Implement Global HTTP(S) Load Balancing with Google Kubernetes Engine Through 'Standalone NEGs'

External request ingress into Kubernetes

Kubernetes comes built-in with the Ingress resource, Ingress resource simplifies the external traffic management into your services running on a Kubernetes cluster by providing a single point of entry for the incoming traffic and then allowing you to configure rules based on host or request path in order to route the traffic to the appropriate back-end. Ingress remains a popular way for organizations to handle external traffic related aspects of their Kubernetes based web applications. However, with the rise of Gateway API, this has started to change lately.

Acknowledging that this article is not about Gateway API (will cover that in a subsequent blog), let's shift focus to Ingress, if you are using a managed Kubernetes services such as GKE, in all likelihood you may also be using the "cluster-scoped" Ingress Controller provided to you by Google Cloud, if not, you maybe using something like Nginx Ingress Controller to address the Ingress resources as they are created and thus provision and configure the load balancing infrastructure for your application. In either of these cases (GKE provided controller or running something like Nginx) you should know that the controller itself is a cluster-scoped resource, that is it is essentially tied to your cluster just like Ingress resource itself. If you have looked at GKE Ingress in details, you would have also noticed the custom resources like BackendConfig, FrontendConfig and ManagedCertificate that allows you to further customize and configure the load balancer (which essentially rolls up to a global HTTP(S) load balancer on Google Cloud).

All of it is fantastic and really simplifies managing the load balancer configurations largely with the help of Ingress resource itself plus the custom resources, but sometimes, this is just not enough, why? Because you may be working in an environment where load balancing is provisioned and managed externally or Kubernetes pods may not be the only backend target, what if you have some legacy VM based applications, a Cloud Storage bucket serving static content or a serverless app hosting a web based form for quick data capture and submission. Clearly the GKE provided controller or the Nginx controller alone can't satisfy this requirement. They are great options when the backend is just Kubernetes services.

This is where we come to realize the value of Standalone NEGs and how GKE can actually indicate the built-in NEG controllers to instead create Standalone Zonal NEGs for a later consumption as backend services by the Global HTTP(S) Load Balancer.

Let's dive in and see this in action.

What are NEGs?

NEGs or the Network Endpoint Groups are like a configuration object or simply put a data structure (like a tuple) consisting of network endpoints or backend service endpoints (basically IPs or IP and Port combination). Google Cloud supports a variety of NEGs:

1. Zonal NEGs
2. Hybrid NEGs
3. Internet NEGs
4. Serverless NEGs
5. PSC NEGs

Each type of NEG listed above has a use case associated with it. For example, with Internet NEGs you can literally put any publicly available endpoint behind Google's HTTP(S) load balancing and leverage Global Load Balancing's features like TLS termination, Cloud Armor for DDoS mitigation, URL Maps for routing and Cloud CDN etc. For a deeper description of these NEGs work, please refer to the documentation here.

For this blog, our focus is going to be the Zonal NEGs. This is what GKE supports to create service endpoints which can then be used with the backend services with the Global HTTP(S) Loadbalancer.

Zonal NEGs can further be divided into two types:

GCE_VM_IP - Only concerning the primary internal IP of the virtual machine
GCE_VM_IP_PORT - This can resolve to primary internal IP of the virtual machine and as well as secondary or alias IPs used on the virtual machine, like the ones used by containers/pods.

Google Kubernetes Engine supports GCE_VM_IP_PORT NEGs, which makes it possible to route traffic directly to the pods running on the VMs in your cluster, the pod IPs are are assigned from the alias IP range that you specify while creating the cluster and honored by the CNI plug-in managing the overlay networking of your Kubernetes cluster, we will not get into the details of Kubernetes networking here as it is out of scope for this blog but well, if you have experience creating GKE clusters in your VPC on Google Cloud, the IP ranges that you assign for pods and services also appear as secondary/alias IP ranges on the subnet on which your cluster resides (or the VMs are connected to), if you choose to create a VPC native cluster (which is selected by default and recommended), the pod IPs become routable on your VPC and that comes in handy when it comes to targeting them as backends directly from the load balancing proxy at the edge of Google's network also referred to as Google Front End or GFE in many contexts.

In this blog I may interchangeably use terms like VPC native clusters and container native load balancing but I would really be referring to the same idea - making pod IPs routable on VPC and target them from the load balancer directly, which has it's own benefits.

Alright, we now understand what NEGs are, let's understand what are Standalone NEGs and why they are so useful. GKE has a built in Ingress Controller which is a "cluster scoped" (as described above) resource and in turn a GKE cluster is a regional resource, you can have zonal clusters too at reduced availability SLAs though, but not recommended at least for production workloads.

When you create a GKE cluster, you can enable the HTTPS load balancing (enabled by default, you can choose not to use it, that is disable it), when this add-on is enabled, GKE will also deploy the built-in Ingress Controller for the cluster, just like other Ingress Controllers that you are familiar with or have worked with like the popular open source Nginx Ingress Controller, the GKE Ingress Controller will watch for the Ingress resources that you create in your cluster and "automatically" configures a Global HTTP(S) load balancer.

This is really useful, given the level of abstraction that you get, GKE Ingress Controller is responsible for appropriately programming your load balancer by looking at your Ingress resource, you can also supply a couple of custom resources (CRDs) provided by GKE, that is the BackendConfig and FrontendConfig to control/configure several other aspects of your load balancer, such as health checks, SSL certificates, CDN, timeouts etc (this was also mentioned briefly above, again not going into their details).

In short, the GKE Ingress Controller is the brain behind provisioning the load balancing infrastructure for you and in this process also create NEGs, which are nothing but the Kubernetes pod IPs and container ports as the eligible endpoints to your backend service component of the load balancer. If you use VPC native or container native load balancing which is something I highly recommend, the traffic entering from Google Cloud Load Balancing will be directly routed to your pods directly, which is excellent for applications with low tolerance to latencies, it reduces unnecessary hops that request has to take otherwise by directly targeting the routable pod IPs, you can simply use ClusterIP services instead of NodePort type here. Though NodePort services work with container native load balancing as well, unless you don't need to use the nodePort assigned, I would recommend to use ClusterIP services with container native load balancing.

Note - To use NEGs with Global Load Balancing, you must choose to use VPC native clusters, private GKE clusters are VPC native by default. Read more on VPC native GKE clusters here.

For a non-container-native load balancing scheme, where you expose your services as Kubernetes NodePort services and essentially, all the traffic first reaches the VM at it's IP and assigned nodePort, NodePort services are basically built atop ClusterIP type, so the traffic is then forwarded to the ClusterIP port and finally via the kube-proxy is routed to one or the another pod under that service, the point here is that the pod may or may not be on the same VM which received the request to begin with from the load balancer. Which means now the traffic can potentially spill over to another VM, that VM could be in the same zone or in another zone etc. thus adding hops and contributing to the latency. Also this may lead to you paying some inter-zone egress charges, unless you control pod placement.
Coming back to the topic of Ingress Controllers and how it programs and manages the load balancing for your GKE clusters, managing NEG also becomes a part of it's job as a result, as new pods are

created, destroyed, it keeps a track of such changes and ensure that the load balancer has the NEGs as up-to date as possible.

So, when one has such a great deal of automation and functionality built into GKE Ingress Controllers, why would anyone not use it? Correct? Well, the answer is that it depends because the situation may dictate you to not use the Ingress and Ingress Controller as we will see in the next section.

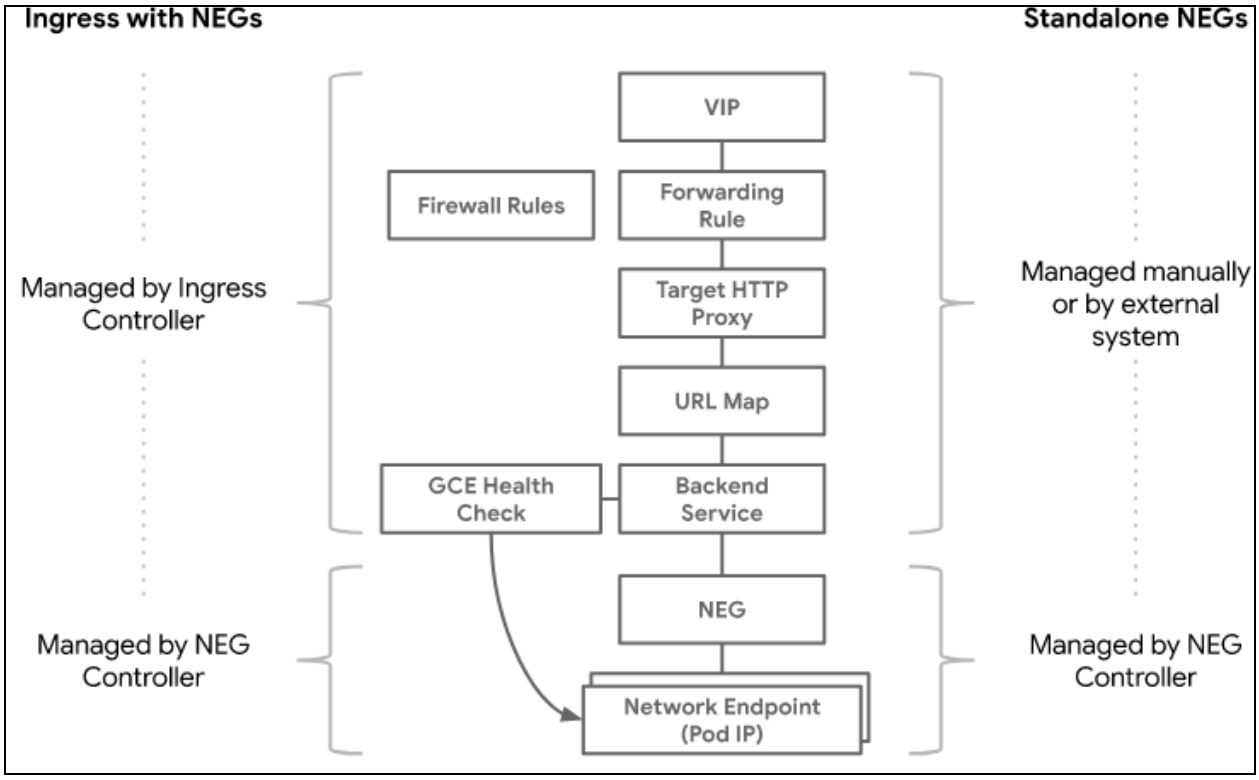
When Standalone NEGs for GKE becomes important?

If you do not manage or provision the Global HTTPS Load Balancer through the GKE Ingress Controller but rather provision and manage it externally as a separate Google Cloud resource, then using stand alone NEGs becomes essential if you would like to have VPC native clusters where incoming traffic can be directly routed to your pods.

This can be a common use case for many organizations. You might have a variety of backends (VMs, storage buckets, internet NEGs & GKE pods) fronted by a single load balancer and not just GKE services, it's also possible that a different team manages and governs load balancing and other edge networking components and security related constructs like TLS certificates.

Standalone NEGs can also be used with Google's TCP Proxy load balancing or SSL Proxy load balancers.

This figure [from Google Cloud GKE documentation] gives a much better understanding of what the resource model may look like with standalone NEG when compared to NEGs created and managed by the Ingress/Ingress Controller.



Limitations of Standalone NEGs when used with GKE

While Standalone NEGs (VM IP & PORT type with GKE) gives plenty of flexibility, one must be aware of the limitations, most importantly the scenarios such as NEG leakage.

1. When a GKE service is deleted, the associated NEG will not be garbage collected if the NEG is still referenced by a backend service. Dereference the NEG from the backend service to allow NEG deletion
2. These NEGs are also retained on cluster deletion and needs separate clean-up
3. Standalone NEGs make no assumptions on the health checks, while Ingress created NEGs are configured with health checks even if not specified. Though you can configure your own health checks through CRDs, it's not the case with standalone NEGs. Compute Engine health checks should always be configured along with the load balancer to prevent the traffic from being sent to backends which are not ready to receive it. If there is no health check status associated with the NEG (usually because no health check is configured), then the NEG controller will mark the pod's readiness gate value to True when its corresponding endpoint is programmed in NEG, leading to unexpected behavior.
4. When programming/configuring the load balancer via the GKE Ingress, firewall rules are auto created to allow the proxy as well as the healthchek through the VPC's firewall, if you are using

NEGs as backends to the proxy, you will have to ensure that a firewall rule exist to allow both the proxy and healthcheck

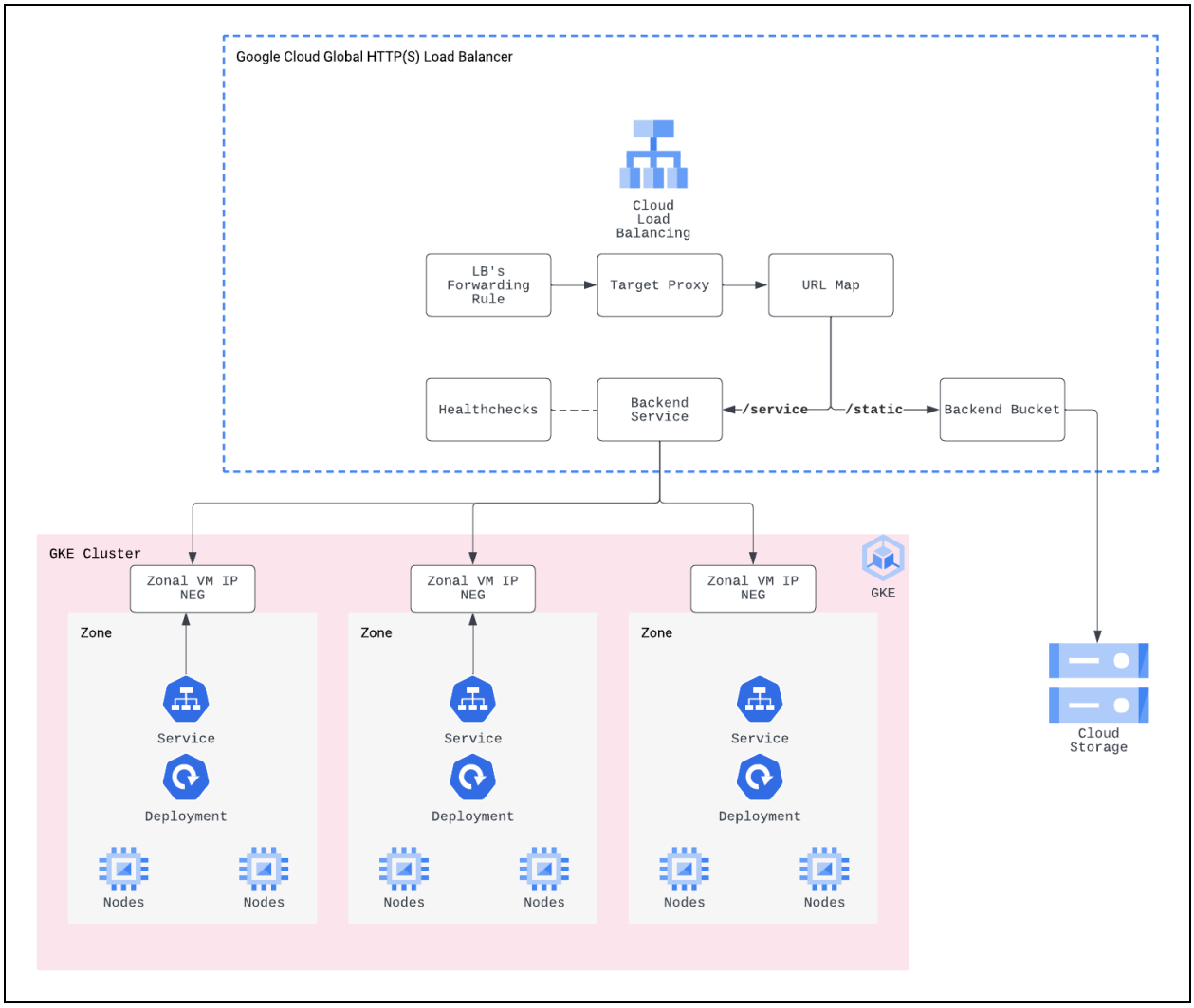
5. A more comprehensive overview of configurations needed by NEG based VPC native load balancing, please refer to the official documentation here.

Next, we will dive into some hands-on stuff, in the Google Cloud documentation, you will find the examples using gcloud commands, I have used Terraform, why? Because I always use Terraform! What's better than having your automation track and remember the state and reclaim the infrastructure in the same order it was created, without you having to hand-pick and delete when you don't need it? Yes, shutting down the project is an easy tool, but what if you have other collaborators in the project? In-short gcloud commands are great but they don't track state and dependencies, as much as possible use automation tools like Terraform and only use console/gcloud for verification purposes.

In the code samples, you will see how a load balancer provisioned externally can communicate with GKE pods via NEGs while also routing all the static content requests to a Cloud Storage Bucket!

The demo application I have used is available on Google Cloud Github page here.

Reference architecture and set-up



You can complete a similar set up as documented in Google Cloud public documentation as described here.

Or, if you love Terraform then here's a quick GKE NEG PoC toolkit that I have put together on Github, you can use that. The benefit of using Terraform is that it's much easier to provision, modify and remove resources, you of course don't want to get billed for these experimental projects. You can easily destroy all the resources created by Terraform without having to worry about any specific sequence or dependencies, Terraform will do that for you.

I will highlight the snippets which matter the most for completing the set up and try-out.

Before we begin, a quick check on the tools you will need:

1. A Google Cloud project with a billing account associated with it
2. Terraform installed on your computer/workstation
3. Google Cloud SDK
4. Kubectl

Prerequisites:

1. Make sure you have a Google Cloud account and owner permissions to the project you are going to use, though I highly recommend that you follow the principle of least privileges in your environments
2. Authenticate the gcloud SDK, follow the instructions here
3. Next set the application default credentials, this is to update the project/billing context and auth info to use when making Google API calls
4. Finally check if everything is looking good with `gcloud config list`
5. You can set the project if not set correctly by `gcloud config set project <project-name>`

Alright, so first start with creating all the resources that we need, we begin with the infrastructure, go ahead and clone the repo.

```
git clone <repo>
```

```
cd gke-infrastructure
```

Here. create a file with name **terraform.tfvars** and update with your project name

```
project = "<your-project-name>"
```

Initialize the Terraform provider, it will go and pull all that it needs to execute the Google Cloud Provider for Terraform.

```
terraform init
```

Generate the plan, make sure there are no errors and you can see the resources that will be created when this code finishes the execution.

```
terraform plan
```

We will end up with following resources:

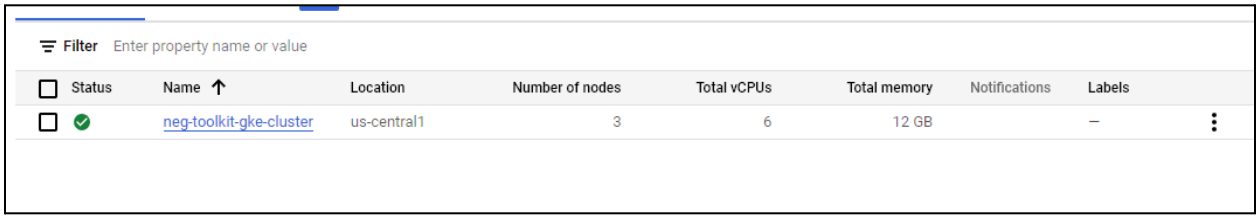
1. A VPC network
2. A VPC subnetwork in us-central1 region
3. A GKE regional cluster, we will pick the zones where nodes will be deployed and the zones will be us-central1-a, us-central1-c and us-central1-f
4. A custom service account will be created and assigned to the nodes in the node pool, we will sway away from the default compute engine account which has broad set of permissions

```
terraform apply -auto-approve
```

Just login to the Google Cloud console and verify that everything is looking good after the terraform application has finished.

The VPC:

The GKE cluster:



Next, let's deploy our application, but since we will be using kubectl, let's authenticate kubectl to our cluster (essentially the kubeconfig file), to do that, run the following gcloud command:

```
gcloud container clusters get-credentials neg-toolkit-gke-cluster --region us-central1
```

Just make sure you can reach the API server and get information:

```
kubectl get nodes -o wide
```

Change to the directory where the kubernetes resource definition is:

```
cd ../kubernetes-resources/
```

Let's look at the YAML file here, the single most important thing to note here the annotation on the Kubernetes service which tells the NEG controller to create network endpoint groups (1 in each zone where the nodes are placed) and then make pod ip and port available in the endpoints groups created.

```
cloud.google.com/neg: '{"exposed_ports": {"80":{}}}'
```

I have left the NEG name as blank and this will lead to auto-generated and unique NEGs. You can of course provide a name of your choice and GKE shall create a NEG with the same name in each zone where your nodes are located.

Apply the manifest:

```
kubectl apply -f gke-neg-demo-service.yaml
```

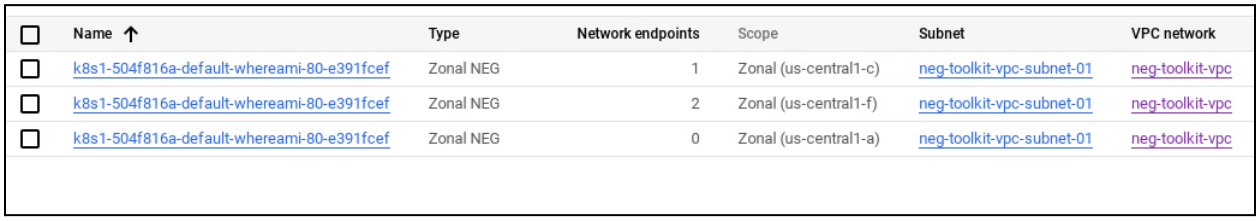
```
deployment.apps/whereami created
service/whereami created
```

Let's check the health of our deployment:

```
kubectl get deployments
```

```
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
whereami  3/3     3            3           3m36s
```

Looking good, so far! Now, let's see the NEGs created by GKE NEG Controller. First validate via the kubectl and gcloud. You can also locate them under Compute Engine -> Network Endpoint Groups on the web console.



Name	Type	Network endpoints	Scope	Subnet	VPC network
k8s1-504f816a-default-whereami-80-e391fcef	Zonal NEG	1	Zonal (us-central1-c)	neg-toolkit-vpc-subnet-01	neg-toolkit-vpc
k8s1-504f816a-default-whereami-80-e391fcef	Zonal NEG	2	Zonal (us-central1-f)	neg-toolkit-vpc-subnet-01	neg-toolkit-vpc
k8s1-504f816a-default-whereami-80-e391fcef	Zonal NEG	0	Zonal (us-central1-a)	neg-toolkit-vpc-subnet-01	neg-toolkit-vpc

```
kubectl get svcneg
```

This should show the NEG with an auto generated name, it will be different for you in your environment if you choose to use auto generated names for the NEG, now let's see the details of the NEG Kubernetes resource (ServiceNetworkEndpointGroup) which was created and then implemented as a Compute Engine Network Endpoint Group by the NEG Controller in GKE.

```
kubectl get svcneg k8s1-504f816a-default-whereami-80-e391fcef -o yaml
```

```
apiVersion: v1
items:
- apiVersion: networking.gke.io/v1beta1
  kind: ServiceNetworkEndpointGroup
  metadata:
    creationTimestamp: "2022-09-03T19:04:02Z"
    finalizers:
    - networking.gke.io/neg-finalizer
    generation: 6
    labels:
      networking.gke.io/managed-by: neg-controller
      networking.gke.io/service-name: whereami
      networking.gke.io/service-port: "80"
    name: k8s1-504f816a-default-whereami-80-e391fcef
    namespace: default
    ownerReferences:
```

```
    - apiVersion: v1
      blockOwnerDeletion: false
      controller: true
      kind: Service
      name: whereami
      uid: 7f6911b1-92e9-4e86-9834-da49774522a1
    resourceVersion: "25633"
    uid: ad815d1c-fa3c-4a21-872b-4dfa6936432b
  spec: {}
  status:
    conditions:
    - lastTransitionTime: "2022-09-03T19:04:24Z"
      message: ""
      reason: NegInitializationSuccessful
      status: "True"
      type: Initialized
    - lastTransitionTime: "2022-09-03T19:04:24Z"
      message: ""
      reason: NegSyncSuccessful
      status: "True"
      type: Synced
    lastSyncTime: "2022-09-03T19:04:28Z"
    networkEndpointGroups:
    - id: "4413616797232536525"
      networkEndpointType: GCE_VM_IP_PORT
      selfLink:
https://www.googleapis.com/compute/v1/projects/rmishra-kubernetes-playground/zones/us-central1-a/networkEndpointGroups/k8s1-504f816a-default-whereami-80-e391fcef
    - id: "4399381806734777285"
      networkEndpointType: GCE_VM_IP_PORT
      selfLink:
https://www.googleapis.com/compute/v1/projects/rmishra-kubernetes-playground/zones/us-central1-c/networkEndpointGroups/k8s1-504f816a-default-whereami-80-e391fcef
    - id: "5368309880203385821"
      networkEndpointType: GCE_VM_IP_PORT
      selfLink:
https://www.googleapis.com/compute/v1/projects/rmishra-kubernetes-playground/zones/us-central1-f/networkEndpointGroups/k8s1-504f816a-default-whereami-80-e391fcef
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

Verify the GCE_VM_IP_PORT type NEGs created by gcloud CLI (your output may look different):

```
gcloud compute network-endpoint-groups list
```

```
NAME: k8s1-504f816a-default-whereami-80-e391fcef
LOCATION: us-central1-a
ENDPOINT_TYPE: GCE_VM_IP_PORT
SIZE: 0

NAME: k8s1-504f816a-default-whereami-80-e391fcef
LOCATION: us-central1-c
ENDPOINT_TYPE: GCE_VM_IP_PORT
SIZE: 1

NAME: k8s1-504f816a-default-whereami-80-e391fcef
LOCATION: us-central1-f
ENDPOINT_TYPE: GCE_VM_IP_PORT
SIZE: 2
```

The SIZE here reflects the pods serving within the NEG, we created 3 replicas in the deployment, as you can see though the NEGs are available in all the zones we need, currently we do not have any pods in the us-central1-a and thus the SIZE is 0, but as we know that pods are ephemeral, they come and go, then we also scale out and scale in the deployment. The NEGs will automatically get updated as the pod IPs are changed or new pods are rolled out or existing pods are removed.

As an example, let's try to scale the deployment manually and increase the replicas to 10:

```
kubectl scale deployment whereami --replicas=10
```

Now you may see pods scheduled on other nodes in the zone where there were none as the scheduler tries to find nodes in the pool which can accommodate these workloads:

```
gcloud compute network-endpoint-groups list
```

```
NAME: k8s1-504f816a-default-whereami-80-e391fcef
LOCATION: us-central1-a
ENDPOINT_TYPE: GCE_VM_IP_PORT
SIZE: 2

NAME: k8s1-504f816a-default-whereami-80-e391fcef
```

```
LOCATION: us-central1-c
ENDPOINT_TYPE: GCE_VM_IP_PORT
SIZE: 4

NAME: k8s1-504f816a-default-whereami-80-e391fcef
LOCATION: us-central1-f
ENDPOINT_TYPE: GCE_VM_IP_PORT
SIZE: 4
```

Alright scale down the deployment to 3 replicas, 10 was just to show how pods automatically get reflected in the NEGs. As we are using a VPC native cluster, the load balancing proxy will directly target these pod IPs, thus optimizing the request paths and reducing the hops.

Now, we need to create the networking infrastructure at the top, that is the load balancer, proxies and backend services (configured with NEGs). We will also create a firewall rule to let the proxied traffic and the health checks reach the pod IPs.

Note: This blog is focused on NEGs created by GKE and how they can be used with a Global Load Balancer on Google Cloud, that's why I have purposefully abstracted away other complex things you do not need, such as primary and secondary ranges and IP allocation to pods and services, the terraform configuration contains the IP ranges used by pods and services.

Note the name of the NEG and network tags on the GKE nodes, we will need them when we later provision the load balancing infrastructure.

```
cd ../global-load-balancing-infrastructure/
```

Note down the network tag on your GCE VMs in the nodepool, this will vary for you, I am just providing the value I see, of course you can add custom network tags as well when creating the GKE cluster:

- gke-neg-toolkit-gke-cluster-1b990638-node
- And the NEG name is k8s1-504f816a-default-whereami-80-e391fcef

Now here under the current directory where all the terraform code for load balancing infrastructure is present, create a file **terraform.tfvars** and update these values:

```
project = "<your project name"
gke_zonal_network_endpoint_group_name = "<NEG name>"
```

```
gke_nodes_target_tags = "<network tag on VMs under your node pool"
```

Go ahead and run the terraform code in this directory.

```
terraform init
```

```
terraform plan
```

```
terraform apply -auto-approve
```

While your load balancing infrastructure is being created, let's pay attention to some specific code blocks in the main.tf under this directory.

The Zonal NEGs, VPC and subnets etc. were created as a part of core GKE infrastructure. While provisioning the global load balancer and it's components, we will need to reference them, such as the firewall rule will need a reference to the VPC, the back end service will need to know the NEGs. We will get them as 'data sources' by querying the infrastructure through API (behind the scenes).

```
data "google_compute_network_endpoint_group" "gke_zonal_network_endpoint_group_zone_a" {
  name = var.gke_zonal_network_endpoint_group_name
  zone = var.zonal_neg_central_a
}

data "google_compute_network_endpoint_group" "gke_zonal_network_endpoint_group_zone_c" {
  name = var.gke_zonal_network_endpoint_group_name
  zone = var.zonal_neg_central_c
}

data "google_compute_network_endpoint_group" "gke_zonal_network_endpoint_group_zone_f" {
  name = var.gke_zonal_network_endpoint_group_name
  zone = var.zonal_neg_central_f
}

data "google_compute_network" "gke_neg_poc_vpc_network" {
  name = var.vpc_network_name
```

```
}
```

The back end service resource will look like this:

```
resource "google_compute_backend_service" "gke_neg_poc_backend_service" {
  name                        = var.gke_neg_backend_name
  enable_cdn                  = false
  timeout_sec                 = 10
  connection_draining_timeout_sec = 10
  health_checks               = [google_compute_health_check.gke_neg_backend_health_check.id]
  backend {
    balancing_mode       = "RATE"
    max_rate_per_endpoint = 100
    group                =
data.google_compute_network_endpoint_group.gke_zonal_network_endpoint_group_zone_a.id
  }
  backend {
    balancing_mode       = "RATE"
    max_rate_per_endpoint = 100
    group                =
data.google_compute_network_endpoint_group.gke_zonal_network_endpoint_group_zone_c.id
  }
  backend {
    balancing_mode       = "RATE"
    max_rate_per_endpoint = 100
    group                =
data.google_compute_network_endpoint_group.gke_zonal_network_endpoint_group_zone_f.id
  }
}
```

Of course you can change the max rate per endpoint to a value that you like, but note that utilization is not supported as a balancing mode with Standalone NEGs.

Also, to demonstrate that this is a load balancer with a set of heterogeneous backends, we will also create a Cloud Storage bucket:

```
resource "google_storage_bucket" "static_content_regional_bucket" {
  name                        = "${var.project}-static"
  location                    = var.region
  uniform_bucket_level_access = true

}
```

Then the URL Map resource will know when to send the request to the GKE backend and when to the Cloud Storage.

```
resource "google_compute_url_map" "global_lb_url_map" {
 name          = var.url_map_name
 default_service = google_compute_backend_bucket.gke_neg_poc_storage_bucket.id
 path_matcher {
  default_service = google_compute_backend_service.gke_neg_poc_backend_service.id
  name = "global-lb-service-path-matcher"
  path_rule {
   paths   = ["/service"]
   service = google_compute_backend_service.gke_neg_poc_backend_service.id
  }
 }
 path_matcher {
  default_service = google_compute_backend_bucket.gke_neg_poc_storage_bucket.id
  name = "global-lb-static-path-matcher"
  path_rule {
   paths   = ["/static"]
   service = google_compute_backend_bucket.gke_neg_poc_storage_bucket.id
  }
 }
 host_rule {
  hosts       = ["api.example.com"]
  path_matcher = "global-lb-service-path-matcher"
 }
 host_rule {
  hosts       = ["static.example.com"]
  path_matcher = "global-lb-static-path-matcher"
 }
}
```

Other parts of this terraform configuration are just routine resources such as health check, firewall rules, static IP address and finally referring to the IP address in the forwarding rule for the load balancer.

Once the code has finished execution, now test it out! Note down the IP address created for your load balancer's forwarding rule, it will be different in your environment.

From the side-bar menu (or hamburger menu!), go to Networking Services > Load Balancer and you will see your load balancer provisioned, click on in and note down the public IP and let us test it.

With the /service prefix (remember our path matching rules!)

```
curl http://<load balancer ip>/service -H "Host: api.example.com"
```

Also, check the static content from the Cloud Storage Bucket. Upload any image that you may have to this bucket and then test it out using the same Global Load Balancer IP.

You can use a Chrome extension called Mod Header to pass the Host header as “static.example.com”. You may also need the bucket to be publicly accessible (readable in this case), if not done already. Read here for details on how to do that. Once you do all of the above, you shall be able to see the image rendered in your browser.

```
http://<load balancer ip>/static/<image-name>.png
```

Clean Up

```
kubectl delete -f ../kubernetes-resources/gke-neg-demo-service.yaml
```

```
terraform destroy -auto-approve
```

```
cd ../gke-infrastructure/
```

```
terraform destroy -auto-approve
```

Summary

Standalone NEGs are really helpful if you are not using Kubernetes Ingress resource as a means to provision and configure your load balancing infrastructure or your GKE services are one of the many backends fronted by the same load balancer. Though they have their own limitations, Standalone NEGs are yet another powerful way to achieve VPC nativer or container native load balancing for your Kubernetes based applications especially when working in a largely heterogeneous environment.