemma



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. In our example, Node 1 chooses Pod 5, which isn't on this node; therefore Node 1 forwards the traffic to Pod 5 on Node 3.

The double-hop dilemma



Google Cloud

Pod 5 directs its responses back via Node 1, the double-hop. Node 1 then forwards the traffic back to the network load balancer, which sends it back to the client.

Therefore, this method has two levels of load balancing (one by the load balancer, and the other by kube-proxy), which results in multiple network hops. The response traffic also follows the same path. Overall, this method is not optimal for load balancing. This process does keep the Pod use even, but at the expense of increased latency and extra network traffic.

Add the Local value for the external-traffic policy

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

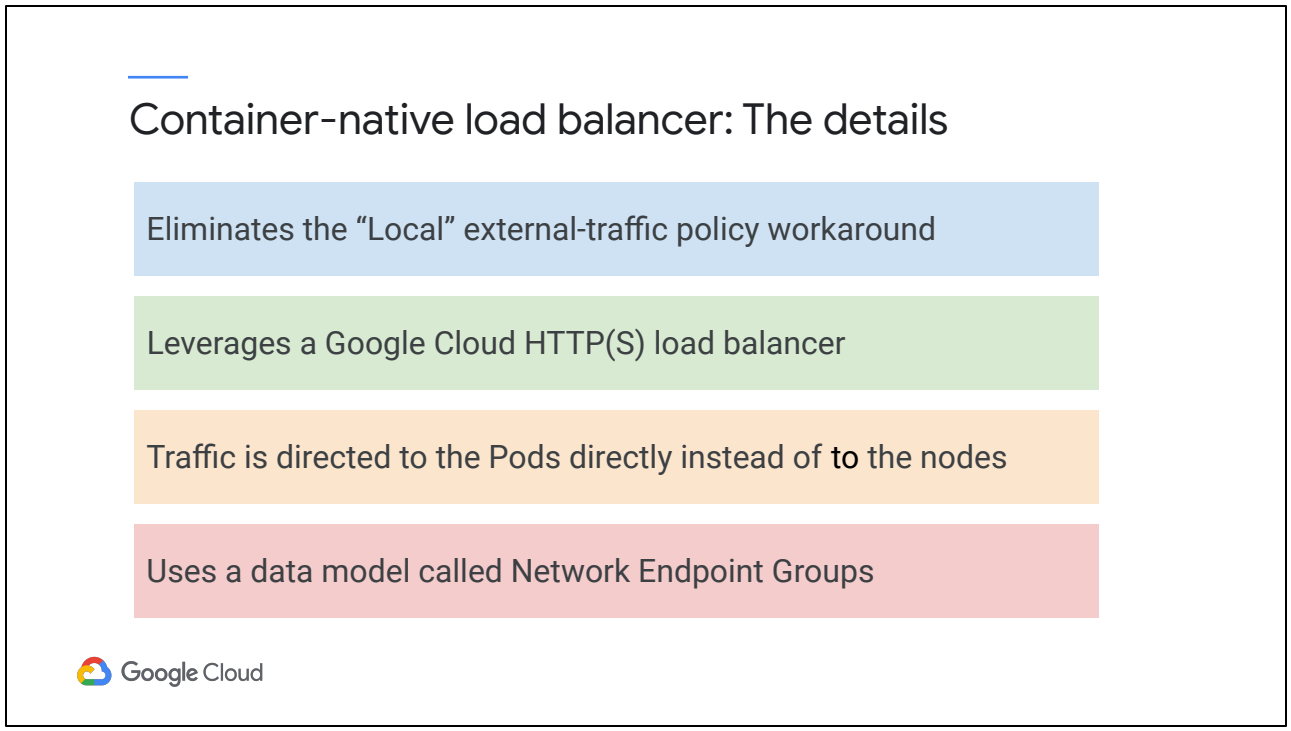
Google Cloud

So if you are using traditional Kubernetes networking, you should choose which is most important to you: the lowest possible latency or the most even cluster load balancing.

Suppose that the lowest possible latency is most important. You can configure the LoadBalancer Service to force kube-proxy to choose a Pod local to the Node that received the client traffic. To do that, set the externalTrafficPolicy field to “Local” in the Service manifest.

This choice eliminates the double-hop to another node. Why? Because kube-proxy will always choose a Pod on the receiving node. In addition, 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 imbalance in the load of the cluster. What’s the best choice? It depends on your application. You can profile both configurations and choose the one that gives the best overall application performance.

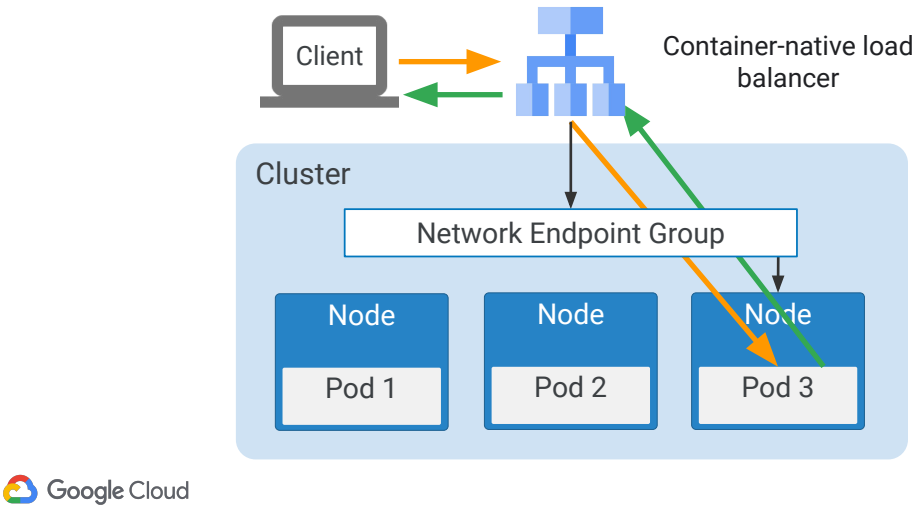


The “Local” external-traffic policy may cause other issues as it imposes constraints on the mechanisms that balance Pod traffic internally, and external the HTTP or HTTPS load balancer continues to forward traffic via Nodes with no awareness of the state of the Pods themselves.

A true container-first approach to load balancing is now available in GKE. The solution still leverages the powerful Google Cloud HTTPS load balancer however, the load balancer now directs the traffic to the Pods directly instead of to the nodes.

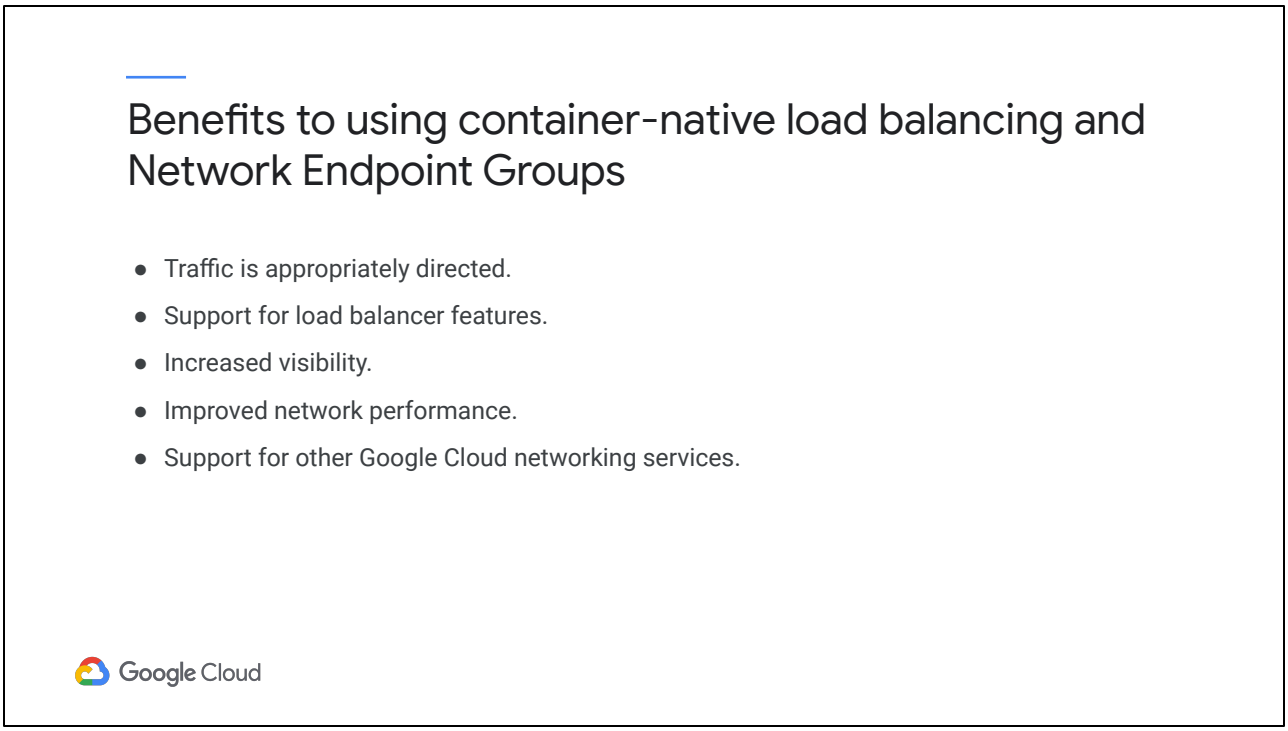
This method requires your GKE cluster to operate in VPC-native mode, and it relies on a data model called *network endpoint groups*, or NEGs. NEGs are a set of network endpoints representing IP-to-port pairs, which means that Pods can simply be just another endpoint within that group, equal in standing to compute instance VMs.

Container-native load balancer in action



Google Cloud

Every connection is made directly between the load balancer and the intended Pod. 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.



There are many benefits to using container-native load balancing and Network Endpoint Groups.

Pods can be specified directly as endpoints for Google Cloud load balancers: the traffic is appropriately directed to the intended Pod, eliminating extra network hops.

Load balancer features, such as traffic shaping and advanced algorithms, are supported. With the direct connection to the Pod, the load balancer can accurately distribute the traffic.

Further, container-native load balancing allows direct visibility to the Pods and more accurate health checks. Source IP is preserved, which provides visibility into round-trip time from the client to the load balancer and helps with troubleshooting. This visibility can be easily extended using Google Cloud's operations suite.

Next, there are fewer network hops in the path, which optimizes the data path. This improves latency and throughput, providing better network performance overall.

Cloud Armor is a Google Cloud service that helps protect your applications against Distributed Denial of Service and other kinds of attacks.

Cloud CDN lets you cache content at Google's network edge locations, so that your users experience low latency connections. Identity-Aware Proxy lets you protect your applications by verifying their users' identity and the context of their network requests. If you choose container-native load balancing, you can use these services and more with your GKE-based application.

Agenda

Pod Networking

Services

Service Types and Load Balancers

Ingress Resource

Container-Native Load Balancing

Network Security

Lab: Configuring Google Kubernetes Engine (GKE) Networking

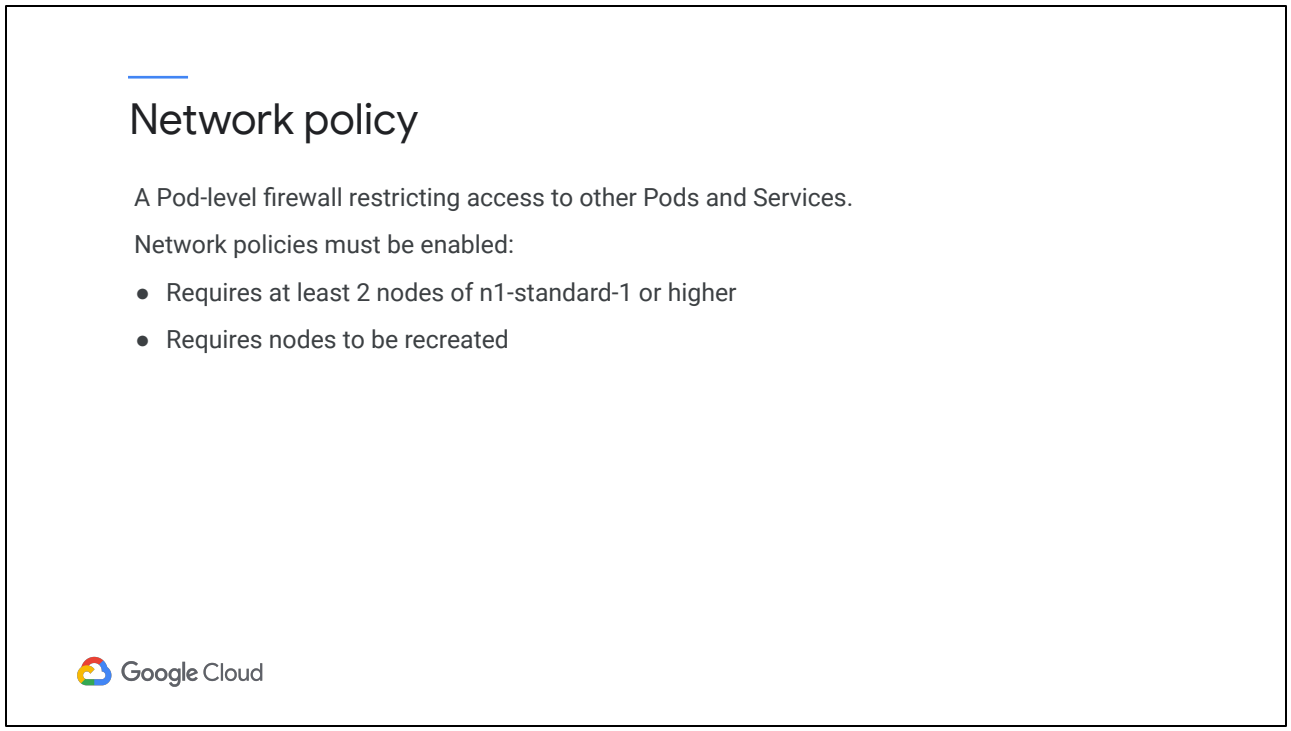
Lab: Creating Services and Ingress Resources

Summary

Google Cloud

Now that you've learned how to access your Services from outside the cluster, let's discuss implementing security.

So far, you've learned that all Pods can communicate with one another by default. But what if you don't want that? What if you'd like to restrict access to certain Pods? The solution is to implement a network policy.



A network policy is a set of firewall rules at the Pod level that restrict access to other Pods and Services inside the cluster. For example, in a multi-layered application, you can restrict access at each stack level using these network policies. 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.

By default, network policies are disabled in GKE. In order to enable them, you need at least 2 nodes of n1-standard-1 instance type or higher. The recommended minimum is 3. Network policies are not supported on f1-micro and g1-small instances.

The policy can be enabled on the creation of the cluster and will always apply. If a policy is applied after the cluster is created, Nodes must be recreated for this process to complete. GKE will do this automatically for you during an active maintenance window, or you can still manually upgrade the cluster.

The cluster will remain vulnerable until the process is completed.

Enabling a network policy

Enable a network policy for a new cluster

```
gcloud container clusters create [NAME] \
    --enable-network-policy
```

Enable a network policy for an existing cluster

```
gcloud container clusters update [NAME] \
    --update-addons-NetworkPolicy=ENABLED
gcloud container clusters update [NAME] \
    --enable-network-policy
```

Google Cloud

You can also enable a network policy using kubectl commands.

A policy can be enabled during the cluster creation, or through a 2-step process for an existing cluster. In the examples, NAME represents a cluster name.

You can also update the cluster from the advanced options in the Cloud Console. By default, enabling network policy enforcement for your cluster also enables enforcement for the cluster's nodes.

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

Writing a network policy (1/2)

```
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
```

Google Cloud

Here, you'll learn how to write a network policy.

First, podSelector enables you to select a set of Pods based on labels. The network policy is applied to these selected Pods. 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. Don't be confused by the terminology. The word "ingress" just means incoming traffic to affected Pods, just as "egress" means traffic going out of those Pods. "Ingress" here has nothing to do with the Ingress objects you learned about in an earlier lesson for load balancing.

Let's dig into the details of the ingress, or incoming traffic, side of the policy. There are two main sections: "from" and "ports." In the example, a single rule is specified, which starts at the keyword "from" and includes the "ports" section. Within that single rule, the traffic can come from any of three sources.

1. The first source is from a specific ipBlock. You can also provide an exception to this rule.
2. The second source is namespaceSelector, which means that only traffic

1. coming from this particular namespace will be allowed.
2. The last source is podSelector, where traffic from Pods that match the label selector will be allowed access. In the example, Pods with the label “frontend” will be allowed access.

Traffic from any one of those sources will be permitted, but only if it’s inbound on TCP port 6379.

Writing a network policy (2/2)

```
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
```

Google Cloud

Now, let's consider the egress rules.

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.

Disabling a network policy

Disable a network policy for a cluster

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

Google Cloud

You can use the on-screen kubectl command to disable a network policy. In the 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.

Network policy defaults

```
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

```
metadata:
  name: allow-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress:
  - {}
```

```
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

```
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Egress
```

```
metadata:
  name: allow-all
spec:
  podSelector: {}
  policyTypes:
  - Egress
  egress:
  - {}
```

Google Cloud

By default, 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. You can, however, set up default network policies. Default policies are “catch-all” conditions that the policy will invoke if no other rules match.

Default-deny policy for ingress, or incoming traffic. If you don’t specify a value for policyTypes: Ingress, then all incoming traffic will be denied. Similarly, you can create a default-deny policy for all outgoing traffic.

Going back to the ingress traffic, you can create a default policy that permits all traffic. The name of this keyword is “allow-all.” Note the difference in wording between this and the “default-deny” keyword.

Under the ingress rule, there is an empty rule which matches all traffic, and therefore this rule allows all traffic. Similarly, there’s an “allow-all” policy for egress traffic. This time, Egress policyType is required.

The last one is a “default-deny” policy that applies to all ingress and all egress traffic.

So, in the real world, why would you use network policies? Well, imagine what would happen if an attacker compromised one of your Pods by exploiting a security vulnerability in it. By default, that attacker would be able to probe outwards from the compromised Pod to all other Pods in your cluster. The attacker might find other vulnerabilities. It would be wise to limit the attack surface of your cluster.

You would know what legitimate traffic in your cluster looks like: that is, you know which Pods should be communicating with which other Pods as part of your cluster's normal operation. Network policies let you lock down network traffic to only those pathways.

A drawback of network policies is that they can be a lot of work to manage. That's one of the design motivations behind Istio, which I mentioned earlier in this module. A service mesh, such as what Istio delivers, is an easier way: not only to ensure that only authorized traffic can occur on your network, but also to ensure that traffic is mutually authenticated.

Agenda

Pod Networking

Services

Service Types and Load Balancers

Ingress Resource

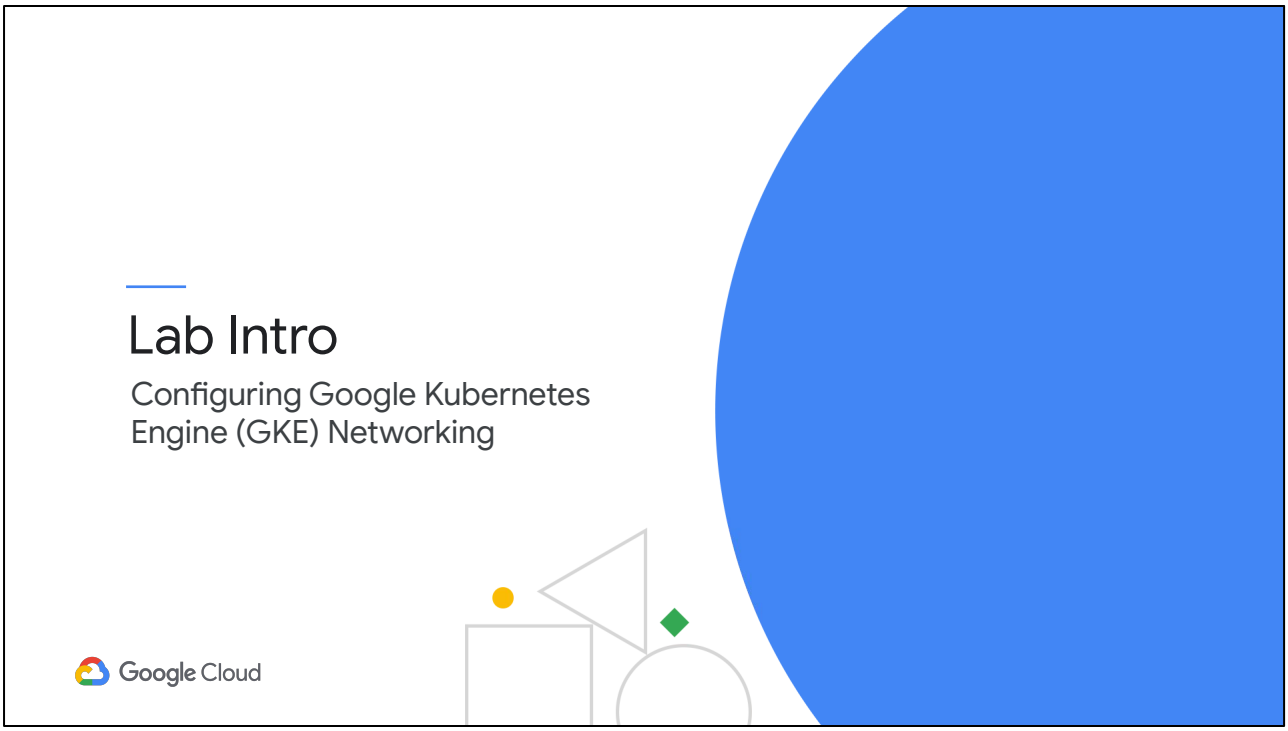
Container-Native Load Balancing

Network Security

Lab: Configuring Google Kubernetes Engine (GKE) Networking

Lab: Creating Services and Ingress Resources

Summary



In this lab, you'll create a private cluster, add an authorized network for API access to it, and then configure a network policy for Pod security. Your first task will be to create a private cluster, consider the options for how private to make it, and then compare your private cluster to your original cluster. You'll then learn how to add an authorized network for cluster control plane access. Your final task will be to create a cluster network policy to restrict communication between Pods.

Agenda

Pod Networking

Services

Service Types and Load Balancers

Ingress Resource

Container-Native Load Balancing

Network Security

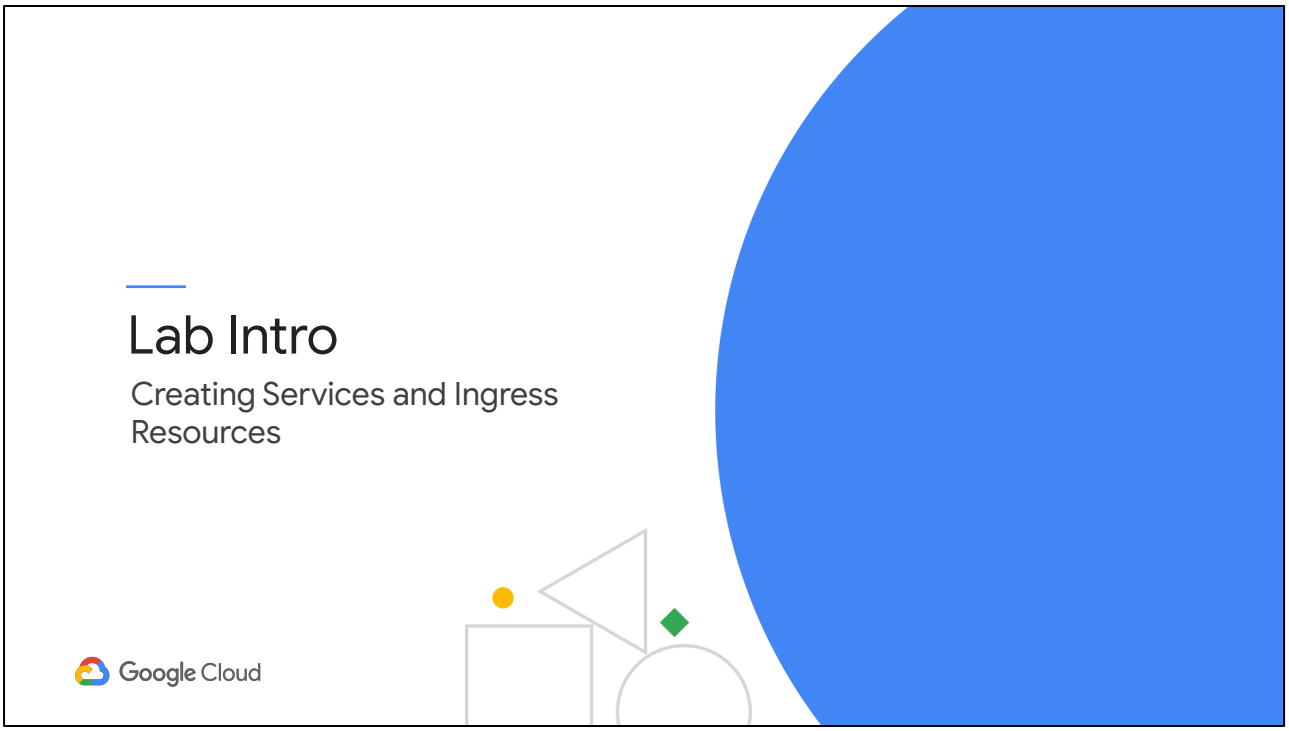
Lab: Configuring Google Kubernetes Engine (GKE) Networking

Lab: Creating Services and Ingress Resources

Summary

Google Cloud

The Kubernetes networking model relies heavily on IP addresses. Services, Pods, containers, and nodes communicate using IP addresses and ports. Kubernetes provides different types of load balancing to direct traffic to the correct Pods. So, let's start by reviewing basic Pod networking.



In this lab, you'll create two deployments of Pods and work with three different types of services, including the Ingress resource.

Your first task will be to create a GKE cluster and a deployment manifest for a set of Pods within the cluster that you'll then use to test DNS resolution of the Pod and service names. Your next task will be to deploy a sample workload and a ClusterIP service. Thereafter, you'll convert your existing ClusterIP service to a NodePort service, and retest access to the service from inside and outside the cluster. In the fourth task you will deploy a new set of Pods running a different version of the application so that you can easily differentiate the two services. You'll then expose the new Pods as a LoadBalancer service and access the service from outside the cluster. Your final task will be to deploy an Ingress resource that will direct traffic to both services based on the URL entered by the user.

Agenda

Pod Networking

Services

Service Types and Load Balancers

Ingress Resource

Container-Native Load Balancing

Network Security

Lab: Configuring Google Kubernetes Engine (GKE) Networking

Lab: Creating Services and Ingress Resources

Summary

Summary

Create and use Deployments.

Create and run Jobs and CronJobs.

Use Helm Charts.

Scale clusters manually and automatically.

Configure Node and Pod affinity.

In this module, you learned how to create Services to expose applications running within Pods, allowing them to communicate with each other, and the outside world. Use load balancers to expose Services to external clients and spread incoming traffic across your cluster as evenly as possible.

You learned how to create Ingress resources for HTTP or HTTPS load balancing, for specialized external load balancing off traffic and leverage container-native load balancing to allow you to directly configure Pods as network endpoints with Cloud Load Balancing.


Google Cloud