Running Dedicated Game Servers in Google Kubernetes Engine

GSP133

Google Cloud Self-Paced Labs

Packaging server applications as containers is quickly gaining traction across the tech landscape, and game companies are no exception. Many game companies are interested in using containers to improve VM utilization as well as take advantage of their isolated run-time paradigm. Despite high interest, many companies don't know where to start.

This lab will show you how to use an expandable architecture for running a real-time, session-based multiplayer dedicated game server using Kubernetes on Google Kubernetes Engine. A scaling manager process is configured to automatically start and stop virtual machine instances as needed. Configuration of the machines as Kubernetes nodes is handled automatically by managed instance groups. The online game structure presented in this lab is intentionally simple, and places where additional complexity might be useful or necessary are pointed out where appropriate.

Objectives

- Create a container image of a popular open-source dedicated game server (DGS) on Linux using Docker. This container image adds only the binaries and necessary libraries to a base Linux image.
- Store the assets on a separate read-only persistent disk volume and mount them in the container at run-time.
- Setup and configure basic scheduler processes using the Kubernetes and Google Cloud APIs to spin up and down nodes to meet demand.

Before you begin start the download of the game client

In this lab you'll be creating a game server, and to test the server, you need to connect to a game client. The OpenArena game client is available for many operating systems, as long as you can install it on your local machine. If you cannot install a game client on the machine you're currently using, consider taking this lab when you are using a machine you can install a game client on. Validating your work by connecting to a game client is not required, you can choose not to. Taking this lab will still show you how to create a dedicated game server, you just won't be able to confirm it works.

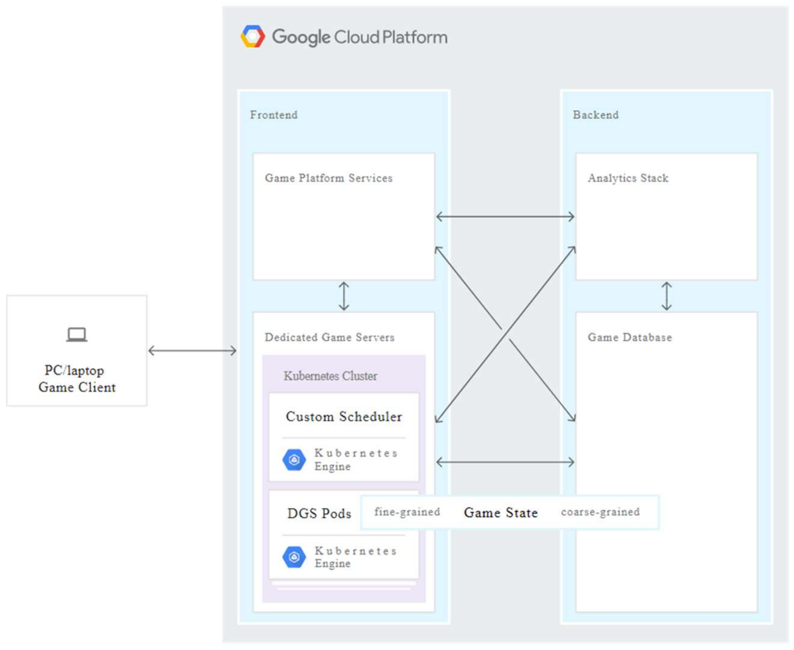
Install the OpenArena game client on your computer to test the connection to the game server at the end of the lab :

```
http://openarena.wikia.com/wiki/Manual/Install
```

This will take a while. Start working on the lab while it downloads, and check back in about 30 min to see if it's finished. Then install the game client.

Architecture overview

The Overview of Cloud Game Infrastructure discusses the high-level components common to many online game architectures. In this lab you implement a Kubernetes DGS cluster frontend service and a scaling manager backend service. A full production game infrastructure would also include many other frontend and backend services which are outside the scope of this solution.



Constructive constraints

In an effort to produce an example that is both instructive and simple enough to extend, this lab assumes the following game constraints:

- This is a match-based real-time game with an authoritative DGS that simulates the game State.
- The DGS communicates over UDP.
- Each DGS process runs one match.
- All DGS processes generates approximately the same amount of load.
- Matches have a maximum length.
- DGS startup time is negligible, and 'pre-warming' of the dedicated game server process isn't necessary.
- Customer experience impact should be avoided as much as possible:
- Once peak time has passed, matches should not be ended prematurely in order to scale down the number of VM instances.
- However, if the dedicated game server process encounters an issue and cannot continue, the match state is lost and the user is expected to join a new match.
- The DGS process loads static assets from disk but does not require write access to the assets.
  These constraints all have precedent within the game industry, and represent a real-world use case.

Note: Many of the commands in this lab require you to substitute your own values for those listed in the code. These substituted variables are capitalized and surrounded by angle brackets <>, such as <PROJECT_ID>.

Setup and Requirements

Qwiklabs 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 Qwiklabs 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.

What you need

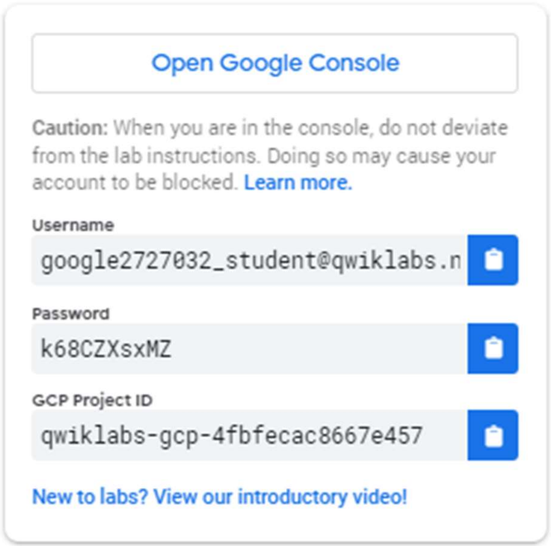
To complete this lab, you need:

- Access to a standard internet browser (Chrome browser recommended).
- Time to complete the lab.
  Note: If you already have your own personal Google Cloud account or project, do not use it for this lab.

Note: If you are using a Pixelbook, open an Incognito window to run this lab.

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 a panel populated with the temporary credentials that you must use for this lab.



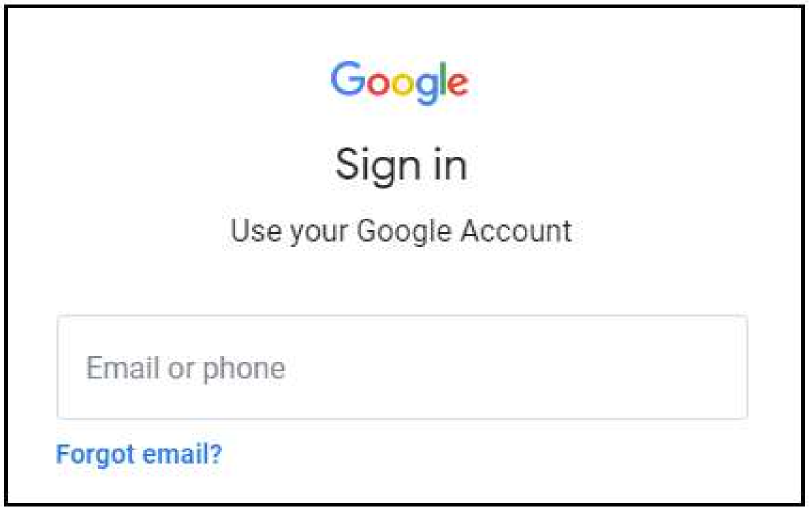
Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked. Learn more.

Username
google2727032_student@qwiklabs.n

Password
k68CZXsxMZ

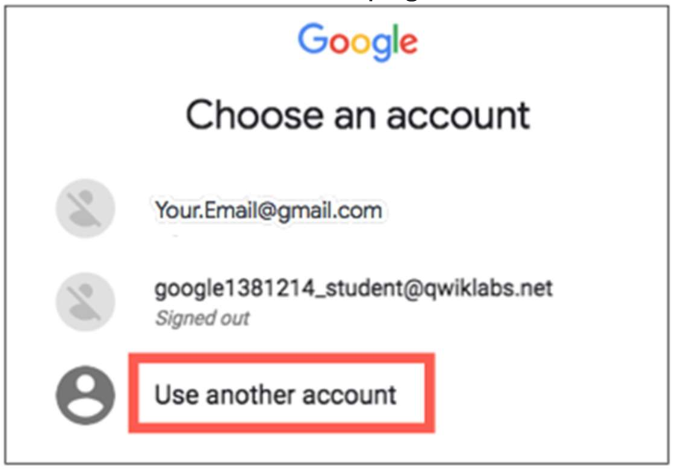
GCP Project ID
qwiklabs-gcp-4fbfecac8667e457

2. Copy the username, and then click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Sign in** page.



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

If you see the **Choose an account** page, click **Use Another Account**.



3. In the **Sign in** page, paste the username that you copied from the Connection Details panel. Then copy and paste the password.

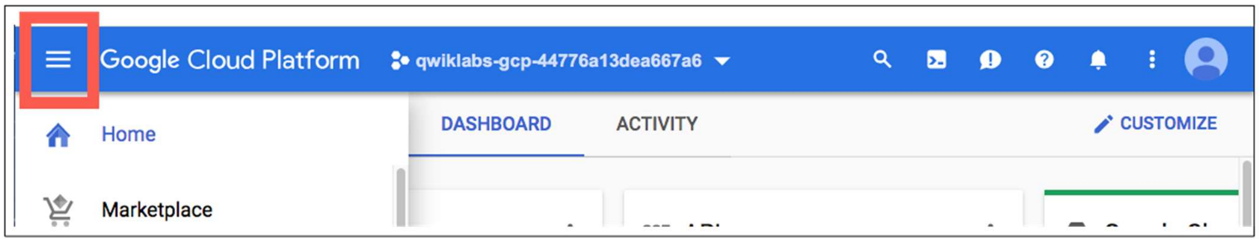
   Important: You must use the credentials from the Connection Details panel. Do not use your Qwiklabs credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).

4. 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 Cloud Console opens in this tab.

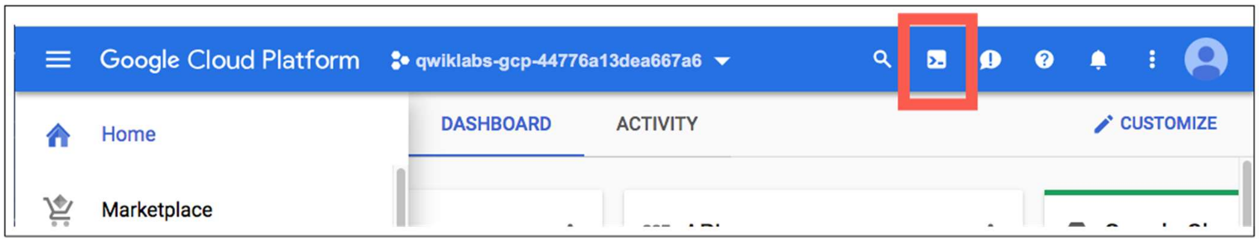
Note: You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-
left.



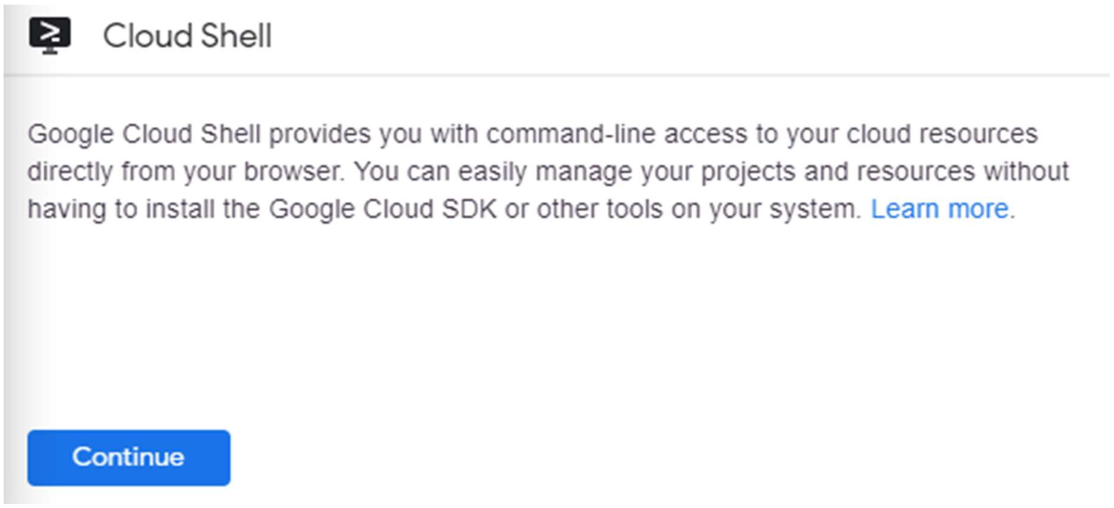
Activate Cloud Shell

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.

In the Cloud Console, in the top right toolbar, click the **Activate Cloud Shell** button.

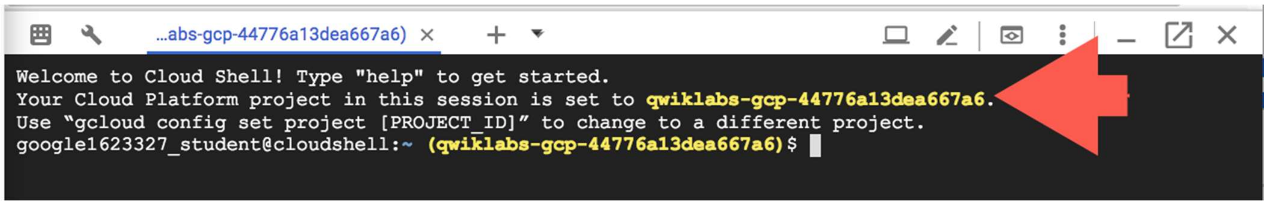


Click **Continue**.


Cloud Shell

Google Cloud Shell provides you with command-line access to your cloud resources directly from your browser. You can easily manage your projects and resources without having to install the Google Cloud SDK or other tools on your system. Learn more.

It takes a few moments to provision and connect to the environment. When you are connected, you are already authenticated, and the project is set to your *PROJECT_ID*. For example:



```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to qwiklabs-gcp-44776a13dea667a6.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
google1623327_student@cloudshell:~ (qwiklabs-gcp-44776a13dea667a6)$
```

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

You can list the active account name with this command:

```
gcloud auth list
```

(Output)

```
Credentialed accounts:
 - <myaccount>@<mydomain>.com (active)
```

(Example output)

```
Credentialed accounts:
 - google1623327_student@qwiklabs.net
```

You can list the project ID with this command:

```
gcloud config list project
```

(Output)

```
[core]
project = <project_ID>
```

(Example output)

```
[core]
project = qwiklabs-gcp-44776a13dea667a6
```

For full documentation of `gcloud` see the gcloud command-line tool overview.

Containerizing the dedicated game server

In this lab, you'll use OpenArena, a "community-produced deathmatch FPS based on GPL idTech3 technology". Although this game's technology is over fifteen years old, it's still a good example of a common DGS pattern:

- A server binary is compiled from the same code base as the game client.
- The only data assets included in the server binary are those necessary for the server to run the simulation.
- The game server container image only adds the binaries and libraries necessary to run the server process to the base OS container image.
- Assets are mounted from a separate volume.
  This architecture has many benefits: it speeds image distribution, reduces the update load as only binaries are replaced, and consumes less disk space.

Creating a dedicated game server binaries container image

For this lab, you cannot use Cloud Shell. You must install the Google Cloud SDK in a client and prepare the environment so that you can create and work with docker images.

Create a VM instance to carry out lab tasks

In the Cloud Console Click **Compute Engine** > **VM Instances**.

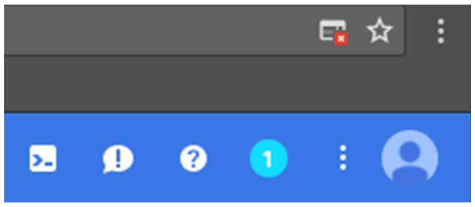
Click **Create**.

In the **Identity and API access** section select **Allow full access to all Cloud APIs**.

Click **Create**.

After your instance has deployed, click the **SSH** button. The remaining tasks for this lab will be carried out in the SSH console of this VM.

If the SSH console does not automatically open check the upper right of your browser window for a pop-up alert icon.



Click the pop-up alert icon, select **Always allow pop-ups from https://console.cloud.google.com** and then click **Done.**

Click the **SSH** button again to open the console.
Click *Check my progress* to verify the objective.

Install kubectl and docker on your VM

Update the apt-get repositories in the VM:

```
sudo apt-get update
```

Install gcloud tools for Kubernetes:

```
sudo apt-get -y install kubectl
```

Install dependencies for docker:

```
sudo apt-get -y install \
     apt-transport-https \
     ca-certificates \
     curl \
     gnupg2 \
     software-properties-common
```

Install Docker's official GPG key:

```
curl -fsSL https://download.docker.com/linux/$(. /etc/os-release; echo "$ID")/gpg |
sudo apt-key add -
```

Check that the GPG fingerprint of the Docker key you downloaded matches the expected value:

```
sudo apt-key fingerprint 0EBFCD88
```

You should see the following fingerprint in the output

```
pub   4096R/0EBFCD88 2017-02-22
      Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
uid                  Docker Release (CE deb) <docker@docker.com>
sub   4096R/F273FCD8 2017-02-22
```

Add the stable Docker repository:

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/$(.
/etc/os-release; echo "$ID")    $(lsb_release -cs) stable"
```

Update the apt-get repositories:

```
sudo apt-get update
```

Install Docker-ce:

```
sudo apt-get -y install docker-ce
```

Configure Docker to run as a non-root user:

```
sudo usermod -aG docker $USER
```

Exit from the SSH session then reconnect by clicking the **SSH** button for your VM instance.

You have to disconnect and reconnect the SSH session because the group permission change does not take effect until the user logs in again.
Confirm that you can run Docker as a non-root user:

```
docker run hello-world
```

You should see the following message in the output.

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Download the Sample Game Server Demo

Clone the demo example:

```
gsutil -m cp -r gs://spls/gsp133/gke-dedicated-game-server .
```

Creating a dedicated game server binaries container image

Prepare the working environment

Select the `gcr.io` region nearest to your Kubernetes Engine cluster (for example, `us` for the United States, `eu` for Europe, or `asia` for Asia, as noted in the documentation). Substitute your <PROJECT_ID> and your **gcr.io** region for <GCR_REGION> in this command:

```
export GCR_REGION=<REGION> PROJECT_ID=<PROJECT_ID>
printf "$GCR_REGION \n$PROJECT_ID\n"
```

The updated command will look something like:

```
export GCR_REGION=us PROJECT_ID=qwiklabs-gcp-e5xxxxxxx4c61da8
```

Generate a new container image

You first create a Dockerfile describing the image to be built. This lab's Debian-based Dockerfile is in the repository at `openarena/Dockerfile`.
Change to the docker directory for the demo:

```
cd gke-dedicated-game-server/openarena
```

Run this Docker build command to generate the container image and tag it. For ease of use with Google Kubernetes Engine, we're using Google Container Registry (**gcr.io**).

```
docker build -t \
${GCR_REGION}.gcr.io/${PROJECT_ID}/openarena:0.8.8 .
```

You may see some warnings and errors during the build but if the command completes with two statements starting with `Successfully built` and `Successfully tagged` then you can proceed.

Upload the container image to an image repository:

```
gcloud docker -- push \
  ${GCR_REGION}.gcr.io/${PROJECT_ID}/openarena:0.8.8
```

Click *Check my progress* to verify the objective.

Generate an assets disk

In most games binaries are orders of magnitude smaller than assets. Because of this fact, it makes sense to create a container image that only contains binaries; assets can be put on a persistent disk and attached to multiple VM instances that run the DGS container. This architecture saves money and eliminates the need to distribute assets to all VM instances.
Create an OpenArena asset disk by following these steps:

Set the default region and store a zone ID in an environment variable:

```
region=us-east1
zone_1=${region}-b
gcloud config set compute/region ${region}
```

Create a small Compute Engine VM instance using `gcloud`.

```
gcloud compute instances create openarena-asset-builder \
   --machine-type f1-micro \
   --image-family debian-9 \
   --image-project debian-cloud \
   --zone ${zone_1}
```

Create and attach an appropriately-sized persistent disk. The persistent disk must be separate from the boot disk, and should be configured to remain undeleted when the virtual machine is removed. Kubernetes persistentVolume functionality works best with persistent disks initialized according to the Compute Engine documentation consisting of a single `ext4` file system without a partition table.

```
gcloud compute disks create openarena-assets \
   --size=50GB --type=pd-ssd\
   --description="OpenArena data disk. Mount read-only at
/usr/share/games/openarena/baseoa/" \
   --zone ${zone_1}
```

Note: Some persistent disk types have more performance when provisioned at larger disk sizes. Please consult the documentation when choosing your disk size. Depending on how read-heavy your engine is, it might make sense to choose a disk larger than needed to hold the data in order to have additional performance.
Wait until the `openarena-asset-builder` instance has fully started up, then attach the persistent disk.

```
gcloud compute instances attach-disk openarena-asset-builder \
   --disk openarena-assets --zone ${zone_1}
```

Once attached, you can SSH into the `openarena-asset-builder` VM instance and format the new persistent disk.

Click *Check my progress* to verify the objective.

Connect to the Asset Builder VM Instance using SSH

In the Console, click **Compute Engine** > **VM Instances**.

Click the **SSH** button next to the `openarena-asset-builder` instance.

This will open a new SSH console that you will use to prepare the persistent volume.

Format and Configure the Assets Disk

New disks are unformatted. You must format and mount a disk before it can be used. First, attach the `openarena-assets` persistent disk to the `openarena-asset-builder` VM instance.

Because the `mkfs.ext4` command in the next step is a destructive command, confirm the device ID for the `openarena-assets` disk. If you're following this lab from the beginning, the ID is `/dev/sdb`.

Verify this using the `lsblk` command to look at the attached disks and their partitions:

```
sudo lsblk
```

The output should show the 10 GB OS disk `sda` with 1 partition `sda1` and the 50 GB `openarena-assets` disk with no partition as device `sdb`:

```
NAME MAJ:MIN  RM SIZE RO TYPE MOUNTPOINT
sda      8:0   0  10G  0 disk
└─sda1   8:1   0  10G  0 part /
sdb      8:16  0  50G  0 disk
```

Note: If you only see `sda` and `sda1` in this list, check that you are running these commands in the SSH console for the `openarena-assets-builder` VM instance.
Format the `openarena-assets` disk:

```
sudo mkfs.ext4 -m 0 -F -E lazy_itable_init=0,lazy_journal_init=0,discard /dev/sdb
```

Install OpenArena on the `openarena-asset-builder` VM instance and copy the compressed asset archives to the `openarena-assets` persistent disk. For this game, the assets are `.pk3` files, located in the `/usr/share/games/openarena/baseoa/` directory.

To save some work, mount the assets disk to this directory before installing, so all the `.pk3` files get put on the disk by the install process:

```
sudo mkdir -p /usr/share/games/openarena/baseoa/

sudo mount -o discard,defaults /dev/sdb \
    /usr/share/games/openarena/baseoa/
sudo apt-get update
sudo apt-get -y install openarena-server

sudo gsutil cp gs://qwiklabs-assets/single-match.cfg
/usr/share/games/openarena/baseoa/single-match.cfg
```

Once the installation has been completed unmount the persistent volume and shut down the instance:

```
sudo umount -f -l /usr/share/games/openarena/baseoa/
sudo shutdown -h now
```

The SSH console will now stop responding, this is expected.

The persistent disk is ready to be used as a `persistentVolume` in Kubernetes and the instance can be safely deleted.

Return to your main lab VM instance SSH console and verify value of zone_1 variable which you set earlier

```
echo $zone_1
```

Expected output:

```
us-east1-b
```

If it is not set, i.e. it returns none, you can set it via following command:

```
region=us-east1
zone_1=${region}-b
```

Now, delete the openarena-asset-builder VM.

```
gcloud compute instances delete openarena-asset-builder --zone ${zone_1}
```

Enter `Y` when prompted to confirm the deletion.

When implementing persistent disks as part of a game development pipeline, configure your build system to create the persistent disk with all the asset files in an appropriate directory structure. This may take the form of a simple script running gcloud commands, or a Google Cloud-specific plugin for your build system of choice. It's also recommended that you create multiple copies of the persistent disk and have VM instances connected to these copies in a balanced manner both for throughput considerations and to manage failure risk.

Setting up a Kubernetes cluster

Creating a Kubernetes cluster on Container Engine

For this lab you're using a standard Container Engine cluster with the n1-standard machine type, which fine for demonstration purposes. In a production environment, the n1-highcpu machine types are more appropriate for the usage profile of OpenArena. The number of vCPUs you'll run on each machine is largely influenced by two factors:

- The largest number of concurrent DGS pods you plan to run:
- There is a limit on the number of nodes that can be in a Kubernetes cluster pool (although the Kubernetes project plans to increase this with future releases). For example, If you run one DGS per Virtual CPU (vCPU), a 1,000 node cluster of n1-highcpu-2 machines provides capacity for only 2,000 DGS pods, while a 1,000 node cluster of n1-highcpu-32 machines allows up to 32,000.
- Your VM instance granularity:
- The simplest way for resources to be added or removed from the cluster is in increments of a single VM instance of the type chosen during cluster creation.
  For this lab the number of vCPUs is limited to 2 per node as that is sufficient.

In **Cloud Shell** create a network and firewall rules for the gaming cluster.

```
gcloud compute networks create game
gcloud compute firewall-rules create openarena-dgs --network game \
    --allow udp:27961-28061
```

Use SSH console for the main **VM instance** which you created earlier to run next commands.

The following `gcloud create` command creates a three-node cluster (`--numnodes 3`) with four virtual CPU cores on each (`--machine-type=n1-standard-2`):

```
gcloud container clusters create openarena-cluster \
   --num-nodes 3 \
   --network game \
   --machine-type=n1-standard-2 \
   --zone=${zone_1}
```

It may take up to 10 minutes for the cluster to start up.

Click *Check my progress* to verify the objective.

Creating a Kubernetes cluster on Container Engine

Check my progress

Once the cluster has started, set up your local shell with the proper Kubernetes authentication credentials to control your new cluster:

```
gcloud container clusters get-credentials openarena-cluster --zone ${zone_1}
```

Although the managed instance groups feature used by Container Engine offers VM instance autoscaling to increase and decrease the number of nodes in the pool based on usage, the previous command disabled this feature using the `--disable-addons=HttpLoadBalancing,HorizontalPodAutoscaling` flag.

Many game developers find it useful to implement a custom scaling manager process which is DGS aware to deal with the specific requirements of this type of workload. The managed instance group does serve an important function: its default Container Engine image template includes all the necessary Kubernetes software and automatically registers the node with the master on startup.
In addition, you disabled HTTP Load Balancing with the `--disable-addons` flag. The pods running in this cluster don't require it.

Configuring the assets disk in Kubernetes

The typical DGS does not need write access to the game assets, so you can have each DGS pod mount the same persistent disk containing read-only assets. To accomplish this, first create the assets disk as described in the Generate an assets disk section, which you've already done. Next, create a `PersistentVolume` in Kubernetes.

In order to include this volume in a Kubernetes DGS pod, you need a PersistentVolumeClaim resource. `YAML` files for these resources are found in the `openarena/k8s/` directory of the solution repository. Once applied, the `PersistentVolume` detects the asset disk, and the `persistentVolumeClaim` allows a DGS pod to mount the disk as read-only.
Run the following commands to create the `PersistentVolume` and the `PersistentVolumeClaim`:

```
kubectl apply -f k8s/asset-volume.yaml
kubectl apply -f k8s/asset-volumeclaim.yaml
```

Note: If you experience connection issues running the `kubectl` commands, make sure you have kubectl set up correctly, and that you've acquired the proper credentials for the new `openarena-cluster`.
If these commands produce an error saying that the path does not exist, double check the directory you are running the commands from. These commands must be run from the `~/gke-dedicated-game-server/openarena` directory.
You can confirm both volume and claim are in `Bound` status by running the following commands and comparing the output:

```
kubectl get persistentVolume
```

Expected output:

```
NAME         CAPACITY ACCESSMODES RECLAIMPOLICY STATUS CLAIM    STORAGECLASS      REASON
AGE
asset.volume 50Gi     ROX         Retain        Bound  default /asset.disk.claim assets
3m
```

Then run:

```
kubectl get persistentVolumeClaim
```

Expected output:

```
NAME             STATUS VOLUME       CAPACITY ACCESSMODES STORAGECLASS AGE
asset.disk.claim Bound  asset.volume 50Gi     ROX         assets 2     5s
```

Click *Check my progress* to verify the objective.

Configuring the DGS pod

Since scheduling and networking tasks are handled by Kubernetes, and the startup and shutdown time of the DGS containers are relatively negligible, this solution has DGS instances spin up on-demand.

Each DGS instance only lasts the length of a single game match. In addition, it's implied that each game match has a defined time limit, specified in the OpenArena DGS server configuration file. Once the match is complete, the container successfully exits and if players want to play again they simply request another game.These assumptions simplify a number of pod lifecycle aspects and form the basis of an autoscaling policy discussed in the scaling manager section.

Although this flow isn't seamless with OpenArena, it's only because the solution doesn't fork and change the game client code. In a commercially released game, requesting another match would be made invisible to the user behind previous match result screens and loading times. The code requiring the server change doesn't represent additional development time as such code is mandatory for handling client reconnections during unforeseen circumstances such as network issues or crashing servers.

For the sake of simplicity, this solution for this lab assumes that the Container Engine nodes have the default network configuration, which assigns each a public IP and allows clients connections.

Managing the dedicated game server process

In commercially produced game servers, all the additional, non-DGS functionality that makes DGSes run as well as containers should be integrated directly into the DGS binaries whenever possible.

As a best practice, the DGS should avoid communicating directly with the matchmaker or scaling manager whenever possible, and instead should expose its state to the Kubernetes API. External processes should read the DGS state from the appropriate Kubernetes endpoints rather than querying the server directly. More information on accessing the Kubernetes API directly can be found here.

Kubernetes resource definitions

At first glance, a single process running in a container with a constrained lifetime and defined success criteria would appear to be a use case for Kubernetes jobs, but in practice it's unnecessary. DGS pods require neither the parallel execution functionality of jobs, nor the ability to guarantee successes by automatically restarting (typically when a session-based DGS dies for some reason, the state is lost, and players simply join another DGS). Due to these factors, scheduling individual Kubernetes pods is preferable for this use case. In production, DGS pods should be started directly by your matchmaker using the Kubernetes API. For the purposes of this lab, a human-readable `YAML` file describing the DGS pod resource is included in the solution repository at `openarena/k8s/oarena_pod.yaml`. When creating configurations for dedicated game server pods, pay close attention to the volume properties to ensure that the asset disk can be mounted as read-only in multiple pods.

Setting up the scaling manager

Note: These commands are run from the repository's `scaling_manager/` directory but assume that you start in the repositories `openarena/` directory.
The scaling manager is a simple process that scales the number of virtual machines used as Container Engine nodes based on the current DGS load. In practice, scaling is accomplished using a set of simple scripts that run forever, inspect the total number of DGS pods running and requested, and resize the node pool as necessary. The scripts are packaged in docker container images with the appropriate libraries and Cloud SDK.
The docker files creating these images are in the lab's repository in the `scaling_manager/` directory, and the docker image can be created and pushed to **gcr.io** using the build and push shell script once a number of environment variables are configured.

Configure the neccessary environment variables, replacing **[GCR_REGION]** with your GCR region ("us", "eu", etc) and **[PROJECT_ID]** with your Qwiklabs Project ID.

```
export GCR_REGION=[GCR_REGION]

export PROJECT_ID=[PROJECT_ID]
```

Build and push the configuration using the following command:

```
cd ../scaling-manager
chmod +x build-and-push.sh
source ./build-and-push.sh
```

These scripts are designed to be run within a kubernetes deployment , ensuring these processes are restarted in the event of a failure.
Click *Check my progress* to verify the objective.

Configure the Openarena Scaling Manager Deployment File

An example kubernetes deployment YAML file is provided in the sample solution in the `scaling_manager/k8s/openarena-scaling-manager-deployment.yaml` directory. This file has to be customized so that the container environment variables it configures use values that describe your lab's OpenArena cluster.

Find the name of the base instance:

```
gcloud compute instance-groups managed list
```

Copy the base instance name. It will be in the fourth column of the output which will look similar to this:

```
BASE_INSTANCE_NAME
gke-openarena-cluster-default-pool-1990bcc7
```

In this next series of commands, replace all of the items in square brackets, including the square brackets themselves, with the appropriate values from your lab. For example, you might replace [ZONE] with "us-east1-b". [BASE_INSTANCE_ID] will vary for each lab instance:

```
export GKE_BASE_INSTANCE_NAME=[BASE_INSTANCE_NAME]

export GCP_ZONE=[ZONE]

printf "$GCR_REGION \n$PROJECT_ID \n$GKE_BASE_INSTANCE_NAME \n$GCP_ZONE \n"
```

Now run the following to apply the variables you set to the `YAML` deployment template:

```
sed -i "s/\[GCR_REGION\]/$GCR_REGION/g" k8s/openarena-scaling-manager-deployment.yaml

sed -i "s/\[PROJECT_ID\]/$PROJECT_ID/g" k8s/openarena-scaling-manager-deployment.yaml

sed -i "s/\[ZONE\]/$GCP_ZONE/g" k8s/openarena-scaling-manager-deployment.yaml

sed -i "s/\gke-openarena-cluster-default-pool-\[REPLACE_ME\]/$GKE_BASE_INSTANCE_NAME/g"
k8s/openarena-scaling-manager-deployment.yaml
```

These commands update the placeholders in the file for the environment variable values listed below with the correct values.

Replacement text	Value	Notes
`[GCR_REGION]`	**gcr.io** region such as us, asia, etc.	Requires replacement. The region of your **gcr.io** repository.
`[PROJECT_ID]`	Qwiklabs Project ID, you can copy this from your lab launch page.	Requires replacement. The name of your project.
`[REPLACE_ME]`	This is the value you identified in the previous step. You just replace the 8 digit unique id at the end. `gke-openarena-cluster-default-pool-`**`[REPLACE_ME]`**	Requires replacement. Different for every ContainerEngine cluster. The value can be obtained from the output of the `gcloud compute instance-groups managed list` command.
`[ZONE]`	Set this to the Google Cloud Zone you selected at that start of the lab. This is usually us-east1-b.	The name of the Google Cloud zone specified in the `gcloud container clusters create` command.

The following environment variable values should remain at their defaults.

`K8S_CLUSTER`	`openarena-cluster`	The name of the Kubernetes cluster.

You can now add the deployment to your Kubernetes cluster by running:

```
kubectl apply -f k8s/openarena-scaling-manager-deployment.yaml
```

Monitor the deployment using `kubectl`.

```
kubectl get pods
```

Wait until you see this report 3/3 nodes ready.

Click *Check my progress* to verify the objective.

Scaling nodes

Concerns

To scale nodes, the scaling manager uses the Kubernetes API to look at current node usage and, as needed, resizes the Container Engine cluster's managed instance group running the underlying virtual machines. Common sticking points for DGS scaling include:

- Standard CPU and memory usage metrics often fail to capture enough information to drive game server instance scaling.
- Keeping a buffer of available, underutilized nodes is critical, as scheduling an optimized DGS container on an existing node takes a matter of seconds, whereas adding a new node can take minutes - an unacceptable latency for a waiting player.
- Many autoscalers aren't able to handle graceful pod shutdowns. It's important to drain pods from nodes being removed due to scaling. Shutting off a node with even one running match is often unacceptable.

Although the scripts supplied by this solution are basic, their simple design makes it easy to add additional logic. Typical DGSes have well-understood performance characteristics, and by making these into metrics you can determine when to add or remove VM instances. Common scaling metrics are number of DGS instances per CPU, as used in this solution, or available player slots.

Scaling up

For this solution, scaling up requires no special handling; simply increasing the number of nodes in the managed instance group is sufficient. For simplicity, this solution uses the limits and requests pod properties in Kubernetes to 'reserve' approximately one vCPU and 500MB of memory for each DGS. Since you created the cluster using the `n1-highcpu` instance family, which has a fixed ratio of 600MB of memory to 1 vCPU, you can safely assume there will be sufficient memory if you schedule one DGS pod per vCPU. Because of this fact, you can scale based on the number of pods in the cluster compared to the number of CPUs in all nodes in the cluster. This ratio determines the remaining resources available, allowing you to add more nodes if the value falls below a threshold. This solution adds nodes if more than 70% of the vCPUs are currently allocated to pods. In a live, production online game backend, it is recommended that you accurately profile DGS CPU, memory, and network usage and chose your `limits` and `requests` pod properties appropriately. For many games, it makes sense to create multiple pod types for different DGS scenarios with different usage profiles, such as game types, specific maps, or number of player slots. Such considerations falls outside the scope of this solution.

Scaling down

Scaling down, unlike scaling up, is a multi-step process and one of the major reasons to run a custom, Kubernetes-aware, DGS scaling manager. In this lab, `scaling_manger.sh` handles the following steps:

- An appropriate node must be selected for removal. Since a full custom game-aware Kubernetes scheduler is outside the scope of this lab, the nodes are selected in the order they are returned by the API.

- The selected node is marked as 'unschedulable' in Kubernetes by using the cordon command. This prevents additional pods from being started on this node.
- The selected node is removed from the managed instance group using the `abandon-instance` command. This prevents the managed instance group from attempting to recreate the instance.
  Separately, the `node_stopper.sh` script monitors abandoned, unschedulable nodes for the absence of DGS pods. Once all matches on a node have finished and the pods successfully exit, the script shuts down the VM instance.

Scaling the number of DGS pods

In typical production game backends, the matchmaker controls when new DGS instances are added. Since DGS pods are configured to successfully exit when matches complete (refer to the constraints), no explicit action is necessary to scale down the number of DGS pods. If there are not enough player requests coming into the matchmaker system to generate new matches, the DGS pods will slowly remove themselves from the Kubernetes cluster as matches end, in effect scaling down the number of pods.

Testing the setup

So far, the OpenArena container image has been created and pushed to the container registry, and the DGS Kubernetes cluster has been started. In addition, the game asset disk has been generated and configured for use in Kubernetes, and the scaling manager deployment has been started. Now it's time to start DGS pods for testing.

Requesting a new DGS instance

In a typical production system, the matchmaker process would directly request an instance using the Kubernetes API when it has appropriate players for a match. For the purposes of testing this solution's setup, the request for an instance can be made using a simple `kubectl` command, specifying the `openarena_pod.yaml` file.

You must first update the file `openarena/k8s/openarena-pod.yaml` file and replace [GCR_REGION] and [PROJECT_ID] in the containers - image: value. Since the variables are already defined, we can use `sed` again to do this:

```
cd ..
sed -i "s/\[GCR_REGION\]/$GCR_REGION/g" openarena/k8s/openarena-pod.yaml
sed -i "s/\[PROJECT_ID\]/$PROJECT_ID/g" openarena/k8s/openarena-pod.yaml
```

Apply the new pod configuration by running:

```
kubectl apply -f openarena/k8s/openarena-pod.yaml
```

A few seconds later, the status of the pod can be confirmed using `kubectl`:

```
kubectl get pods
```

Repeat the kubectl `get pods` command until you get the following output:

```
NAME           READY  STATUS   RESTARTS  AGE
openarena.dgs  1/1    Running  0         25s
```

Click *Check my progress* to verify the objective.

Only if your pod errors out at launch try deleting the pod, changing the mount location, and re-creating it:

```
kubectl delete pod openarena.dgs
sed -i "s/\/usr\/share\/games\/openarena\/baseoa/\/usr\/lib\/openarena-
server\/baseoa/g"  openarena/k8s/openarena-pod.yaml
kubectl apply -f openarena/k8s/openarena-pod.yaml
```

Connecting to the DGS

After the pod has started, the following commands will identify the game server ip address that you can use to connect to the game instance on port 27461.

```
export NODE_NAME=$(kubectl get pod openarena.dgs \
    -o jsonpath="{.spec.nodeName}")
export DGS_IP=$(gcloud compute instances list \
    --filter="name=( ${NODE_NAME} )" \
    --format='table[no-heading](EXTERNAL_IP)')

printf "Node Name: $NODE_NAME \nNode IP  : $DGS_IP \nPort         : 27961\n"
printf " launch client with: \nopenarena +connect $DGS_IP +set net_port 27961\n"
```

Launch the OpenArena client on your own computer once the game has been installed. Select **Multiplayer** and then **Specify**.

Configure the server ip-address and port using those listed by the output of the last command.

Click **Connect**.

Note: The configuration settings for this solution limit the game server duration to a few minutes, so don't wait too long before testing the client connection or you might have to start another pod.
Close the OpenArena client.

Testing the scaling manager

Since the scaling manager scales the number of VM instances in the Kubernetes cluster based on the number of DGS pods, testing it requires requesting a number of pods over a period of time and checking that the number of nodes scale appropriately.

For testing purposes, a script is provided in the solutions repository which adds four DGS pods per minute for 5 minutes:

```
source ./scaling-manager/tests/test-loader.sh
```

In your own environment, make sure to set the match length in the server configuration file to an appropriate limit so that the pods eventually exit and you can see the nodes scale back down! For this lab, the server configuration file places a five minute time limit on the match, and can be used as an example. It's located in the `repo` at `openarena/single-match.cfg`, and is used by default.

Open the Cloud Console and click **Kubernetes Engine**, then click **Workloads**.

You will see a sequence of workloads startup called `openarena-dgs.s`, `openarena-dgs.2` up to `openarena-dgs.15`. Because there are a limited the number of vCPUs in this cluster, many of the test containers will initially show an error state with a status of "Unschedulable". As the startup load in each container reduces, some of the later game server containers will successfully start up. All containers will start after about 10 minutes.

If you explore any of the failed instances you will see that the reason for the failure is a lack of CPU resources. If you explore any of the running instances you will see that the game server has initialized on those instances.

Congratulations

You have now successfully completed the Running Dedicated Game Servers in Google Kubernetes Engine.

Finish your Quest

This self-paced lab is part of the Google Cloud Solutions I: Scaling Your Infrastructure and Kubernetes Solutions Quests. A Quest is a series of related labs that form a learning path. Completing this Quest earns you the badge above, to recognize your achievement. You can make your badge public and link to them in your online resume or social media account. Enroll in a Quest and get immediate completion credit if you've taken this lab. See other available Qwiklabs Quests.

Take your next lab

Continue your Quest with the next lab, or check out these suggestions:

- Awwvision: Cloud Vision API from a Kubernetes Cluster
- Managing Deployments Using Kubernetes Engine

Next steps / learn more

This lab sketches out a bare-bones architecture for running dedicated game servers in containers and autoscaling a Kubernetes cluster based on game load. Many additional features, such as seamless player transition from session to session, can be enabled by engineering some basic client-side support. Other features, such as allowing players to form groups and moving the groups from server to server, are commonly accomplished by creating a separate platform service living alongside the matchmaking service. This service is used to form groups; send, accept, or reject group invitations; and send groups of players into dedicated game server instances together.

Another common feature is a custom Kubernetes scheduler capable of choosing nodes for your DGS pods in a more intelligent, game-centric way. For most games, a custom scheduler that packs pods together is highly desirable, making it easy for you to prioritize the order in which nodes should be removed when scaling down after peak.

- Learn more about Kubernetes

Google Cloud Training & Certification

...helps you make the most of Google Cloud technologies. Our classes include technical skills and best practices to help you get up to speed quickly and continue your learning journey. We offer fundamental to advanced level training, with on-demand, live, and virtual options to suit your busy schedule. Certifications help you validate and prove your skill and expertise in Google Cloud technologies.
Manual Last Updated April 05, 2021
Lab Last Tested April 05, 2021
Copyright 2021 Google LLC All rights reserved. Google and the Google logo are trademarks of Google LLC. All other company and product names may be trademarks of the respective companies with which they are associated.