Step 1: Verify the API

Step 2: Login to cloudshell

Step 3: Create a subnet in default VPC

gcloud compute networks subnets create gke-deep-dive-subnet --network=default --range=10.10.0.0/24

Step 4: Verify the created network and it's subnet

No Secondary IP ranges

Step 5: Create VPC native cluster

gcloud container clusters create **gke-deep-dive** --num-nodes=1 --disk-type=pd-standard --disk-size=10 --enable-ip-alias --subnetwork=gke-deep-dive-subnet

Step 6: Verify the secondary IP ranges created

Step 7: Verify the cluster is deployed okay

Step 8: Check the VPC-native traffic routing under networking section - enabled

Step 9: Create necessary files:

Next, we'll deploy a workload to the cluster.

The following sample Deployment, `gke-demo-app`, runs a single instance of a containerized HTTP server.

1. **gke-deep-dive-app.yaml**

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  labels:
    app: gke-deep-dive-app
  name: gke-deep-dive-app
spec:
  selector:
    matchLabels:
      app: gke-deep-dive-app
  template:
    metadata:
      labels:
        app: gke-deep-dive-app
    spec:
      containers:
      - image: gcr.io/google-containers/serve_hostname
        name: gke-deep-dive
        ports:
        - containerPort: 9376
          protocol: TCP
```

2. **gke-deep-dive`-svc.yaml`**

```
apiVersion: v1
kind: Service
metadata:
  name: gke-deep-dive-svc # Name of Service
  annotations:
    cloud.google.com/neg: '{"ingress": true}' # Creates a NEG after an Ingress is created
spec:
  type: ClusterIP
  selector:
    app: gke-deep-dive-app # Selects Pods labelled app: gke-demo-app
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 9376
```

3. **gke-deep-dive`-ing.yaml`**

```
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```
metadata:
  name: gke-deep-dive-ing
spec:
  defaultBackend:
    service:
      name: gke-deep-dive-svc # Name of the Service targeted by the Ingress
      port:
        number: 80 # Should match the port used by the Service
```

Step 10: If using google cloud shell, kube config would have already been updated with the new cluster deployed for you

Check by running : kubectl config view

If not set or using other machine than cloud shell, follow lab 2 to configure kubectl for cluster

Step 11: **Create deployment**

kubectl apply -f gke-deep-dive-app.yaml

Step 12: Create a service

kubectl apply -f gke-deep-dive-svc.yaml

Step 13: Create an ingress

kubectl apply -f gke-deep-dive-ing.yaml

Step 14: Verify the load balancer and network endpoint groups in console

Upon creating the Ingress, an HTTP(S) load balancer is created in the project, and NEGs are created in each zone in which the cluster runs. The endpoints in the NEG and the endpoints of the Service are kept in sync.

Step 15: Verify ingress

kubectl describe ingress gke-demo-ing

Step 16: Verify the container-native load balancer functionality

Wait few minutes for the HTTP(S) load balancer to be configured.

We can verify that the container-native load balancer is functioning by visiting the Ingress' IP address.

To get the Ingress IP address, run the following command:

kubectl get ingress gke-deep-dive-ing

Visit the IP address in a web browser.

Step 17: Verify the ingress functionality

Let's first update our Deployment to scale from one instance to two instances:

```
kubectl scale deployment gke-deep-dive-app --replicas 2
```

To verify that the rollout is complete:

```
kubectl get deployment gke-deep-dive-app
```

verify that there are two available replicas:

Let's check the response from load balancer now by reaching the ingress IP in a new browser. Refresh the page multiple times. We should observe that the number of distinct responses is the same as the number of replicas, indicating that all backend Pods are serving traffic

Step 18: Cleanup

gcloud container clusters delete gke-deep-dive