HANDS-ON TUTORIALS

A Hands-on Tutorial to Continuous Deployment Pipelines for ML-based Web Apps on Google Cloud

A comprehensive guide to ML App Deployment using Flask

Dr. Sohini Roychowdhury Feb 17 · 7 min read

Machine Learning (ML) models typically leverage the capability to learn patterns from previously seen (training) data and apply them to predict the outcome for new (test) data. Deploying ML models as web apps can help test the efficacy of trained ML models by allowing access to multiple users and testing environments, thereby gathering test performance metrics. However, ML web app deployment in production can be a highly complex process that may need to ensure minimal down-time for users in the event of app update processes. Cloud-based deployment solutions such as Google Cloud Platform (GCP) have highly simplified the process of continuous integration and continuous deployment (CI/CD) through pipelines and triggers that can be designed to ensure sanity checks as well as the integrity of the integrated code base for updated application runs.

> *In this tutorial, we will review a detailed example wherein we deploy a ML model web app on GCP through a CD pipeline. To follow the steps from data analysis to final app deployment, fork the github repository at [1] and follow the steps below.*

The first step is understanding the data followed by ML model creation and subsequent deployment. For this step we refer to the file *model.py* under the folder *app_files*.

> *For this tutorial, we will be using the 50_Startup.csv data from [2] as shown in Fig. 1, where the goal is to train a regression model (f(X)) using the three features (x1,x2,x3) as (*R&D Spend, Administration Cost, Marketing Cost*), and predict the company* 'Profit' *(Y).*

	R&D Spend	Administration	Marketing	Profit
2	165349.2	136897.8	471784.1	192261.83
3	162597.7	151377.59	443898.53	191792.06
4	153441.51	101145.55	407934.54	191050.39
5	144372.41	118671.85	383199.62	182901.99

Fig 1: A snapshot of the training data set for regression data modeling.

Our goal is to estimate Profit as a function of its features as $f(x)=g(x_1,x_2,x_3)$, where $g(.)$ represents linear or non-linear combinations of the features x_1, x_2, x_3, respectively, that can be estimated from the training data.

While Linear Regression is the simplest solution to the above data set, one of the most scalable data models for large data sets continues to be ensemble models that are based on Decision Trees, since they enable data aggregation and expendability. In this work, we implement a non-linear Decision Tree-based Regression model as shown in Fig 2. To train and visualize the regression model, run the file *model.py* with code snippet shown below.

```
from sklearn.tree import DecisionTreeRegressor
reg=DecisionTreeRegressor(max_depth=5)
reg.fit(X,y)
y_pred=reg.predict(X)
print(reg.score)
```

model.py hosted with ❤ by GitHub view raw

A Decision Tree Regressor proceeds training by partitioning sub-regions or blocks in the feature subspace such that similar samples flock together to minimize the residual sum

respective sub-region as the estimated outcome [3]. The non-linear nature of the trained decision tree regression model with a maximum depth of 5 is shown in Fig. 2. The code to visualize the trained model across each feature is shown below.

```python
#To generate a best fit model
X_range=np.zeros((50,3))
y_range=np.zeros((50,))
for i in range(3):
    Xi=X[:,i]
    vals=plt.hist(Xi,49)
    plt.xlabel("Feature")
    plt.ylabel("Frequency")
    X_range[:,i]=np.transpose(vals[1])
y_range=model.predict(X_range)


# Plot the results
plt.figure()
plt.scatter(X[:,0], y, s=20, edgecolor="black", c="darkorange", label="train data")
plt.scatter(x_test[:,0], model.predict(x_test), s=30, color="yellowgreen", label="test d
plt.plot(X_range[:,0], y_range, color="cornflowerblue",
         label="Regression_model", linewidth=2)
plt.xlabel("R&D Cost")
plt.ylabel("Profit")
plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```

model.py hosted with ❤ by GitHub view raw

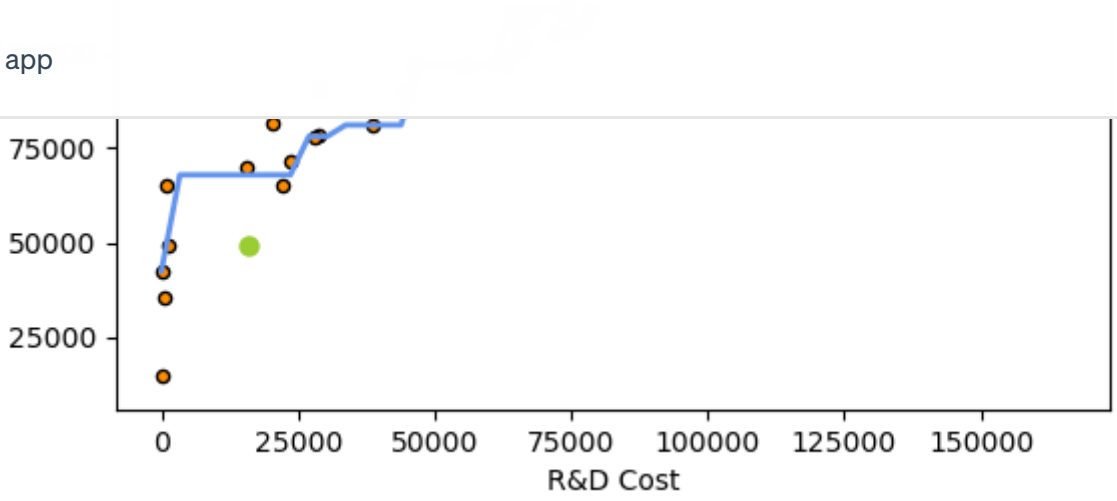


Fig 2: Example of Decision Tree Regression model fit on the data set visualized on the single feature of R&D cost. The test data set is [16000, 135000, 450000].

Step 1 Command at terminal: python model.py

Step 2: ML Web App using Flask

Having built the ML model, the next step is to package it and serve as a Web App. Here, we refer to the file *app.py* and html files under the *templates* folder contained in *app_files*.

We utilize Flask-based RESTful api from [4]. REpresentational State Transfer (REST) is a standard architectural design for web APIs and Flask has been the choice for python web API serves since its inception in 2010. The two major components for designing the Flask-API are [5]:

1. Flask library and POST HTTP method to serve the predictions.
2. Front end HTML files that are useful for users to interact with the ML model.

One significant change in the app.py file from general Flask-based python deployments in [4–5] is that the running *app.py* launches an API that runs at http://0.0.0:8000 instead of its default settings.

```
1  if __name__=='__main__':
2      app.run( debug=True, host='0.0.0.0', port=int(os.environ.get('PORT', 8080)) )
3           label="Regression_model", linewidth=2)
```

app.py hosted with ❤ by GitHub view raw

port 8080. For details on ways to debug a GCR app see [6].

Step 2 Command at terminal: python app.py

Step 3: Create a Docker Container for Google Cloud

Now that we have a functional web app, we need to package it for a Google Cloud Run. For this purpose, we will create a Docker container, as in [7] that is a standalone software and can be installed on any physical or virtual machine to run containerized applications. The Docker file for this tutorial, as shown below, must include 4 key components.

```
FROM python:3.6-slim-buster

RUN python -m pip install --upgrade pip
COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . /app
WORKDIR /app/app_files

CMD ["python", "app.py"]
```

Dockerfile hosted with ❤️ by GitHub view raw

1. The operating system needed by the application;
2. List of dependencies such as libraries and packages in the *requirements.txt* file;
3. Application files;
4. Command to launch the application.

Step 4: Create a YAML file for the Cloud Run

The final component required to run the CD pipeline is a **YAML** Ain't Markup Language file based on the application in [7]. YAML files follow the data serialization standard to build configuration settings for a CD pipeline. Details on building a YAML file for your

```yaml
steps:
  # Step 1: pull the container image if it is already built, if fails do not exit
  - name: 'gcr.io/cloud-builders/docker'
    entrypoint: 'bash'
    args:
      - '-c'
      - 'docker pull gcr.io/$PROJECT_ID/appcicd:latest || exit 0'
   # Step 2: Create a docker image if none exists, esle load existing one
  - name: 'gcr.io/cloud-builders/docker'
    args:
      - 'build'
      - '-t'
      - 'gcr.io/$PROJECT_ID/appcicd:latest'
      - '-t'
      - 'gcr.io/$PROJECT_ID/appcicd:$COMMIT_SHA'
      - '--cache-from'
      - 'gcr.io/$PROJECT_ID/appcicd:latest'
      - '.'
  # Step 3: Push the docker image to the run
  - name: 'gcr.io/cloud-builders/docker'
    args:
      - 'push'
      - 'gcr.io/$PROJECT_ID/appcicd'
  # Step 4: Deploy container image to Cloud Run
  - name: 'gcr.io/cloud-builders/gcloud'
    args:
      - 'run'
      - 'deploy'
      - 'flaskappml'
      - '--image'
      - 'gcr.io/$PROJECT_ID/appcicd:latest'
      - '--region'
      - 'us-central1'
      - '--platform'
      - 'managed'
      - '--allow-unauthenticated'
```

cloudbuild.yaml hosted with ❤️ by **GitHub** view raw

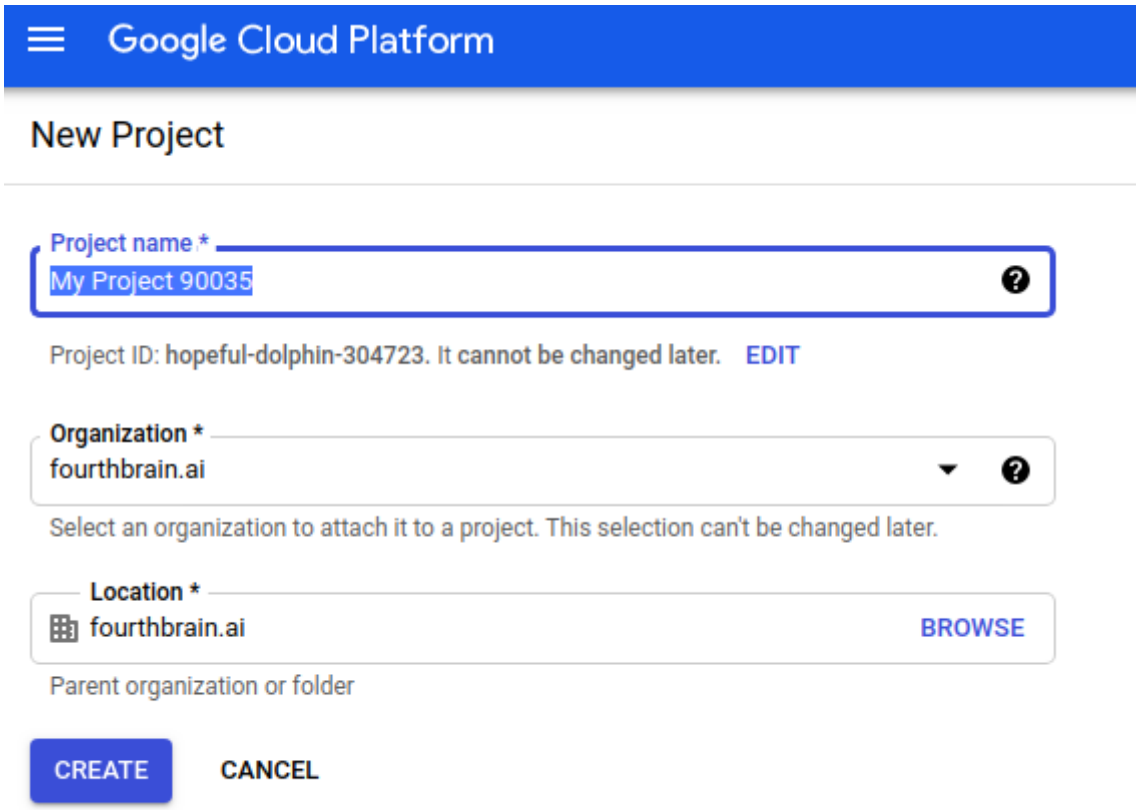
2. Create a build if none exists so far, otherwise use the cache build.
3. Push the docker image to GCR
4. Deploy the application to GCR.

Now that we have all the components to run and deploy the ML web app, lets create a trigger to deploy the app on *Google Cloud*.

Step 5: Create the CD Pipeline and Run

Now that we have an app that runs locally, the next task is to set up a project in GCP that will generate triggers and the cloud run. Follow the steps below to ensure you have a viable Google cloud project to begin the CD process.

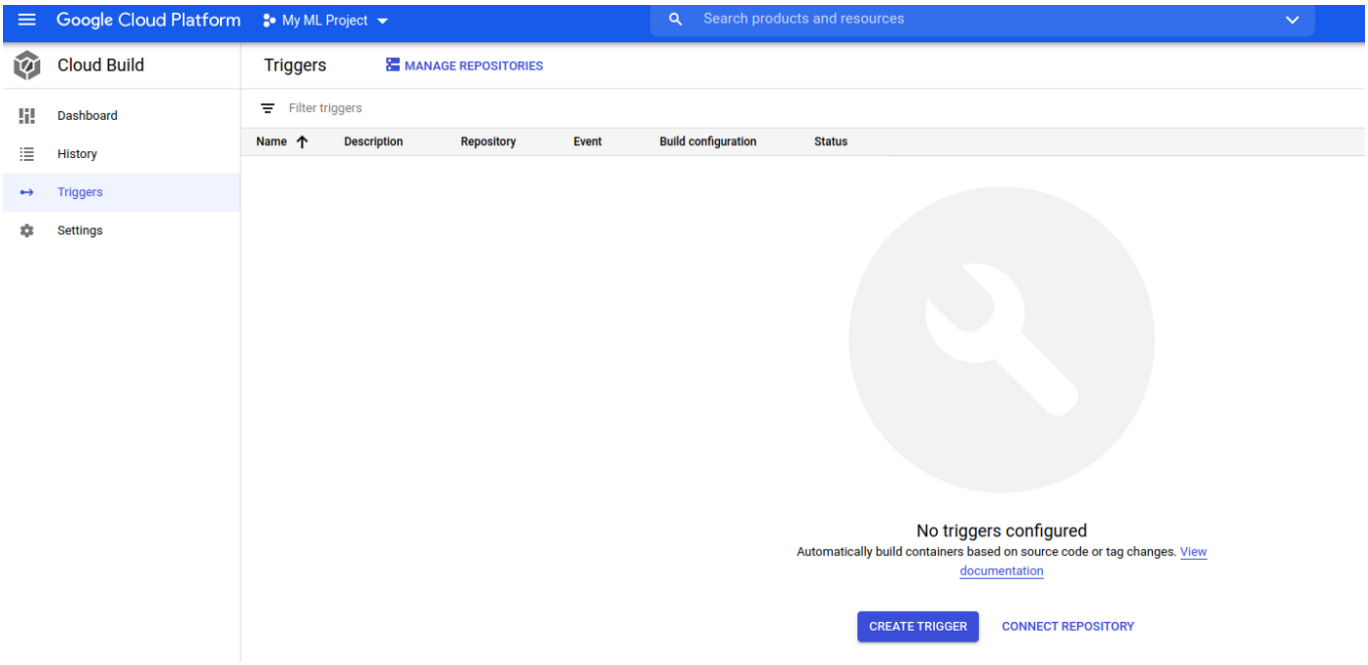
1. Sign up at Google Console: https://console.cloud.google.com/home/
2. Create a new Project as shown below:



3. Ensure 'Billing' is activated for the project.

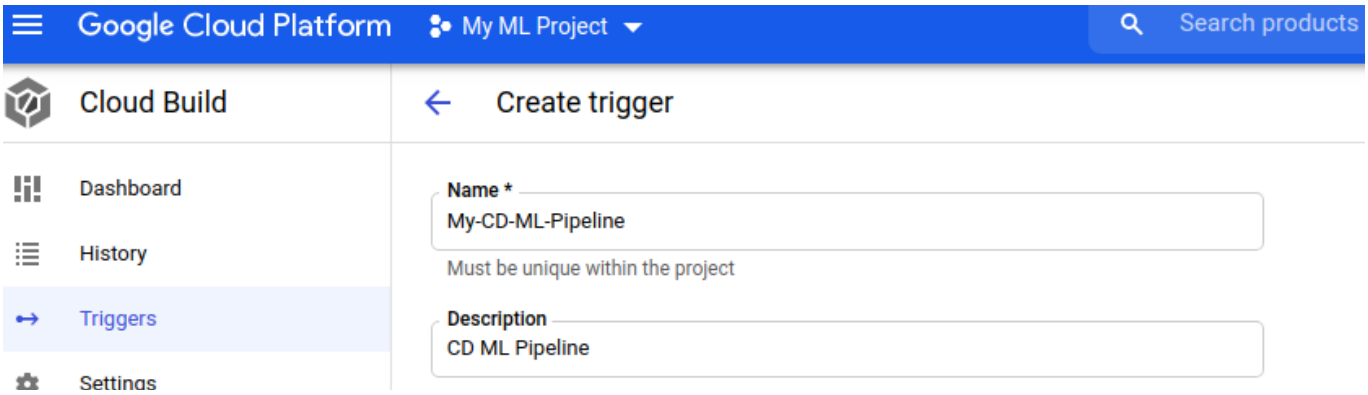
4. Start from Github code to create a CD pipeline. Use the forked Github repository from your Github account for this purpose.

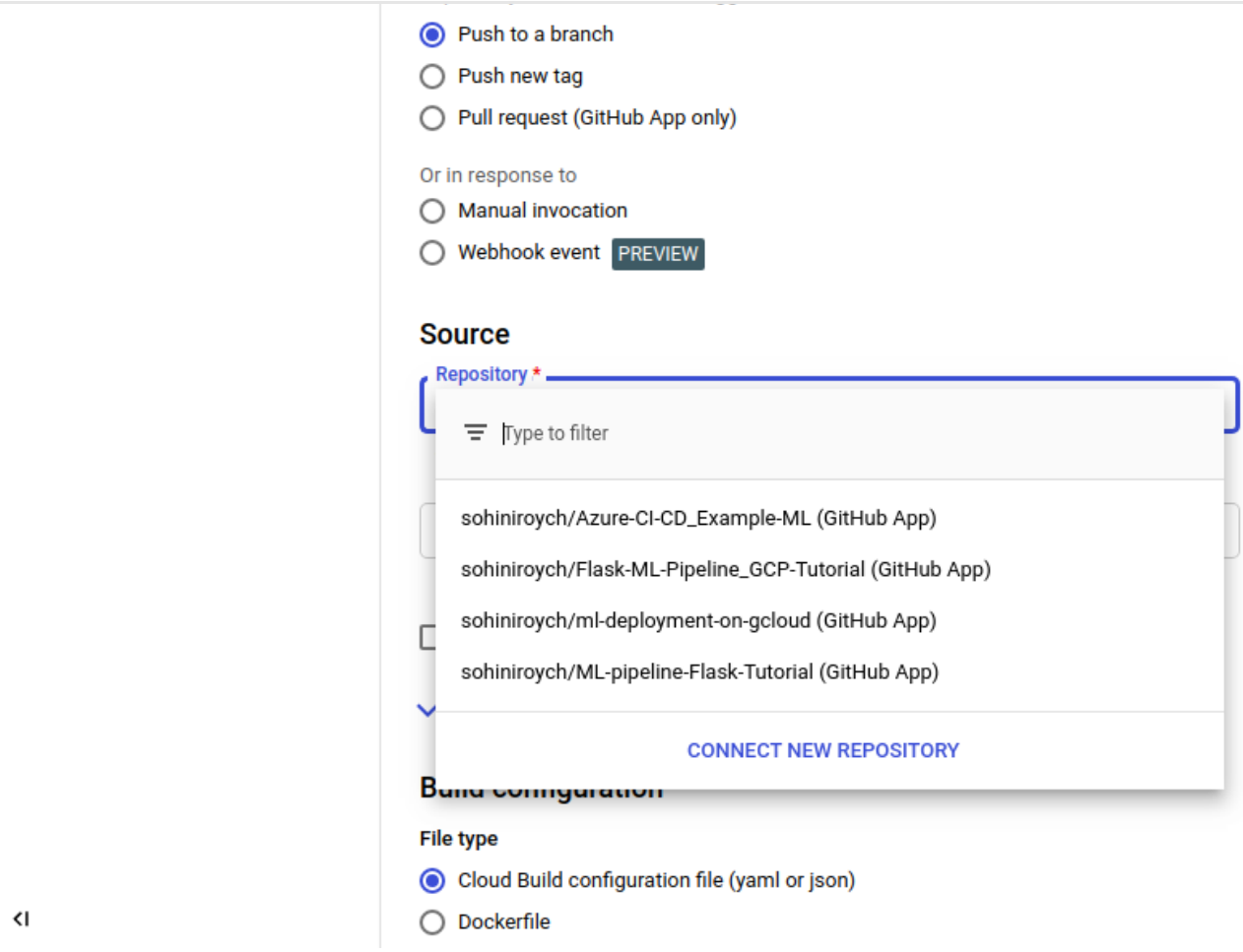
5. Go to the page: https://console.cloud.google.com/cloud-build/triggers. Ensure your project is selected on the top panel. Next click 'Create Trigger' button at the bottom as shown below.



Build Triggers in GCP

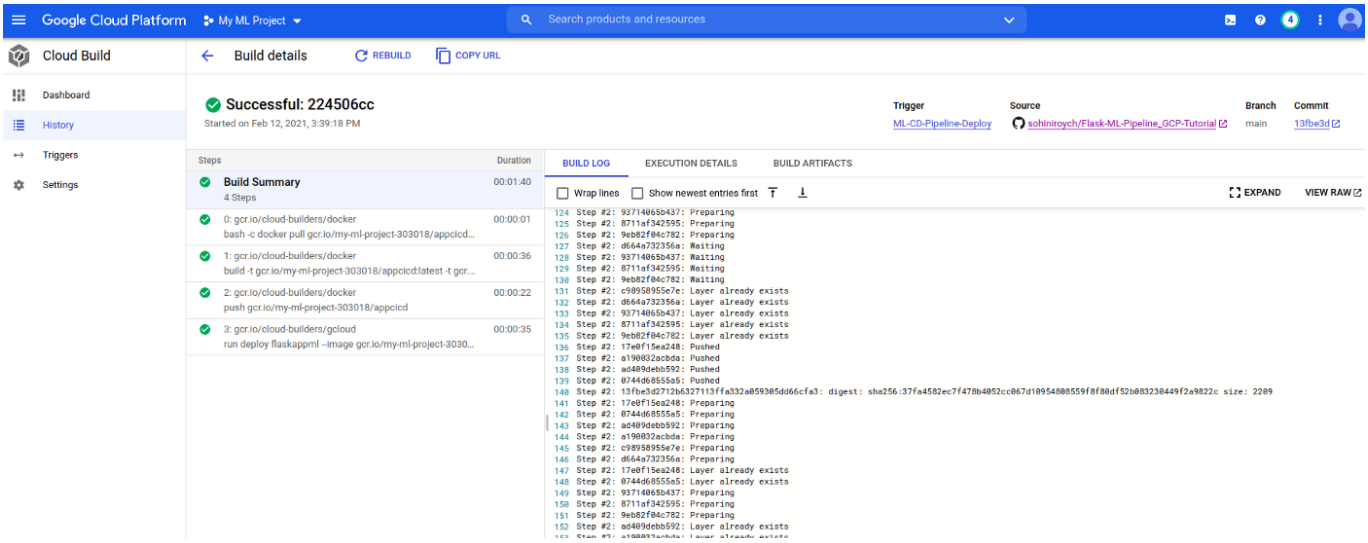
6. This will take you to the following page where you will need to create a 'Push to a branch' event for your forked Github repository. To deploy the app, select the 'Cloud Build' configuration file option as shown below and hit 'Create' at the bottom.





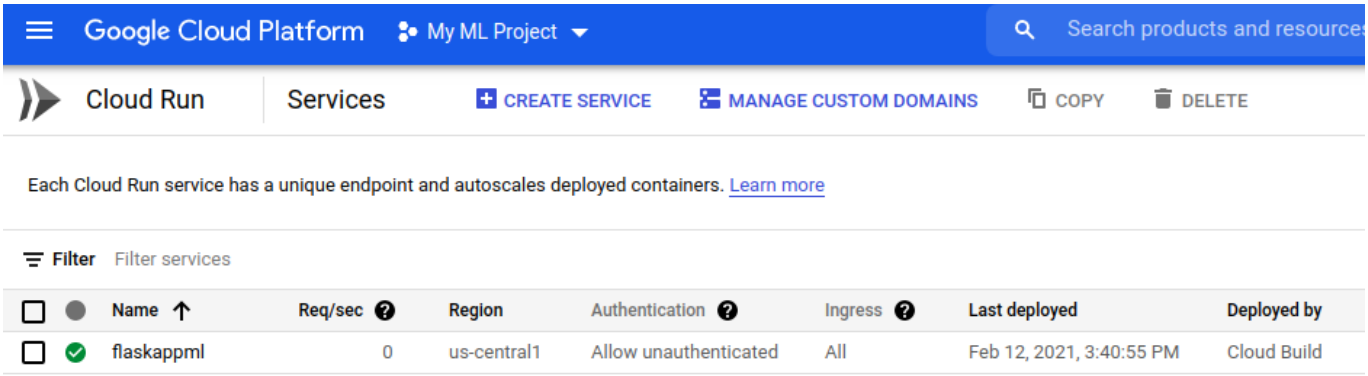
Steps to Create a Push Trigger using Github code

7. Now a trigger gets created. Hit RUN option next to the trigger and look at the logs that get generated. This would take a few minutes to run and deploy. At the end you should have something that looks like below.



Outcome of the Push Trigger Run

8. Now go to the GCR console at: https://console.cloud.google.com/run and notice your app (flaskappml) running as shown below:



Deployed App in GCR

> *Clicking on 'flaskappml' takes you to the GCR page with the URL displayed on top right. This is the URL to your deployed webapp that you can now test on any device.*

Don't forget to clean up the triggers and the GCR before your leave. This is crucial to ensure appropriate resource usage.

Conclusion

Building ML-models and deploying them on cloud platforms not only enhances usability, but it also helps gather insights into the limiting conditions for the model. To ensure robust deployed web apps, CI pipelines enable code to be committed to a repository in a systematic manner while CD pipelines enable code sanity triggers and checks to ensure successful app deployment. Extending this tutorial to other web apps is a relatively simple process. This will require the following 4 major changes:

1. Updating the app_files to reflect your application of interest.
2. Updating the Dockerfile and requirements.txt files to represent the libraries needed for your application.

4. Committing an updated code as a continuous integration pipeline to a GitHub or Git repo.

Now you have the resources needed to build CI/CD pipelines for your own ML-based use cases.

References:

[1] S. Roychowdhury. Flask ML Model CD Pipeline Tutorial [Online]: https://github.com/sohiniroych/Flask-ML-Pipeline_GCP-Tutorial/

[2] K. Veerakumar. Startup — Multiple Linear Regression. [Online]:https://www.kaggle.com/karthickveerakumar/startup-logistic-regression

[3] L. Li. Classification and Regression Analysis with Decision Trees [Online]: Classification and Regression Analysis with Decision Trees | by Lorraine Li | Towards Data Science

[4] A. Jaleel. Startup:REST API with Flask [Online]: https://www.kaggle.com/ahammedjaleel/startup-rest-api-with-flask

[5] M. Grinberg. Designing a RESTful API with Python and Flask [Online]: https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask

[6] Google Cloud. Trouble shooting. [Online] https://cloud.google.com/run/docs/troubleshooting

[7] J. Araujo. Machine Learning deployment pipeline on Google Cloud Run [Online]: https://github.com/jgvaraujo/ml-deployment-on-gcloud

[8]Google Cloud. app.yaml Reference [Online]: https://cloud.google.com/appengine/docs/standard/python/config/appref

Thanks to Anne Bonner.

Google Cloud Run Continuous Deployment Editors Pick Ml Deployment On Gcp

Hands On Tutorials