Network overview

AUTOPILOT (/KUBERNETES-ENGINE/DOCS/CONCEPTS/AUTOPILOT-OVERVIEW)

STANDARD (/KUBERNETES-ENGINE/DOCS/CONCEPTS/TYPES-OF-CLUSTERS)

This page provides a guide to the main aspects of Google Kubernetes Engine (GKE) networking. This information is useful to those who are just getting started with Kubernetes, as well as experienced cluster operators or application developers who need more knowledge about Kubernetes networking in order to better design applications or configure Kubernetes workloads.

Kubernetes lets you declaratively define how your applications are deployed, how applications communicate with each other and with the Kubernetes control plane, and how clients can reach your applications. This page also provides information about how GKE configures Google Cloud services, where it is relevant to networking.

When you use Kubernetes to orchestrate your applications, it's important to change how you think about the network design of your applications and their hosts. With Kubernetes, you think about how Pods, Services, and external clients communicate, rather than thinking about how your hosts or virtual machines (VMs) are connected.

Kubernetes' advanced software-defined networking (SDN) enables packet routing and forwarding for Pods, Services, and nodes across different zones in the same regional cluster. Kubernetes and Google Cloud also dynamically configure IP filtering rules, routing tables, and firewall rules on each node, depending on the declarative model of your Kubernetes deployments and your cluster configuration on Google Cloud.

Warning: Don't manually make changes to nodes because they are overridden by GKE, and your cluster may not function correctly. The only reason to access a node directly is to debug problems with your configuration.

Prerequisites

This page uses terminology related to the Transport (https://en.wikipedia.org/wiki/Transport_layer), Internet (https://en.wikipedia.org/wiki/Internet_layer), and Application (https://en.wikipedia.org/wiki/Application_layer) layers of the Internet protocol (https://en.wikipedia.org/wiki/Internet_protocol_suite) suite, including HTTP

(https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol), and DNS (https://en.wikipedia.org/wiki/Domain_Name_System), but you don't need to be an expert on these topics.

In addition, you may find this content easier to understand if you have a basic understanding of Linux network management concepts and utilities such as `iptables` rules and routing.

Terminology related to IP addresses in Kubernetes

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. All of these mechanisms are described in more detail later. Keep the following terms in mind as you read:

- **ClusterIP:** The IP address assigned to a Service. In other documents, it might be called the "Cluster IP". This address is stable for the lifetime of the Service, as discussed in Services (#services).
- **Pod IP address:** The IP address assigned to a given Pod. This is ephemeral, as discussed in Pods (#pods).
- **Node IP address:** The IP address assigned to a given node.

Cluster networking requirements

Both public and private clusters require connectivity to `*.googleapis.com`, `*.gcr.io`, and the control plane IP address. This requirement is satisfied by the implied allow egress rules (/vpc/docs/firewalls#default_firewall_rules) and the automatically created firewall rules (/kubernetes-engine/docs/concepts/firewall-rules) that GKE creates.

For public clusters, if you add firewall rules that deny egress with a higher priority, you must create firewall rules to allow `*.googleapis.com`, `*.gcr.io`, and the control plane IP address.

For more information private cluster requirements, see Requirements, restrictions, and limitations (/kubernetes-engine/docs/how-to/private-clusters#req_res_lim)

Networking inside the cluster

This section discusses networking within a Kubernetes cluster, as it relates to IP allocation (#ip-allocation), Pods (#pods), Services (#services), DNS (#dns), and the control plane (#control-plane).

IP allocation

Kubernetes uses various IP ranges to assign IP addresses to nodes, Pods, and Services.

- Each node has an IP address assigned from the cluster's Virtual Private Cloud (VPC) network. This node IP provides connectivity from control components like `kube-proxy` and the `kubelet` to the Kubernetes API server. This IP address address is the node's connection to the rest of the cluster.
- Each node has a pool of IP addresses that GKE assigns Pods running on that node (a /24 CIDR block by default (https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing)). You can optionally specify the range of IPs when you create the cluster. The Flexible Pod CIDR range feature (/kubernetes-engine/docs/how-to/flexible-pod-cidr) lets you reduce the size of the range for Pod IP addresses for nodes in a node pool.

Note: For Standard clusters, you can run a maximum of 256 Pods on a node with a /23 range, not 512 as you might expect. This provides a buffer so that Pods don't become unschedulable due to a transient lack of IP addresses in the Pod IP address range for a given node. For all ranges, at most half as many Pods can be scheduled as IP addresses in the range. Autopilot clusters can run a maximum of 32 Pods per node.

- Each Pod has a single IP address assigned from the Pod CIDR range of its node. This IP address is shared by all containers running within the Pod, and connects them to other Pods running in the cluster.
- Each Service has an IP address, called the ClusterIP, assigned from the cluster's VPC network. You can optionally customize the VPC network when you create the cluster.
- Each control plane has a public or internal IP address based on the type of cluster, version, and creation date. For more information, see control plane (#control-plane) description.

GKE networking model doesn't allow IP addresses to be reused across the network. When you migrate to GKE, you must plan your IP address allocation to Reduce internal IP address usage in GKE

(/kubernetes-engine/docs/concepts/gke-ip-address-mgmt-strategies#reduce-private-ip-address-usage-in-gke)
.

MTU

The MTU selected for a Pod interface is dependent on the Container Network Interface (CNI) used by the cluster Nodes and the underlying VPC MTU setting. For more information, see Pods (/kubernetes-engine/docs/concepts/network-overview#pods).

Autopilot is configured to always inherit the VPC MTU.

The Pod interface MTU value is either `1460` or inherited from the primary interface of the Node.

CNI	MTU	GKE Standard
kubenet	1460	Default
kubenet (GKE version 1.26.1 and later)	Inherited	Default
Calico	1460	Enabled by using `--enable-network-policy`. For details, see Control communication between Pods and Services using network policies (/kubernetes-engine/docs/how-to/network-policy).
netd	Inherited	Enabled by using any of the following: • Intranode visibility (/kubernetes-engine/docs/how-to/intranode-visibility) • Workload Identity Federation for GKE (/kubernetes-engine/docs/how-to/workload-identity) • IPv4/IPv6 dual-stack networking (/kubernetes-engine/docs/concepts/alias-ips#dual_stack_network)
GKE Dataplane V2	Inherited	Enabled by using `--enable-dataplane-v2`. For details, see Using GKE Dataplane V2 (/kubernetes-engine/docs/how-to/dataplane-v2).

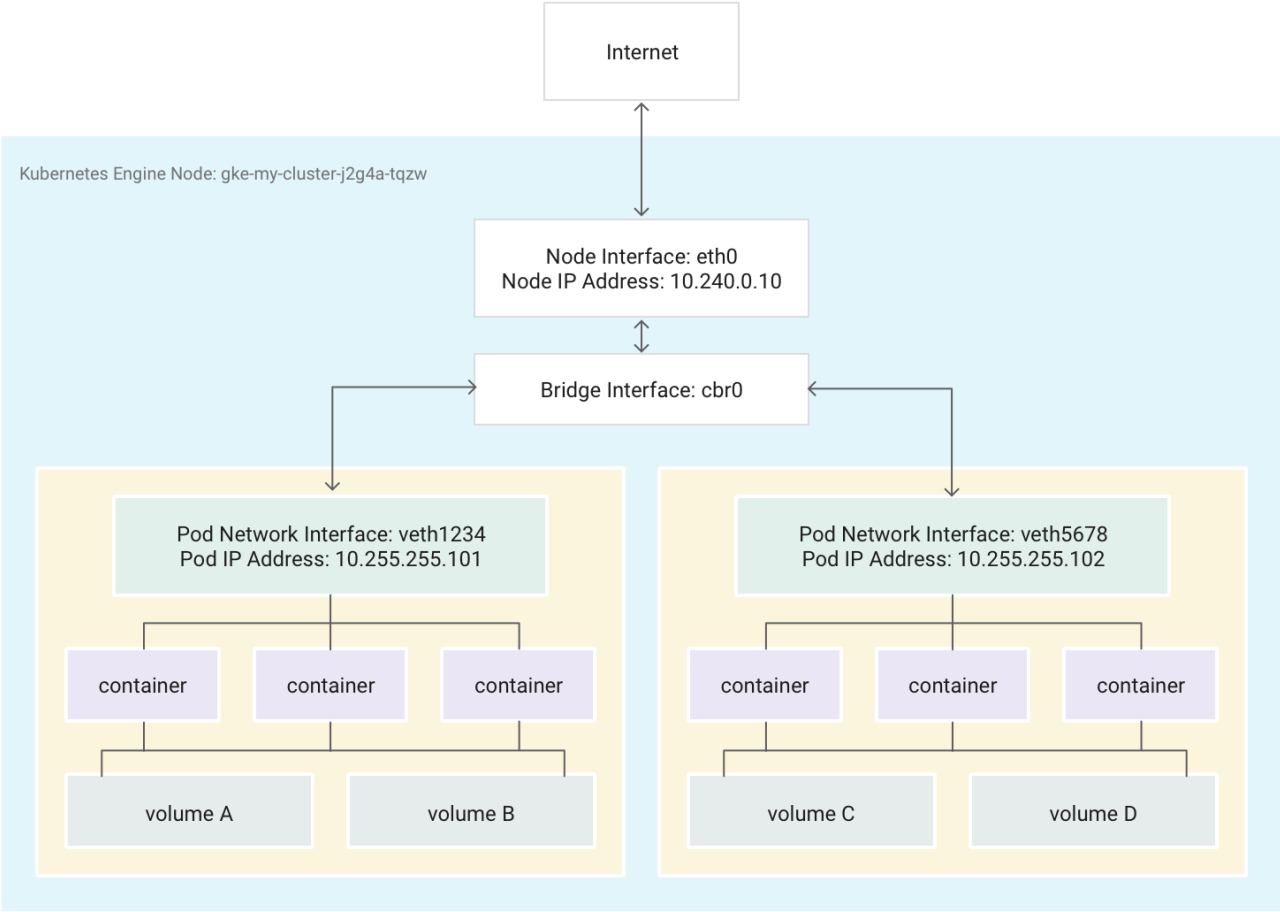
For more information, see VPC-native clusters
(/kubernetes-engine/docs/concepts/alias-ips#cluster_sizing).

Pods

In Kubernetes, a Pod (https://kubernetes.io/docs/concepts/workloads/pods) is the most basic deployable unit within a Kubernetes cluster. A Pod runs one or more containers. Zero or more Pods run on a node. Each node in the cluster is part of a node pool (/kubernetes-engine/docs/concepts/node-pools).

In GKE, these nodes are virtual machines, each running as an instance in Compute Engine.

Pods can also attach to external storage volumes and other custom resources. This diagram shows a single node running two Pods, each attached to two volumes.



When Kubernetes schedules a Pod to run on a node, it creates a network namespace (http://man7.org/linux/man-pages/man8/ip-netns.8.html) for the Pod in the node's Linux kernel. This network namespace connects the node's physical network interface, such as `eth0`, with the Pod using a virtual network interface, so that packets can flow to and from the Pod. The associated virtual network interface in the node's root network namespace connects to a Linux bridge that allows communication among Pods on the same node. A Pod can also send packets outside of the node using the same virtual interface.

Kubernetes assigns an IP address (the Pod IP address) to the virtual network interface in the Pod's network namespace from a range of addresses reserved for Pods on the node. This address range is a subset of the IP address range assigned to the cluster for Pods, which you can configure when you create a cluster.

A container running in a Pod uses the Pod's network namespace. From the container's point of view, the Pod appears to be a physical machine with one network interface. All containers in the Pod see this same network interface. Each container's `localhost` is connected, through the Pod, to the node's physical network interface, such as `eth0`.

Note that this connectivity differs drastically depending on whether you use GKE's Container Network Interface (CNI) or choose to use Calico's implementation by enabling Network policy (/kubernetes-engine/docs/how-to/network-policy) when you create the cluster.

- If you use GKE's CNI, one end of the Virtual Ethernet Device (veth) pair is attached to the Pod in its namespace, and the other is connected to the Linux bridge device `cbr0`.[1] (#fn1) In this case, the following command shows the various Pods' MAC addresses attached to `cbr0`:

  ```
  arp -n
  ```

  Running the following command in the toolbox container shows the root namespace end of each veth pair attached to `cbr0`:

  ```
  brctl show cbr0
  ```

- If Network Policy is enabled, one end of the veth pair is attached to the Pod and the other to `eth0`. In this case, the following command shows the various Pods' MAC addresses attached to different veth devices:

  ```
  arp -n
  ```

  Running the following command in the toolbox container shows that there is not a Linux bridge device named `cbr0`:

  ```
  brctl show
  ```

The iptables rules that facilitate forwarding within the cluster differ from one scenario to the other. It is important to have this distinction in mind during detailed troubleshooting of connectivity issues.

By default, each Pod has unfiltered access to all the other Pods running on all nodes of the cluster, but you can limit access among Pods (#limit-connectivity). Kubernetes regularly tears down and recreates Pods. This happens when a node pool is upgraded, when changing the Pod's declarative configuration or changing a container's image, or when a node becomes unavailable. Therefore, a Pod's IP address is an implementation detail, and you shouldn't rely on them. Kubernetes provides stable IP addresses using Services (#services).

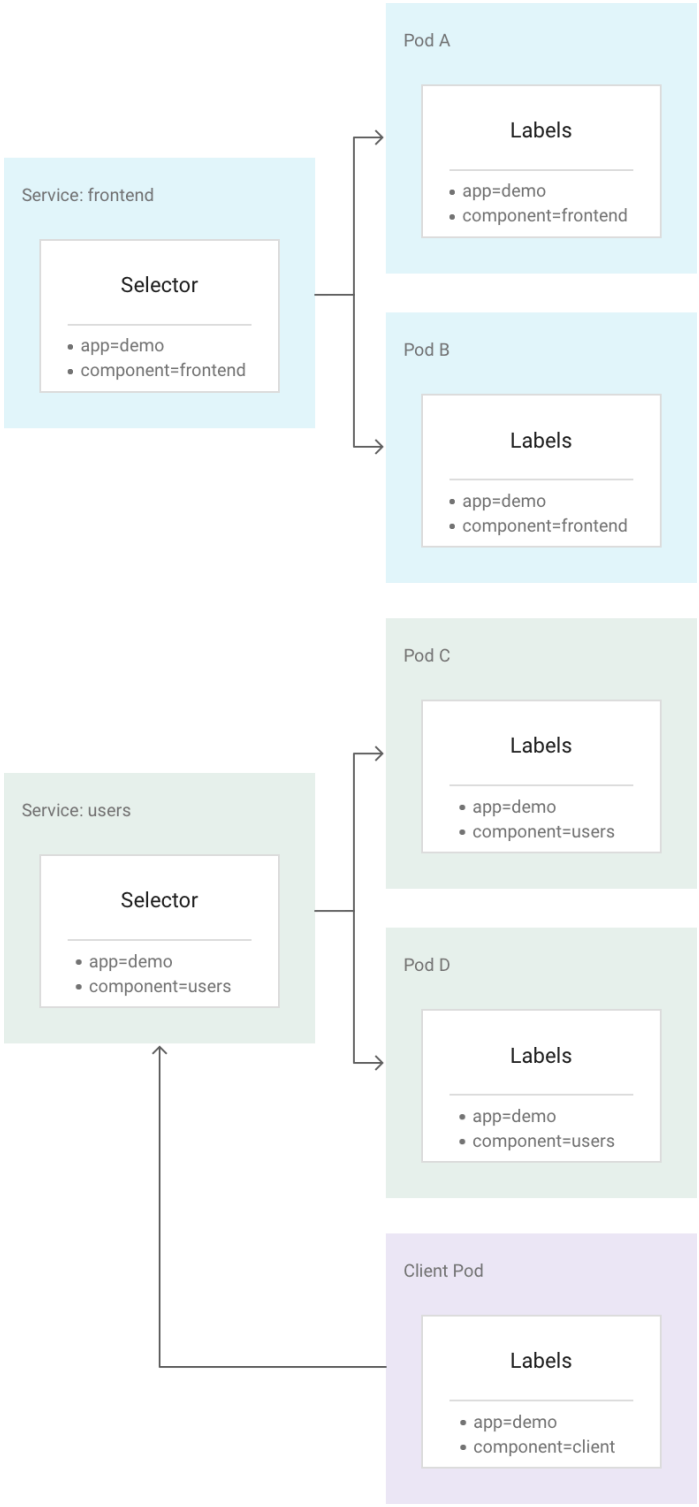
1. The virtual network bridge `cbr0` is only created if there are Pods which set `hostNetwork: false`.↩ (#fnref1)

Services

In Kubernetes, you can assign arbitrary key-value pairs called labels (/kubernetes-engine/docs/how-to/creating-managing-labels) to any Kubernetes resource. Kubernetes uses labels to group multiple related Pods into a logical unit called a Service. A Service has a stable IP address and ports, and provides load balancing among the set of Pods whose labels match all the labels you define in the label selector (https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors) when you create the Service.

Note: Services in Kubernetes are completely unrelated to services in Docker. A Docker service is nearer to a Kubernetes Pod (#pods).

The following diagram shows two separate Services, each of which is comprised of multiple Pods. Each of the Pods in the diagram has the label `app=demo`, but their other labels differ. Service "frontend" matches all Pods with both `app=demo` and `component=frontend`, while Service "users" matches all Pods with `app=demo` and `component=users`. The Client Pod does not match either Service selector exactly, so it is not a part of either Service. However, the Client Pod can communicate with either of the Services because it runs in the same cluster.



Kubernetes assigns a stable, reliable IP address to each newly-created Service (the ClusterIP (https://v1-25.docs.kubernetes.io/docs/reference/generated/kubernetes-api/v1.25/#servicespec-v1-core)) from the cluster's pool of available Service IP addresses (#ip-allocation). Kubernetes also assigns a hostname to the ClusterIP, by adding a DNS entry (https://kubernetes.io/docs/concepts/services-networking/service/#dns). The ClusterIP and

hostname are unique within the cluster and don't change throughout the lifecycle of the Service. Kubernetes only releases the ClusterIP and hostname if the Service is deleted from the cluster's configuration. You can reach a healthy Pod running your application using either the ClusterIP or the hostname of the Service.

At first glance, a Service may seem to be a single point of failure for your applications. However, Kubernetes spreads traffic as evenly as possible across the full set of Pods, running on many nodes, so a cluster can withstand an outage affecting one or more (but not all) nodes.

Kube-Proxy

Kubernetes manages connectivity among Pods and Services using the `kube-proxy` component, which traditionally runs as a static Pod on each node.

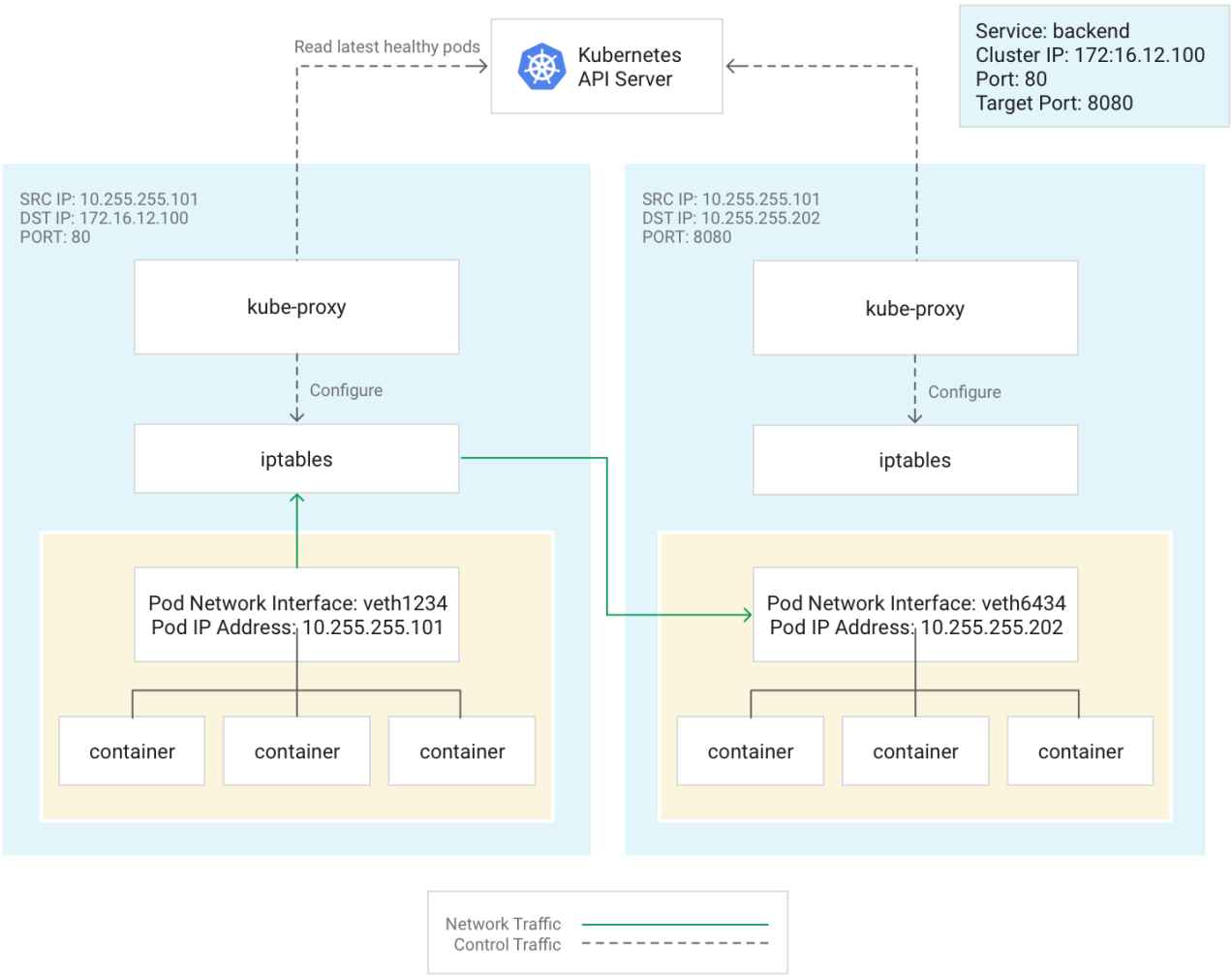
`kube-proxy`, which is **not** an inline proxy, but an egress-based load-balancing controller, watches the Kubernetes API server and continually maps the ClusterIP to healthy Pods by adding and removing destination NAT (DNAT) rules to the node's `iptables` subsystem. When a container running in a Pod sends traffic to a Service's ClusterIP, the node selects a Pod at random and routes the traffic to that Pod.

When you configure a Service, you can optionally remap its listening port by defining values for `port` and `targetPort`.

- The `port` is where clients reach the application.
- The `targetPort` is the port where the application is actually listening for traffic within the Pod.

`kube-proxy` manages this port remapping by adding and removing `iptables` rules on the node.

This diagram illustrates the flow of traffic from a client Pod to a server Pod on a different node. The client connects to the Service at `172.16.12.100:80`. The Kubernetes API server maintains a list of Pods running the application. The `kube-proxy` process on each node uses this list to create an `iptables` rule to direct traffic to an appropriate Pod (such as `10.255.255.202:8080`). The client Pod does not need to be aware of the topology of the cluster or any details about individual Pods or containers within them.



How `kube-proxy` is deployed depends on the GKE version of the cluster:

- For GKE versions 1.16.0 and 1.16.8-gke.13, `kube-proxy` is deployed as a DaemonSet.
- For GKE versions later than 1.16.8-gke.13, `kube-proxy` is deployed as a static Pod for nodes.

DNS

GKE provides the following managed cluster DNS options to resolve service names and external names:

- kube-dns: a cluster add-on that is deployed by default in all GKE clusters. For more information, see Using kube-dns (/kubernetes-engine/docs/how-to/kube-dns).
- Cloud DNS: a cloud-managed cluster DNS infrastructure that replaces kube-dns in the cluster. For more information, see Use Cloud DNS for GKE (/kubernetes-engine/docs/how-to/cloud-dns).

GKE also provides NodeLocal DNSCache (/kubernetes-engine/docs/how-to/nodelocal-dns-cache) as an optional add-on with kube-dns or Cloud DNS to improve cluster DNS performance.

To learn more about how GKE provides DNS, see Service discovery and DNS (/kubernetes-engine/docs/concepts/service-discovery).

Control plane

In Kubernetes, the control plane (/kubernetes-engine/docs/concepts/cluster-architecture#control_plane) manages the control plane processes, including the Kubernetes API server. How you access the control plane depends on the version of your GKE Autopilot (/kubernetes-engine/docs/concepts/choose-cluster-mode#why-autopilot) or Standard (/kubernetes-engine/docs/concepts/choose-cluster-mode#why-standard) cluster.

Clusters with Private Service Connect

Private or public clusters that meet any of the following conditions, use Private Service Connect to privately connect nodes and the control plane:

- New public clusters in version 1.23 on or after March 15, 2022.
- New private clusters in version 1.29 after January 28, 2024.

Existing *public* clusters that don't meet the preceding conditions are being migrated to Private Service Connect. Therefore, these clusters might already use Private Service Connect. To check if your cluster uses Private Service Connect, run the gcloud container clusters describe (/sdk/gcloud/reference/container/clusters/describe) command. If your public cluster uses Private Service Connect, `privateClusterConfig` resource has the following values:

- The `peeringName` field is empty or doesn't exist.
- The `privateEndpoint` field has a value assigned.

However, existing *private* clusters that don't meet the preceding conditions are not migrated yet.

You can create clusters that use Private Service Connect and change the cluster isolation (/kubernetes-engine/docs/how-to/change-cluster-isolation).

Use authorized networks (/kubernetes-engine/docs/how-to/authorized-networks#psc) to *restrict* the access to your cluster's control plane by defining the origins that can reach the control

plane.

Warning: Public clusters with Private Service Connect created before January 30, 2022 use a Private Service Connect *endpoint* and *forwarding rule*. Both resources are named **`gke-[cluster-name]-[cluster-hash:8]-[uuid:8]-pe`** and permit the control plane and nodes to privately connect. GKE creates these resources automatically with no cost. If you remove these resources, cluster network issues including downtime will occur.

Networking outside the cluster

This section explains how traffic from outside the cluster reaches applications running within a Kubernetes cluster. This information is important when designing your cluster's applications and workloads.

You've already read about how Kubernetes uses Services (#services) to provide stable IP addresses for applications running within Pods (#pods). By default, Pods don't expose an external IP address, because `kube-proxy` manages all traffic on each node. Pods and their containers can communicate freely, but connections outside the cluster cannot access the Service. For example, in the previous illustration, clients outside the cluster cannot access the frontend Service using its ClusterIP.

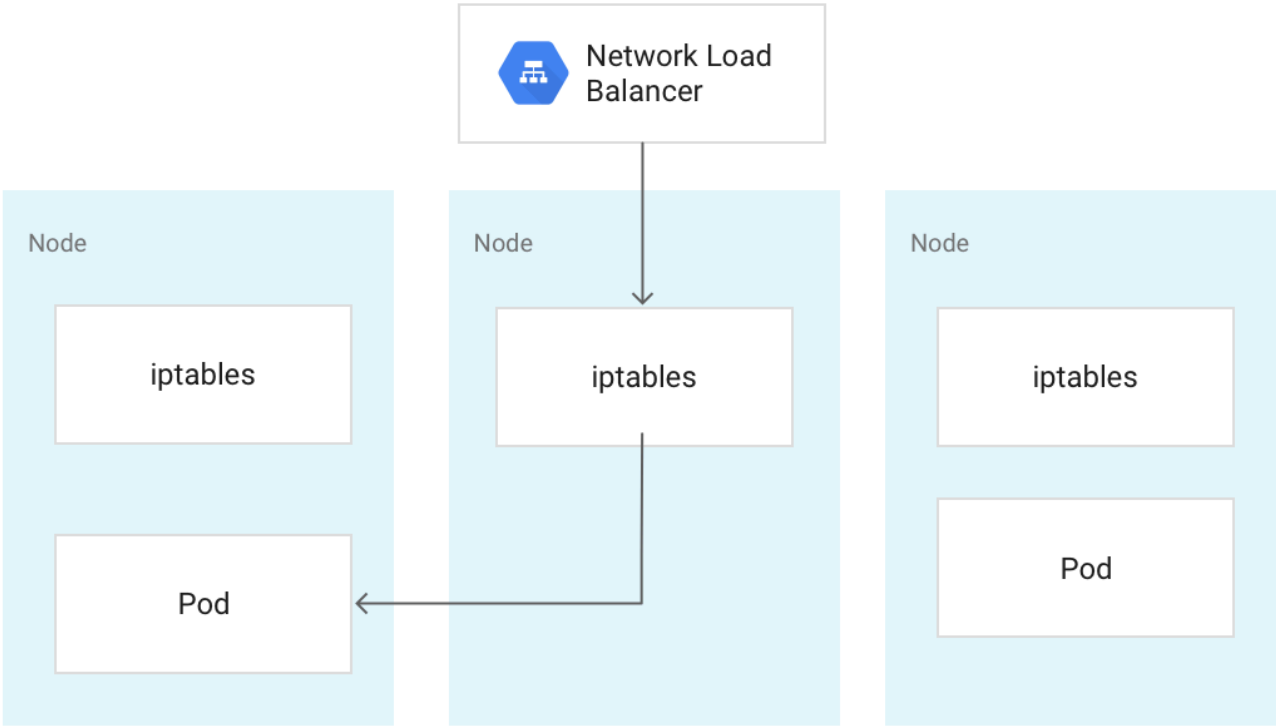
GKE provides three different types of load balancers to control access and to spread incoming traffic across your cluster as evenly as possible. You can configure one Service to use multiple types of load balancers simultaneously.

- **External load balancers** (#ext-lb) manage traffic coming from outside the cluster and outside your Google Cloud VPC network. They use forwarding rules associated with the Google Cloud network to route traffic to a Kubernetes node.
- **Internal load balancers** (#int-lb) manage traffic coming from within the same VPC network. Like external load balancers, they use forwarding rules associated with the Google Cloud network to route traffic to a Kubernetes node.
- **Application Load Balancers** (#https-lb) are specialized external load balancers used for HTTP(S) traffic. They use an Ingress resource rather than a forwarding rule to route traffic to a Kubernetes node.

When traffic reaches a Kubernetes node, it is handled the same way, regardless of the type of load balancer. The load balancer is not aware of which nodes in the cluster are running Pods for its Service. Instead, it balances traffic across all nodes in the cluster, even those

not running a relevant Pod. On a regional cluster, the load is spread across all nodes in all zones for the cluster's region. When traffic is routed to a node, the node routes the traffic to a Pod, which may be running on the same node or a different node. The node forwards the traffic to a randomly chosen Pod by using the `iptables` rules that `kube-proxy` manages on the node.

In the following diagram, the external passthrough Network Load Balancer directs traffic to the middle node, and the traffic is redirected to a Pod on the first node.



When a load balancer sends traffic to a node, the traffic might get forwarded to a Pod on a different node. This requires extra network hops. If you want to avoid the extra hops, you can specify that traffic must go to a Pod that is on the same node that initially receives the traffic.

To specify that traffic must go to a Pod on the same node, set `externalTrafficPolicy` to `Local` in your Service manifest:

```
apiVersion: v1
kind: Service
metadata:
  name: my-lb-service
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  selector:
    app: demo
```

```
    component: users
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

When you set `externalTrafficPolicy` to `Local`, the load balancer sends traffic only to nodes that have a healthy Pod that belongs to the Service. The load balancer uses a health check to determine which nodes have the appropriate Pods.

External load balancer

If your Service needs to be reachable from outside the cluster and outside your VPC network, you can configure your Service as a LoadBalancer (https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types), by setting the Service's `type` field to `Loadbalancer` when defining the Service. GKE then provisions an external passthrough Network Load Balancer (/load-balancing/docs/network) in front of the Service. The external passthrough Network Load Balancer is aware of all nodes in your cluster and configures your VPC network's firewall rules to allow connections to the Service from outside the VPC network, using the Service's external IP address. You can assign a static external IP address to the Service.

For more information, see Configuring domain names with static IP addresses (/kubernetes-engine/docs/tutorials/configuring-domain-name-static-ip).

To learn more about firewall rules, see Automatically created firewall rules (/kubernetes-engine/docs/concepts/firewall-rules).

Technical details

When using the external load balancer, arriving traffic is initially routed to a node using a forwarding rule associated with the Google Cloud network. After the traffic reaches the node, the node uses its `iptables` NAT table to choose a Pod. `kube-proxy` manages the `iptables` rules on the node.

Internal load balancer

For traffic that needs to reach your cluster from within the same VPC network, you can configure your Service to provision an internal passthrough Network Load Balancer (/kubernetes-engine/docs/how-to/internal-load-balancing). The internal passthrough Network Load Balancer chooses an IP address from your cluster's VPC subnet instead of an external

IP address. Applications or services within the VPC network can use this IP address to communicate with Services inside the cluster.

Technical details

Internal load balancing is provided by Google Cloud. When the traffic reaches a given node, that node uses its `iptables` NAT table to choose a Pod, even if the Pod is on a different node. `kube-proxy` manages the `iptables` rules on the node.

For more information about internal load balancers, see Using an internal passthrough Network Load Balancer (/kubernetes-engine/docs/how-to/internal-load-balancing).

Application Load Balancer

Many applications, such as RESTful web service APIs, communicate using HTTP(S). You can allow clients external to your VPC network to access this type of application using a Kubernetes Ingress (https://kubernetes.io/docs/concepts/services-networking/ingress/).

An Ingress lets you map hostnames and URL paths to Services within the cluster. When using an Application Load Balancer, you must configure the Service to use a NodePort (https://kubernetes.io/docs/concepts/services-networking/service/#nodeport), as well as a ClusterIP. When traffic accesses the Service on a node's IP at the NodePort, GKE routes traffic to a healthy Pod for the Service. You can specify a NodePort or allow GKE to assign a random unused port.

When you create the Ingress resource, GKE provisions an external Application Load Balancer (/load-balancing/docs/https) in the Google Cloud project. The load balancer sends a request to a node's IP address at the NodePort. After the request reaches the node, the node uses its `iptables` NAT table to choose a Pod. `kube-proxy` manages the `iptables` rules on the node.

This Ingress definition routes traffic for `demo.example.com` to a Service named `frontend` on port 80, and `demo-backend.example.com` to a Service named `users` on port 8080.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: demo
spec:
  rules:
  - host: demo.example.com
    http:
```

```
    paths:
    - backend:
        service:
          name: frontend
          port:
            number: 80
  - host: demo-backend.example.com
    http:
      paths:
      - backend:
          service:
            name: users
            port:
              number: 8080
```

Visit GKE Ingress for Application Load Balancer (/kubernetes-engine/docs/concepts/ingress) for more information.

Technical details

When you create an Ingress object, the GKE Ingress controller configures an Application Load Balancer according to the rules in the Ingress manifest and the associated Service manifests. The client sends a request to the Application Load Balancer. The load balancer is an actual proxy; it chooses a node and forwards the request to that node's `NodeIP:NodePort` combination. The node uses its `iptables` NAT table to choose a Pod. `kube-proxy` manages the `iptables` rules on the node.

Limiting connectivity between nodes

Creating ingress or egress firewall rules targeting nodes in your cluster may have adverse effects. For example, applying egress deny rules to nodes in your cluster could break features such as `NodePort` and `kubectl exec`.

Limiting connectivity to Pods and Services

By default, all Pods running within the same cluster can communicate freely. However, you can limit connectivity within a cluster in different ways, depending on your needs.

Limiting access among Pods

You can limit access among Pods using a network policy (/kubernetes-engine/docs/how-to/network-policy). Network policy definitions allow you to restrict the ingress (/kubernetes-engine/docs/tutorials/network-policy#step_2_restrict_incoming_traffic_to_pods) and egress (/kubernetes-engine/docs/tutorials/network-policy#step_3_restrict_outgoing_traffic_from_the_pods) of Pods based on an arbitrary combination of labels, IP ranges, and port numbers.

By default, there is no network policy, so all traffic among Pods in the cluster is allowed. As soon as you create the first network policy in a namespace, all other traffic is denied.

After creating a network policy, you must explicitly enable it for the cluster. For more information, see Configuring network policies for applications (/kubernetes-engine/docs/tutorials/network-policy).

Limiting access to an external load balancer

If your Service uses an external load balancer (#ext-lb), traffic from any external IP address can access your Service by default. You can restrict which IP address ranges can access endpoints within your cluster, by configuring the `loadBalancerSourceRanges` option when configuring the Service. You can specify multiple ranges, and you can update the configuration of a running Service at any time. The `kube-proxy` instance running on each node configures that node's `iptables` rules to deny all traffic that does not match the specified `loadBalancerSourceRanges`. No VPC firewall rule is created.

Limiting access to an Application Load Balancer

If your service uses an Application Load Balancer (#https-lb), you can use a Google Cloud Armor security policy (/armor/docs/security-policy-concepts) to limit which external IP addresses can access your Service and which responses to return when access is denied because of the security policy. You can configure Cloud Logging (/logging/docs) to log information about these interactions.

If a Google Cloud Armor security policy is not fine-grained enough, you can enable the Identity-Aware Proxy (/iap) on your endpoints to implement user-based authentication and authorization for your application. Visit the detailed tutorial for configuring IAP (/iap/docs/enabling-kubernetes-howto) for more information.

Known issues

This section covers the known issues.

Containerd-enabled node unable to connect to `172.17/16` range

A node VM with containerd enabled cannot connect to a host that has an IP within `172.17/16`. For more information, see Conflict with 172.17/16 IP address range (/kubernetes-engine/docs/troubleshooting/container-runtime#ip-address-conflict).

What's next

- Learn about Services (/kubernetes-engine/docs/concepts/service).
- Learn about Pods (https://kubernetes.io/docs/concepts/workloads/pods/).
- Set up a cluster with Shared VPC (/kubernetes-engine/docs/how-to/cluster-shared-vpc).

Last updated 2024-03-28 UTC.