

Packaging server applications as container images is quickly gaining traction across the tech landscape. Many game companies are also interested in using containers to improve VM utilization, as well as take advantage of the isolated run-time paradigm offered by containers. Despite this high interest, many game companies don't know where to start. We recommend using the container orchestration framework [Kubernetes](http://kubernetes.io/) (<http://kubernetes.io/>) to deploy production-scale fleets of dedicated game servers.

This tutorial describes an expandable architecture for running real-time, session-based multiplayer [dedicated game servers](/solutions/gaming/cloud-game-infrastructure#dedicated_game_server) ([/solutions/gaming/cloud-game-infrastructure#dedicated\\_game\\_server](/solutions/gaming/cloud-game-infrastructure#dedicated_game_server)) on [Google Kubernetes Engine](/kubernetes-engine/) (</kubernetes-engine/>). A scaling manager process automatically starts and stops virtual machine instances as needed. Configuration of the machines as Kubernetes nodes is handled automatically by [managed instance groups](/compute/docs/instance-groups/) (</compute/docs/instance-groups/>).

The online game structure presented in this tutorial is intentionally simple so that it's easy to understand and implement. Places where additional complexity might be useful are pointed out where appropriate.

- Create a container image of OpenArena, a popular open source dedicated game server (DGS) on Linux using [Docker](https://www.docker.com/) (<https://www.docker.com/>). 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.
- Set up and configure basic scheduler processes using the Kubernetes and Google Cloud Platform APIs to spin nodes up and down to meet demand.

This tutorial uses the following billable components of Google Cloud Platform:

- GKE
- [Persistent Disk \(/persistent-disk/\)](/persistent-disk/)

You can use the [pricing calculator \(/products/calculator/\)](/products/calculator/) to generate a cost estimate based on your projected usage.

This tutorial is intended to be run from a Linux or macOS environment.

1. In the Cloud Console, on the project selector page, select or create a Cloud project.

★ **Note:** If you don't plan to keep the resources that you create in this procedure, create a project instead of selecting an existing project. After you finish these steps, you can delete the project, removing all resources associated with the project.

[Go to the project selector page \(https://console.cloud.google.com/projectselector2/home/dashboard\)](https://console.cloud.google.com/projectselector2/home/dashboard)

2. Make sure that billing is enabled for your Google Cloud project. [Learn how to confirm billing is enabled for your project \(/billing/docs/how-to/modify-project/\)](/billing/docs/how-to/modify-project/).

3. Enable the Compute Engine API.

[Enable the API \(https://console.cloud.google.com/flows/enableapi?apiid=compute\\_engine\)](https://console.cloud.google.com/flows/enableapi?apiid=compute_engine)

4. [Install and initialize the Cloud SDK \(/sdk/docs/\)](/sdk/docs/).

**Note:** For this tutorial, you cannot use Cloud Shell. You must install the Cloud SDK.

5. Install `kubect1`, the [command-line interface for Kubernetes \(https://kubernetes.io/docs/user-guide/kubectl-overview/\)](https://kubernetes.io/docs/user-guide/kubectl-overview/):

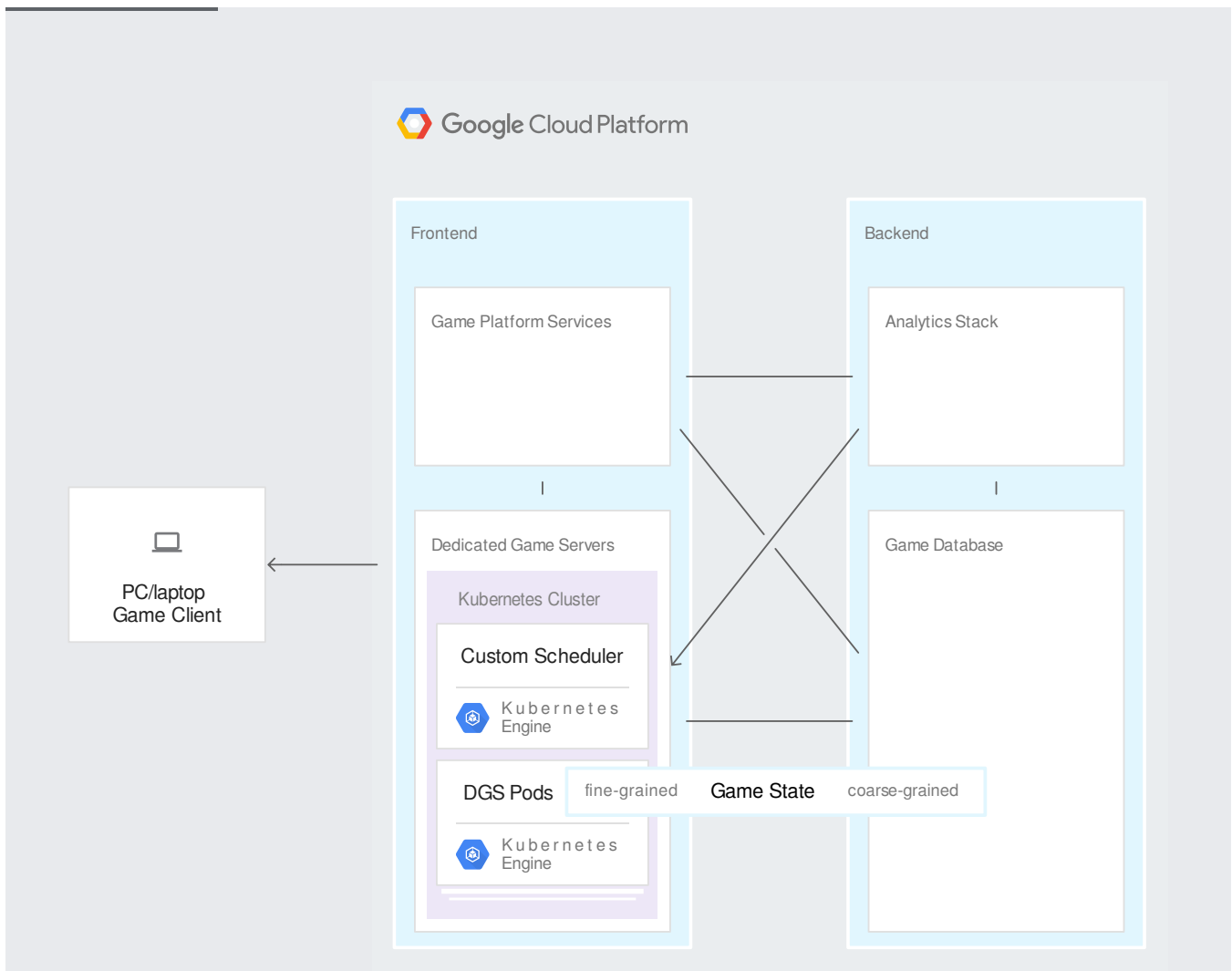
6. Clone the tutorial's repository from GitHub:

## 7. Install Docker (<https://docs.docker.com/engine/installation/>)

This tutorial does not run Docker commands as the root user, so be sure to also follow the [post-installation instructions for managing Docker as a non-root user](https://docs.docker.com/install/linux/linux-postinstall/)

(<https://docs.docker.com/install/linux/linux-postinstall/>).

8. (Optional) If you want to test a connection to the game server at the end of the tutorial, [install the OpenArena game client](http://openarena.wikia.com/wiki/Manual/Install) (<http://openarena.wikia.com/wiki/Manual/Install>). Running the game client requires a desktop environment. This tutorial includes instructions for testing using Linux or macOS.



### The Overview of Cloud Game Infrastructure

([/solutions/gaming/cloud-game-infrastructure#high-level\\_components](/solutions/gaming/cloud-game-infrastructure#high-level_components)) page discusses the high-level components common to many online game architectures. In this tutorial, 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, but those are outside the scope of this tutorial.

To produce an example that is both instructive and simple enough to extend, this tutorial 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 with the client over UDP.
- Each DGS process runs 1 match.
- All DGS processes generate approximately the same load.
- Matches have a maximum time length.
- DGS startup time is negligible, and pre-warming the dedicated game server process isn't necessary.
- When scaling down after peak, matches are not ended prematurely in an attempt to save cost. The priority is on avoiding impact to the player experience.
- If a DGS process encounters an error and cannot continue, the match state is lost and the player is expected to use the game client to join another 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.

To make it easier to run `gcloud` commands, you can set properties so that you don't have to supply options for these properties with each command.

1. Set your default project, using your project ID for `[PROJECT_ID]`:
2. Set your default Compute Engine zone, using your preferred zone for `[ZONE]`:

In this tutorial, you'll use [OpenArena](http://www.openarena.ws/smfnews.php) (<http://www.openarena.ws/smfnews.php>), which is described as "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 adds only the binaries and libraries to the base OS container image that are required in order to run the server process.
- Assets are mounted from a separate volume.

This architecture has many benefits: it speeds image distribution, reduces the update load because only binaries are replaced, and consumes less disk space.

A [Dockerfile](https://docs.docker.com/engine/reference/builder/) (<https://docs.docker.com/engine/reference/builder/>) describes the image to be built. The Dockerfile for this tutorial is provided in the repository at `openarena/Dockerfile`. From the `openarena/` directory, run the [Docker build command](https://docs.docker.com/engine/reference/commandline/build/) (<https://docs.docker.com/engine/reference/commandline/build/>) to generate the container image and tag it as version 0.8.8 of the OpenArena server:

In most games, binaries are orders of magnitude smaller than assets. Because of this, it makes sense to create a container image that contains only 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.

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

## 2. Create a persistent disk:

The persistent disk must be separate from the boot disk, and you must configure it to remain undeleted when the virtual machine is removed. Kubernetes [persistentVolume](https://kubernetes.io/docs/concepts/storage/persistent-volumes/) (<https://kubernetes.io/docs/concepts/storage/persistent-volumes/>) functionality works best in GKE with persistent disks. According to the [Compute Engine](/compute/docs/disks/add-persistent-disk#formatting) (</compute/docs/disks/add-persistent-disk#formatting>), these persistent disks should consist of a single `ext4` file system without a partition table.

★ **Note:** Some persistent disk types have better performance when provisioned at larger disk sizes. For details, see the [documentation about optimizing persistent disk performance](/compute/docs/disks/performance) (</compute/docs/disks/performance>). Depending on how read-heavy your engine is, for better performance it can make sense to choose a disk that's larger than is needed just to hold the data.

## 3. Attach the `openarena-assets` persistent disk to the `openarena-asset-builder` VM instance:


## 4. Format the new disk.

- a. Log in to the `openarena-asset-builder` VM instance and format the disk.

- b. Because the `mkfs.ext4` command in the next step is a **destructive** command, be sure to confirm the device ID for the `openarena-assets` disk. If you're following this tutorial from the beginning, and you are using a fresh project, the ID is `/dev/sdb`. Verify this using the `lsblk` command to look at the attached disks and their partitions:

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`:

- c. Format the `openarena-assets` disk:

 **Caution:** The `mkfs.ext4` command is a **destructive command**. Be sure to use the device ID you verified previously.

5. 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, the following sequence of commands mounts the assets disk to this directory before installing, so all the `.pk3` files are put on the disk by the install process. Be sure to use the device ID you verified previously.



## 6. Exit from the instance and then delete it:

The disk is ready to be used as a persistent volume in Kubernetes.

When you implement persistent disks as part of your game development pipeline, configure your build system to create the persistent disk with all the asset files in an appropriate directory structure. This might take the form of a simple script running `gcloud` commands, or a GCP-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 additional throughput and to manage the risk of failure.

Kubernetes is an open source community project and as such can be configured to run in most environments, including on-premises.

This tutorial uses a standard Kubernetes Engine cluster outfitted with the [n1-highcpu machine type](/compute/docs/machine-types) (`/compute/docs/machine-types`), which fits the usage profile of OpenArena.

1. Create a VPC network for the game named `game`:

2. Create a firewall rule for OpenArena:

★ **Note:** This tutorial opens 100 ports and naively increments the port number for each DGS created. In a production system, you should create a more advanced form of port management.

3. Use `gcloud` to create a 3-node cluster with 4 virtual CPU cores on each that uses your `game` network:

4. After the cluster has started, [set up your local shell with the proper Kubernetes authentication credentials](#) (`/sdk/gcloud/reference/container/clusters/get-credentials`) to control your new cluster:

In a production cluster, 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](#) (<https://kubernetes.io/docs/setup/cluster-large/>) (although the Kubernetes project plans to increase this with future releases). For example, if you run 1 DGS per virtual CPU (vCPU), a 1000-node cluster of `n1-highcpu-2` machines provides capacity for only 2000 DGS pods. In contrast, a 1000-node cluster of `n1-highcpu-32` machines allows up to 32,000 pods.
- **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. Therefore, don't choose a 32 vCPU machine if you want to be able to add or remove capacity in smaller amounts than 32 vCPUs at a time.

The managed instance groups feature that's used by default by GKE includes VM instance autoscaling and HTTP load balancing features. However, the command you used to create the Kubernetes cluster disabled these features by using the `--disable-addons HttpLoadBalancing,HorizontalPodAutoscaling` flag.

HTTP load balancing is not needed because the DGS communicates with clients using UDP, not TCP. The autoscaler can currently only scale the instance group based on CPU usage, which can be a misleading indicator of DGS load. Many DGSs are designed to consume idle cycles in an effort to optimize the game's simulation.

As a result, many game developers implement a custom scaling manager process that is DGS aware (`#setting_up_the_scaling_manager`) to deal with the specific requirements of this type of workload. The managed instance group does still serve an important function, however—its default GKE image template includes all the necessary Kubernetes software and automatically registers the node with the master on startup.

Google offers private Docker image storage in Container Registry (<https://gcr.io>) (`gcr.io`).

1. Select the `gcr.io` region nearest to your GKE cluster (for example, `us` for the United States, `eu` for Europe, or `asia` for Asia, as noted in the documentation (`/container-registry/docs/pushing-and-pulling`)) and put the region information in an environment variable along with your project ID:
2. Tag your container image with the `gcr.io` registry name:

### 3. Upload the container image to the image repository:

After the push is complete, the container image will be available to run in your GKE cluster. Make a note of your final container image tag, because you'll need to put it in the pod specification file later.

The typical DGS does not need write access to the game assets, so you can have each DGS pod mount the same persistent disk that contains read-only assets. This is accomplished using the [persistentVolume](http://kubernetes.io/docs/user-guide/persistent-volumes/#persistent-volumes) (<http://kubernetes.io/docs/user-guide/persistent-volumes/#persistent-volumes>) and [persistentVolumeClaim](http://kubernetes.io/docs/user-guide/persistent-volumes/#persistentvolumeclaims) (<http://kubernetes.io/docs/user-guide/persistent-volumes/#persistentvolumeclaims>) resources in Kubernetes.

1. Apply `asset-volume.yaml`, which contains the definition of a Kubernetes `persistentVolume` resource that will bind to the assets disk you created before:

2. Apply `asset-volumeclaim.yaml`. It contains the definition of a Kubernetes `persistentVolumeClaim` resource, which will allow pods to mount the assets disk:

Confirm that the volume is in `Bound` status by running the following command:

Expected output:

Similarly, confirm that the claim is in bound status:

Expected output:

Because scheduling and networking tasks are handled by Kubernetes, and because the startup and shutdown times of the DGS containers are negligible, in this tutorial, DGS instances spin up on demand.

Each DGS instance lasts only the length of a single game match, and a game match has a defined time limit specified in the OpenArena DGS server configuration file. When the match is complete, the container successfully exits. If players want to play again, they request another game. This design simplifies a number of pod lifecycle aspects and forms the basis of the autoscaling policy discussed later in the [scaling manager section](#) (`#setting_up_the_scaling_manager`).

Although this flow isn't seamless with OpenArena, it's only because the tutorial 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 that requires the client to connect to a new server instance between matches doesn't represent additional development time, because that code is mandatory anyway for handling client reconnections for unforeseen circumstances such as network issues or crashing servers.

For the sake of simplicity, this tutorial assumes that the GKE nodes have the default network configuration, which assigns each node a public IP address and allows client connections.

In commercially produced game servers, all of the additional, non-DGS functionality that makes a DGS run well in a container 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, 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 about [accessing the Kubernetes API directly](#)

(<http://kubernetes.io/docs/user-guide/accessing-the-cluster/#directly-accessing-the-rest-api>) can be found in the Kubernetes documentation.

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](#)

(<https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/>), but in practice it's unnecessary to use jobs. DGS processes don't require the parallel execution functionality of jobs. They also don't require 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 tutorial, a human-readable YAML file describing the DGS pod resource is included in the tutorial repository at [openarena/k8s/openarena-pod.yaml](#). When you create configurations for dedicated game server pods, pay close attention to the volume properties to ensure that the asset disk can be mounted read-only in multiple pods.

The scaling manager is a simple process that scales the number of virtual machines used as GKE nodes, based on the current DGS load. Scaling is accomplished using a set of scripts that run forever, that inspect the total number of DGS pods running and requested, and that resize the node pool as necessary. The scripts are packaged in Docker container images that include the appropriate libraries and the Cloud SDK. The Docker images can be created and pushed to gcr.io using the following procedure.

1. If necessary, put the gcr.io `GCR_REGION` value and your `PROJECT_ID` into environment variables for the build and push script. You can skip this step if you already did it earlier when you uploaded the container image (`#uploading_the_container_image_to_gcp`).

2. Change to the script directory:

3. Run the build script to build all the container images and push them to gcr.io:

4. Using a text editor, open the Kubernetes deployment file at `scaling-manager/k8s/openarena-scaling-manager-deployment.yaml`.

The scaling manager scripts are designed to be run within a Kubernetes deployment, which ensures that these processes are restarted in the event of a failure.

5. Change the environment variables to values for your deployment, as listed in the following table:

Environment Variable	Default Value	Notes
<code>REGION</code>	<code>[GCR_REGION]</code>	<b>Requires replacement.</b> The region of your gcr.io repository.
<code>PROJECT_ID</code>	<code>[PROJECT_ID]</code>	<b>Requires replacement.</b> The name of your project.
<code>GKE_BASE_INSTANCE_NAME</code>	<code>gke-openarena-cluster-default-pool-[REPLACE_ME]</code>	<b>Requires replacement.</b> Different for every GKE cluster. To get the value for <code>[REPLACE_ME]</code> , run the <code>gcloud compute instance-groups managed list</code> command.

Environment Variable	Default Value	Notes
GCP_ZONE	[ZONE]	<b>Requires replacement.</b> The name of the GCP zone that you <u>specified</u> ( <code>#preparing_your_gcp_working_environment</code> ) at the beginning of this tutorial.
K8S_CLUSTER	<code>openarena-cluster</code>	The name of the Kubernetes cluster.

6. Return to the parent directory:

7. Add the deployment to your Kubernetes cluster:

To scale nodes, the scaling manager uses the Kubernetes API to look at current node usage. As needed, the manager resizes the Kubernetes Engine cluster's managed instance group (`/compute/docs/instance-groups/`) that runs 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, because scheduling an optimized DGS container on an existing node takes a matter of seconds. However, adding a node can take minutes, which is an unacceptable latency for a waiting player.



- Many autoscalers aren't able to handle pod shutdowns gracefully. It's important to drain pods from nodes that are being removed. Shutting off a node with even one running match is often unacceptable.

Although the scripts supplied by this tutorial are basic, their simple design makes it easy to add additional logic. Typical DGSs 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 the number of DGS instances per CPU, as used in this tutorial, or the number of available player slots.

Scaling up requires no special handling in this tutorial. For simplicity, this tutorial sets the **limits and requests pod properties**

(<https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/#resource-requests-and-limits-of-pod-and-container>)

in the pod's YAML file (`openarena/k8s/openarena-pod.yaml`) to reserve approximately 1 vCPU for each DGS, which is sufficient for OpenArena.

Because the cluster was created using the `n1-highcpu` instance family, which has a fixed ratio of 600 MB of memory to 1 vCPU, there should be sufficient memory if 1 DGS pod is scheduled per vCPU. Therefore, you can scale up 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 tutorial adds nodes if more than 70% of the vCPUs are currently allocated to pods.

In a production online game backend, it is recommended that you accurately profile DGS CPU, memory, and network usage, and then set the **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 fall outside the scope of this tutorial.

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 tutorial, `scaling-manager.sh` automatically handles the following steps:

1. Selecting an appropriate node for removal. Because a full custom game-aware Kubernetes scheduler is outside the scope of this tutorial, the nodes are selected in the order they are returned by the API.
2. Marking the selected node as unavailable in Kubernetes (<http://kubernetes.io/docs/admin/node/#manual-node-administration>). This prevents additional pods from being started on the node.
3. Removing the selected node from the managed instance group using the abandon-instance (</sdk/gcloud/reference/compute/instance-groups/managed/abandon-instances>) 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. After all matches on a node have finished and the pods exit, the script shuts down the VM instance.

In typical production game backends, the matchmaker controls when new DGS instances are added. Because DGS pods are configured to exit when matches finish (refer to the design constraints ([#design-constraints](#)) earlier), 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 slowly remove themselves from the Kubernetes cluster as matches end.

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

In a typical production system, when the matchmaker process has appropriate players for a match, it directly requests an instance using the Kubernetes API. For the purposes of testing this tutorial's setup, you can directly make the request for an instance.

1. Open `openarena/k8s/openarena-pod.yaml` in a text editor, and find the line that specifies the container image to run.
2. Change the value to match your `openarena` container image tag by running the `docker tag` command as [described earlier in this tutorial](#) (`#docker-tag-image`).
3. Run the `kubectl apply` command, specifying the `openarena-pod.yaml` file:
  
4. Wait for a short time and then confirm the status of the pod:

The output should look similar to this:

After the pod has started, you can verify that you can connect to the DGS by launching the OpenArena client.

From a macOS or Linux desktop:

**Note:** The tutorial configuration settings limit the game server duration to a few moments, so don't wait too long before testing the client connection or you might have to start another pod.

The scaling manager scales the number of VM instances in the Kubernetes cluster based on the number of DGS pods. Therefore, testing the scaling manager requires requests for a number of pods over a period of time and checking that the number of nodes scales appropriately. In order to see the nodes scale back down, the match length in the server configuration file must have a time limit. The tutorial server configuration file at `openarena/single-match.cfg` places a 5-minute limit on the match, and is used by the tutorial DGS pods by default. To test, run the following script, which adds DGS pods at regular intervals for 5 minutes:

You should be able to see the number of nodes scale up and back down by running `kubectl get nodes` at regular intervals.

To avoid incurring charges to your Google Cloud Platform account for the resources used in this tutorial:

The easiest way to eliminate billing is to delete the project that you created for the tutorial.

To delete the project:




**Caution:** Deleting a project has the following effects:

- **Everything in the project is deleted.** If you used an existing project for this tutorial, when you delete it, you also delete any other work you've done in the project.
- **Custom project IDs are lost.** When you created this project, you might have created a custom project ID that you want to use in the future. To preserve the URLs that use the project ID, such as an `appspot.com` URL, delete selected resources inside the project instead of deleting the whole project.

If you plan to explore multiple tutorials and quickstarts, reusing projects can help you avoid exceeding project quota limits.

1. In the Cloud Console, go to the **Manage resources** page.

[Go to the Manage resources page](https://console.cloud.google.com/iam-admin/projects) (https://console.cloud.google.com/iam-admin/projects)

2. In the project list, select the project you want to delete and click **Delete** .

3. In the dialog, type the project ID, and then click **Shut down** to delete the project.

If you don't want to delete the whole project, run the following command to delete the GKE cluster:

To delete a persistent disk:

1. In the Cloud Console, go to the Compute Engine Disks page.

[Go to the Compute Engine Disks page](https://console.cloud.google.com/compute/disks) (https://console.cloud.google.com/compute/disks)

2. Select the disk you want to delete.

3. Click the **Delete** button at the top of the page.

This tutorial sketches out a bare-bones architecture for running dedicated game servers in containers and autoscaling a Kubernetes cluster based on game load. You can add many features, such as seamless player transition from session to session, by programming some basic client-side support. To add other features, such as allowing players to form groups and moving the groups from server to server, you can create a separate platform service living alongside the matchmaking service. You can then use the service 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.

More guides to running a DGS on GCP:

- [Overview of Cloud Game Infrastructure](/solutions/gaming/cloud-game-infrastructure) (/solutions/gaming/cloud-game-infrastructure)
- [Dedicated Game Server Migration Guide](/solutions/gaming/dedicated-game-server-migration-guide) (/solutions/gaming/dedicated-game-server-migration-guide)
- [Setting Up a Minecraft Server on Google Compute Engine](/solutions/gaming/minecraft-server) (/solutions/gaming/minecraft-server)
- Try out other Google Cloud features for yourself. Have a look at our [tutorials](/docs/tutorials) (/docs/tutorials).