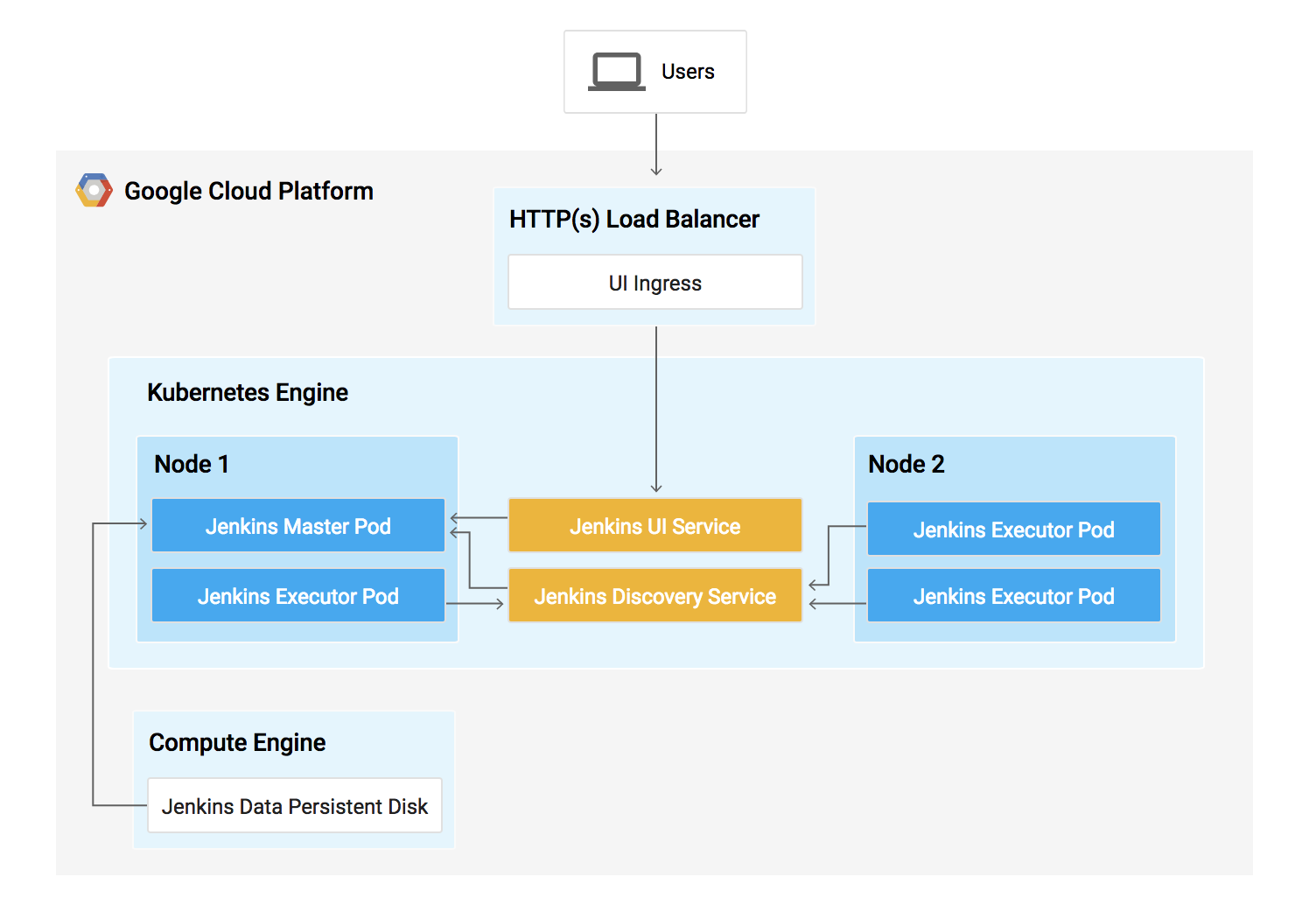
Continuous Deployment with Jenkins v1.6

2 hours 30 minutes9 Credits

Rate Lab

**Overview**

In this lab, you will learn how to set up a continuous delivery pipeline with **Jenkins** on Kubernetes engine. Jenkins is the go-to automation server used by developers who frequently integrate their code in a shared repository. The solution you'll build in this lab will be similar to the following diagram:



You can find more details about running Jenkins on Kubernetes [here](https://cloud.google.com/solutions/jenkins-on-kubernetes-engine).

What you'll do

In this lab, you will complete the following tasks:

* Provision a Jenkins environment on a Kubernetes Engine Cluster, using the Helm Package Manager
* Explore the features of a Jenkins application
* Create and exercise Jenkins pipelines
* Deploy an application using development, canary, and production pipelines

Prerequisites

This is an **expert level** lab. Before taking it, you should be comfortable with at least the basics of shell programming, Kubernetes, and Jenkins. Here are some Qwiklabs that can get you up to speed:

* [Introduction to Docker](https://google.qwiklabs.com/catalog_lab/944)
* [Hello Node Kubernetes](https://google.qwiklabs.com/catalog_lab/468)
* [Managing Deployments Using Kubernetes Engine](https://google.qwiklabs.com/catalog_lab/572)
* [Setting up Jenkins on Kubernetes Engine](https://google.qwiklabs.com/catalog_lab/1093)

Once your prepared, scroll down to learn more about Kubernetes, Jenkins, and Continuous Delivery.

**What is Kubernetes Engine?**

Google Kubernetes Engine (GKE) is the hosted version of Kubernetes on Google Cloud Platform (GCP). GKE is a powerful cluster manager and orchestration system for containers.

Kubernetes is an open source project that can run on many different environments—from laptops to high-availability multi-node clusters; from virtual machines to bare metal. As mentioned before, Kubernetes apps are built on Containers - these are lightweight applications bundled with all the necessary dependencies and libraries to run them. This underlying structure makes Kubernetes applications highly available, secure, and quick to deploy—an ideal framework for cloud developers.

**What is Jenkins?**

[Jenkins](https://jenkins.io/) is an open-source automation server that lets you flexibly orchestrate your build, test, and deployment pipelines. Jenkins allows developers to iterate quickly on projects without worrying about overhead issues that can stem from continuous delivery.

**What is Continuous Delivery / Continuous Deployment?**

When you need to set up a continuous delivery (CD) pipeline, deploying Jenkins on Kubernetes Engine provides important benefits over a standard VM-based deployment.

When your build process uses containers, one virtual host can run jobs on multiple operating systems. Kubernetes Engine provides ephemeral build executors — these are only utilized when builds are actively running, which leaves resources for other cluster tasks such as batch processing jobs. Another benefit of ephemeral build executors is *speed* — they launch in a matter of seconds.

Kubernetes Engine also comes pre-equipped with Google's global load balancer, which you can use to automate web traffic routing to your instance(s). The load balancer handles SSL termination and utilizes a global IP address that's configured with Google's backbone network—coupled with your web frontend, this load balancer will always set your users on the fastest possible path to an application instance.

Now that you've learned a little bit about Kubernetes, Jenkins, and how the two interact in a CD pipeline, let's go build one.

**Set up**

**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 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 the Google Cloud Platform 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 GCP account or project, do not use it for this lab.

**How to start your lab and sign in to the 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.



1. Copy the username, and then click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Choose an account** page.

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

1. On the Choose an account page, click **Use Another Account**.



1. The Sign in page opens. 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 GCP account, do not use it for this lab (avoids incurring charges).

1. 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 GCP console opens in this tab.

**Note:** You can view the menu with a list of GCP Products and Services by clicking the **Navigation menu** at the top-left, next to “Google Cloud Platform”. 

Activate Google Cloud Shell

Google 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. Google Cloud Shell provides command-line access to your GCP resources.

1. In GCP console, on the top right toolbar, click the Open Cloud Shell button.



1. Click **Continue**. 

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:



**gcloud** is the command-line tool for Google Cloud Platform. 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

Full documentation of **gcloud** is available on [Google Cloud gcloud Overview](https://cloud.google.com/sdk/gcloud).

**Clone Repository**

Let's get set up. You'll first set your zone and then clone the lab's sample code into your Cloud Shell:

gcloud config set compute/zone us-east1-d

git clone https://github.com/GoogleCloudPlatform/continuous-deployment-on-kubernetes.git

cd continuous-deployment-on-kubernetes

The Git repository contains Kubernetes manifests that you'll use to deploy an application. The manifests and their settings are described in [Configuring Jenkins for Kubernetes Engine](https://cloud.google.com/solutions/jenkins-on-kubernetes-engine).

**Create a Service Account with permissions**

Using a service account is the recommended way to avoid granting extra permissions in Jenkins and the cluster.

1. Create a Google Cloud Platform (GCP) service account.
2. gcloud iam service-accounts create jenkins-sa \

--display-name "jenkins-sa"

Output (do not copy):

Created service account [jenkins-sa].

1. Add required permissions, to the service account, using predefined roles.

Most of these permissions are related to Jenkins use of *Cloud Build*, and storing/retrieving build artifacts in *Cloud Storage*. Also, the service account needs to enable the Jenkins agent to read from a repo you will create in *Cloud Source Repositories (CSR)*.

gcloud projects add-iam-policy-binding $GOOGLE\_CLOUD\_PROJECT \

--member "serviceAccount:jenkins-sa@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com" \

--role "roles/viewer"

gcloud projects add-iam-policy-binding $GOOGLE\_CLOUD\_PROJECT \

--member "serviceAccount:jenkins-sa@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com" \

--role "roles/source.reader"

gcloud projects add-iam-policy-binding $GOOGLE\_CLOUD\_PROJECT \

--member "serviceAccount:jenkins-sa@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com" \

--role "roles/storage.admin"

gcloud projects add-iam-policy-binding $GOOGLE\_CLOUD\_PROJECT \

--member "serviceAccount:jenkins-sa@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com" \

--role "roles/storage.objectAdmin"

gcloud projects add-iam-policy-binding $GOOGLE\_CLOUD\_PROJECT \

--member "serviceAccount:jenkins-sa@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com" \

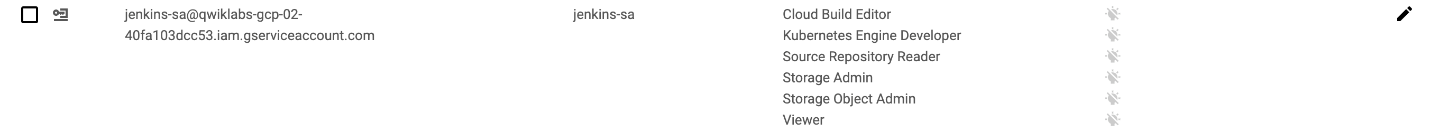
--role "roles/cloudbuild.builds.editor"

gcloud projects add-iam-policy-binding $GOOGLE\_CLOUD\_PROJECT \

--member "serviceAccount:jenkins-sa@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com" \

--role "roles/container.developer"

You can check the permissions added using **IAM & admin** in Cloud Console.



1. Export the service account credentials to a JSON key file in Cloud Shell:
2. gcloud iam service-accounts keys create ~/jenkins-sa-key.json \

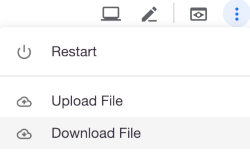
--iam-account "jenkins-sa@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com"

Output (do not copy):

created key [...] of type [json] as [/home/.../jenkins-sa-key.json] for [jenkins-sa@myproject.aiam.gserviceaccount.com]

1. Download the JSON key file to your local machine.

Click **Download File** from **More** on the Cloud Shell toolbar:



1. Enter the **File path** as jenkins-sa-key.json and click **Download**.

The file will be downloaded to your local machine, for use later.

**Create a Kubernetes cluster for Jenkins and your application**

1. Provision a Kubernetes cluster, on Google Kubernetes Engine (GKE):
2. gcloud container clusters create jenkins-cd \
3. --num-nodes 2 \
4. --machine-type n1-standard-2 \
5. --cluster-version latest \
6. --service-account "jenkins-sa@$GOOGLE\_CLOUD\_PROJECT.iam.gserviceaccount.com"

This step can take up to several minutes to complete.

Click *Check my progress* to verify the objective.

Create a Kubernetes cluster (zone: us-east1-d)

Check my progress

1. Confirm that your cluster is running by running the following command:
2. gcloud container clusters list
3. Now, get the credentials for your cluster:
4. gcloud container clusters get-credentials jenkins-cd

Kubernetes Engine uses these credentials to access your newly provisioned cluster.

1. Confirm that you can connect to the cluster:
2. kubectl cluster-info
3. Add yourself as a cluster administrator, in the cluster's RBAC:

This is needed so that you can give Jenkins permissions in the cluster.

kubectl create clusterrolebinding cluster-admin-binding --clusterrole=cluster-admin --user=$(gcloud config get-value account)

Output (do not copy):

Your active configuration is: [cloudshell-...]

clusterrolebinding.rbac.authorization.k8s.io/cluster-admin-binding created

**Install Helm**

In this lab, you will use **Helm** to install Jenkins from its *Charts* repository. Helm is a package manager that makes it easy to configure and deploy Kubernetes applications. Once you have Jenkins installed, you'll be able to set up your CI/CD pipeline.

1. Download and install the helm binary:
2. wget https://storage.googleapis.com/kubernetes-helm/helm-v2.14.3-linux-amd64.tar.gz
3. Unzip the file in Cloud Shell:
4. tar zxfv helm-v2.14.3-linux-amd64.tar.gz
5. cp linux-amd64/helm .
6. Grant Tiller, the server side of Helm, the cluster-admin role in your cluster:
7. kubectl create serviceaccount tiller --namespace kube-system
8. kubectl create clusterrolebinding tiller-admin-binding --clusterrole=cluster-admin --serviceaccount=kube-system:tiller

Output (do not copy):

serviceaccount/tiller created

clusterrolebinding.rbac.authorization.k8s.io/tiller-admin-binding created

1. Initialize Helm. This ensures that the server side of Helm (Tiller) is properly installed in your cluster.
2. ./helm init --service-account=tiller

Output (do not copy):

...

Tiller (the Helm server-side component) has been installed into your Kubernetes Cluster.

...

1. Update your local repo with the latest charts.

./helm repo update

Output (do not copy):

Hang tight while we grab the latest from your chart repositories...

...Skip local chart repository

...Successfully got an update from the "stable" chart repository

Update Complete.

1. Ensure Helm is properly installed:
2. ./helm version

You should see versions appear for both the server and the client of v2.14.3.

If you see an error, such as could not find a ready tiller pod, wait a bit and try again.

Click *Check my progress* to verify the objective.

Install Helm

Check my progress

**Configure and Install Jenkins**

You will use a custom values file to add the GCP specific plugins necessary to use service account credentials to reach your Cloud Source Repository.

1. Use the Helm CLI to deploy the chart with your configuration settings:
2. ./helm install -n cd stable/jenkins -f jenkins/values.yaml --version 1.7.3 --wait

This may take a minute.

1. The Jenkins pod **STATUS** should change to Running when it's ready:
2. kubectl get pods

Output (do not copy):

NAME READY STATUS RESTARTS AGE

cd-jenkins-7c786475dd-vbhg4 1/1 Running 0 1m

Click *Check my progress* to verify the objective.

Configure and Install Jenkins

Check my progress

1. Apply the cluster-admin role to the Jenkins service account:
2. kubectl create clusterrolebinding jenkins-deploy \
3. --clusterrole=cluster-admin --serviceaccount=default:cd-jenkins

In this tutorial, the Jenkins service account needs cluster-admin permissions so that it can create Kubernetes namespaces and any other resources that the app requires. For production use, you should catalog the individual permissions necessary and apply them to the service account individually.

1. Set up port forwarding to the Jenkins UI, from Cloud Shell:
2. export JENKINS\_POD\_NAME=$(kubectl get pods -l "app.kubernetes.io/component=jenkins-master" -o jsonpath="{.items[0].metadata.name}")
3. kubectl port-forward $JENKINS\_POD\_NAME 8080:8080 >> /dev/null &
4. Now, check that the Jenkins Service was created properly:
5. kubectl get svc

Output (do not copy):

NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE

cd-jenkins 10.35.249.67 <none> 8080/TCP 3h

cd-jenkins-agent 10.35.248.1 <none> 50000/TCP 3h

kubernetes 10.35.240.1 <none> 443/TCP 9h

This Jenkins configuration is using the [Kubernetes Plugin](https://wiki.jenkins-ci.org/display/JENKINS/Kubernetes+Plugin), so that builder nodes will be automatically launched as necessary when the Jenkins master requests them. Upon completion of the work, the builder nodes will be automatically be turned down, and their resources added back to the clusters resource pool.

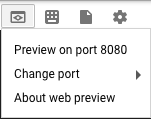
Notice that this service exposes ports 8080 and 50000 for any pods that match the selector. This will expose the Jenkins web UI and builder/agent registration ports within the Kubernetes cluster. Additionally, the jenkins-ui services are exposed using a ClusterIP so that it is not accessible from outside the cluster.

**Connect to Jenkins**

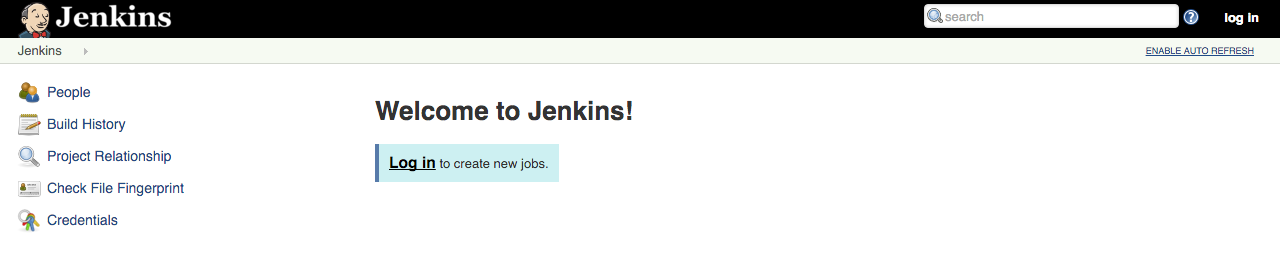
1. The Jenkins chart will automatically create an admin password for you. To retrieve it, run:

printf $(kubectl get secret cd-jenkins -o jsonpath="{.data.jenkins-admin-password}" | base64 --decode);echo

1. To get to the Jenkins user interface, click on the **Web Preview** button in cloud shell, then click **Preview on port 8080**.



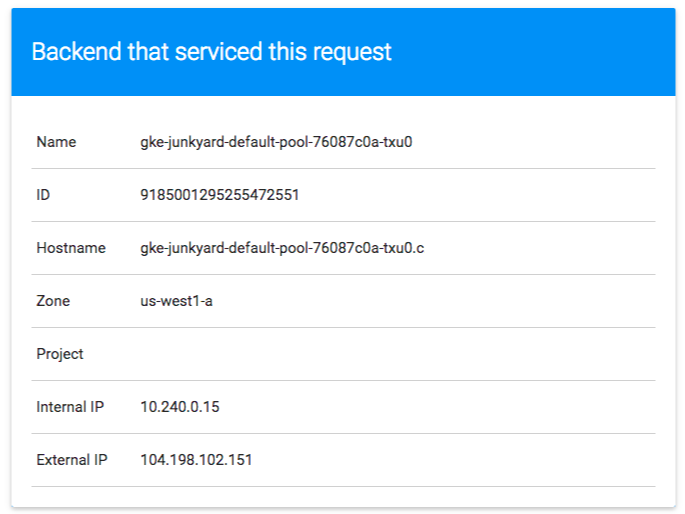
1. You should now be able to log in with username admin and your auto-generated password.



You now have Jenkins set up in your Kubernetes cluster! Jenkins will drive your automated CI/CD pipelines in the next sections.

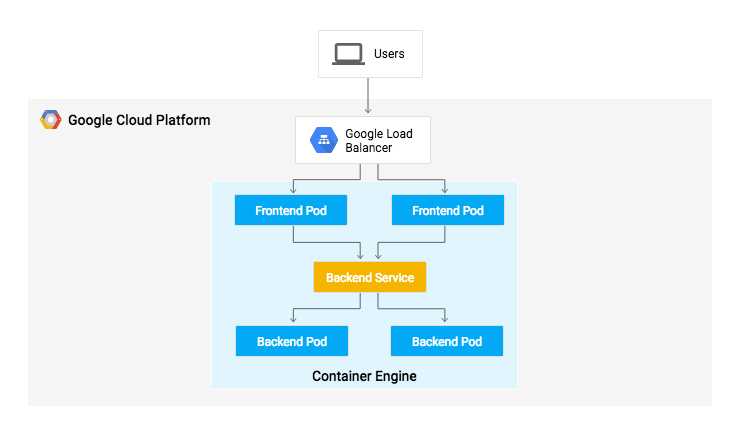
**Understand the Application**

You'll deploy the sample application, gceme, in your continuous deployment pipeline. The application is written in the Go language and is located in the repo's sample-app directory. When you run the gceme binary on GKE, the app displays the instance's metadata in an info card.



The application mimics a microservice by supporting two operating modes.

* In **backend mode**: gceme listens on port 8080 and returns host node instance metadata in JSON format.
* In **frontend mode**: gceme queries the backend gceme service and renders the resulting JSON in the user interface.



Both the frontend and backend modes of the application support two additional URLs:

* /version prints the version of the binary (declared as a const in main.go)
* /healthz reports the health of the application. In frontend mode, health will be OK if the backend is reachable.

**Deploy the Application to Kubernetes**

You will deploy the application into two different environments:

* **Production**: The live site that your users access.
* **Canary**: A smaller-capacity site that receives only a percentage of your user traffic. Use this environment to validate your software with live traffic before it's released to all of your users.

1. In Google Cloud Shell, navigate to the sample application directory:
2. cd sample-app
3. Create the Kubernetes namespace to logically isolate the deployment:
4. kubectl create ns production

Output (do not copy):

namespace/production created

1. Create the production Deployments for frontend and backend:
2. kubectl --namespace=production apply -f k8s/production

Output (do not copy):

deployment.extensions/gceme-backend-production created

deployment.extensions/gceme-frontend-production created

1. Create the canary Deployments for frontend and backend:
2. kubectl --namespace=production apply -f k8s/canary

Output (do not copy):

deployment.extensions/gceme-backend-canary created

deployment.extensions/gceme-frontend-canary created

1. Create the Services for frontend and backend:
2. kubectl --namespace=production apply -f k8s/services

Output (do not copy):

service/gceme-backend created

service/gceme-frontend created

Click *Check my progress* to verify the objective.

Create the production and canary deployments in production namespace

Check my progress

1. Scale the production, frontend service:

By default, only one replica of the frontend is deployed. Use the kubectl scale command to ensure that there are at least 4 replicas running at all times. This will be used to set the ratio of *production* pods to *canary* pods.

kubectl --namespace=production scale deployment gceme-frontend-production --replicas=4

Output (do not copy):

deployment.extensions/gceme-frontend-production scaled

This may take upto a minute for all the pods to be running.

1. Confirm that you have 5 pods running for the frontend:

4 are for production traffic and 1 is for canary releases. Changes to the canary release will only affect 1 out of 5 (20%) users.

kubectl get pods -n production -l app=gceme -l role=frontend

1. Also confirm that you have 2 pods for the backend:

1 is for production and 1 is for canary.

kubectl get pods -n production -l app=gceme -l role=backend

1. Retrieve the External IP for the production services:
2. kubectl --namespace=production get service gceme-frontend

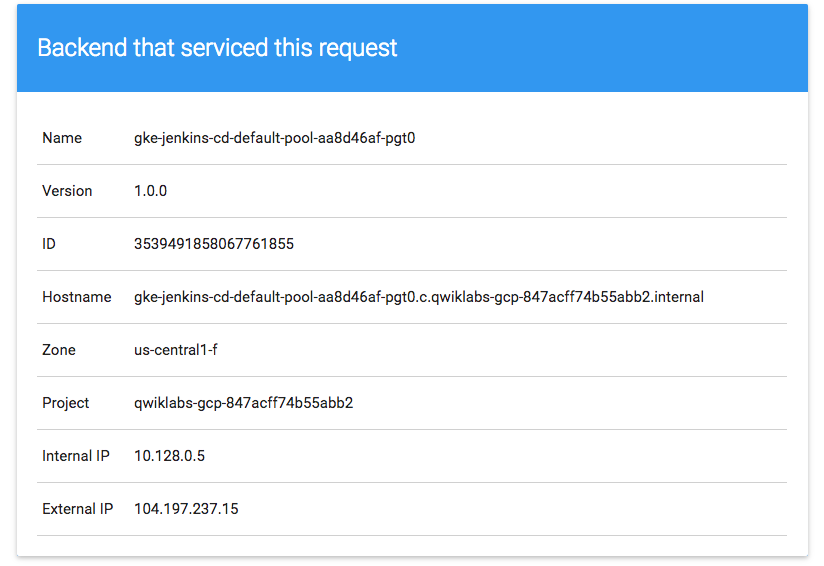
It can take several minutes before you see the load balancer external IP address.

Output (do not copy):

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

gceme-frontend LoadBalancer 10.79.241.131 104.196.110.46 80/TCP 5h

1. Confirm that both services are working by opening the frontend EXTERNAL-IP in your browser



1. Check the version output of the service:
2. export FRONTEND\_SERVICE\_IP=$(kubectl get -o jsonpath="{.status.loadBalancer.ingress[0].ip}" --namespace=production services gceme-frontend)
3. curl http://$FRONTEND\_SERVICE\_IP/version

Output (do not copy):

1.0.0

You have successfully deployed the sample application!

Create a repository to host the sample app source code

1. Create your own copy of the gceme sample app in [Cloud Source Repository](https://cloud.google.com/source-repositories/docs/).
2. gcloud source repos create gceme

You can ignore the warning, you will not be billed for this repository.

Click *Check my progress* to verify the objective.

Create a repository

Check my progress

1. Initialize the sample-app directory as its own local Git repository:
2. git init
3. git config credential.helper gcloud.sh
4. Add a *git remote* for the new repo in CSR.
5. git remote add origin https://source.developers.google.com/p/$GOOGLE\_CLOUD\_PROJECT/r/gceme
6. Review the *remotes* created:
7. git remote -v
8. origin https://source.developers.google.com/p/qwiklabs-gcp-.../r/gceme (fetch)
9. origin https://source.developers.google.com/p/qwiklabs-gcp-.../r/gceme (push)
10. Set the username and email address for your Git commits.
11. git config --global user.email "$USER@qwiklabs.net"
12. git config --global user.name "$USER"
13. Review the username and email address.
14. git config --global -l
15. Add, commit, and push the files to CSR:
16. git add .
17. git commit -m "Initial commit"
18. git push origin master

You have successfully pushed the sample application source to a new repo. Next, you will set up a pipeline for deploying your changes continuously and reliably.

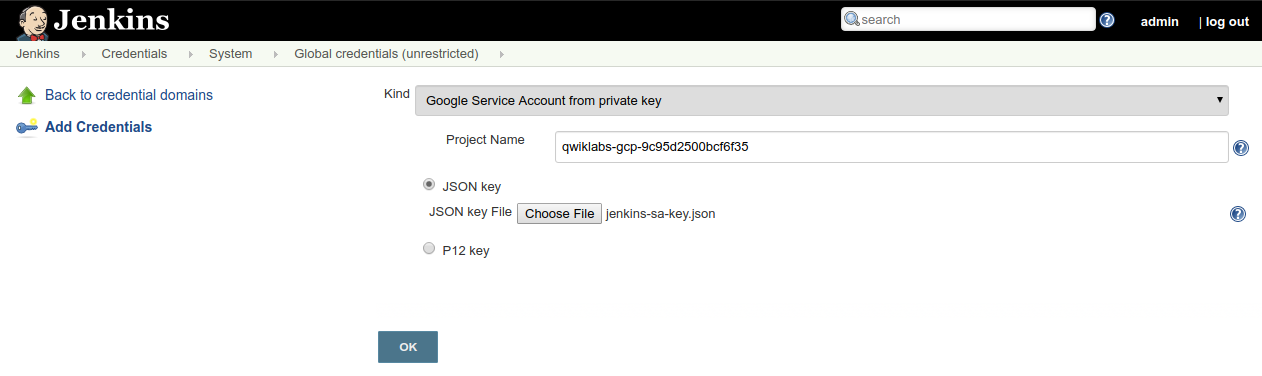
**Create the Jenkins Pipeline**

You'll now use Jenkins to define and run a pipeline that will test, build, and deploy your copy of gceme to your Kubernetes cluster.

Add your service account credentials

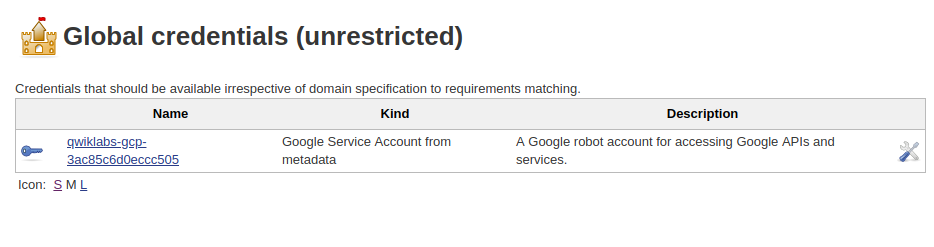
Configure your credentials to allow Jenkins to access the code repository. Jenkins will use the service account credentials created earlier to download code from the Cloud Source Repositories.

1. In the **Jenkins UI**, click **Credentials** in the left navigation.
2. Click **(global)**.
3. Click **Add Credentials** in the left navigation.
4. Select **Google Service Account from private key** from the **Kind** drop-down.
5. Enter the **Project Name** from your project
6. Leave **JSON key** selected, and click **Choose File**.
7. Select the jenkins-sa-key.json file downloaded earlier, then click **Open**.



1. Click **OK**

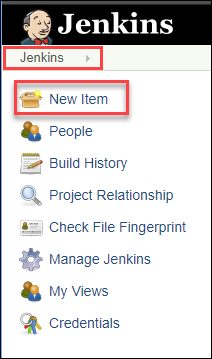
The global credential has been added. The name of the credential is the GCP Project ID found in the CONNECTION DETAILS section of the lab.



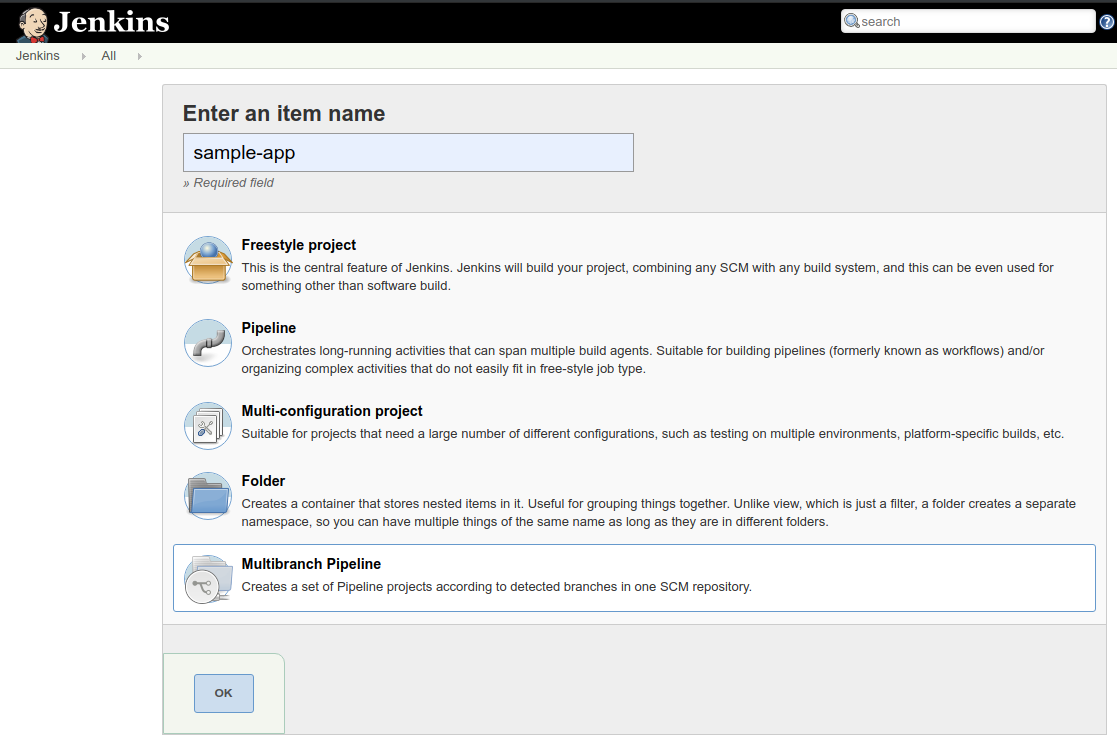
Create the Jenkins job

Navigate to your Jenkins user interface and follow these steps to configure a Pipeline job.

1. Click **Jenkins** > **New Item** in the left navigation:



1. Use **sample-app** as the item name, then choose the **Multibranch Pipeline** option and click **OK**.



1. On the next page, in the **Branch Sources** section, click **Add Source** and select **git**.
2. Paste the **HTTPS clone URL** of your gceme repo in Cloud Source Repositories into the **Project Repository** field.

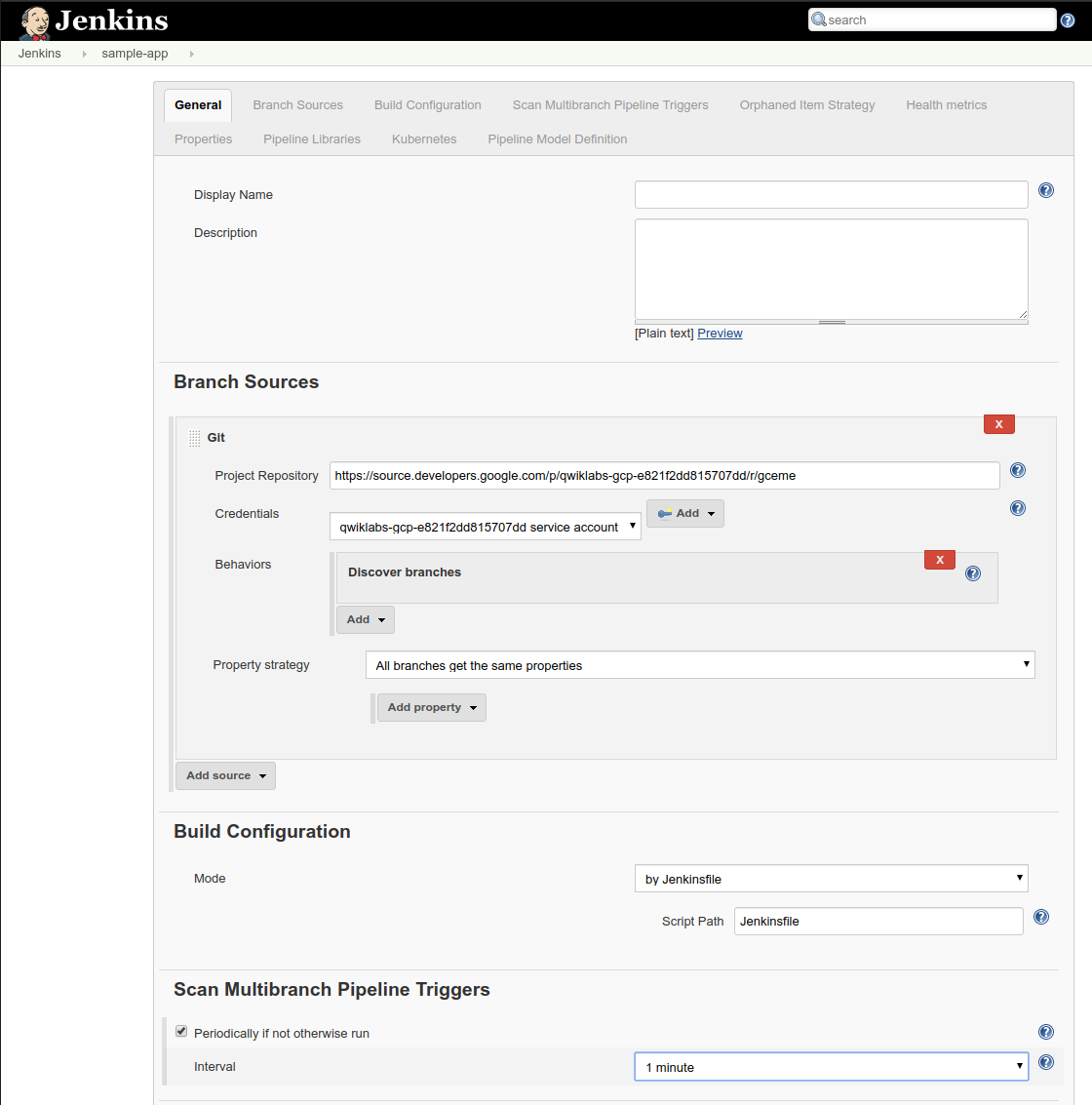
Use the output of this command:

echo https://source.developers.google.com/p/$GOOGLE\_CLOUD\_PROJECT/r/gceme

1. From the **Credentials** drop-down, select the name of the credentials you created when adding your service account in the previous steps. It should have the format PROJECT\_ID service account.

If you did not paste your project repository correctly, you will only see 1 choice: - none -. Try fixing your repo name until you see another credential name.

1. Under **Scan Multibranch Pipeline Triggers** section, check the **Periodically if not otherwise run** box and set the **Interval** value to 1 minute.



1. Click **Save** leaving all other options with their defaults.

After you complete these steps, a job named Branch indexing runs. This meta-job identifies the branches in your repository and ensures changes haven't occurred in existing branches. If you click sample-app in the top left, the master job should be seen.

The first run of the master job will fail until you make a few code changes in the next step.

You have successfully created a Jenkins pipeline! Next, you'll create the development environment for continuous integration.

**Creating the Development Environment**

Development branches are a set of environments your developers use to test their code changes before submitting them for integration into the live site. These environments are scaled-down versions of your application, but need to be deployed using the same mechanisms as the live environment.

Creating a development branch

To create a development environment from a feature branch, you can push the branch to the Git server and let Jenkins deploy your environment.

Create a development branch and push it to the Git server:

git checkout -b new-feature

Modifying the pipeline definition

The Jenkinsfile that defines that pipeline is written using the [Jenkins Pipeline Groovy syntax](https://jenkins.io/doc/book/pipeline/syntax/). Using a Jenkinsfile allows an entire build pipeline to be expressed in a single file that lives alongside your source code. Pipelines support powerful features like parallelization and require manual user approval.

In order for the pipeline to work as expected, you need to modify the Jenkinsfile to set your project ID.

1. Open the Jenkinsfile in your terminal editor, for example vi:
2. vi Jenkinsfile
3. Start insert-mode in vi:
4. i
5. Update the correct **PROJECT** environment variable.

Look for the line that begins with PROJECT =. Replace the string with your PROJECT\_ID, found in the CONNECTION DETAILS section of the lab. You can also use the results of gcloud config get-value project.

**Be sure to replace REPLACE\_WITH\_YOUR\_PROJECT\_ID with your project name.**

1. Save the Jenkinsfile file:

For vi, hit **Esc** then, :wq, for write-quit.

:wq

Modify the application in your new-feature branch

To demonstrate changing the application, change the gceme cards from **blue** to **orange**.

1. Open html.go:
2. vi html.go
3. Start insert-mode in vi:
4. i
5. Change the two instances of <div class="card blue"> to the following:
6. <div class="card orange">
7. Save the html.go file:

For vi, hit **Esc** then, :wq, for write-quit.

:wq

1. Open main.go:
2. vi main.go
3. Start insert-mode in vi:
4. i
5. Change the version string:

Update this:

const version string = "1.0.0"

To this:

const version string = "2.0.0"

1. Save the main.go:

For vi, hit **Esc** then, :wq, for write-quit.

:wq

**Kick off Deployment for new-feature in development**

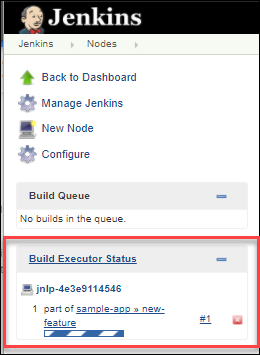
Build and deploy the changes

1. Commit and push your changes:
2. git add -A
3. git commit -m "Version 2.0.0"
4. git push origin new-feature

This will kick off a build of your development environment.

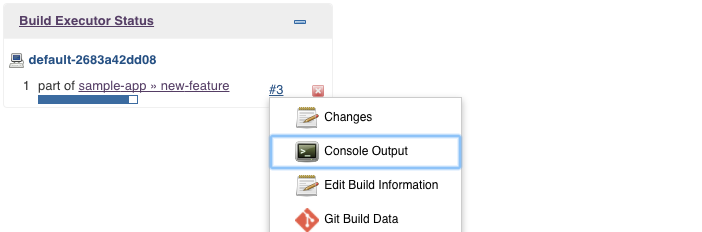
1. Navigate back to the Jenkins UI and review the new-feature job:

After the change is pushed to the Git repository, you can see that your build started automatically for the new-feature branch. It can take up to a minute for the changes to be picked up by Jenkins.



1. After the build is running, follow the build output:

Click the down arrow next to the **build#** in the **Build Executor Status** area. Select **Console output**.



Track the output of the build for a few minutes. When you see Finished: SUCCESS, your new-feature branch has been deployed to your cluster.

Verify the changes

In a development scenario, you wouldn't use a public-facing load balancer to service your application. To help secure your application, you can use port-forwarding check your deployed changes, without exposing your service to the public internet.

If you missed the build in Build Executor, not to worry. Just go to the Jenkins UI > **sample-app**. Verify that the new-feature pipeline has been created.

1. Once the build is complete, start port-forwarding on 8001 in the background:
2. export DEV\_POD\_NAME=$(kubectl get pods -n new-feature -l "app=gceme,env=dev,role=frontend" -o jsonpath="{.items[0].metadata.name}")
3. kubectl port-forward -n new-feature $DEV\_POD\_NAME 8001:80 >> /dev/null &
4. Verify that your application is accessible:

Send a request to localhost; kubectl will forward it to your service.

curl http://localhost:8001/api/v1/namespaces/new-feature/services/gceme-frontend:80/

You should see it respond with 2.0.0, which is the version that is now running.

You have set up the development environment! Next, you will build on what you learned in the previous module by deploying a **canary** release to test out a new feature.

**Deploying a Canary Release**

You've verified that your app is running the latest code in the development environment, so let's deploy that code to the canary environment.

1. Create a canary branch and push it to the Git server:
2. git checkout -b canary
3. git push origin canary

In the Jenkins UI, you should notice the **canary** pipeline has kicked off.

1. Once complete, check the service URL for the new version:

Ensure that some of the traffic is being served by your new version. You should see about 1 in 5 requests (in no particular order) returning version 2.0.0.

export FRONTEND\_SERVICE\_IP=$(kubectl get -o jsonpath="{.status.loadBalancer.ingress[0].ip}" --namespace=production services gceme-frontend)

while true; do curl http://$FRONTEND\_SERVICE\_IP/version; sleep 1; done

If you keep seeing 1.0.0, try checking and re-running the above commands again. Eventually, you should verify 2.0.0 is returned at times.

1. End the while loop, with **Ctrl+C**.

That's it! You have deployed a canary release. Next, you will deploy the new version to production.

**Deploying to production**

Now that the canary release was successful, and you haven't heard any customer complaints, deploy to the rest of your production fleet.

1. Change to the master branch, merge canary, and push to Git:
2. git checkout master
3. git merge canary
4. git push origin master
5. Return to the Jenkins UI, and look for the master pipeline to kick off.
6. Once complete, check the external service URL:

Ensure that **all** of the traffic is being served by your new version, 2.0.0.

while true; do curl http://$FRONTEND\_SERVICE\_IP/version; sleep 1; done

Once again, if you see responses of 1.0.0, try running the above commands again.

1. Stop the command by pressing **Ctrl+C**.

Output (do not copy):

gcpstaging9854\_student@qwiklabs-gcp-df93aba9e6ea114a:~/continuous-deployment-on-kubernetes/sample-app$ while true; do curl http://$FRONTEND\_SERVICE\_IP/version; sleep 1; done

2.0.0

2.0.0

2.0.0

2.0.0

2.0.0

2.0.0

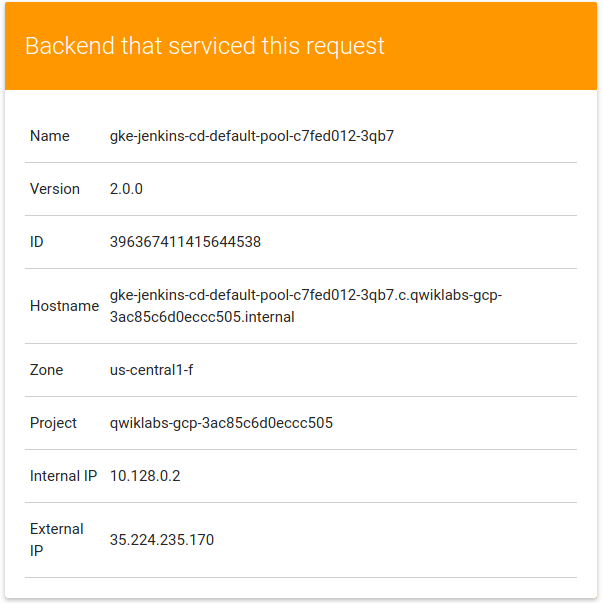
^C

1. Navigate to the External-IP, where gceme displays the info cards.

The card color changed from blue to orange. Here's the command again to get the External-IP address so you can check it out:

kubectl get service gceme-frontend -n production

**Output:**



**You're done!**

Awesome job, you have successfully deployed your application to production!

**Congratulations!**

This concludes this hands-on lab deploying and working with Jenkins in Kubernetes Engine to enable a Continuous Delivery / Continuous Deployment pipeline.

You've had the opportunity to deploy a **significant** DevOps tool in Kubernetes Engine and configure it for production use. You've worked with the kubectl command-line tool and deployment configurations in YAML files, and have learned a bit about setting up Jenkins pipelines for a development / deployment process.

With this practical hands-on experience, you should feel comfortable applying these tools in your own DevOps shop.

**End your lab**