GKE Ingress for Application Load Balancers

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

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

This page provides a general overview of what Ingress for external Application Load Balancers is and how it works. Google Kubernetes Engine (GKE) provides a built-in and managed Ingress controller called GKE Ingress. This controller implements Ingress resources as Google Cloud load balancers for HTTP(S) workloads in GKE.

Overview

In GKE, an Ingress object (https://kubernetes.io/docs/concepts/services-networking/ingress/) defines rules for routing HTTP(S) traffic to applications running in a cluster. An Ingress object is associated with one or more Service objects (/kubernetes-engine/docs/concepts/service), each of which is associated with a set of Pods. To learn more about how Ingress exposes applications using Services, see Service networking overview (/kubernetes-engine/docs/concepts/service-networking).

When you create an Ingress object, the GKE Ingress controller (https://github.com/kubernetes/ingress-gce) creates a Google Cloud HTTP(S) Load Balancer (/load-balancing/docs/https) and configures it according to the information in the Ingress and its associated Services.

To use Ingress, you must have the HTTP load balancing add-on enabled. GKE clusters have HTTP load balancing enabled by default; you must not disable it.

Ingress for external and internal traffic

GKE Ingress resources come in two types:

- Ingress for external Application Load Balancers (/kubernetes-engine/docs/concepts/ingress-xlb) deploys the classic Application Load Balancer (/load-balancing/docs/https). This internet-facing load balancer is deployed globally across Google's edge network as a managed and scalable pool of load balancing resources. Learn how to set up and use Ingress for external Application Load Balancers (/kubernetes-engine/docs/how-to/load-balance-ingress).

- Ingress for internal Application Load Balancers (/kubernetes-engine/docs/concepts/ingress-ilb) deploys the internal Application Load Balancer (/load-balancing/docs/l7-internal). These internal Application Load Balancers are powered by Envoy proxy systems outside of your GKE cluster, but within your VPC network. Learn how to set up and use Ingress for internal Application Load Balancers (/kubernetes-engine/docs/how-to/internal-load-balance-ingress).

Important: Whenever GKE creates an external Application Load Balancer or an internal Application Load Balancer through an Ingress object, you should avoid changing the load balancer's configuration using methods outside of GKE. Your customizations to settings for the load balancer's objects – forwarding rules, target proxies, URL maps, backend services, and health checks – are **overwritten** when new resources changes are applied, during periodic syncs, or during cluster upgrades. If you need to manage an external Application Load Balancer or an internal Application Load Balancer outside of GKE, use container native load balancing for standalone NEGs (/kubernetes-engine/docs/how-to/standalone-neg) instead.

GKE Ingress controller behavior

For clusters running GKE versions 1.18 and later, whether or not the GKE Ingress controller processes an Ingress depends on the value of the `kubernetes.io/ingress.class` annotation:

`kubernetes.io/ingress.class` value	`ingressClassName` value	GKE Ingress controller behavior
Not set	Not set or any value. This field is ignored.	Process the Ingress manifest and create an external Application Load Balancer.
`gce`	Any value. This field is ignored.	Process the Ingress manifest and create an external Application Load Balancer.
`gce-internal`	Any value. This field is ignored.	Process the Ingress manifest and create an internal Application Load Balancer.
Set to a value other than `gce` or `gce-internal`	Any value	Takes no action. The Ingress manifest could be processed by a third-party Ingress controller if one has been deployed.

`kubernetes.io/ingress.class` value	`ingressClassName` value	GKE Ingress controller behavior
Not set	Any value	Takes no action. The Ingress manifest could be processed by a third-party Ingress controller if one has been deployed.

For clusters running older GKE versions, the GKE controller processes any Ingress that does not have the annotation `kubernetes.io/ingress.class`, or has the annotation with the value `gce` or `gce-internal`.

`kubernetes.io/ingress.class` annotation deprecation

Although the `kubernetes.io/ingress.class` annotation is deprecated in Kubernetes (https://kubernetes.io/docs/concepts/services-networking/ingress/#deprecated-annotation), GKE continues to use this annotation.

You cannot use the `ingressClassName` field to specify a GKE Ingress. You must use the `kubernetes.io/ingress.class` annotation.

Features of external Application Load Balancers

An external Application Load Balancer, configured by Ingress, includes the following features:

- Flexible configuration for Services. An Ingress defines how traffic reaches your Services and how the traffic is routed to your application. In addition, an Ingress can provide a single IP address for multiple Services in your cluster.
- Integration with Google Cloud network services
- Support for multiple TLS certificates. An Ingress can specify the use of multiple TLS certificates for request termination.

For a comprehensive list, see Ingress configuration (/kubernetes-engine/docs/how-to/ingress-configuration).

Container-native load balancing

Note: This feature is not supported with Windows Server node pools.

Container-native load balancing (/kubernetes-engine/docs/how-to/container-native-load-balancing) is the practice of load balancing directly to Pod endpoints in GKE using Network Endpoint Groups (NEGs) (/load-balancing/docs/negs).

When using Instance Groups, Compute Engine load balancers send traffic to VM IPs as backends. When running containers on VMs, in which containers share the same host interface, this introduces some limitations:

- It incurs two hops of load balancing - one hop from the load balancer to the VM `NodePort` and another hop through kube-proxy routing to the Pod IP (which may reside on a different VM).
- Additional hops add latency and make the traffic path more complex.
- The Compute Engine load balancer has no direct visibility to Pods resulting in suboptimal traffic balancing.
- Environmental events like VM or Pod loss are more likely to cause intermittent traffic loss due to the double traffic hop.

With NEGs, traffic is load balanced from the load balancer directly to the Pod IP as opposed to traversing the VM IP and kube-proxy networking. In addition, Pod readiness gates (https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-readiness-gate) are implemented to determine the health of Pods from the perspective of the load balancer and not just the Kubernetes in-cluster health probes. This improves overall traffic stability by making the load balancer infrastructure aware of lifecycle events such as Pod startup, Pod loss, or VM loss. These capabilities resolve the above limitations and result in more performant and stable networking.

Container-native load balancing is enabled by default for Services when all of the following conditions are true:

- For Services created in GKE clusters 1.17.6-gke.7 and up
- Using VPC-native clusters
- Not using a Shared VPC
- Not using GKE Network Policy

In these conditions, Services will be annotated automatically with `cloud.google.com/neg: '{"ingress": true}'` indicating that a NEG should be created to mirror the Pod IPs within the Service. The NEG is what allows Compute Engine load balancers to communicate directly with Pods. Note that existing Services created prior to GKE 1.17.6-gke.7+ will not be automatically annotated by the Service controller.

For GKE 1.17.6-gke.7+ clusters where NEG annotation is automatic, it is possible to disable NEGs and force the Compute Engine external load balancer to use an instance group as its backends if necessary. This can be done by explicitly annotating Services with `cloud.google.com/neg: '{"ingress": false}'`. It is not possible to disable NEGs with Ingress for internal Application Load Balancers.

For clusters where NEGs are not the default, it is still **strongly recommended** to use container-native load balancing, but it must be enabled explicitly on a per-Service basis. The annotation should be applied to Services in the following manner:

```
kind: Service
...
  annotations:
    cloud.google.com/neg: '{"ingress": true}'
...
```

Caution: When you add this annotation, a new BackendService is created for the existing Service. This can result in a temporary outage for your Service.

Shared VPC

Ingress and MultiClusterIngress resources are supported in Shared VPC (/vpc/docs/shared-vpc), but they require additional preparation. The Ingress controller runs on the GKE control plane and makes API calls to Google Cloud using the GKE service account of the cluster's project. By default, when a cluster that is located in a Shared VPC service project uses a Shared VPC network, the Ingress controller cannot use the service project's GKE service account to create and update ingress allow firewall rules in the host project.

You can grant the service project's GKE service account permissions to create and manage VPC firewall rules (/kubernetes-engine/docs/how-to/cluster-shared-vpc#managing_firewall_resources) in the host project. By granting these permissions, GKE creates ingress allow firewall rules for the following:

- Google Front End (GFE) proxies and health check systems used by external Application Load Balancers for external Ingress. For more information, see the External Application Load Balancer overview (/load-balancing/docs/https#firewall-rules).
- Health check systems for internal Application Load Balancers used by internal Ingress.

★ **Note:** Whether or not you're using Shared VPC, if your cluster uses an internal Ingress, you must manually create ingress allow firewall rules (/kubernetes-engine/docs/concepts/firewall-rules#ingress-fws) to permit traffic from the proxy-only subnet (/load-balancing/docs/proxy-only-subnets) in the cluster's VPC network and region. Manual creation of these rules is needed because the GKE Ingress controller cannot create ingress allow firewall rules to permit traffic for an internal Application Load Balancer's proxy-only subnet. For more information, see Required networking environment in Ingress for internal Application Load Balancers (/kubernetes-engine/docs/concepts/ingress-ilb#required_networking_environment).

Manually provision firewall rules from the host project

If your security policies only allow firewall management from the host project, then you can provision these firewall rules manually. When deploying Ingress in a Shared VPC, the Ingress resource event provides the specific firewall rule you need to add necessary to provide access.

To manually provision a rule:

1. View the Ingress resource event:

```
kubectl describe ingress INGRESS_NAME
```

   Replace ***INGRESS_NAME*** with the name of your Ingress.

   You should see output similar to the following example:

```
Events:
Type    Reason  Age                    From                     Message
----    ------  ----                   ----                     -------
Normal  Sync    9m34s (x237 over 38h)  loadbalancer-controller  Firewall
```

The suggested required firewall rule appears in the `Message` column.

2. Copy and apply the suggested firewall rules from the host project. Applying the rule provides access to your Pods from the load balancer and Google Cloud health checkers.

★ **Note:** Different types of Ingress require different rules. GKE might show a firewall rule warning even if you have your own custom ingress firewall rules to allow the traffic. You can ignore the warning in that case.

Note: For Multi Cluster Ingress, this manual provision step is the only supported way to install firewall rules in a host project. Changing the service account permissions for automatic provisioning is not supported.

Providing the GKE Ingress controller permission to manage host project firewall rules

If you want a GKE cluster in a service project to create and manage the firewall resources in your host project, the service project's GKE service account must be granted the appropriate IAM permissions using one of the following strategies:

- Grant the service project's GKE service account the Compute Security Admin role (/compute/docs/access/iam#compute.securityAdmin) to the host project. The following example demonstrates this strategy.
- For a finer grained approach, create a custom IAM role (/iam/docs/creating-custom-roles#creating_a_custom_role) that includes only the following permissions: `compute.networks.updatePolicy`, `compute.firewalls.list`, `compute.firewalls.get`, `compute.firewalls.create`, `compute.firewalls.update`, and `compute.firewalls.delete`. Grant the service project's GKE service account that custom role to the host project.

If you have clusters in more than one service project, you must choose one of the strategies and repeat it for each service project's GKE service account.

```
gcloud projects add-iam-policy-binding HOST_PROJECT_ID \
  --member=serviceAccount:service-SERVICE_PROJECT_NUMBER@container-engine-r
  --role=roles/compute.securityAdmin
```

Replace the following:

- *`HOST_PROJECT_ID`*: the project ID (/resource-manager/docs/creating-managing-projects#identifying_projects) of the Shared VPC host project (/vpc/docs/shared-vpc#shared_vpc_host_project_and_service_project_associations).
- *`SERVICE_PROJECT_NUMBER`*: the project number (/resource-manager/docs/creating-managing-projects#identifying_projects) of the service project (/vpc/docs/shared-vpc#shared_vpc_host_project_and_service_project_associations) that contains your cluster.

Multiple backend services

Each external Application Load Balancer or internal Application Load Balancer uses a single URL map, which references one or more backend services. One backend service corresponds to each Service referenced by the Ingress.

For example, you can configure the load balancer to route requests to different backend services depending on the URL path. Requests sent to your-store.example could be routed to a backend service that displays full-price items, and requests sent to your-store.example/discounted could be routed to a backend service that displays discounted items.

You can also configure the load balancer to route requests according to the hostname. Requests sent to your-store.example could go to one backend service, and requests sent to your-experimental-store.example could go to another backend service.

In a GKE cluster, you create and configure an HTTP(S) load balancer by creating a Kubernetes Ingress object. An Ingress object must be associated with one or more Service objects, each of which is associated with a set of Pods.

Here is a manifest for an Ingress called `my-ingress`:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - http:
      paths:
```

```
      - path: /*
        pathType: ImplementationSpecific
        backend:
          service:
            name: my-products
            port:
              number: 60000
      - path: /discounted
        pathType: ImplementationSpecific
        backend:
          service:
            name: my-discounted-products
            port:
              number: 80
```

Note: To use Ingress, you must have the external Application Load Balancer add-on enabled. GKE clusters have external Application Load Balancers enabled by default; you must not disable it.

When you create the Ingress, the GKE ingress controller creates and configures an external Application Load Balancer or an internal Application Load Balancer according to the information in the Ingress and the associated Services. Also, the load balancer is given a stable IP address that you can associate with a domain name.

In the preceding example, assume you have associated the load balancer's IP address with the domain name your-store.example. When a client sends a request to your-store.example, the request is routed to a Kubernetes Service named `my-products` on port 60000. And when a client sends a request to your-store.example/discounted, the request is routed to a Kubernetes Service named `my-discounted-products` on port 80.

The only supported wildcard character for the `path` field of an Ingress is the `*` character. The `*` character must follow a forward slash (`/`) and must be the last character in the pattern. For example, `/*`, `/foo/*`, and `/foo/bar/*` are valid patterns, but `*`, `/foo/bar*`, and `/foo/*/bar` are not.

A more specific pattern takes precedence over a less specific pattern. If you have both `/foo/*` and `/foo/bar/*`, then `/foo/bar/bat` is taken to match `/foo/bar/*`.

For more information about path limitations and pattern matching, see the URL Maps documentation (/load-balancing/docs/url-map).

The manifest for the `my-products` Service might look like this:

```
apiVersion: v1
kind: Service
metadata:
  name: my-products
spec:
  type: NodePort
  selector:
    app: products
    department: sales
  ports:
  - protocol: TCP
    port: 60000
    targetPort: 50000
```

In the Service manifest, you must use `type: NodePort` unless you're using container native load balancing (/kubernetes-engine/docs/concepts/container-native-load-balancing). If using container native load balancing, use the `type: ClusterIP`.

In the Service manifest, the `selector` field says any Pod that has both the `app: products` label and the `department: sales` label is a member of this Service.

When a request comes to the Service on port 60000, it is routed to one of the member Pods on TCP port 50000.

Each member Pod must have a container listening on TCP port 50000.

The manifest for the `my-discounted-products` Service might look like this:

```
apiVersion: v1
kind: Service
metadata:
  name: my-discounted-products
spec:
  type: NodePort
  selector:
    app: discounted-products
    department: sales
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

In the Service manifest, the `selector` field says any Pod that has both the `app: discounted-products` label and the `department: sales` label is a member of this Service.

When a request comes to the Service on port 80, it is routed to one of the member Pods on TCP port 8080.

Each member Pod must have a container listening on TCP port 8080.

Default backend

You can specify a default backend by providing a `defaultBackend` field in your Ingress manifest. Any requests that don't match the paths in the `rules` field are sent to the Service and port specified in the `defaultBackend` field. For example, in the following Ingress, any requests that don't match `/` or `/discounted` are sent to a Service named `my-products` on port 60001.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  defaultBackend:
    service:
      name: my-products
      port:
        number: 60001
  rules:
  - http:
      paths:
      - path: /
        pathType: ImplementationSpecific
        backend:
          service:
            name: my-products
            port:
              number: 60000
      - path: /discounted
        pathType: ImplementationSpecific
        backend:
          service:
            name: my-discounted-products
            port:
              number: 80
```

If you don't specify a default backend, GKE provides a default backend that returns 404. This is created as a `default-http-backend` NodePort service on the cluster in the `kube-system` namespace.

The 404 HTTP response is similar to the following:

```
response 404 (backend NotFound), service rules for the path non-existent
```

To set up GKE Ingress with a customer default backend, see the GKE Ingress with custom default backend (https://github.com/GoogleCloudPlatform/gke-networking-recipes/tree/main/ingress/single-cluster/ingress-custom-default-backend) .

Ingress to Compute Engine resource mappings

The GKE Ingress controller deploys and manages Compute Engine load balancer resources based on the Ingress resources that are deployed in the cluster. The mapping of Compute Engine resources depends on the structure of the Ingress resource.

The following manifest describes an Ingress:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - http:
      paths:
      - path: /*
        pathType: ImplementationSpecific
        backend:
          service:
            name: my-products
            port:
              number: 60000
      - path: /discounted
        pathType: ImplementationSpecific
        backend:
          service:
            name: my-discounted-products
            port:
              number: 80
```

This Ingress manifest instructs GKE to create the following Compute Engine resources:

- A forwarding rule and IP address.
- Compute Engine firewall rules (/kubernetes-engine/docs/concepts/firewall-rules) that permit traffic for load balancer health checks and application traffic from Google Front Ends or Envoy proxies.
- A target HTTP proxy and a target HTTPS proxy, if you configured TLS.
- A URL map which with a single host rule referencing a single path matcher. The path matcher has two path rules, one for `/*` and another for `/discounted`. Each path rule maps to a unique backend service.
- NEGs which hold a list of Pod IP addresses from each Service as endpoints. These are created as a result of the `my-discounted-products` and `my-products` Services.

Options for providing SSL certificates

There are three ways to provide SSL certificates to an HTTP(S) load balancer:

Google-managed certificates

Google-managed SSL certificates (/kubernetes-engine/docs/how-to/managed-certs) are provisioned, deployed, renewed, and managed for your domains. Managed certificates do not support wildcard domains.

Self-managed certificates shared with Google Cloud

You can provision your own SSL certificate and create a certificate resource in your Google Cloud project. You can then list the certificate resource in an annotation on an Ingress to create an HTTP(S) load balancer that uses the certificate. Refer to instructions for pre-shared certificates (/kubernetes-engine/docs/how-to/ingress-multi-ssl#using_pre-shared_certificates) for more information.

Self-managed certificates as Secret resources

You can provision your own SSL certificate and create a Secret (https://kubernetes.io/docs/tasks/configmap-secret/managing-secret-using-kubectl/#create-a-secret) to hold it. You can then refer to the Secret in an Ingress specification to create an HTTP(S) load balancer that uses the certificate. Refer to the instructions for using certificates in Secrets (/kubernetes-engine/docs/how-to/ingress-multi-ssl) for more information.

Health checks

When you expose one or more Services through an Ingress using the default Ingress controller, GKE creates a classic Application Load Balancer (/kubernetes-engine/docs/concepts/ingress-xlb) or an internal Application Load Balancer (/kubernetes-engine/docs/concepts/ingress-ilb). Both of these load balancers support multiple backend services (/load-balancing/docs/backend-service) on a single URL map (/load-balancing/docs/url-map-concepts). Each of the backend services corresponds to a Kubernetes Service, and each backend service must reference a Google Cloud health check (/load-balancing/docs/health-check-concepts). This health check is *different* from a Kubernetes liveness or readiness probe because the health check is implemented outside of the cluster.

Note: Load balancer health checks are specified *per backend service*. While it's possible to use the same health check for all backend services of the load balancer, the health check reference isn't specified for the whole load balancer (at the Ingress object itself).

GKE uses the following procedure to create a health check for each backend service corresponding to a Kubernetes Service:

- If the Service references a `BackendConfig` CRD (#direct_hc) with `healthCheck` information, GKE uses that to create the health check. Both the GKE Enterprise Ingress controller and the GKE Ingress controller support creating health checks this way.
- If the Service does *not* reference a `BackendConfig` CRD:
    - GKE can infer some or all of the parameters for a health check if the Serving Pods use a Pod template with a container whose readiness probe has attributes that can be interpreted as health check parameters. See Parameters from a readiness probe (#interpreted_hc) for implementation details and Default and

inferred parameters (#def_inf_hc) for a list of attributes that can be used to create health check parameters. Only the GKE Ingress controller supports inferring parameters from a readiness probe.

- If the Pod template for the Service's serving Pods does **not** have a container with a readiness probe whose attributes can be interpreted as health check parameters, the default values (#def_inf_hc) are used to create the health check. Both the GKE Enterprise Ingress controller and the GKE Ingress controller can create a health check using only the default values.

Note: **`containerPort`** field must be defined under pod spec and **`targetPort`** field must be defined under service spec for health check to infer parameters defined under readiness probe. If they are not defined, the health check for the backend service defaults to "/" path even if parameters are defined under readiness probe.

Default and inferred parameters

The following parameters are used when you do **not** specify health check parameters for the corresponding Service using a `BackendConfig` CRD (#direct_hc).

Health check parameter	Default value	Inferable value
Protocol (/load-balancing/docs/health-check-concepts#category_and_protocol)	HTTP	if present in the Service annotation **`cloud.goo`**
Request path (/load-balancing/docs/health-check-concepts#criteria-protocol-http)	**`/`**	if present in the serving Pod's **`spec`**: **`containers[].readinessProbe.httpGet`**
Request Host header (/load-balancing/docs/health-check-concepts#criteria-protocol-http)	**`Host:`** ***`backend-ip-address`***	if present in the serving Pod's **`spec`**: **`containers[].readinessProbe.httpGet`**
Expected response (/load-balancing/docs/health-check-concepts#criteria-protocol-http)	HTTP 200 (OK)	HTTP 200 (OK) cannot be changed

Check interval (/load-balancing/docs/health-check-concepts#probes)	• for zonal NEGs: 15 seconds • for instance groups: 60 seconds	if present in the serving Pod's **`spec`**: • for zonal NEGs: **`containers[].readinessProbe.peri`** • for instance groups: **`containers[].readinessProbe.peri`**
Check timeout (/load-balancing/docs/health-check-concepts#probes)	5 seconds	if present in the serving Pod's **`spec`**: **`containers[].readinessProbe.timeout`**
Healthy threshold (/load-balancing/docs/health-check-concepts#health_state)	1	1 cannot be changed
Unhealthy threshold (/load-balancing/docs/health-check-concepts#health_state)	• for zonal NEGs: 2 • for instance groups: 10	same as default: • for zonal NEGs: 2 • for instance groups: 10
Port specification (/load-balancing/docs/health-check-concepts#category_and_port_specification)	• for zonal NEGs: the Service's **`port`** • for instance groups: the Service's **`nodePort`**	The health check probes are sent to the port nu **`containers[].readinessProbe.httpGet`** following are also true: • The readiness probe's port number must ma **`containers[].spec.ports.containe`** • The serving Pod's **`containerPort`** matche • The Ingress *service backend port specificati* **`spec.ports[]`** of the Service. This can be ◦ **`spec.rules[].http.paths[].ba`** the Ingress matches **`spec.ports[]`** corresponding Service ◦ **`spec.rules[].http.paths[].ba`** in the Ingress matches **`spec.ports`** corresponding Service
Destination IP address (/load-balancing/docs/health-check-concepts#hc-packet-dest)	• for zonal NEGs: the Pod's IP address • for instance groups:	same as default: • for zonal NEGs: the Pod's IP address • for instance groups: the node's IP address

	the node's IP address

Note: A valid Kubernetes readiness probe supports setting multiple HTTP headers in `readinessProbe.httpGet` (https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.25/#httpgetaction-v1-core). If `readinessProbe.httpGet.httpHeaders` specifies more than just the `Host` header, the load balancer's health check parameters are set to default values instead of values inferred from the readiness probe. This limitation exists because health checks (/load-balancing/docs/health-check-concepts#headers) only support setting the `Host` header.

Parameters from a readiness probe

When GKE creates the health check for the Service's backend service, GKE can copy certain parameters from *one container's* readiness probe used by that Service's serving Pods. This option is *only* supported by the GKE Ingress controller.

Supported readiness probe attributes that can be interpreted as health check parameters are listed along with the default values in Default and inferred parameters (#def_inf_hc). Default values are used for any attributes not specified in the readiness probe or if you don't specify a readiness probe at all.

If serving Pods for your Service contain *multiple* containers, or if you're using the GKE Enterprise Ingress controller, you should use a `BackendConfig` CRD to define health check parameters. For more information, see When to use a `BackendConfig` CRD instead (#direct_hc_instead).

Warning: When a backend service's health check parameters are inferred from a serving Pod's readiness probe, GKE does **not** keep the readiness probe and health check synchronized:

- If you change the readiness probe for the serving Pods of a Service referenced by an Ingress after GKE has created the external Application Load Balancer or the internal Application Load Balancer for that Ingress, the changes you make to the readiness probe will **not** be copied to the health check for the corresponding backend service on the load balancer.
- Conversely, if you make changes to the health check used by a backend service of a load balancer created on behalf of an Ingress, GKE will **not** update any readiness probe, nor will it revert any changes you make to the health check.

When to use `BackendConfig` CRDs instead

Instead of relying on parameters from Pod readiness probes, you should explicitly define health check parameters for a backend service by creating a `BackendConfig` CRD (#direct_hc) for the Service in these situations:

- **If you're using GKE Enterprise:** The GKE Enterprise Ingress controller does **not** support obtaining health check parameters from the readiness probes of serving Pods. It can only create health checks using implicit parameters (#def_inf_hc) or as defined in a `BackendConfig` CRD (#direct_hc).
- **If you have more than one container in the serving Pods:** GKE does not have a way to select the readiness probe of a *specific container* from which to infer health check parameters. Because each container can have its own readiness probe, and because a readiness probe isn't a required parameter for a container, you should define the health check for the corresponding backend service by referencing a `BackendConfig` CRD (#direct_hc) on the corresponding Service.
- **If you need control over the port used for the load balancer's health checks:** GKE only uses the readiness probe's `containers[].readinessProbe.httpGet.port` for the backend service's health check when that port matches the service port for the Service referenced in the Ingress `spec.rules[].http.paths[].backend.servicePort`.

Parameters from a `BackendConfig` CRD

You can specify the backend service's health check parameters using the `healthCheck` parameter of a `BackendConfig` CRD (/kubernetes-engine/docs/how-to/ingress-configuration#direct_health) referenced by the corresponding Service. This gives you more flexibility and control over health checks for a classic Application Load Balancer or an internal Application Load Balancer created by an Ingress. See Ingress configuration (/kubernetes-engine/docs/how-to/ingress-configuration) for GKE version compatibility.

This example `BackendConfig` CRD defines the health check protocol (type), a request path, a port, and a check interval in its `spec.healthCheck` attribute:

```
apiVersion: cloud.google.com/v1
kind: BackendConfig
metadata:
  name: http-hc-config
spec:
  healthCheck:
    checkIntervalSec: 15
    port: 15020
    type: HTTPS
    requestPath: /healthz
```

To configure all the fields available when configuring a `BackendConfig` health check, use the custom health check configuration (/kubernetes-engine/docs/how-to/ingress-configuration#direct_health) example.

To set up GKE Ingress with a custom HTTP health check, see GKE Ingress with custom HTTP health check (https://github.com/GoogleCloudPlatform/gke-networking-recipes/tree/main/ingress/single-cluster/ingress-custom-http-health-check)
.

Using multiple TLS certificates

Suppose you want an HTTP(S) load balancer to serve content from two hostnames: your-store.example and your-experimental-store.example. Also, you want the load balancer to use one certificate for your-store.example and a different certificate for your-experimental-store.example.

You can do this by specifying multiple certificates in an Ingress manifest. The load balancer chooses a certificate if the Common Name (CN) in the certificate matches the hostname used in the request. For detailed information on how to configure multiple certificates, see Using multiple SSL certificates in HTTPS Load Balancing with Ingress (/kubernetes-engine/docs/how-to/ingress-multi-ssl).

Kubernetes Service compared to Google Cloud backend service

A Kubernetes Service (https://kubernetes.io/docs/concepts/services-networking/service/) and a Google Cloud backend service (/load-balancing/docs/backend-service) are different things. There is a strong relationship between the two, but the relationship is not necessarily one to one. The GKE ingress controller creates a Google Cloud backend service for each (`service.name`, `service.port`) pair in an Ingress manifest. So it is possible for one Kubernetes Service object to be related to several Google Cloud backend services.

Limitations

- In clusters using versions earlier than 1.16, the total length of the namespace and name of an Ingress must not exceed 40 characters. Failure to follow this guideline may cause the GKE Ingress controller to act abnormally. For more information, see this Github issue about long names (https://github.com/kubernetes/ingress-gce/issues/537).
- In clusters using NEGs, ingress reconciliation time may be affected by the number of ingresses. For example, a cluster with 20 ingresses, each containing 20 distinct NEG backends, may result in a latency of more than 30 minutes for an ingress change to be reconciled. This especially impacts regional clusters due to the increased number of NEGs needed.
- Quotas for URL maps (/load-balancing/docs/quotas#url_maps) apply.
- Quotas for Compute Engine resources (/compute/docs/resource-quotas) apply.
- If you're not using NEGs with the GKE ingress controller (https://github.com/kubernetes/ingress-gce) then GKE clusters have a limit of 1000 nodes. When services are deployed with NEGs, there is no GKE node limit. Any non-NEG Services exposed through Ingress do not function correctly on clusters above 1000 nodes.
- For the GKE Ingress controller to use your `readinessProbes` as health checks, the Pods for an Ingress must exist at the time of Ingress creation. If your replicas are scaled to 0, the default health check applies. For more information, see this Github issue comment about health checks (https://github.com/kubernetes/ingress-gce/issues/241#issuecomment-384749607).
- Changes to a Pod's `readinessProbe` do not affect the Ingress after it is created.
- An external Application Load Balancer terminates TLS in locations that are distributed globally, to minimize latency between clients and the load balancer. If you require geographic control over where TLS is terminated, you should use a custom ingress controller (https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/) exposed through a GKE Service of type `LoadBalancer` instead, and terminate TLS on backends that are located in regions appropriate to your needs.
- Combining multiple Ingress resources into a single Google Cloud load balancer is not supported.
- You must turn off mutual TLS on your application because it is not supported for external Application Load Balancers (/load-balancing/docs/https#limitations).

External Ingress and routes-based clusters

If you use routes-based clusters with external Ingress, the GKE Ingress controller cannot use container-native load balancing using `GCE_VM_IP_PORT` network endpoint groups (NEGs). Instead, the Ingress controller uses unmanaged instance group backends that include all nodes in all node pools. If these unmanaged instance groups are also used by `LoadBalancer` Services, it can cause issues related to the Single load-balanced instance group limitation (/kubernetes-engine/docs/concepts/service-load-balancer#limit-lb-ig).

Some older external Ingress objects created in VPC-native clusters might use instance group backends on the backend services of each external Application Load Balancer they create. This is not relevant to internal Ingress because internal Ingress resources always use `GCE_VM_IP_PORT` NEGs and require VPC-native clusters.

To learn how to troubleshoot 502 errors with external Ingress, see External Ingress produces HTTP 502 errors (/kubernetes-engine/docs/troubleshooting/troubleshoot-load-balancing#ingress-502s).

Implementation details

- The Ingress controller performs periodic checks of service account permissions by fetching a test resource from your Google Cloud project. You will see this as a `GET` of the (non-existent) global `BackendService` with the name `k8s-ingress-svc-acct-permission-check-probe`. As this resource should not normally exist, the `GET` request will return "not found". This is expected; the controller is checking that the API call is not rejected due to authorization issues. If you create a BackendService with the same name, then the `GET` will succeed instead of returning "not found".

Templates for the Ingress configuration

- In GKE Networking Recipes (https://github.com/GoogleCloudPlatform/gke-networking-recipes/tree/main), you can find templates provided by GKE on many common Ingress usage under the Ingress (https://github.com/GoogleCloudPlatform/gke-networking-recipes/tree/main/ingress/single-cluster) section.

What's next

- Learn about GKE Networking Recipes (https://github.com/GoogleCloudPlatform/gke-networking-recipes/tree/main)
- Learn more about load balancing in Google Cloud (/load-balancing/docs/https).
- Read an overview of networking in GKE (/kubernetes-engine/docs/concepts/network-overview).
- Learn how to configure Ingress for internal Application Load Balancers (/kubernetes-engine/docs/how-to/internal-load-balance-ingress).
- Learn how to configure Ingress for external Application Load Balancers (/kubernetes-engine/docs/how-to/load-balance-ingress).

Last updated 2023-12-04 UTC.