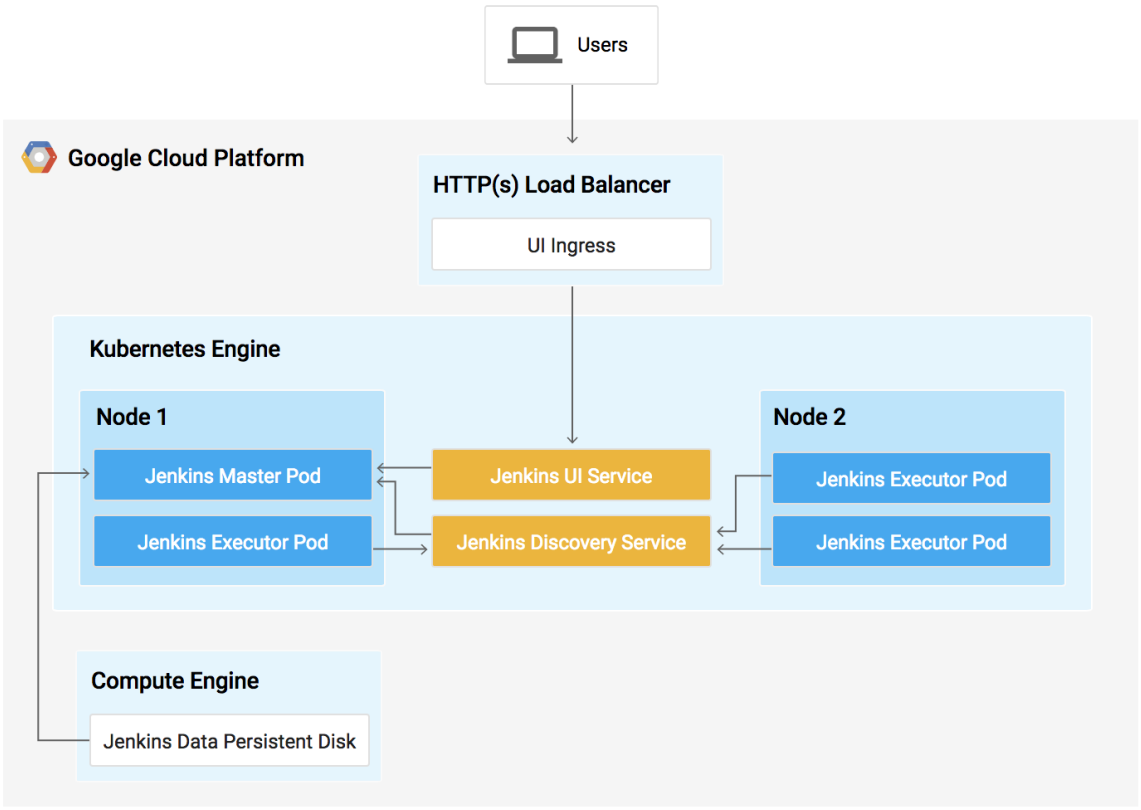
Jenkins on GCP Demo:

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:



Jenkins on GCP:
https://cloud.google.com/architecture/jenkins-on-kubernetes-engine

You will complete the following tasks during the demo:

- Provision a Jenkins application into a Kubernetes Engine Cluster
- Set up your Jenkins application using Helm Package Manager
- Explore the features of a Jenkins application
- Create and exercise a Jenkins pipeline

What is Kubernetes Engine?

Kubernetes Engine is Google Cloud's hosted version of Kubernetes - 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 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 front, this load balancer will always set your users on the fastest possible path to an application instance.

Clone the repository

- To get set up, open a new session in Cloud Shell and run the following command to set your zone us-west1-c:

```
gcloud services enable compute.googleapis.com

gcloud config set compute/zone us-west1-c
```

- Then copy the lab's sample code:

```
gsutil cp gs://spls/gsp051/continuous-deployment-on-kubernetes.zip .
unzip continuous-deployment-on-kubernetes.zip
```

- Now change to the correct directory:

```
cd continuous-deployment-on-kubernetes
```

Provisioning Jenkins and Creating a Kubernetes Cluster:

- Enable container service:

```
gcloud services enable container.googleapis.com
```

- Now, run the following command to provision a Kubernetes cluster:

```
gcloud container clusters create jenkins-cd \
--num-nodes 2 \
--machine-type n1-standard-2 \
--scopes "https://www.googleapis.com/auth/source.read_write,cloud-platform" \
--no-enable-ip-alias
```

This step can take up to several minutes to complete. The extra scopes enable Jenkins to access Cloud Source Repositories and Google Container Registry.

- Confirm that your cluster is running by executing the following command:

gcloud container clusters list

- Get the credentials for your cluster:

gcloud container clusters get-credentials jenkins-cd

- Check kubectl command on cluster:

kubectl cluster-info

Provisioning the Helm Charts:

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.

- Add Helm's stable chart repo:

helm repo add jenkins https://charts.jenkins.io

- Ensure the repo is up to date:

helm repo update

Configure and Install Jenkins:

When installing Jenkins, a `values` file can be used as a template to provide values that are necessary for setup.
You will use a custom `values` file to automatically configure your Kubernetes Cloud and add the following necessary plugins:

- Kubernetes:1.29.4
- Workflow-multibranch:latest
- Git:4.7.1
- Configuration-as-code:1.51
- Google-oauth-plugin:latest

- Google-source-plugin:latest
- Google-storage-plugin:latest

This will allow Jenkins to connect to your cluster and your GCP project.

- Download the custom `values` file:

gsutil cp gs://spls/gsp330/values.yaml jenkins/values.yaml

- Use the Helm CLI to deploy the chart with your configuration settings:

helm install cd jenkins/jenkins -f jenkins/values.yaml --wait

This command may take a couple minutes to complete.

- Ensure the Jenkins pod goes to the `Running` state and the container is in the READY state:

kubectl get pods

- Configure the Jenkins service account to be able to deploy to the cluster:

kubectl create clusterrolebinding jenkins-deploy --clusterrole=cluster-admin --serviceaccount=default:cd-jenkins

Following will be the output:

```
clusterrolebinding.rbac.authorization.k8s.io/jenkins-deploy created
```

- Run the following command to setup port forwarding to the Jenkins UI from the Cloud Shell:

export POD_NAME=$(kubectl get pods --namespace default -l "app.kubernetes.io/component=jenkins-master" -l "app.kubernetes.io/instance=cd" -o jsonpath="{.items[0].metadata.name}")
kubectl port-forward $POD_NAME 8080:8080 >> /dev/null &

Press Enter if prompt is hung

- Now, check that the Jenkins Service was created properly:

kubectl get svc

Example Output:

```
NAME              CLUSTER-IP     EXTERNAL-IP   PORT(S)     AGE
cd-jenkins        10.35.249.67                 8080/TCP    3h
cd-jenkins-agent  10.35.248.1                  50000/TCP   3h
kubernetes        10.35.240.1                  443/TCP     9h
```

You are using the Kubernetes Plugin so that our builder nodes will be automatically launched as necessary when the Jenkins master requests them. Upon completion of their work, they will 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 is exposed using a ClusterIP so that it is not accessible from outside the cluster.

Connect to Jenkins:

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

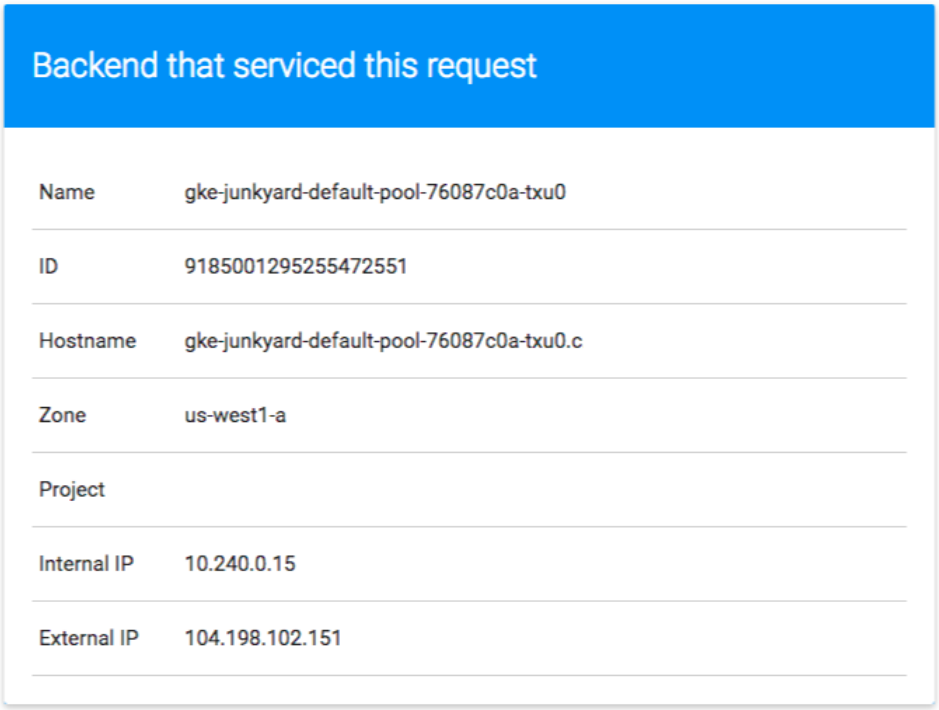
printf $(kubectl get secret cd-jenkins -o jsonpath="{.data.jenkins-admin-password}" | base64 --decode);echo

- To get to the Jenkins user interface, click on the **Web Preview** button in Cloud Shell, then click **Preview on port 8080**:
- You should now be able to log in with username `admin` and your auto-generated password.

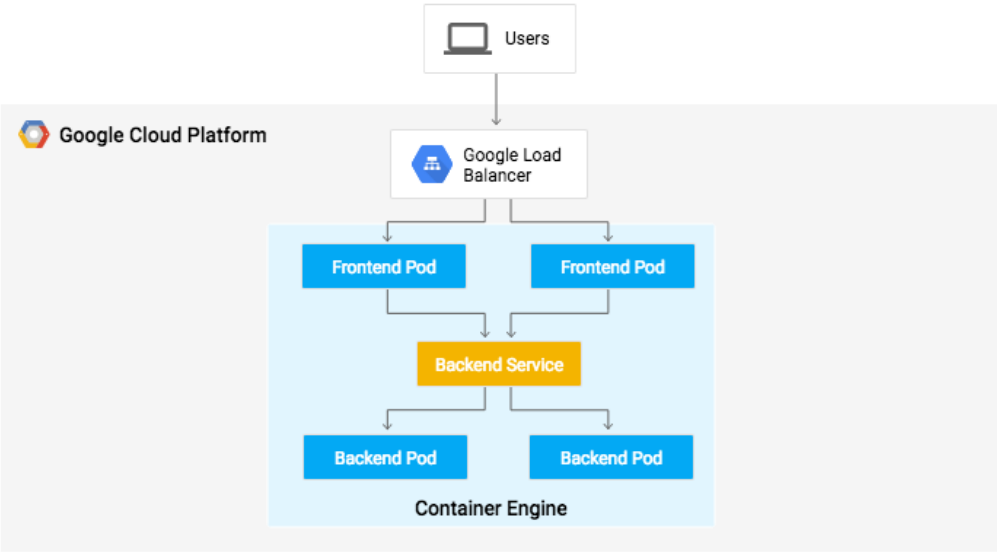
You now have Jenkins set up in your Kubernetes cluster! Jenkins will drive your automated CI/CD pipelines in the next sections.

Understanding 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 a Compute Engine instance, the app displays the instance's metadata in an info card.



- The application mimics a microservice by supporting two operation modes.
    - In **backend mode**: gceme listens on port 8080 and returns Compute Engine instance metadata in JSON format.
    - In **frontend mode**: gceme queries the backend gceme service and renders the resulting JSON in the user interface.



Deploying the Application:

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.

- In Google Cloud Shell, navigate to the sample application directory:

cd sample-app

- Create the Kubernetes namespace to logically isolate the deployment:

kubectl create ns production

- Create the production and canary deployments, and the services using the `kubectl apply` commands:

kubectl apply -f k8s/production -n production

kubectl apply -f k8s/canary -n production

kubectl apply -f k8s/services -n production

- 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.

  Scale up the production environment frontends by running the following command:

kubectl scale deployment gceme-frontend-production -n production --replicas 4

- Now confirm that you have 5 pods running for the frontend, 4 for production traffic and 1 for canary releases (changes to the canary release will only affect 1 out of 5 (20%) of users):

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

- Also confirm that you have 2 pods for the backend, 1 for production and 1 for canary:

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

- Retrieve the external IP for the production services:

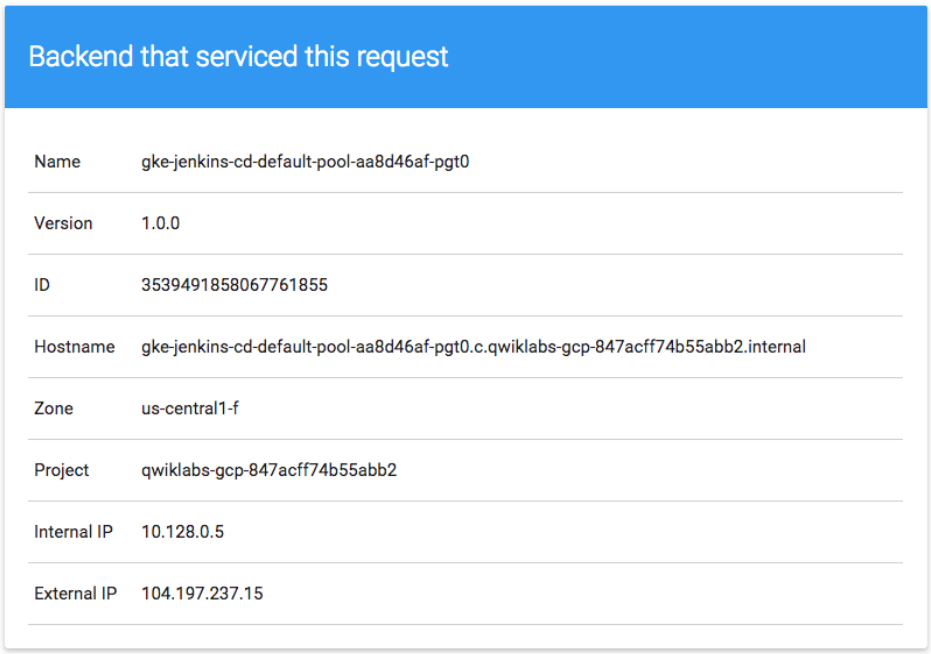
kubectl get service gceme-frontend -n production

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

Example Output:

```
NAME             TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)
AGE
gceme-frontend   LoadBalancer   10.79.241.131  104.196.110.46  80/TCP    5h
```

- Paste **External IP** into a browser to see the info card displayed on a card—you should get a similar page:



- Now, store the *frontend service* load balancer IP in an environment variable for use later:

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

- Confirm that both services are working by opening the frontend external IP address in your browser. Check the version output of the service by running the following command (it should read 1.0.0):

```
curl http://$FRONTEND_SERVICE_IP/version
```

You have successfully deployed the sample application! Next, you will set up a pipeline for deploying your changes continuously and reliably.

Creating the Jenkins Pipeline

Creating a repository to host the sample app source code

- Create a copy of the gceme sample app and push it to a Cloud Source Repository:

gcloud services enable sourcerepo.googleapis.com
gcloud services enable cloudbuild.googleapis.com
gcloud source repos create default

- Run: git init

- Initialize the sample-app directory as its own Git repository:

git config credential.helper gcloud.sh

- Run the following command:

git remote add origin
https://source.developers.google.com/p/$DEVSHELL_PROJECT_ID/r/default

- Set the username and email address for your Git commits. Replace [EMAIL_ADDRESS] with your Git email address and [USERNAME] with your Git username:

git config --global user.email "Email of your free tier account"

git config --global user.name "Your Name"

- Add, commit, and push the files:

git add .

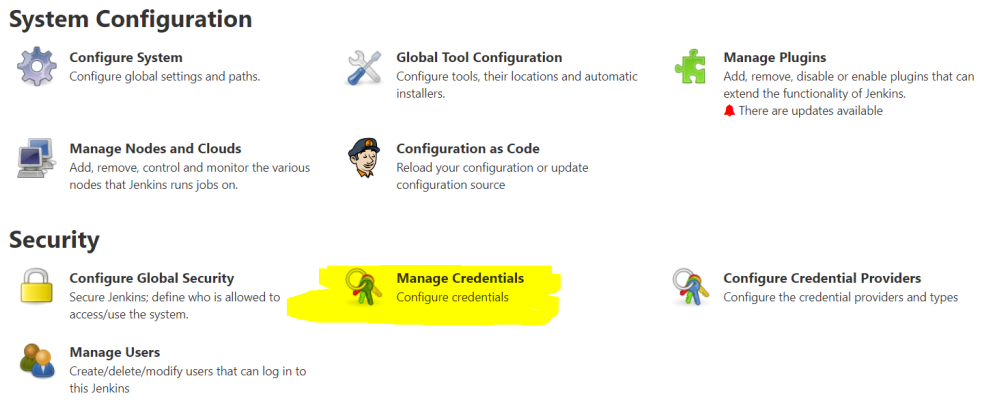
git commit -m "Initial commit"

git push origin master

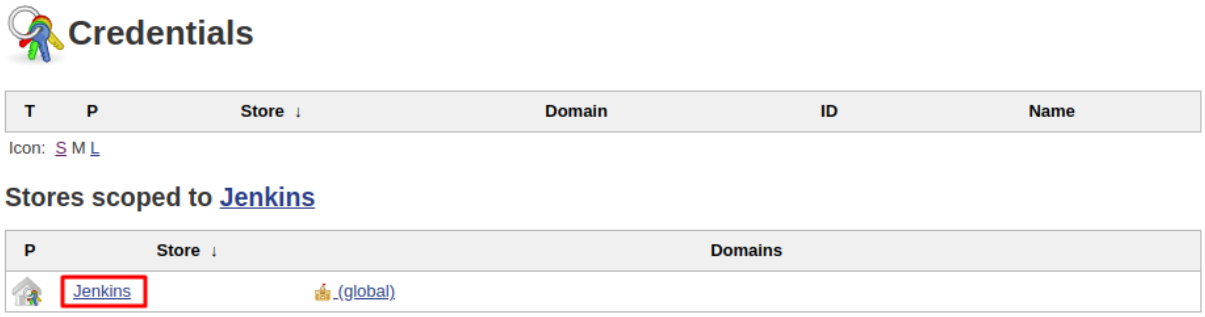
Adding your service account credentials

Configure your credentials to allow Jenkins to access the code repository. Jenkins will use your cluster's service account credentials in order to download code from the Cloud Source Repositories.

Step 1: In the Jenkins user interface, click **Manage Jenkins** in the left navigation then click **Manage Credentials**.

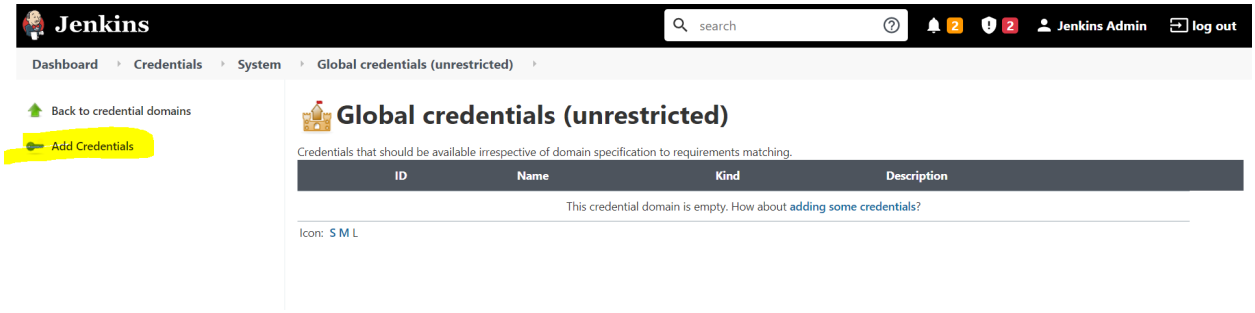


Step 2: Click **Jenkins**

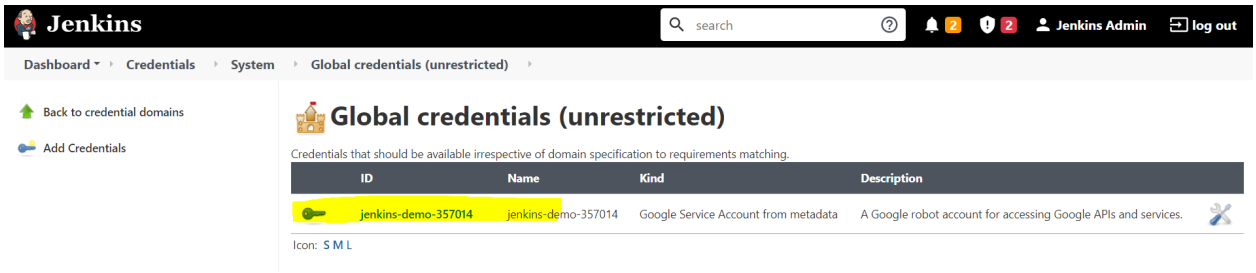


Step 3: Click **Global credentials (unrestricted)**.

Step 4: Click **Add Credentials** in the left navigation.



Step 5: Select **Google Service Account from metadata** from the **Kind** drop-down and click **OK**.

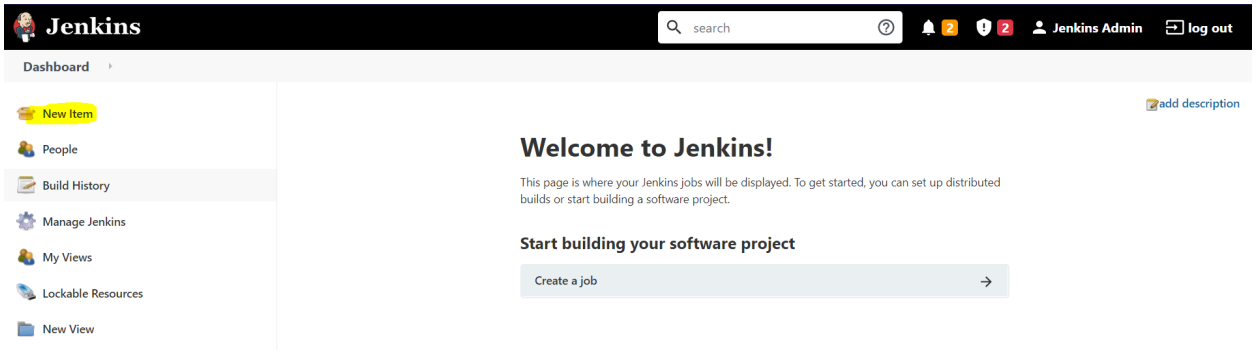


The global credentials has been added. The name of the credential is the `Project ID` found in the `CONNECTION DETAILS` section of the lab.

Creating the Jenkins job

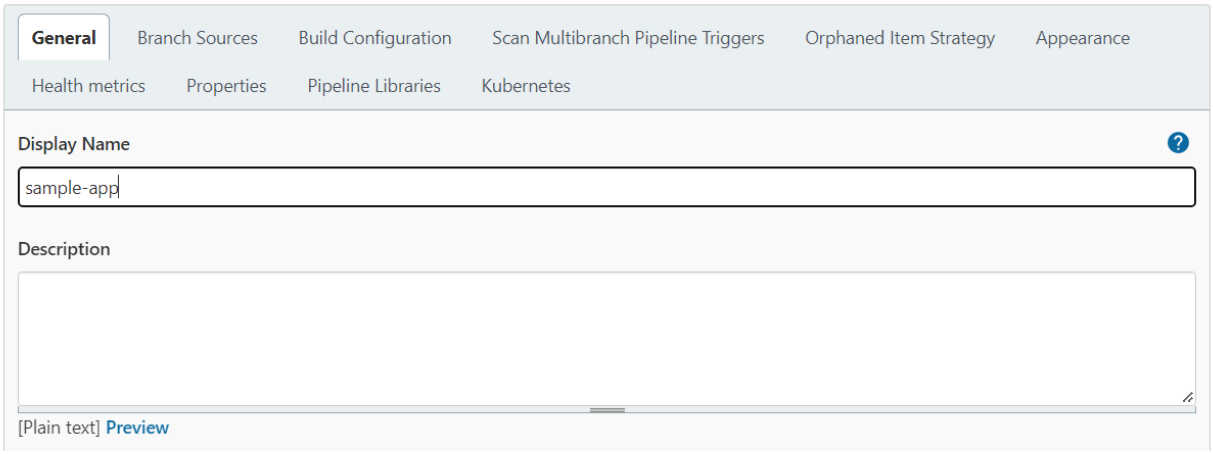
Navigate to your Jenkins user interface and follow these steps to configure a Pipeline job.

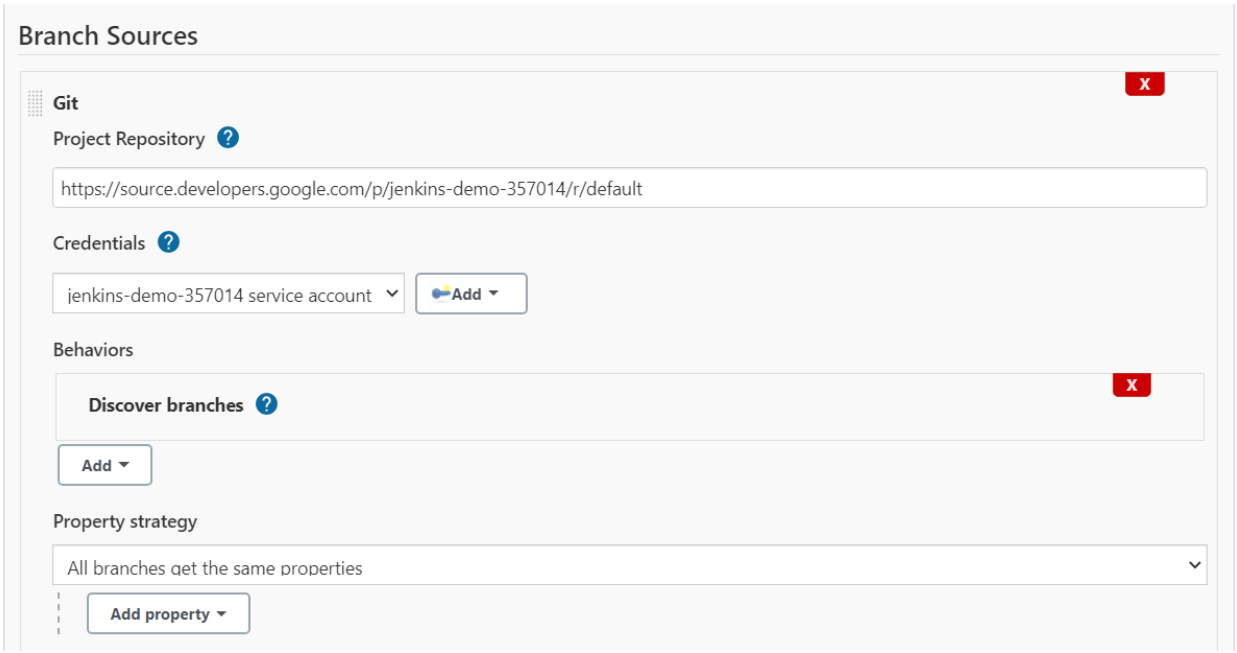
Step 1: **Click Dashboard** > **New Item** in the left panel.



Step 2: Name the project **sample-app**, then choose the **Multibranch Pipeline** option and click **OK**.
Step 3: On the next page, in the **Branch Sources** section, click **Add Source** and select **git**.
Step 4: Paste the **HTTPS clone URL** of your sample-app repo in Cloud Source Repositories into the **Project Repository** field. Replace [`PROJECT_ID`] with your **Project ID**:
https://source.developers.google.com/p/[PROJECT_ID]/r/default

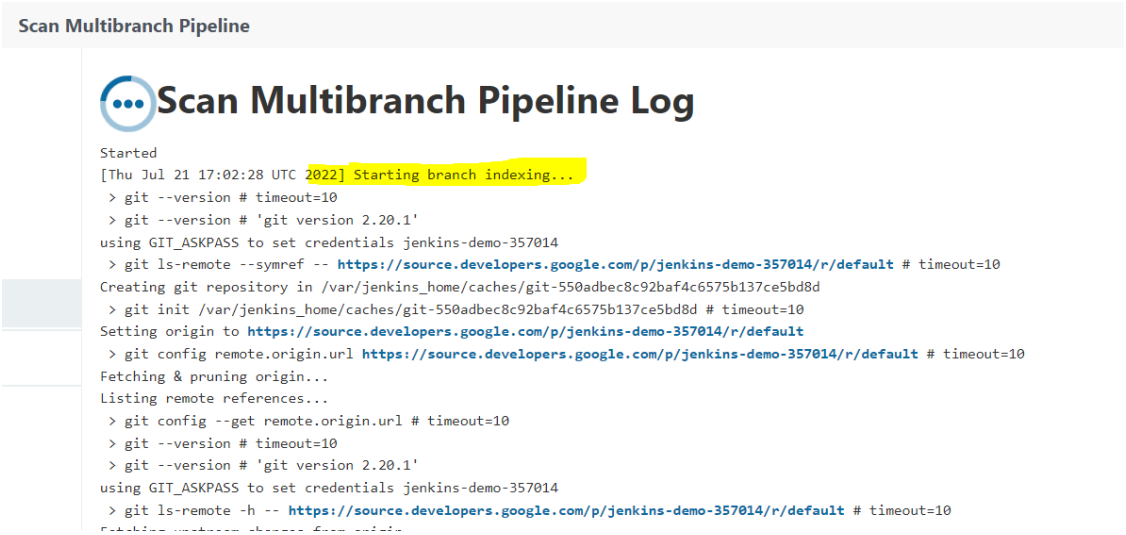
Step 5: From the **Credentials** drop-down, select the name of the credentials you created when adding your service account in the previous steps.
Step 6: Under **Scan Multibranch Pipeline Triggers** section, check the **Periodically if not otherwise run** box and set the **Interval** value to 1 minute.
Step 7: Your job configuration should look like this:


General
Branch Sources
Build Configuration
Scan Multibranch Pipeline Triggers
Orphaned Item Strategy
Appearance
Health metrics
Properties
Pipeline Libraries
Kubernetes
Display Name
sample-app
Description
[Plain text] Preview


Branch Sources
Git
Project Repository
https://source.developers.google.com/p/jenkins-demo-357014/r/default
Credentials
jenkins-demo-357014 service account
Add
Behaviors
Discover branches
Add
Property strategy
All branches get the same properties
Add property


Build Configuration
Mode
by Jenkinsfile
Script Path
Jenkinsfile
Scan Multibranch Pipeline Triggers
Periodically if not otherwise run
Interval
1 minute

Step 8: 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.



Note: The first run of the master job might 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. 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.

- Open the Jenkinsfile in your terminal editor, for example vi:

```
vi Jenkinsfile
```

- Start the editor:

```
i
```

- Add your `PROJECT_ID` to the `REPLACE_WITH_YOUR_PROJECT_ID` value. (Your `PROJECT_ID` is your Project ID found in the `CONNECTION DETAILS` section of the lab. You can also run `gcloud config get-value project` to find it.

- Check the `CLUSTER_ZONE` = same as GKE cluster's zone

- Save the `Jenkinsfile` file: hit **Esc** then (for vi users):

```
:wq
```

Modify the site

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

- Open html.go:

vi html.go

- Start the editor:

i

- Change the two instances of `<div class="card blue">` with following:

- Save the html.go file: press **Esc** then:

:wq

- Open main.go:

vi main.go

- Start the editor:

i

- The version is defined in this line:
  From: const version string = "1.0.0" to const version string = "2.0.0"

- Save the main.go file one more time: **Esc** then:

:wq

Kick off Deployment

- Commit and push your changes:

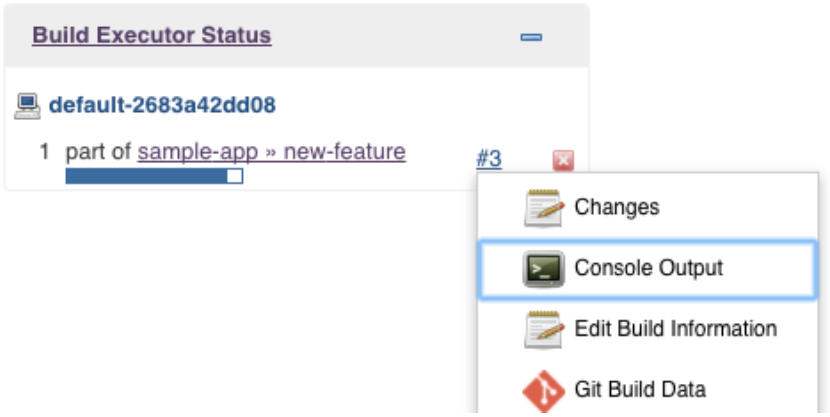
git add Jenkinsfile html.go main.go

git commit -m "Version 2.0.0"

git push origin new-feature

This will kick off a build of your development environment.
After the change is pushed to the Git repository, navigate to the Jenkins user interface where you can see that your build started for the `new-feature` branch. It can take up to a minute for the changes to be picked up.

- After the build is running, click the down arrow next to the **build** in the left navigation and select **Console output**:



- Track the output of the build for a few minutes and watch for the `kubectl --namespace=new-feature apply...` messages to begin. Your new-feature branch will now be deployed to your cluster.

Note: In a development scenario, you wouldn't use a public-facing load balancer. To help secure your application, you can use kubectl proxy. The proxy authenticates itself with the Kubernetes API and proxies requests from your local machine to the service in the cluster without exposing your service to the Internet.

If you didn't see anything in `Build Executor`, not to worry. Just go to the Jenkins homepage > sample app. Verify that the `new-feature` pipeline has been created.

- Once that's all taken care of, start the proxy in the background: Go to Cloud shell and run:

kubectl proxy &

- If it stalls, press **Ctrl + C** to exit out. Verify that your application is accessible by sending a request to localhost and letting kubectl proxy forward it to your service:

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

You should see it respond with 2.0.0, which is the version that is now running. If you receive a similar error:

```
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {
  },
  "status": "Failure",
  "message": "no endpoints available for service \"gceme-frontend:80\"",
  "reason": "ServiceUnavailable",
  "code": 503
```

- It means your frontend endpoint hasn't propagated yet—wait a little bit and try the curl command again. Move on when you get the following output:

2.0.0

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 have verified that your app is running the latest code in the development environment, so now deploy that code to the canary environment.

- Create a canary branch and push it to the Git server:

```
git checkout -b canary
```

```
git push origin canary
```

- In Jenkins, you should see the **canary** pipeline has kicked off. Once complete, you can check the service URL to 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 running the above commands again. Once you've verified that the above works, end the command with **Ctrl + C**.

You have deployed a canary release. Next you will deploy the new version to production.

Deploying to production

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

- Create a canary branch and push it to the Git server:

```
git checkout master

git merge canary

git push origin master
```

In Jenkins, you should see the master pipeline has kicked off.

- *Once complete* (**which may take a few minutes**), you can check the service URL to ensure that all of the traffic is being served by your new 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
```

- Once again, if you see instances of 1.0.0 try running the above commands again. You can stop this command by pressing **Ctrl + C**.

Example output:

```
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
```

You can also navigate to site on which the gceme application 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