GKE Workload Optimization

1 hour 30 minutes No cost

GSP769

Overview

One of the many benefits of using Google Cloud is its billing model that bills you for only the resources you use. With that in mind, it's imperative that you not only allocate a reasonable amount of resources for your apps and infrastructure, but that you make the most efficient use of them. With GKE there are a number of tools and strategies available to you that can reduce the use of different resources and services while also improving your application's availability.

This lab will walk through a few concepts that will help increase the resource efficiency and availability of your workloads. By understanding and fine-tuning your cluster's workload, you can better ensure you are only using the resources you need and optimizing your cluster's costs.

Setup and requirements

Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

To complete this lab, you need:

- Access to a standard internet browser (Chrome browser recommended).

> **Note:** Use an Incognito or private browser window to run this lab. This prevents any conflicts between your personal account and the Student account, which may cause extra charges incurred to your personal account.

- Time to complete the lab---remember, once you start, you cannot pause a lab.

> **Note:** If you already have your own personal Google Cloud account or project, do not use it for this lab to avoid extra charges to your account.

How to start your lab and sign in to the Google Cloud console

1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is the **Lab Details** panel with the following:
    - The **Open Google Cloud console** button
    - Time remaining

- The temporary credentials that you must use for this lab
- Other information, if needed, to step through this lab

2. Click **Open Google Cloud console** (or right-click and select **Open Link in Incognito Window** if you are running the Chrome browser).

   The lab spins up resources, and then opens another tab that shows the **Sign in** page.

   Tip: Arrange the tabs in separate windows, side-by-side.

   > **Note:** If you see the **Choose an account** dialog, click **Use Another Account.**

3. If necessary, copy the **Username** below and paste it into the **Sign in** dialog.

   ```
   "Username"
   ```

   content_c

   You can also find the **Username** in the **Lab Details** panel.

4. Click **Next**.

5. Copy the **Password** below and paste it into the **Welcome** dialog.

   ```
   "Password"
   ```

   content_c

   You can also find the **Password** in the **Lab Details** panel.

6. Click **Next**.

   > **Important:** You must use the credentials the lab provides you. Do not use your Google Cloud account credentials.

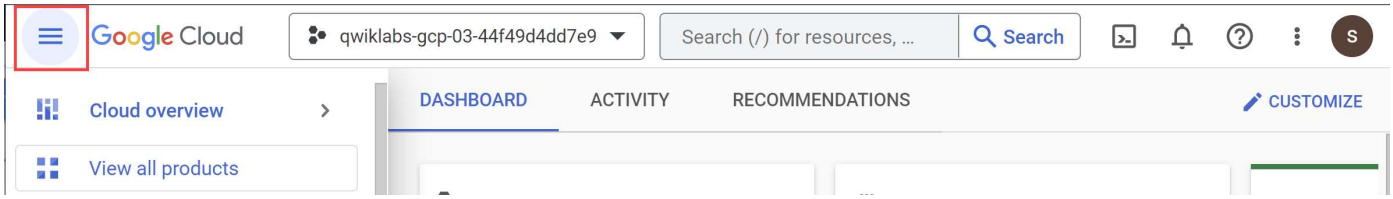
> **Note:** Using your own Google Cloud account for this lab may incur extra charges.

7. Click through the subsequent pages:

    - Accept the terms and conditions.
    - Do not add recovery options or two-factor authentication (because this is a temporary account).
    - Do not sign up for free trials.

After a few moments, the Google Cloud console opens in this tab.

> **Note:** To view a menu with a list of Google Cloud products and services, click the **Navigation menu** at the top-left.



Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

1. Click **Activate Cloud Shell** at the top of the Google Cloud console.

When you are connected, you are already authenticated, and the project is set to your **Project_ID**, `PROJECT_ID`. The output contains a line that declares the **Project_ID** for this session:

```
Your Cloud Platform project in this session is set to "PROJECT_ID"
```

`gcloud` is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

2. (Optional) You can list the active account name with this command:

```
gcloud auth list
```

content_c

3. Click **Authorize**.

Output:

```
ACTIVE: *
ACCOUNT: "ACCOUNT"

To set the active account, run:
    $ gcloud config set account `ACCOUNT`
```

4. (Optional) You can list the project ID with this command:

```
gcloud config list project
```

content_c

Output:

```
[core]
project = "PROJECT_ID"
```

Note: For full documentation of `gcloud`, in Google Cloud, refer to the gcloud CLI overview guide.

Provision lab environment

1. Set your default zone to " ZONE ":

```
gcloud config set compute/zone ZONE
```

content_c

2. Click **Authorize**.

3. Create a three node cluster:

```
gcloud container clusters create test-cluster --num-nodes=3  --enable-ip-
```

content_c

The `--enable-ip-alias` flag is included in order to enable the use of alias IPs for pods which will be required for container-native load balancing through an ingress.

For this lab, you'll use a simple HTTP web app that you will first deploy as a single pod.

4. Create a manifest for the `gb-frontend` pod:

```
cat << EOF > gb_frontend_pod.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: gb-frontend
  name: gb-frontend
spec:
    containers:
    - name: gb-frontend
      image: gcr.io/google-samples/gb-frontend-amd64:v5
      resources:
        requests:
          cpu: 100m
          memory: 256Mi
      ports:
      - containerPort: 80
EOF
```

content_c

5. Apply the newly created manifest to your cluster:

```
kubectl apply -f gb_frontend_pod.yaml
```

content_c

Click **Check my progress** to verify the objective.

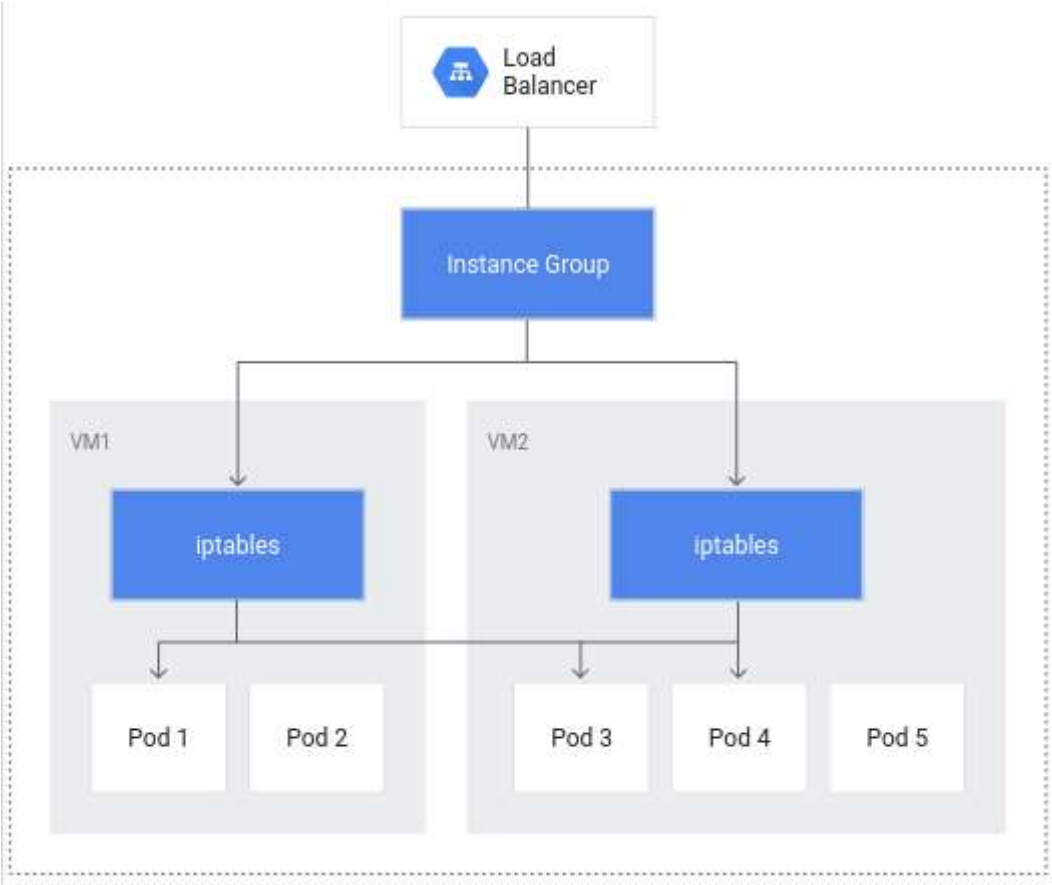
Provision Lab Environment

Check my progress

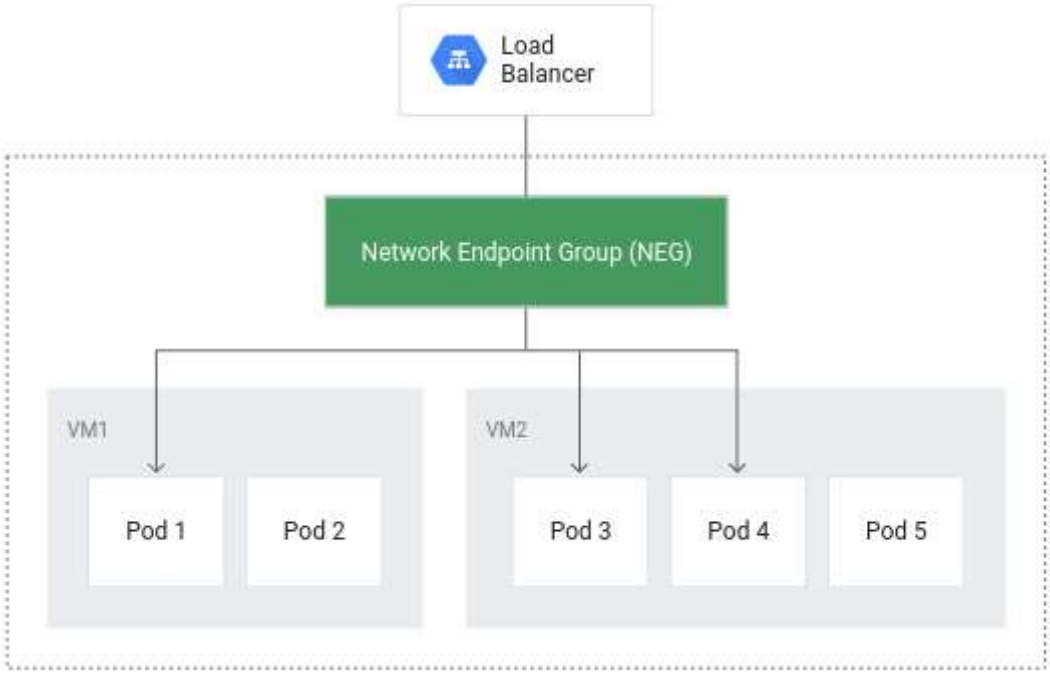
Task 1. Container-native load balancing through ingress

Container-native load balancing allows load balancers to target Kubernetes Pods directly and to evenly distribute traffic to pods.

Without container-native load balancing, load balancer traffic would travel to node instance groups and then be routed via `iptables` rules to pods which may or may not be in the same node:



Container-native load balancing allows pods to become the core objects for load balancing, potentially reducing the number of network hops:



In addition to more efficient routing, container-native load balancing results in substantially reduced network utilization,improved performance, even distribution of traffic across Pods, and application-level health checks.

In order to take advantage of container-native load balancing, the VPC-native setting must be enabled on the cluster. This was indicated when you created the cluster and included the `--enable-ip-alias` flag.

1. The following manifest will configure a `ClusterIP` service that will be used to route traffic to your application pod to allow GKE to create a network endpoint group:

```
cat << EOF > gb_frontend_cluster_ip.yaml
apiVersion: v1
kind: Service
metadata:
  name: gb-frontend-svc
  annotations:
    cloud.google.com/neg: '{"ingress": true}'
spec:
  type: ClusterIP
  selector:
    app: gb-frontend
  ports:
  - port: 80
    protocol: TCP
```

content_c

```
    targetPort: 80
EOF
```

The manifest includes an `annotations` field where the annotation for `cloud.google.com/neg` will enable container-native load balancing on for your application when an ingress is created.

2. Apply the change to your cluster:

```
kubectl apply -f gb_frontend_cluster_ip.yaml
```

content_c

3. Next, create an ingress for your application:

```
cat << EOF > gb_frontend_ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: gb-frontend-ingress
spec:
  defaultBackend:
    service:
      name: gb-frontend-svc
      port:
        number: 80
EOF
```

content_c

4. Apply the change to your cluster:

```
kubectl apply -f gb_frontend_ingress.yaml
```

content_c

When the ingress is created, an HTTP(S) load balancer is created along with an NEG (Network Endpoint Group) in each zone in which the cluster runs. After a few minutes, the ingress will be assigned an external IP.

The load balancer it created has a backend service running in your project that defines how Cloud Load Balancing distributes traffic. This backend service has a health status associated with it.

5. To check the health status of the backend service, first retrieve the name:

```
BACKEND_SERVICE=$(gcloud compute backend-services list | grep NAME | cut
```
content_c

6. Get the health status for the service:

```
gcloud compute backend-services get-health $BACKEND_SERVICE --global
```
content_c

It will take a few minutes before your health check returns a healthy status.

Output will look something like this:

```
---
backend: https://www.googleapis.com/compute/v1/projects/qwiklabs-gcp-00-
27ced9534cde/zones/us-central1-a/networkEndpointGroups/k8s1-95c051f0-default-
gb-frontend-svc-80-9b127192
status:
  healthStatus:
  - healthState: HEALTHY
    instance: https://www.googleapis.com/compute/v1/projects/qwiklabs-gcp-00-
27ced9534cde/zones/us-central1-a/instances/gke-test-cluster-default-pool-
7e74f027-47qp
    ipAddress: 10.8.0.6
    port: 80
  kind: compute#backendServiceGroupHealth
```

Note: These health checks are part of the Google Cloud load balancer and are distinct from the liveness and readiness probes provided by the Kubernetes API which can be used to determine the health of individual pods. The Google Cloud load balancer health checks use special routes outside of your project's VPC to perform health checks and determine the success or failure of a backend.

Once the health state for each instance reports as HEALTHY, you can access the application via its external IP.

7. Retrieve it with:

```
kubectl get ingress gb-frontend-ingress
```
content_c

8. Entering the external IP in a browser window will load the application.

Click **Check my progress** to verify the objective.

Container-native Load Balancing Through Ingress

Check my progress

Task 2. Load testing an application

Understanding your application capacity is an important step to take when choosing resource requests and limits for your application's pods and for deciding the best auto-scaling strategy.

At the start of the lab, you deployed your app as a single pod. By load testing your application running on a single pod with no autoscaling configured, you will learn how many concurrent requests your application can handle, how much CPU and memory it requires, and how it might respond to heavy load.

To load test your pod, you'll use Locust, an open source load-testing framework.

1. Download the Docker image files for Locust in your Cloud Shell:

```
gsutil -m cp -r gs://spls/gsp769/locust-image .
```

content_c

The files in the provided `locust-image` directory include Locust configuration files.

2. Build the Docker image for Locust and store it in your project's container registry:

```
gcloud builds submit \
    --tag gcr.io/${GOOGLE_CLOUD_PROJECT}/locust-tasks:latest locust-image
```

content_c

3. Verify the Docker image is in your project's container registry:

```
gcloud container images list
```

content_c

Expected output:

```
NAME
gcr.io/qwiklabs-gcp-01-343cd312530e/locust-tasks
Only listing images in gcr.io/qwiklabs-gcp-01-343cd312530e.
```

Locust consists of a main and a number of worker machines to generate load.

4. Copy and apply the manifest will create a single-pod deployment for the main and a 5-replica deployment for the workers:

```
gsutil cp gs://spls/gsp769/locust_deploy_v2.yaml .
sed 's/${GOOGLE_CLOUD_PROJECT}/'$GOOGLE_CLOUD_PROJECT'/g' locust_deploy_v
```

content_c

5. To access the Locust UI, retrieve the external IP address of its corresponding LoadBalancer service:

```
kubectl get service locust-main
```

content_c

If your **External IP** value is **<pending>**, wait a minute and rerun the previous command until a valid value is displayed.

6. In a new browser window, navigate to `[EXTERNAL_IP_ADDRESS]:8089` to open the Locust web page:

Click **Check my progress** to verify the objective.

Load Testing an Application

Check my progress

Locust allows you to *swarm* your application with many simultaneous users. You are able to simulate traffic by entering a number of users that are spawned at a certain rate.

7. For this example, to represent a typical load, enter **200** for the number of users to simulate and **20** for the hatch rate.

8. Click **Start swarming**.

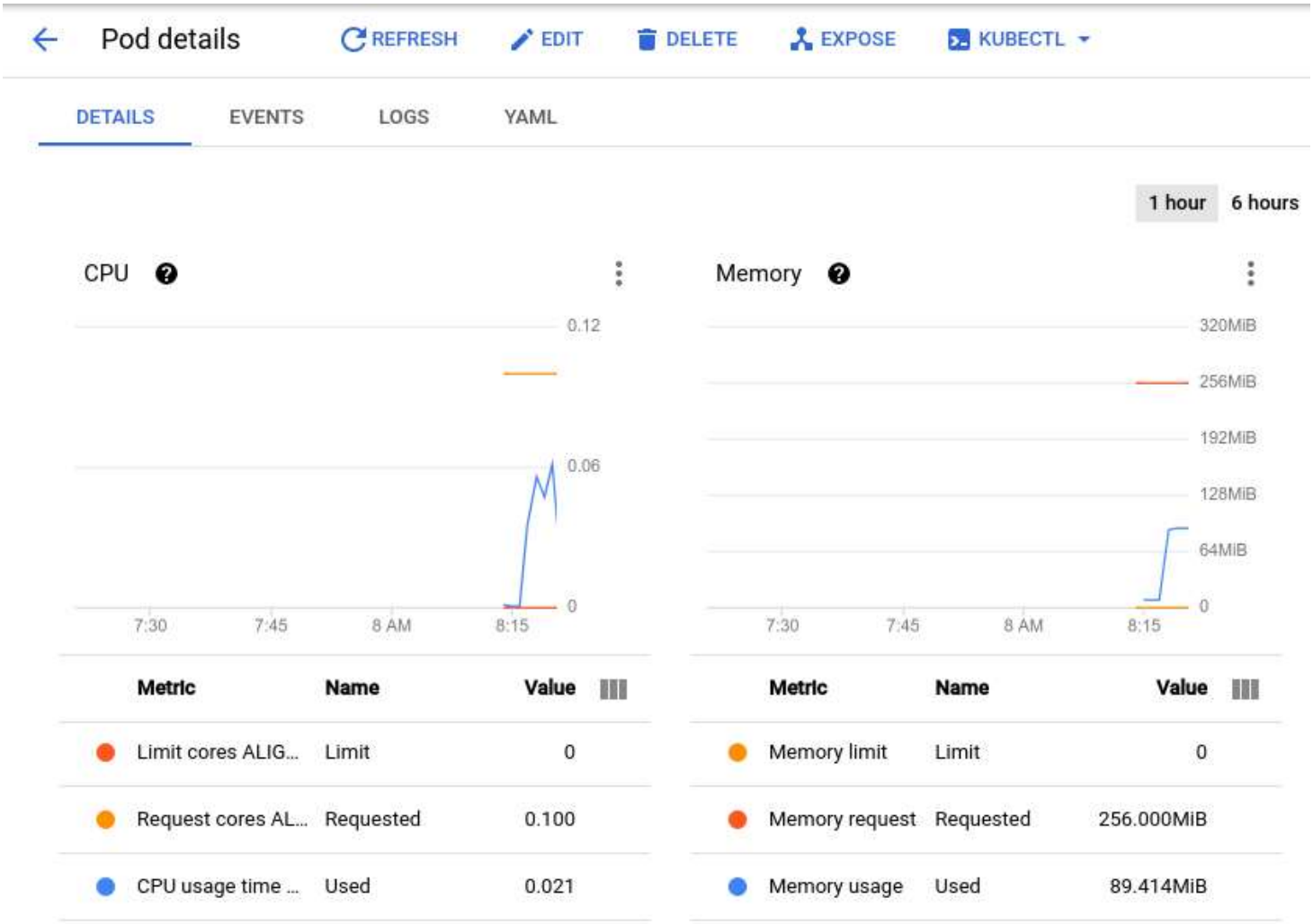
After a few seconds, the **status** should read **Running** with 200 users and about 150 requests per second (RPS).

9. Switch to the Cloud Console and click **Navigation menu (☰) > Kubernetes Engine**.

10. Select **Workloads** from the left pane.

11. Then click on your deployed **gb-frontend** pod.

This will bring you to the pod details page where you can view a graph of the CPU and memory utilization of your pod. Observe the **used** values and the **requested** values.



Note: In order to see the metric values listed below the graph, click the three dots at the top right portion of the graph and select **Expand chart legend** from the dropdown.

With the current load test at about 150 requests per second, you may see the CPU utilization vary from as low as **.04** and as high as **.06.** This represents **40-60%** of your one pod's CPU request. On the other hand, memory utilization stays at around **80Mi**, well below the requested 256Mi. This is your **per-pod capacity**. This information will be useful when configuring your Cluster Autoscaler, resource requests and limits, and choosing how or whether to implement a horizontal or vertical pod autoscaler.

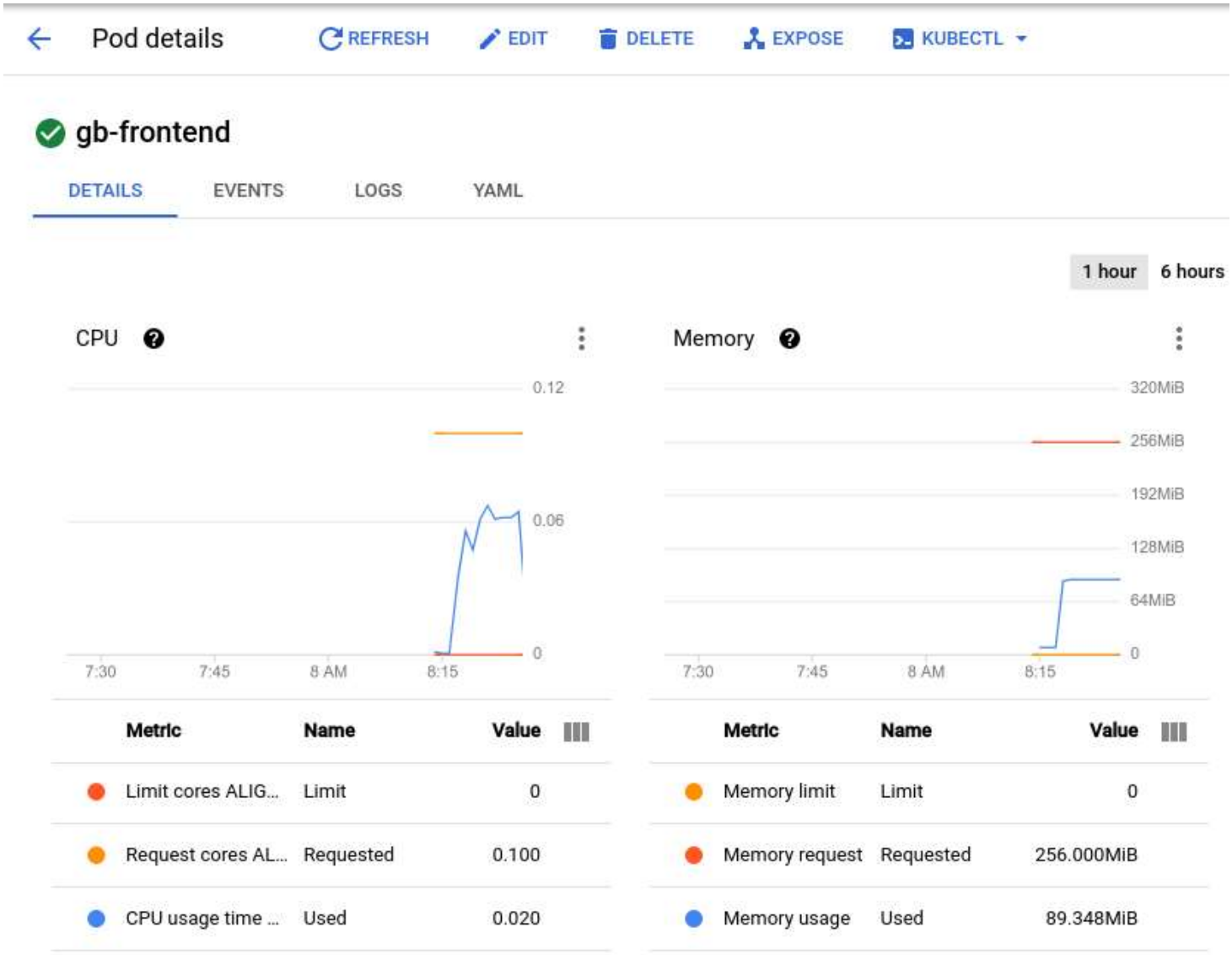
Along with a baseline, you should also take into account how your application may perform after sudden bursts or spikes.

12. Return to the Locust browser window and click **Edit** under the status at the top of the page.

13. This time, enter **900** for the number of users to simulate and **300** for the hatch rate.

14. Click **Start Swarming**.

Your pod will suddenly receive 700 additional requests within 2 - 3 seconds. After the RPS value reaches about 150 and the status indicates 900 users, switch back to the **Pod Details page** and observe the change in the graphs.



While memory stays the same you'll see that CPU peaked at almost .07 - that's 70% of the CPU request for your pod. If this app were a deployment, you could probably safely reduce the total memory request to a lower amount and configure your horizontal autoscaler to trigger on CPU usage.

Task 3. Readiness and liveness probes

Setting up a liveness probe

If configured in the Kubernetes pod or deployment spec, a liveness probe will continuously run to detect whether a container requires a restart and trigger that restart. They are helpful for automatically restarting deadlocked applications that may still be in a running state. For example, a kubernetes-managed load balancer (such as a service) would only send traffic to a pod backend if all of its containers pass a readiness probe.

1. To demonstrate a liveness probe, the following will generate a manifest for a pod that has a liveness probe based on the execution of the cat command on a file created on creation:

```
cat << EOF > liveness-demo.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    demo: liveness-probe
  name: liveness-demo-pod
spec:
  containers:
  - name: liveness-demo-pod
    image: centos
    args:
    - /bin/sh
    - -c
    - touch /tmp/alive; sleep infinity
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/alive
      initialDelaySeconds: 5
      periodSeconds: 10
EOF
```

content_c

2. Apply the manifest to your cluster to create the pod:

```
kubectl apply -f liveness-demo.yaml
```

content_c

The `initialDelaySeconds` value represents how long before the first probe should be performed after the container starts up. The periodSeconds value indicates how often the probe will be performed.

> **Note:** Pods can also be configured to include a `startupProbe` which indicates whether the application within the container is started. If a `startupProbe` is present, no other probes will perform until it returns a `Success` state. This is recommended for applications that may have variable start-up times in order to avoid interruptions from a liveness probe.

In this example the liveness probe is essentially checking if the file /tmp/alive exists on the container's file system.

3. You can verify the health of the pod's container by checking the pod's events:

```
kubectl describe pod liveness-demo-pod
```

content_c

At the bottom of the output there should be an Events section with the pod's last 5 events. At this point, the pod's events should only include the events related to its creation and start up:

```
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  19s   default-scheduler  Successfully assigned
default/liveness-demo-pod to gke-load-test-default-pool-abd43157-rgg0
  Normal  Pulling    18s   kubelet            Pulling image "centos"
  Normal  Pulled     18s   kubelet            Successfully pulled image
"centos"
  Normal  Created    18s   kubelet            Created container liveness-demo-
pod
  Normal  Started    18s   kubelet            Started container liveness-demo-
pod
```

This events log will include any failures in the liveness probe as well as restarts triggered as a result.

4. Manually delete the file being used by the liveness probe:

```
kubectl exec liveness-demo-pod -- rm /tmp/alive
```

content_c

5. With the file removed, the `cat` command being used by the liveness probe should return a non-zero exit code.

6. Once again, check the pod's events:

```
kubectl describe pod liveness-demo-pod
```

content_c

As the liveness probe fails, you'll see events added to the log showing the series of steps that are kicked off. It will begin with the liveness probe failing (`Liveness probe failed: cat: /tmp/alive: No such file or directory`) and end with the container starting up once again (`Started container`):

```
Events:
  Type     Reason     Age                 From               Message
  ----     ------     ----                ----               -------
  Normal   Scheduled  2m21s               default-scheduler  Successfully
assigned default/liveness-demo-pod to gke-load-test-default-pool-abd43157-rgg0
  Warning  Unhealthy  36s (x3 over 56s)   kubelet            Liveness probe
failed: cat: /tmp/alive: No such file or directory
  Normal   Killing    36s                 kubelet            Container
liveness-demo-pod failed liveness probe, will be restarted
  Normal   Pulling    6s (x2 over 2m20s)  kubelet            Pulling image
"centos"
  Normal   Pulled     6s (x2 over 2m20s)  kubelet            Successfully
pulled image "centos"
  Normal   Created    6s (x2 over 2m20s)  kubelet            Created container
liveness-demo-pod
  Normal   Started    6s (x2 over 2m20s)  kubelet            Started container
liveness-demo-pod
```

Note: The example in this lab uses a command probe for its `livenessProbe` that depends on the exit code of a specified command. In addition to a command probe, a `livenessProbe` could be configured as an **HTTP probe** that will depend on HTTP response, or a **TCP probe** that will depend on whether a TCP connection can be made on a specific port.

Setting up a readiness probe

Although a pod could successfully start and be considered healthy by a liveness probe, it's likely that it may not be ready to receive traffic right away. This is common for deployments that serve as a backend to a service such

as a load balancer. A **readiness probe** is used to determine when a pod and its containers are ready to begin receiving traffic.

1. To demonstrate this, create a manifest to create a single pod that will serve as a test web server along with a load balancer:

```
cat << EOF > readiness-demo.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    demo: readiness-probe
  name: readiness-demo-pod
spec:
  containers:
  - name: readiness-demo-pod
    image: nginx
    ports:
    - containerPort: 80
    readinessProbe:
      exec:
        command:
        - cat
        - /tmp/healthz
      initialDelaySeconds: 5
      periodSeconds: 5
---
apiVersion: v1
kind: Service
metadata:
  name: readiness-demo-svc
  labels:
    demo: readiness-probe
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
  selector:
    demo: readiness-probe
EOF
```

content_c

2. Apply the manifest to your cluster and create a load balancer with it:

```
kubectl apply -f readiness-demo.yaml
```

content_c

3. Retrieve the external IP address assigned to your load balancer (it may take a minute after the previous command for an address to be assigned):

```
kubectl get service readiness-demo-svc
```

content_c

4. Enter the IP address in a browser window and you'll notice that you'll get an error message signifying that the site cannot be reached.

5. Check the pod's events:

```
kubectl describe pod readiness-demo-pod
```

content_c

The output will reveal that the readiness probe has failed:

```
Events:
  Type     Reason     Age                   From               Message
  ----     ------     ----                  ----               -------
  Normal   Scheduled  2m24s                 default-scheduler  Successfully
assigned default/readiness-demo-pod to gke-load-test-default-pool-abd43157-rgg0
  Normal   Pulling    2m23s                 kubelet            Pulling image
"nginx"
  Normal   Pulled     2m23s                 kubelet            Successfully
pulled image "nginx"
  Normal   Created    2m23s                 kubelet            Created
container readiness-demo-pod
  Normal   Started    2m23s                 kubelet            Started
container readiness-demo-pod
  Warning  Unhealthy  35s (x21 over 2m15s)  kubelet            Readiness probe
failed: cat: /tmp/healthz: No such file or directory
```

Unlike the liveness probe, an unhealthy readiness probe does not trigger the pod to restart.

6. Use the following command to generate the file that the readiness probe is checking for:

```
kubectl exec readiness-demo-pod -- touch /tmp/healthz
```

content_c

The `Conditions` section of the pod description should now show `True` as the value for `Ready`.

```
kubectl describe pod readiness-demo-pod | grep ^Conditions -A 5
```

Output:

```
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
```

7. Now, refresh the browser tab that had your **readiness-demo-svc** external IP. You should see a "**Welcome to nginx!"** message properly displayed.

Setting meaningful readiness probes for your application containers ensures that pods are only receiving traffic when they are ready to do so. An example of a meaningful readiness probe is checking to see whether a cache your application relies on is loaded at startup.

Click **Check my progress** to verify the objective.

Readiness and Liveness Probes

Check my progress

Task 4. Pod disruption budgets

Part of ensuring reliability and uptime for your GKE application relies on leveraging pod disruption budgets (pdp). `PodDisruptionBudget` is a Kubernetes resource that limits the number of pods of a replicated application that can be down simultaneously due to voluntary disruptions.

Voluntary disruptions include administrative actions like deleting a deployment, updating a deployment's pod template and performing a rolling update, draining nodes that an application's pods reside on, or moving pods to different nodes.

First, you'll have to deploy your application as a deployment.

1. Delete your single pod app:

```
kubectl delete pod gb-frontend
```

content_c

2. And generate a manifest that will create it as a deployment of 5 replicas:

```
cat << EOF > gb_frontend_deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gb-frontend
  labels:
    run: gb-frontend
spec:
  replicas: 5
  selector:
    matchLabels:
      run: gb-frontend
  template:
    metadata:
      labels:
        run: gb-frontend
    spec:
      containers:
        - name: gb-frontend
          image: gcr.io/google-samples/gb-frontend-amd64:v5
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
          ports:
            - containerPort: 80
              protocol: TCP
EOF
```

content_c

3. Apply this deployment to your cluster:

```
kubectl apply -f gb_frontend_deployment.yaml
```

content_c

Click **Check my progress** to verify the objective.

Create Pod Disruption Budgets

Check my progress

Before creating a PDB, you will drain your cluster's nodes and observe your application's behavior without a PDB in place.

4. Drain the nodes by looping through the output of the `default-pool` 's nodes and running the `kubectl drain` command on each individual node:

```
for node in $(kubectl get nodes -l cloud.google.com/gke-nodepool=default-
  kubectl drain --force --ignore-daemonsets --grace-period=10 "$node";
done
```

content_c

The command above will evict pods from the specified node and cordon the node so that no new pods can be created on it. If the available resources allow, pods are redeployed on a different node.

5. Once your node has been drained, check in on your `gb-frontend` deployment's replica count:

```
kubectl describe deployment gb-frontend | grep ^Replicas
```

content_c

The output may resemble something like this:

```
Replicas:               5 desired | 5 updated | 5 total | 0 available | 5
unavailable
```

After draining a node, your deployment could have as little as 0 replicas available, as indicated by the output above. Without any pods available, your application is effectively down. Let's try draining the nodes again, except this time with a pod disruption budget in place for your application.

6. First bring the drained nodes back by uncordoning them. The command below allows pods to be scheduled on the node again:

```
for node in $(kubectl get nodes -l cloud.google.com/gke-nodepool=default-
  kubectl uncordon "$node";
done
```

content_c

7. Once again check in on the status of your deployment:

```
kubectl describe deployment gb-frontend | grep ^Replicas
```

content_c

The output should now resemble the following, with all 5 replicas available:

```
Replicas:               5 desired | 5 updated | 5 total | 5 available | 0
unavailable
```

8. Create a pod disruption budget that will declare the minimum number of available pods to be 4:

```
kubectl create poddisruptionbudget gb-pdb --selector run=gb-frontend --mi
```

content_c

9. Once again, drain one of your cluster's nodes and observe the output:

```
for node in $(kubectl get nodes -l cloud.google.com/gke-nodepool=default-
  kubectl drain --timeout=30s --ignore-daemonsets --grace-period=10 "$nod
done
```

content_c

After successfully evicting one of your application's pods, it will loop through the following:

```
evicting pod default/gb-frontend-597d4d746c-fxsdg
evicting pod default/gb-frontend-597d4d746c-tcrf2
evicting pod default/gb-frontend-597d4d746c-kwvmv
evicting pod default/gb-frontend-597d4d746c-6jdx5
error when evicting pod "gb-frontend-597d4d746c-fxsdg" (will retry after 5s):
Cannot evict pod as it would violate the pod's disruption budget.
error when evicting pod "gb-frontend-597d4d746c-tcrf2" (will retry after 5s):
Cannot evict pod as it would violate the pod's disruption budget.
error when evicting pod "gb-frontend-597d4d746c-6jdx5" (will retry after 5s):
Cannot evict pod as it would violate the pod's disruption budget.
error when evicting pod "gb-frontend-597d4d746c-kwvmv" (will retry after 5s):
Cannot evict pod as it would violate the pod's disruption budget.
```

10. Press **CTRL+C** to exit the command.

11. Check your deployments status once again:

```
kubectl describe deployment gb-frontend | grep ^Replicas
```

content_c

The output should now read:

```
Replicas:               5 desired | 5 updated | 5 total | 4 available | 1
unavailable
```

Until Kubernetes is able to deploy a 5th pod on a different node in order to evict the next one, the remaining pods will remain available in order to adhere to the PDB. In this example, the pod disruption budget was configured to indicate **min-available** but a PDB can also be configured to define a **max-unavailable**. Either value can be expressed as an integer representing a pod count, or a percentage of total pods.

Congratulations!

You learned how you can create a container-native load balancer through ingress in order to take advantage of more efficient load balancing and routing. You ran a simple load test on a GKE application and observed its baseline CPU and memory utilization, as well as how it responds to spikes in traffic. Additionally you configured liveness and readiness probes along with a pod disruption budget to ensure your applications' availability. These tools and techniques in conjunction with each other contribute to an overall efficiency to how your application can run on GKE by minimizing extraneous network traffic, defining meaningful indicators of a well-behaved application and improving application availability.

Next steps / Learn more

- Best practices for running cost-optimized Kubernetes applications on GKE: Prepare Cloud Based Kubernetes Applications

Google Cloud training and certific