Optimize Costs for Google Kubernetes Engine: Challenge Lab

1 hour 30 minutes No cost

GSP343

Introduction

In a challenge lab you're given a scenario and a set of tasks. Instead of following step-by-step instructions, you will use the skills learned from the labs in the quest to figure out how to complete the tasks on your own! An automated scoring system (shown on this page) will provide feedback on whether you have completed your tasks correctly.

When you take a challenge lab, you will not be taught new Google Cloud concepts. You are expected to extend your learned skills, like changing default values and reading and researching error messages to fix your own mistakes.

To score 100% you must successfully complete all tasks within the time period!

This lab is only recommended for students who are enrolled in the Optimize Costs for Google Kubernetes Engine quest. Are you ready for the challenge?

Setup

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.

Challenge scenario

You are the lead Google Kubernetes Engine admin on a team that manages the online shop for **OnlineBoutique**.

You are ready to deploy your team's site to Google Kubernetes Engine but you are still looking for ways to make sure that you're able to keep costs down and performance up.

You will be responsible for deploying the **OnlineBoutique** app to GKE and making some configuration changes that have been recommended for cost optimization.

Here are some guidelines you've been requested to follow when deploying:

- Create the cluster in the `europe-west1-c` zone.
- The naming scheme is team-resource-number, e.g. a cluster could be named `onlineboutique-cluster-300`.
- For your initial cluster, start with machine size `e2-standard-2 (2 vCPU, 8G memory)`.
- Set your cluster to use the **rapid** `release-channel`.

Task 1. Create our cluster and deploy our app

1. Before you can deploy the application, you'll need to create a cluster and name it as `onlineboutique-cluster-300`.

2. Start small and make a zonal cluster with only two (2) nodes.

3. Before you deploy the shop, make sure to set up some namespaces to separate resources on your cluster in accordance with the 2 environments - `dev` and `prod`.

4. After that, deploy the application to the `dev` namespace with the following command:

```
git clone https://github.com/GoogleCloudPlatform/microservices-demo.git &&
cd microservices-demo && kubectl apply -f ./release/kubernetes-manifests.yaml --namespace dev
```

content_c

Note: You can check that your **OnlineBoutique** store is up and running by navigating to the IP address for your **frontend-external** service.

Click *Check my progress* to verify the objective.

Create the cluster `onlineboutique-cluster-300` and deploy an app

Check my progress

Please create a zonal cluster with only two (2) nodes having machine size e2-standard-2.

Task 2. Migrate to an optimized node pool

1. After successfully deploying the app to the dev namespace, take a look at the node details:

CPU requested	CPU allocatable	Memory requested	Memory allocatable
1.29 CPU	1.93 CPU	1.1 GB	5.92 GB
1.38 CPU	1.93 CPU	1.45 GB	5.92 GB

You come to the conclusion that you should make changes to the cluster's node pool:

- There's plenty of left over RAM from the current deployments so you should be able to use a node pool with machines that offer **less RAM**.
- Most of the deployments that you might consider increasing the replica count of will require only 100mcpu per additional pod. You could potentially use a node pool with **less total CPU** if you configure it to use smaller machines. However, you also need to consider how many deployments will need to scale, and how much they need to scale by.

2. Create a new node pool named `optimized-pool-4384` with **custom-2-3584** as the machine type.

3. Set the **number of nodes** to **2**.

4. Once the new node pool is set up, migrate your application's deployments to the new nodepool by **cordoning off and draining** `default-pool`.

5. **Delete** the default-pool once the deployments have safely migrated.

Click *Check my progress* to verify the objective.

Migrate to an optimized node pool

Check my progress

Task 3. Apply a frontend update

You just got it all deployed, and now the dev team wants you to push a last-minute update before the upcoming release! That's ok. You know this can be done without the need to cause down time.

1. Set a pod disruption budget for your **frontend** deployment.

2. Name it **onlineboutique-frontend-pdb**.

3. Set the **min-availability** of your deployment to **1**.

Now, you can apply your team's update. They've changed the file used for the home page's banner and provided you an updated docker image:

```
gcr.io/qwiklabs-resources/onlineboutique-frontend:v2.1
```

content_c

4. **Edit** your **frontend** deployment and change its image to the updated one.

5. While editing your deployment, change the **ImagePullPolicy** to **Always**.

Click *Check my progress* to verify the objective.

Apply a Frontend Update

Check my progress

Task 4. Autoscale from estimated traffic

A marketing campaign is coming up that will cause a traffic surge on the OnlineBoutique shop. Normally, you would spin up extra resources in advance to handle the estimated traffic spike. However, if the traffic spike is larger than anticipated, you may get woken up in the middle of the night to spin up more resources to handle the load.

You also want to avoid running extra resources for any longer than necessary. To both lower costs and save yourself a potential headache, you can configure the Kubernetes deployments to scale automatically when the load begins to spike.

1. Apply **horizontal pod autoscaling** to your **frontend deployment** in order to handle the traffic surge.

2. Scale based on a target cpu percentage of 50.

3. Set the pod scaling between 1 minimum and `12` maximum.

Of course, you want to make sure that users won't experience downtime while the deployment is scaling.

4. To make sure the scaling action occurs without downtime, set the deployment to scale with a target cpu percentage of 50%. This should allow plenty of space to handle the load as the autoscaling occurs.

5. Set the deployment to scale between 1 minimum and `12` maximum pods.

But what if the spike exceeds the compute resources you currently have provisioned? You may need to add additional compute nodes.

6. Next, ensure that your cluster is able to automatically spin up additional compute nodes if necessary. However, handling scaling up isn't the only case you can handle with autoscaling.

7. Thinking ahead, you configure both a minimum number of nodes, and a maximum number of nodes. This way, the cluster can add nodes when traffic is high, and reduce the number of nodes when traffic is low.

8. Update your **cluster autoscaler** to scale between **1 node minimum** and **6 nodes maximum**.

Click *Check my progress* to verify the objective.

Autoscale from estimated traffic

Check my progress

9. Lastly, run a load test to simulate the traffic surge.

Fortunately, **OnlineBoutique** was designed with built-in load generation. Currently, your dev instance is simulating traffic on the store with ~10 concurrent users.

10. In order to better replicate the traffic expected for this event, run the load generation from your `loadgenerator` pod with a higher number of concurrent users with this command. Replace **YOUR_FRONTEND_EXTERNAL_IP** with the IP of the frontend-external service:

```
kubectl exec $(kubectl get pod --namespace=dev | grep 'loadgenerator'
| cut -f1 -d ' ') -it --namespace=dev -- bash -c 'export USERS=8000;
locust --host="http://YOUR_FRONTEND_EXTERNAL_IP" --headless -u "8000"
2>&1'
```

content_c

11. Now, observe your **Workloads** and monitor how your cluster handles the traffic spike.

You should see your `recommendationservice` crashing or, at least, heavily struggling from the increased demand.

12. Apply **horizontal pod autoscaling** to your **recommendationservice** deployment. Scale based off a target cpu percentage of 50 and set the pod scaling between 1 minimum and 5 maximum.

Note: The process of scaling from the load test and having to provision any new nodes will take a couple of minutes altogether.

Task 5. (Optional) Optimize other services

While applying horizontal pod autoscaling to your frontend service keeps your application available during the load test, if you monitor your other workloads, you'll notice that some of them are being pushed heavily for certain resources.

If you still have time left in the lab, inspect some of your other workloads and try to optimize them by applying autoscaling towards the proper resource metric.

You can also see if it would be possible to further optimize your resource utilization with **Node Auto Provisioning**.

Congratulations!