Implement DevOps Workflows in Google Cloud

TASK1 : CREATE THE LAB RESOURCES:

In this section, you initialize your Google Cloud project for the demo environment. You enable the required APIs, configure Git in Cloud Shell, create an Artifact Registry Docker repository, and create a GKE cluster to run your production and development applications on.

1. Run the following command to enable the APIs for GKE, Cloud Build, and GitHub Repositories:

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

2. Add the Kubernetes Developer role for the Cloud Build service account:

```
export PROJECT_ID=$(gcloud config get-value project)
gcloud projects add-iam-policy-binding $PROJECT_ID \
--member=serviceAccount:$(gcloud projects describe $PROJECT_ID \
--format="value(projectNumber)")@cloudbuild.gserviceaccount.com --
role="roles/container.developer"
```

3. In Cloud Shell, run the following commands to configure Git and GitHub:

```
4. curl -sS https://webi.sh/gh | sh
5. gh auth login
6. gh api user -q ".login"
7. GITHUB_USERNAME=$(gh api user -q ".login")
8. git config --global user.name "${GITHUB_USERNAME}"
9. git config --global user.email "${USER_EMAIL}"
10.  echo ${GITHUB_USERNAME}
   echo ${USER_EMAIL}
```

- Press ENTER to accept the default options.
- Read the instructions in the command output to log in to GitHub with a web browser.

When you have successfully logged in, your GitHub username appears in the output in Cloud Shell.

11. Create an Artifact Registry Docker repository named **my-repository** in the `REGION` region to store your container images.

12. Create a GKE Standard cluster named `hello-cluster` with the following configuration:

Setting	Value
Zone	`ZONE`
Release channel	**Regular**
Cluster version	`1.29` *or newer*
Cluster autoscaler	**Enabled**
Number of nodes	**3**
Minimum nodes	**2**
Maximum nodes	**6**

*gcloud container clusters create hello-cluster *
*--zone us-east1-d *
*--release-channel regular *
*--cluster-version latest *
*--enable-autoscaling *
*--num-nodes 3 *
*--min-nodes 2 *
*--max-nodes 6 *
*--enable-ip-alias *
*--enable-autorepair *
--enable-autoupgrade

6. Create the **`prod`** and **`dev`** namespaces on your cluster.

Get credentials to interact with your cluster

```
gcloud container clusters get-credentials hello-cluster --zone us-east1-d
```

Create **`prod`** and **`dev`** namespaces

```
kubectl create namespace prod
kubectl create namespace dev
```

TASK 2: CREATE A REPOSITORY IN GITHUB REPOSITORIES

In this task, you create a repository **sample-app** in GitHub Repositories and initialize it with some sample code. This repository holds your Go application code, and be the primary source for triggering builds.

1. Create an empty repository named **sample-app** in GitHub Repositories.

2. Clone the **sample-app** GitHub Repository in Cloud Shell.

cd ~

git clone https://github.com/<your-username>/sample-app.git

cd sample-app

1. Use the following command to copy the sample code into your `sample-app` directory:

```
cd ~
gsutil cp -r gs://spls/gsp330/sample-app/* sample-app
```

4. Run the following command, which will automatically replace the `<your-region>` and `<your-zone>` placeholders in the `cloudbuild-`

`dev.yaml` and `cloudbuild.yaml` files with the assigned region and zone of your project:

```
export REGION="REGION"
export ZONE="ZONE"
for file in sample-app/cloudbuild-dev.yaml sample-app/cloudbuild.yaml;
do
    sed -i "s/<your-region>/${REGION}/g" "$file"
    sed -i "s/<your-zone>/${ZONE}/g" "$file"
done
```

5. Create a GitHub repository with name `sample-app`

6. After creating repository make your first commit with the sample code added to your `sample-app` directory, and push the changes to the **master** branch.

7. Create a branch named **dev**. Make a commit with the sample code added to your `sample-app` directory and push the changes to the **dev** branch.

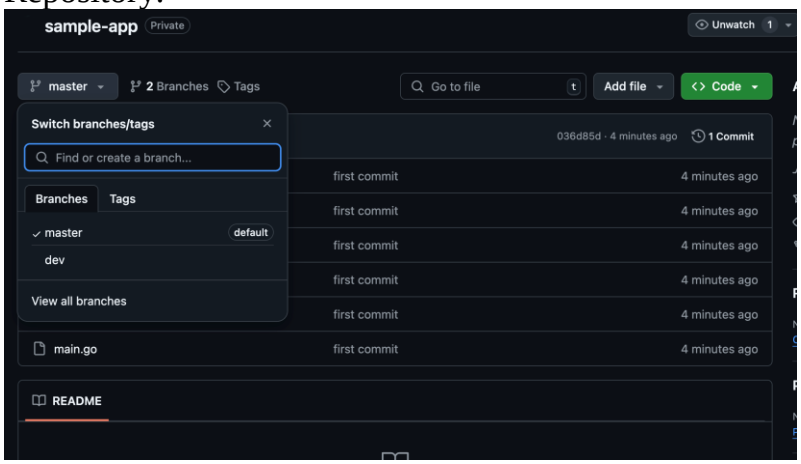
Commit and push to **`main`** (or **`master`**)

```
git add .
git commit -m "Initial commit with sample app code"
git branch -M main
git push origin main
```

Create and push **`dev`** branch

```
git checkout -b dev
git push origin dev
```

8. Verify you have the sample code and branches stored in the GitHub Repository.

9. 

TASK 3: CREATE THE CLOUD BUILD TRIGGERS

In this section, you create two Cloud Build Triggers.

- The first trigger listens for changes on the **`master`** branch and builds a **Docker image** of your application, pushes it to Google Artifact Registry, and deploys the latest version of the image to the **prod** namespace in your GKE cluster.

- The second trigger listens for changes on the **`dev`** branch and build a Docker image of your application and push it to Google Artifact Registry, and deploy the latest version of the image to the **dev** namespace in your GKE cluster.

1. Create a Cloud Build Trigger named **sample-app-prod-deploy** that with the following configurations:

    - Event: **Push to a branch**
    - Source:
        - Connect to a new repository and select the source code management provider: `GitHub (Cloud Build GitHub App)`
        - Choose the GitHub repository: `sample-app`
    - Branch: `^master$`
    - Cloud Build Configuration File: `cloudbuild.yaml`
2. Create a Cloud Build Trigger named **sample-app-dev-deploy** that with the following configurations:

    - Event: **Push to a branch**
    - Source, Choose the GitHub repository : `sample-app`
    - Branch: `^dev$`
    - Cloud Build Configuration File: `cloudbuild-dev.yaml`

After setting up the triggers, any changes to the branches trigger the corresponding Cloud Build pipeline, which builds and deploy the application as specified in the `cloudbuild.yaml` files.

TASK 4: DEPLOY THE FIRST VERSIONS OF THE APPLICATION

Build the first development deployment

1. In Cloud Shell, inspect the `cloudbuild-dev.yaml` file located in the **sample-app** directory to see the steps in the build process. In `cloudbuild-dev.yaml` file, replace the `<version>` on lines 9 and 13 with `v1.0`.

2. Navigate to the `dev/deployment.yaml` file and Update the `<todo>` on line 17 with the correct container image name. Also, replace the **`PROJECT_ID`** variable with actual project ID in the container image name.

Note: Make sure you have same container image name in **dev/deployment.yaml** and **cloudbuild-dev.yaml** file.

3. Make a commit with your changes on the **`dev`** branch and push changes to trigger the **sample-app-dev-deploy** build job.

   gcloud config get-value project

   Make sure the image name matches exactly what you used in cloudbuild-dev.yaml.

4 Verify your build executed successfully in **Cloud build History** page, and verify the **development-deployment** application was deployed onto the **`dev`** namespace of the cluster.

Commit and Push Changes

git add cloudbuild-dev.yaml dev/deployment.yaml

git commit -m "Updated image version and deployment config for dev deployment"

git push origin dev

This will trigger the sample-app-dev-deploy Cloud Build trigger.

Verify the Build and Deployment

Go to the Cloud Build History page.

Confirm the sample-app-dev-deploy trigger ran successfully.

Then, check that the app was deployed to the dev namespace:

kubectl get pods -n dev

kubectl get deployments -n dev

5 Expose the **development-deployment** deployment to a **LoadBalancer** service named `dev-deployment-service` on port 8080, and set the target port of the container to the one specified in the Dockerfile.

Expose the Deployment via LoadBalancer

*kubectl expose deployment development-deployment *

*--type=LoadBalancer *

*--name=dev-deployment-service *

*--port=8080 *

*--target-port=8080 *

--namespace=dev

8080 is the container port (check Dockerfile for confirmation, usually EXPOSE 8080).

6 Navigate to the Load Balancer IP of the service and add the `/blue` entry point at the end of the URL to verify the application is up and running. It should resemble something like the following: `http://34.135.97.199:8080/blue`.

Wait a few moments and then get the external IP:

kubectl get service dev-deployment-service -n dev

Look for the EXTERNAL-IP. Once it's available, open it in your browser:

http://[EXTERNAL-IP]:8080/blue

Build the first production deployment

1. Switch to the **`master`** branch. Inspect the `cloudbuild.yaml` file located in the **sample-app** directory to see the steps in the build process.
   In `cloudbuild.yaml` file, replace the `<version>` on lines **11** and **16** with `v1.0`.

2. Navigate to the `prod/deployment.yaml` file and update the `<todo>` on line 17 with the correct container image name. Also, replace the **`PROJECT_ID`** variable with actual project ID in the container image name.

Note: Make sure you have same container image name in **prod/deployment.yaml** and **cloudbuild.yaml** file.

3. Make a commit with your changes on the **`master`** branch and push changes to trigger the **sample-app-prod-deploy** build job.

Commit and push changes to **`master`**

```
git add cloudbuild.yaml prod/deployment.yaml
git commit -m "Updated production image version and deployment config"
git push origin master
```

This should automatically trigger the `sample-app-prod-deploy` Cloud Build trigger (assuming you already created it like the dev one).

Verify Cloud Build success

Go to Cloud Build History and check if the **prod deployment** was successful.

4 Verify your build executed successfully in **Cloud build History** page, and verify the **production-deployment** application was deployed onto the **`prod`** namespace of the cluster.

Verify deployment on GKE

Check deployments in the `prod` namespace:

```
kubectl get deployments -n prod
```

You should see `production-deployment`.

5 Expose the **production-deployment** deployment on the **`prod`** namespace to a **LoadBalancer** service named `prod-deployment-service` on port 8080, and set the target port of the container to the one specified in the Dockerfile.

```
kubectl expose deployment production-deployment \
 --type=LoadBalancer \
 --name=prod-deployment-service \
 --port=8080 \
 --target-port=8080 \
 --namespace=prod
```

Get the Load Balancer IP

kubectl get service prod-deployment-service -n prod

Wait until you see an external IP under EXTERNAL-IP.

6 Navigate to the Load Balancer IP of the service and add the `/blue` entry point at the end of the URL to verify the application is up and running. It should resemble something like the following: `http://34.135.245.19:8080/blue`.

http://<EXTERNAL-IP>:8080/blue

TASK 5: DEPLOY THE SECOND VERSIONS OF THE APPLICATION

Build the second development deployment

1. Switch back to the **`dev`** branch.

Note: Before proceeding, make sure you are on **dev** branch to create deployment for **dev** environment.

git checkout dev

2. In the `main.go` file, update the `main()` function to the following:

```
func main() {
        http.HandleFunc("/blue", blueHandler)
        http.HandleFunc("/red", redHandler)
        http.ListenAndServe(":8080", nil)
}
```

3. Add the following function inside of the `main.go` file:

```
func redHandler(w http.ResponseWriter, r *http.Request) {
        img := image.NewRGBA(image.Rect(0, 0, 100, 100))
        draw.Draw(img, img.Bounds(), &image.Uniform{color.RGBA{255, 0,
0, 255}}, image.ZP, draw.Src)
        w.Header().Set("Content-Type", "image/png")
        png.Encode(w, img)
}
```

4. Inspect the `cloudbuild-dev.yaml` file to see the steps in the build process. Update the version of the Docker image to `v2.0`.

5. Navigate to the `dev/deployment.yaml` file and update the container image name to the new version (`v2.0`).

6. Make a commit with your changes on the **`dev`** branch and push changes to trigger the **sample-app-dev-deploy** build job.

   Now, commit your changes to the `dev` branch and push them:

   ```
   git add main.go cloudbuild-dev.yaml dev/deployment.yaml
   git commit -m "Updated main.go to include /red endpoint and updated
   image to v2.0"
   git push origin dev
   ```

   This will trigger the **`sample-app-dev-deploy`** Cloud Build trigger and start the deployment process.

   7 Verify your build executed successfully in **Cloud build History** page, and verify the **development-deployment** application was deployed onto the `dev` namespace of the cluster and is using the `v2.0` image.

   Go to the Cloud Build History page.

   Verify that the **build** ran successfully and check for any errors.

   Once the build finishes, verify the **deployment** in the `dev` namespace:

   ```
   kubectl get deployments -n dev
   ```

Check that `development-deployment` is using the new `v2.0` image.

8 Navigate to the Load Balancer IP of the service and add the `/red` entry point at the end of the URL to verify the application is up and running. It should resemble something like the following: `http://34.135.97.199:8080/red`.

Get the **External IP** of the service:

```
kubectl get service dev-deployment-service -n dev
```

Once the `EXTERNAL-IP` is available, open the URL with the `/red` endpoint to verify that the application is running with the updated image:

http://<EXTERNAL-IP>:8080/red

Build the second production deployment

1. Switch to the **`master`** branch.
   Note: Before proceeding, make sure you are on **master** branch to create deployment for **master** environment.

```
git checkout master
```

2. In the `main.go` file, update the `main()` function to the following:

```
func main() {
  http.HandleFunc("/blue", blueHandler)
  http.HandleFunc("/red", redHandler)
  http.ListenAndServe(":8080", nil)
}
```

3. Add the following function inside of the `main.go` file:

```
func redHandler(w http.ResponseWriter, r *http.Request) {
  img := image.NewRGBA(image.Rect(0, 0, 100, 100))
  draw.Draw(img, img.Bounds(), &image.Uniform{color.RGBA{255, 0,
0, 255}}, image.ZP, draw.Src)
  w.Header().Set("Content-Type", "image/png")
  png.Encode(w, img)
}
```

4. Inspect the `cloudbuild.yaml` file to see the steps in the build process. Update the version of the Docker image to `v2.0`.

5. Navigate to the `prod/deployment.yaml` file and update the container image name to the new version (`v2.0`).

6. Make a commit with your changes on the `master` branch and push changes to trigger the **sample-app-prod-deploy** build job.

Now, commit your changes to the `master` branch and push them:

```
git add main.go cloudbuild.yaml prod/deployment.yaml
git commit -m "Updated main.go to include /red endpoint and updated
image to v2.0"
git push origin master
```

This will trigger the **`sample-app-prod-deploy`** Cloud Build trigger and start the deployment process.

7 Verify your build executed successfully in **Cloud build History** page, and verify the **production-deployment** application was deployed onto the **`prod`** namespace of the cluster and is using the `v2.0` image.

Go to the Cloud Build History page and verify that the **build** ran successfully and check for any errors.

Once the build finishes, verify the **production deployment** in the `prod` namespace:

```
kubectl get deployments -n prod
```

Check that `production-deployment` is using the new `v2.0` image.

Expose the `production-deployment` to a LoadBalancer service:

```
kubectl expose deployment production-deployment \
  --type=LoadBalancer \
  --name=prod-deployment-service \
  --port=8080 \
  --target-port=8080 \
  --namespace=prod
```

9 Verify the Application on the Load Balancer

After a few minutes, get the **External IP** of the service:

```
kubectl get service prod-deployment-service -n prod
```

Once the `EXTERNAL-IP` is available, open the URL with the `/red` endpoint to verify that the application is running with the updated image:

`http://<EXTERNAL-IP>:8080/red`

TASK 6: ROLL BACK THE PRODCTION DEPLOYMENT

In this section, you roll back the production deployment to a previous version.

1. Roll back the **production-deployment** to use the `v1.0` version of the application.

Hint: Using Cloud build history, you can easily rollback/rebuild the deployments with the previous versions.

2. Navigate to the Load Balancer IP of the service and add the `/red` entry point at the end of the URL of the production deployment and response on the page should be **`404`**.

1. Go to the **Cloud Build History** page.
2. Locate the **build that used version `v1.0`** — it will usually show in the logs or description something like:

   ```
   gcr.io/YOUR_PROJECT_ID/sample-app:v1.0
   ```

3. Click on the build entry for version `v1.0`.
4. In the top right, click the **"Rebuild"** button to re-run the build and redeploy version `v1.0` via the Cloud Build trigger.

   This will re-deploy the production application with the `v1.0` Docker image.

Verify Rollback in Kubernetes

After the build completes:

```
kubectl describe deployment production-deployment -n prod
```

Ensure the container image shown under `Containers:` is:

```
gcr.io/YOUR_PROJECT_ID/sample-app:v1.0
```

Confirm Rollback via Load Balancer

1. Get the **external IP** of the production service:

   ```
   kubectl get svc prod-deployment-service -n prod
   ```

2. Open the URL in your browser:

```
http://<EXTERNAL-IP>:8080/red
```

If the rollback was successful and version `v1.0` is deployed (which doesn't include the `/red` endpoint), the browser should return:

```
404 page not found
```