

Kubeflow Pipelines

Overview

An introduction to Kubeflow in GCP for folks using ML workflows.

Introduction

Kubeflow is a Machine Learning toolkit for Kubernetes. The project is dedicated to making deployments of Machine Learning (ML) workflows on Kubernetes simple, portable, and scalable. The goal is to provide a straightforward way to deploy best-of-breed open-source systems for ML to diverse infrastructures.

A machine learning workflow can involve many steps with dependencies on each other, from data preparation and analysis, to training, to evaluation, to deployment, and more. It's hard to compose and track these processes in an ad-hoc manner—for example, in a set of notebooks or scripts—and things like auditing and reproducibility become increasingly problematic. Kubeflow Pipelines (KFP) helps solve these issues by providing a way to deploy robust, repeatable machine learning pipelines along with monitoring, auditing, version tracking, and reproducibility. Cloud AI Pipelines makes it easy to set up a KFP installation.

What you'll build

In this experiment, you will build a web app that summarizes GitHub issues using Kubeflow Pipelines to train and serve a model. Upon completion, your infrastructure will contain:

- A Google Kubernetes Engine (GKE) cluster with Kubeflow Pipelines installed (via Cloud AI Pipelines).
- A pipeline that trains a Tensor2Tensor model on GPUs
- A serving container that provides predictions from the trained model
- A UI that interprets the predictions to provide summarizations for GitHub issues
- A notebook that creates a pipeline from scratch using the Kubeflow Pipelines (KFP) SDK

What you'll learn

The pipeline you will build trains a [Tensor2Tensor](#) model on GitHub issue data, learning to predict issue titles from issue bodies. It then exports the trained model and deploys the exported model using [Tensorflow Serving](#). The final step in the pipeline launches a web app, which interacts with the TF-Serving instance in order to get model predictions.

- How to install Kubeflow Pipelines on a GKE cluster
- How to build and run ML workflows using Kubeflow Pipelines
- How to define and run pipelines from an [AI Platform Notebook](#)

What you'll need

- A basic understanding of [Kubernetes](#) will be helpful but not necessary
- An active [GCP project](#) for which you have Owner permissions
- (Optional) A [GitHub](#) account
- Access to the [Google Cloud Shell](#), available in the [Google Cloud Platform \(GCP\) Console](#)

Setup

Cloud Shell

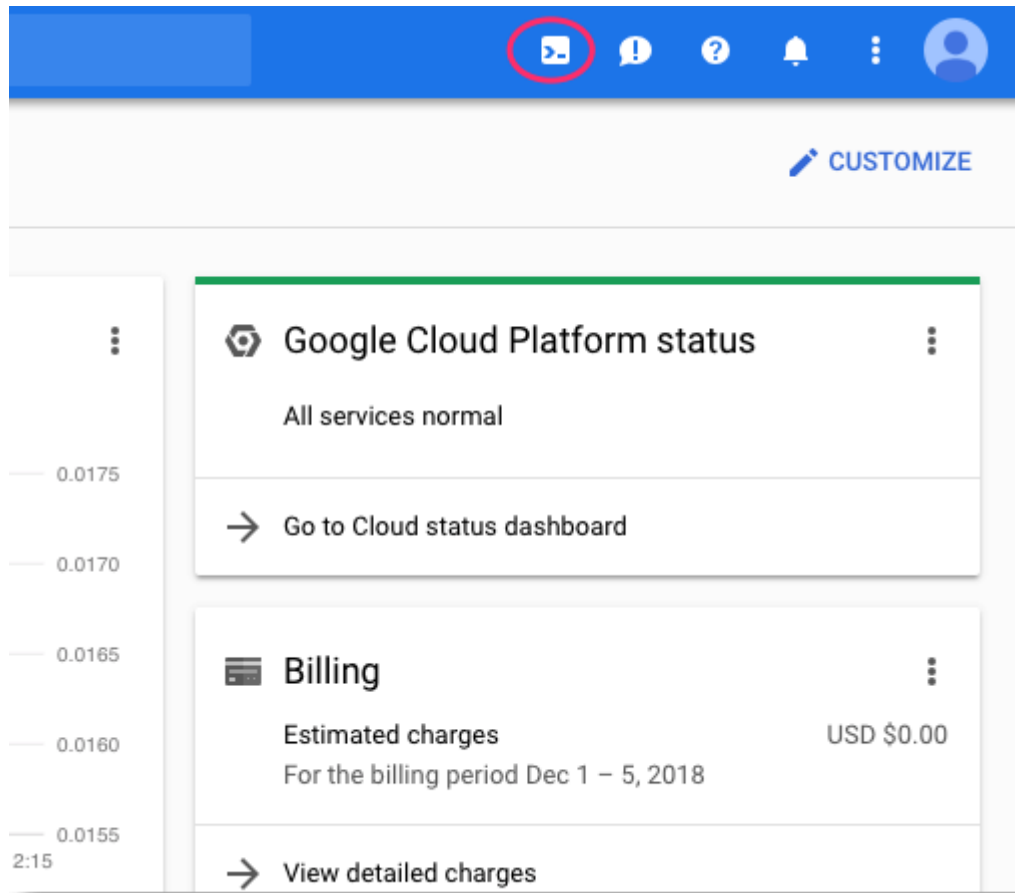
Visit the GCP Console in the browser and log in with your project credentials:

[Open the GCP Console](#)

Click "Select a project" if needed, so that you're working with your experiment project.

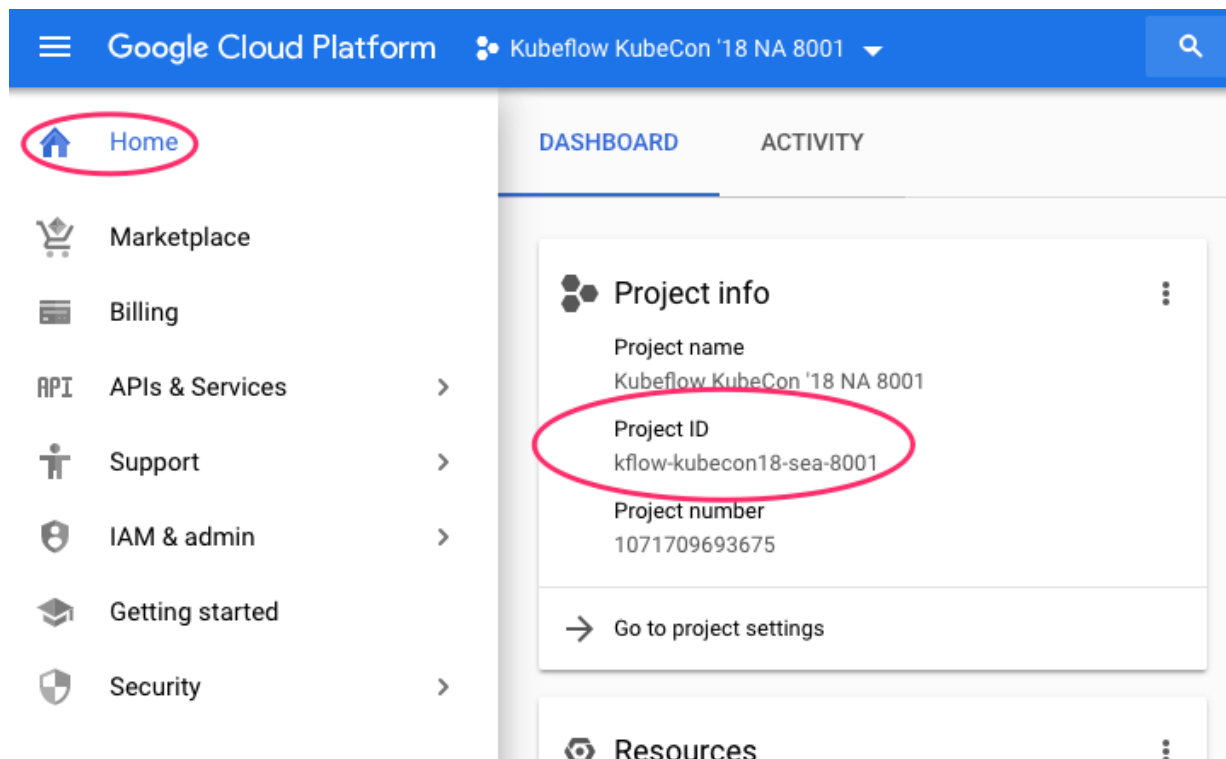


Then click the "Activate Cloud Shell" icon in the top right of the console to start up a [Cloud Shell](#).



When you start up the Cloud Shell, it will tell you the name of the project it's set to use. Check that this setting is correct.

To find your project ID, visit the GCP Console's Home panel. If the screen is empty, click on 'Yes' at the prompt to create a dashboard.



Then, in the Cloud Shell terminal, run these commands if necessary to configure `gcloud` to use the correct project:

```
export PROJECT_ID=<your_project_id> gcloud  
config set project ${PROJECT_ID}
```

Create a storage bucket

Note: Bucket names must be unique across all of GCP, not just your organization

Create a Cloud Storage bucket for storing pipeline files. You'll need to use a globally unique ID, so it is convenient to define a bucket name that includes your project ID. Create the bucket using the `gsutil` `mb` (make bucket) command:

```
export PROJECT_ID=<your_project_id> export  
BUCKET_NAME=kubeflow-${PROJECT_ID} gsutil  
mb gs://${BUCKET_NAME}
```

Alternatively, you can create a bucket via the GCP Console.

Optional**: Create a GitHub token**

This experiment calls the GitHub API to retrieve publicly available data. To prevent rate-limiting, especially at events where a large number of anonymized requests are sent to the GitHub APIs, set up an access token with no permissions. This is simply to authorize you as an individual rather than anonymous user.

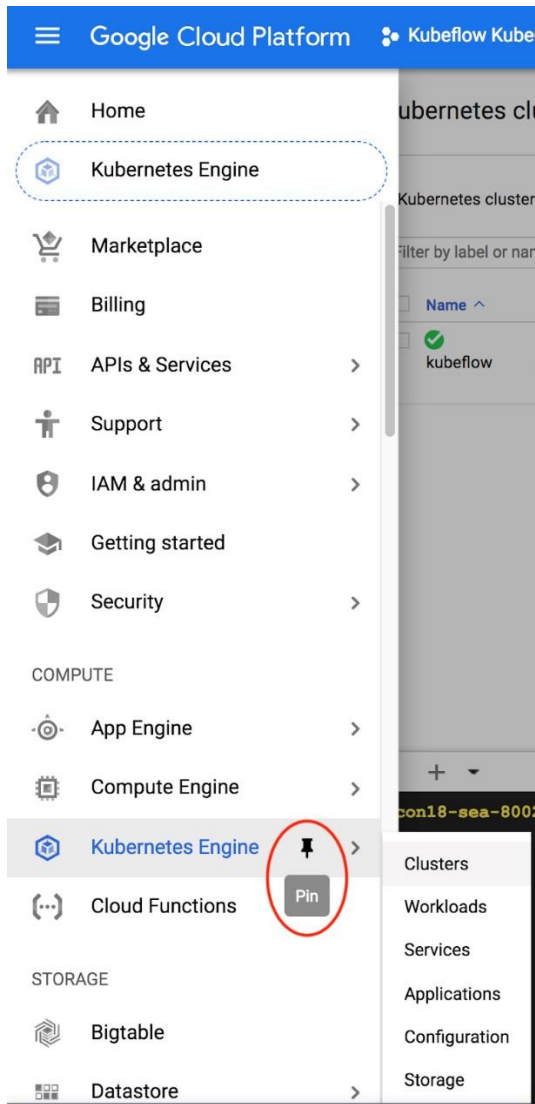
1. Navigate to <https://github.com/settings/tokens> and generate a new token with no scopes.

2. Save it somewhere safe. If you lose it, you will need to delete and create a new one.

If you skip this step, the lab will still work – you will just be a bit more limited in your options for generating input data to test your model.

Optional: **Pin useful dashboards**

In the GCP console, pin the Kubernetes Engine and Storage dashboards for easier access.





Create an AI Platform Pipelines (Hosted Kubeflow Pipelines) installation

Follow the instructions in the 'Before you begin' and 'Set up your instance' sections


<https://console.cloud.google.com/marketplace/product/google-cloud-ai-platform/kubeflow-pipelines>

to set up a GKE instance with KFP installed. Be sure to check the Allow access to the following Cloud APIs box as indicated in the documentation. (If you don't, the example pipeline won't run successfully). Leave the installation namespace as `default`.


Click to CONFIGURE your KFP

 Google Cloud Platform 

Deploy Kubeflow Pipelines



Google Cloud Marketplace does not permit the reselling of any Marketplace solutions, as stated in the [Marketplace Customer Terms of Service](#). If you are transacting under a reseller billing account, you cannot purchase Marketplace solutions. Contact your Google Cloud Partner Sales Manager for more information.

**Kubeflow**
Solution provider

Pricing

Note: There is no usage-based pricing for Kubeflow Pipelines on Google Cloud Platform. Please see the [pricing page](#) for more information.

Documentation


- [Setting up AI Platform Pipelines](#)
- [Upgrade Notes](#)
- [AI Platform Pipelines](#)
- [Kubeflow Pipelines](#)
- [AI Platform Pipelines](#)
- [AI Platform Pipelines](#)
- [Kubeflow Pipelines](#)

Your app will use compute instances managed in a logical grouping called a "cluster", which will be configured in a way that's great for getting started with Kubernetes. For more options, visit the Kubernetes engine [cluster creation page](#).

Zone
us-central1-a


Network
default

Subnetwork
default

☒ Allow access to the following Cloud APIs 
<https://www.googleapis.com/auth/cloud-platform>

You'll need to pick a zone that supports Nvidia k80s. You can use us-central1-a or us-central1-c as defaults.

Note the GKE cluster name and zone listed for your installation in the [AI Pipelines dashboard](#) once installation is complete, and for convenience set environment variables to these values.

Filter						
<input type="checkbox"/>	Status	Name ↑		Zone	Version	Cluster
<input type="checkbox"/>	✓	kubeflow-pipelines-1	OPEN PIPELINES DASHBOARD	us-central1-c	1.0.0	cluster-1
						default
				SETTINGS		

```
export ZONE=<your zone>
```

```
export CLUSTER_NAME=<your cluster name>
```

Set up kubectl to use your new GKE cluster's credentials

After the GKE cluster has been created, configure `kubectl` to use the credentials of the new cluster by running the following command in your Cloud Shell:

```
gcloud container clusters get-credentials ${CLUSTER_NAME} \
--project ${PROJECT_ID} \
--zone ${ZONE}
```

Alternatively, click on the name of the cluster in the AI Pipelines dashboard to visit its GKE page, then click "Connect" at the top of the page. From the popup, paste the command into your Cloud Shell.

This configures your `kubectl` context so that you can interact with your cluster. To verify the config, run the following command:

```
kubectl get nodes -o wide
```

You should see nodes listed with a status of "Ready", and other information about node age, version, external IP address, OS image, kernel version, and container runtime.

Configure the cluster to install the Nvidia driver on gpu-enabled node pools

Next, we'll apply a `daemonset` to the cluster, which will install the Nvidia driver on any GPU-enabled cluster nodes:

```
kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/container-engineaccelerators/master/nvidia-driver-installer/cos/daemonset-preloaded.yaml
```

Then run the following command, which gives the KFP components permission to create new Kubernetes resources:

```
kubectl create clusterrolebinding sa-admin --clusterrole=cluster-admin -
serviceaccount=kubeflow:pipeline-runner
```

Create a GPU node pool

Then, we'll set up a GPU node pool with a size of 1:

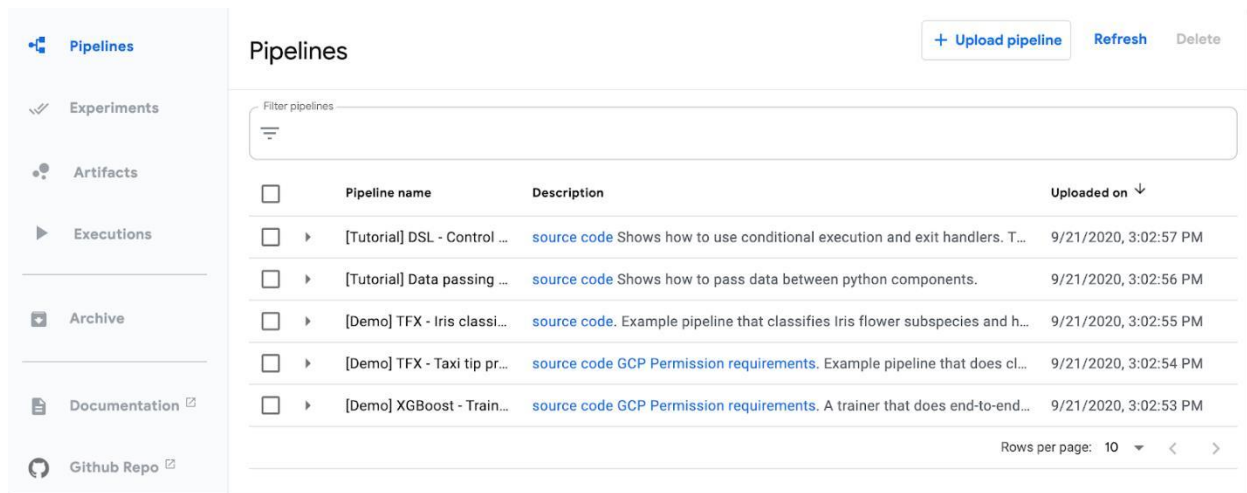
```
gcloud container node-pools create gpu-pool \
--cluster=${CLUSTER_NAME} \
--zone ${ZONE} \
--num-nodes=1 \
--machine-type n1-highmem-8 \
--scopes cloud-platform --verbosity error \
--accelerator=type=nvidia-tesla-k80,count=1
```

Notes: This can take about 5 minutes. If you see a GPU quota error when creating the node pool, you can [request more quota](#).

Run a pipeline from the Pipelines dashboard

Open the Pipelines dashboard

In the Cloud Console, visit the [Pipelines panel](#) if you're not already there. Then click on "**OPEN PIPELINES DASHBOARD**" for your installation, and click on **Pipelines** in the left menu bar. If you get a load error, refresh the tab. You should see a new page like this:



Pipeline description

The pipeline you will run has several steps, review the code details in this experiment for more detail:

1. An existing model checkpoint is copied to your bucket.
2. A [Tensor2Tensor](#) model is trained using preprocessed data.
 - Training starts from the existing model checkpoint copied in the first step, then trains for a few more hundred steps. (It would take too long to fully train it during the experiment).
 - When training finishes, the pipeline step exports the model in a form suitable for serving by [TensorFlow serving](#).
3. A TensorFlow-serving instance is deployed using that model.
4. A web app is launched for interacting with the served model to retrieve predictions.

Download and compile the pipeline

In this section, we'll see how to compile a pipeline definition. The first thing we need to do is install the KFP SDK. Run the following in the Cloud Shell:

```
pip3 install -U kfp
```

To download the pipeline definition file, execute this command from the Cloud Shell:

```
curl -O https://raw.githubusercontent.com/GeorgeNiece/gcp-data-pipelineengineering/main/experiments/gh_summ_hosted_kfp.py
```

Then compile the pipeline definition file by running it like this:

```
python3 gh_summ_hosted_kfp.py
```

You will see the file `gh_summ_hosted_kfp.py.tar.gz` appear as a result.

Note: If you get an error, make sure you have installed the Pipelines SDK and are using Python 3.

Upload the compiled pipeline

In the Kubeflow Pipelines web UI, click on **Upload pipeline**, and select **Import by URL**. Copy, then paste in the following URL, which points to the same pipeline that you just compiled. (It's a few extra steps to upload a file from Cloud Shell, so we're taking a shortcut).

```
https://github.com/GeorgeNiece/gcp-data-pipeline-engineering/blob/main/experiments/gh_summ_hosted_kfp.py.tar.gz
```

 Give

the pipeline a name (e.g. `gh_summ`).

← Upload Pipeline or Pipeline Version

☒ Create a new pipeline ☐ Create a new pipeline version under an existing pipeline

Upload pipeline with the specified package.

Pipeline Name *

gh_summ

Pipeline Description *

URL must be publicly accessible.

For expected file format, refer to [Compile Pipeline Documentation](#).

☐ Upload a file

File *

Choose file

☒ Import by url

Package Url

`https://github.com/GeorgeNiece/gcp-cata-pipeline-engineering/blob/main/experiments/gh_summ_hostec_kfp.py|.tar.gz`

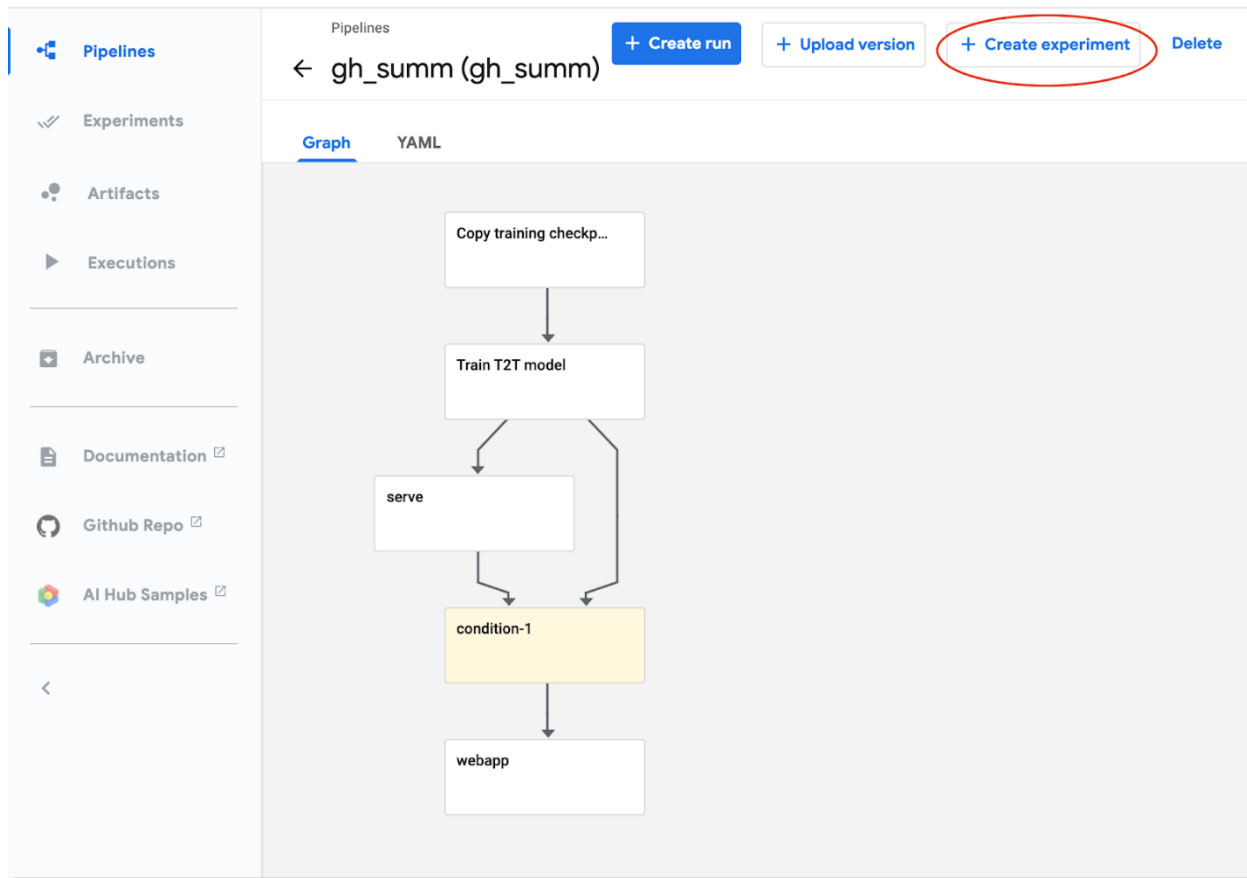
Code Source (optional)

Create

Cancel

Run the pipeline

Click on the uploaded pipeline in the list —this lets you view the pipeline's static graph— then click on **Create experiment** to create a new **Experiment** using the pipeline. An Experiment is a way to group together semantically related runs.



Give the Experiment a name (e.g. the same name as the pipeline, `gh_summ`), then click **Next** to create.

The screenshot shows the 'New experiment' form in the Kubeflow interface. The header bar is dark blue with the Kubeflow logo and name. Below the header, the page is titled 'Experiments' and 'New experiment'. The form is titled 'Experiment details' and includes a sub-header: 'Think of an Experiment as a space that contains the history of all pipelines and their associated runs'. There are two input fields: 'Experiment name *' with the value 'gh_summ' entered, and 'Description (optional)' which is empty. At the bottom of the form are two buttons: 'Next' (in blue) and 'Cancel' (in grey).

This will bring up a page where you can enter the parameters for a Run and start it off. You

may want to execute the following commands in Cloud Shell to help fill in the parameters.

```
gcloud config get-value project echo  
"gs://${BUCKET_NAME}/codelab"
```

The Run name will be auto-filled, but you can give it a different name if you like.

Then fill in three parameter fields:

- `project`
- (optional) `github-token`
- `working-dir`

For the `working-dir`, enter some path under the GCS bucket you created. Include the `'gs://'` prefix. For the `github-token` field, enter either the token that you optionally generated earlier, or leave the placeholder string as is if you did not generate a token.

Run parameters

Specify parameters required by the pipeline

train_steps	2019300
project	YOUR_PROJECT_HERE
github_token	YOUR_GITHUB_TOKEN_HERE
working_dir	gs://YOUR_GCS_DIR_HERE
checkpoint_dir	gs://aju-dev-demos-codelabs/kubecon/model_output_tbase.bak2019000/
deploy_webapp	true
data_dir	gs://aju-dev-demos-codelabs/kubecon/t2t_data_gh_all/

Start

Cancel

After filling in the fields, click **Start**, then click on the listed run to view its details. While a given pipeline step is running, you can click on it to get more information about it, including viewing its *pod* logs. (You can also view the logs for a pipeline step via the link to its [Cloud Logging \(Ops Suite\)](#) logs, even if the cluster node has been torn down).

Note: The pipeline will take approximately 15 minutes to complete.

View the pipeline definition

While the pipeline is running, you may want to take a closer look at how it is put together and what it is doing.

A look at the code

Defining the pipeline

The pipeline used in this experiment is defined in `gh_summ_hosted_kfp.py` in our GitHub repo.

Let's take a look at how it is defined, as well as how its components (steps) are defined. We'll cover some highlights, but see the [documentation](#) for more details.

Note: While not illustrated in this example, you may also be interested to explore [KFP lightweight python components](#). Lightweight python components do not require you to build a new container image for every code change, and are helpful for fast iteration and prototyping. Kubeflow Pipeline steps are container-based. When you're building a pipeline, you can use [pre-built components](#), with already-built container images, or build your own components. For this experiment, we've built our own.

Four of the pipeline steps are defined from reusable components, accessed via their [component definition files](#). In this first code snippet, we're accessing these component definition files via their URL, and using these definitions to create 'ops' that we'll use to create a pipeline step.

```
import kfp.dsl as dsl import
kfp.gcp as gcp import
kfp.components as comp

...

copydata_op = comp.load_component_from_url(
    'https://raw.githubusercontent.com/kubeflow/examples/master/github_issue_summarization/pipelines/components/t2t/datacopy_component.yaml'
)

train_op = comp.load_component_from_url(
    'https://raw.githubusercontent.com/kubeflow/examples/master/github_issue_summarization/pipelines/components/t2t/train_component.yaml'
)
```

Below is one of the component definitions, for the training op, in yaml format. You can see that its inputs, outputs, container image, and container entrypoint args are defined.

```
name: Train T2T model description:
|
  A Kubeflow Pipeline component to train a Tensor2Tensor
model metadata: labels:
  add-pod-env: 'true' inputs:
- name: train_steps  description: '...'  type: Integer  default: 2019300 - name: data_dir  description:
  '...'  type: GCSPath - name: model_dir
  description: '...'
  type: GCSPath
- name: action
description: '...'
type: String
- name: deploy_webapp
  description: '...'
type: String outputs:
- name: launch_server
  description: '...'
type: String
- name: train_output_path
  description: '...'
type: GCSPath
```

```
- name: MLPipeline UI metadata  type: UI metadata implementation: container:  image:
  gcr.io/google-samples/ml-pipeline-t2ttrain:v3ap  args: [
    --data-dir, {inputValue: data_dir},
    --action, {inputValue: action},
    --model-dir, {inputValue: model_dir},
    --train-steps, {inputValue: train_steps},
    --deploy-webapp, {inputValue: deploy_webapp},
    --train-output-path, {outputPath: train_output_path}
  ]
  env:
    KFP_POD_NAME: "{{pod.name}}"
  fileOutputs:
    launch_server: /tmp/output
    MLPipeline UI metadata: /mlpipeline-ui-metadata.json
```

You can also define a pipeline step via the `dsl.ContainerOp` constructor, as we will see below.

Below is the bulk of the pipeline definition. We're defining the pipeline inputs (and their default values). Then we define the pipeline steps. For most we're using the 'ops' defined above, but we're also defining a 'serve' step inline via `ContainerOp`, specifying the container image and entrypoint arguments directly.

You can see that the train, log_model, and serve steps are accessing outputs from previous steps as inputs. You can read more about how this is specified [here](#).

```
@dsl.pipeline( name='Github issue summarization',
  description='Demonstrate Tensor2Tensor-based training and TF-Serving'
)
def gh_summ( #pylint: disable=unused-argument train_steps: 'Integer' =
  2019300, project: str = 'YOUR_PROJECT_HERE', github_token: str =
  'YOUR_GITHUB_TOKEN_HERE', working_dir: 'GCSPath' =
  'gs://YOUR_GCS_DIR_HERE', checkpoint_dir: 'GCSPath' = 'gs://aju-dev-
  demoscodelabs/kubecon/model_output_tbase.bak2019000/', deploy_webapp:
  str = 'true', data_dir: 'GCSPath' = 'gs://aju-dev-demos-
  codelabs/kubecon/t2t_data_gh_all/'
):

  copydata = copydata_op( data_dir=data_dir, checkpoint_dir=checkpoint_dir,
    model_dir='%s/%s/model_output' % (working_dir, dsl.RUN_ID_PLACEHOLDER),
    action=COPY_ACTION,
  )
```

```
train = train_op( data_dir=data_dir,
  model_dir=copydata.outputs['copy_output_path'],
```



```

action=TRAIN_ACTION, train_steps=train_steps,
deploy_webapp=deploy_webapp
)

```

```

serve = dsl.ContainerOp(  name='serve',  image='gcr.io/google-
samples/ml-pipeline-kubeflow-tfserve:v6',  arguments=["--model_name",
'ghsumm-%s' % (dsl.RUN_ID_PLACEHOLDER,),
"--model_path", train.outputs['train_output_path']
]
)

```

```

train.set_gpu_limit(1)

```

Note that we're requiring the 'train' step to run on a node in the cluster that has at least 1 GPU available. `train.set_gpu_limit(1)`

The final step in the pipeline— also defined inline— is conditional. It will run after the 'serve' step is finished, only if the training step `launch_server` output is the string 'true'. It launches the 'prediction web app', that we used to request issue summaries from the trained T2T model.

```

with dsl.Condition(train.outputs['launch_server'] == 'true'):
    webapp = dsl.ContainerOp(  name='webapp',  image='gcr.io/google-
samples/ml-pipeline-webapp-launcher:v1',  arguments=["--model_name",
'ghsumm-%s' % (dsl.RUN_ID_PLACEHOLDER,),  "--github_token",
github_token]

    )
    webapp.after(serve)

```

The component container image definitions

The Kubeflow Pipeline documentation describes some [best practices](#) for building your own components. As part of this process, you will need to define and build a container image. You can see the component steps for this experiment's pipeline [here](#). The Dockerfile definitions are in the `containers` subdirectories, e.g. [here](#).

Use preemptible VMs with GPUs for training

Preemptible VMs are [Compute Engine VM](#) instances that last a maximum of 24 hours and provide no availability guarantees. The pricing of preemptible VMs is lower than that of standard Compute Engine VMs.

With [Google Kubernetes Engine \(GKE\)](#), it is easy to set up a cluster or node pool [that uses preemptible VMs](#). You can set up such a node pool with [GPUs attached to the preemptible instances](#). These work the same as regular GPU-enabled nodes, but the GPUs persist only for the life of the instance.

You can set up a preemptible, GPU-enabled node pool for your cluster by running a command similar to the following, editing the following command with your cluster name and zone, and adjusting the accelerator type and count according to your requirements. You can optionally define the node pool to autoscale based on current workloads.

```
gcloud container node-pools create preemptible-gpu-pool \
  --cluster=<your-cluster-name> \
  --zone <your-cluster-zone> \
  --enable-autoscaling --max-nodes=4 --min-nodes=0 \
  --machine-type n1-highmem-8 \
  --preemptible \
  --node-taints=preemptible=true:NoSchedule \
  --scopes cloud-platform --verbosity error \
  --accelerator=type=nvidia-tesla-k80,count=4
```

You can also set up a node pool via the [Cloud Console](#).

Defining a Kubeflow Pipeline that uses the preemptible GKE nodes

If you're running Kubeflow on GKE, it is now easy to [define and run Kubeflow Pipelines](#) in which one or more pipeline steps (components) [run on preemptible nodes](#), reducing the cost of running a job. For use of preemptible VMs to give correct results, the steps that you identify as preemptible should either be [idempotent](#) (that is, if you run a step multiple times, it will have the same result), or should checkpoint work so that the step can pick up where it left off if interrupted.

When you're defining a Kubeflow Pipeline, you can indicate that a given step should run on a preemptible node by modifying the op like this:

```
your_pipelines_op.apply(gcp.use_preemptible_nodepool())
```

See the [documentation](#) for details.

You'll presumably also want to retry the step some number of times if the node is preempted. You can do this as follows— here, we're specifying 5 retries.

```
your_pipelines_op.set_gpu_limit(1).apply(gcp.use_preemptible_nodepool()).set_retry(5)
```

Try editing the Kubeflow pipeline we used in this experiment to run the training step on a preemptible VM.

Change the following line in the pipeline specification to additionally use a preemptible nodepool (make sure you have created one as indicated above) above, and to retry 5 times:

```
train.set_gpu_limit(1)
```

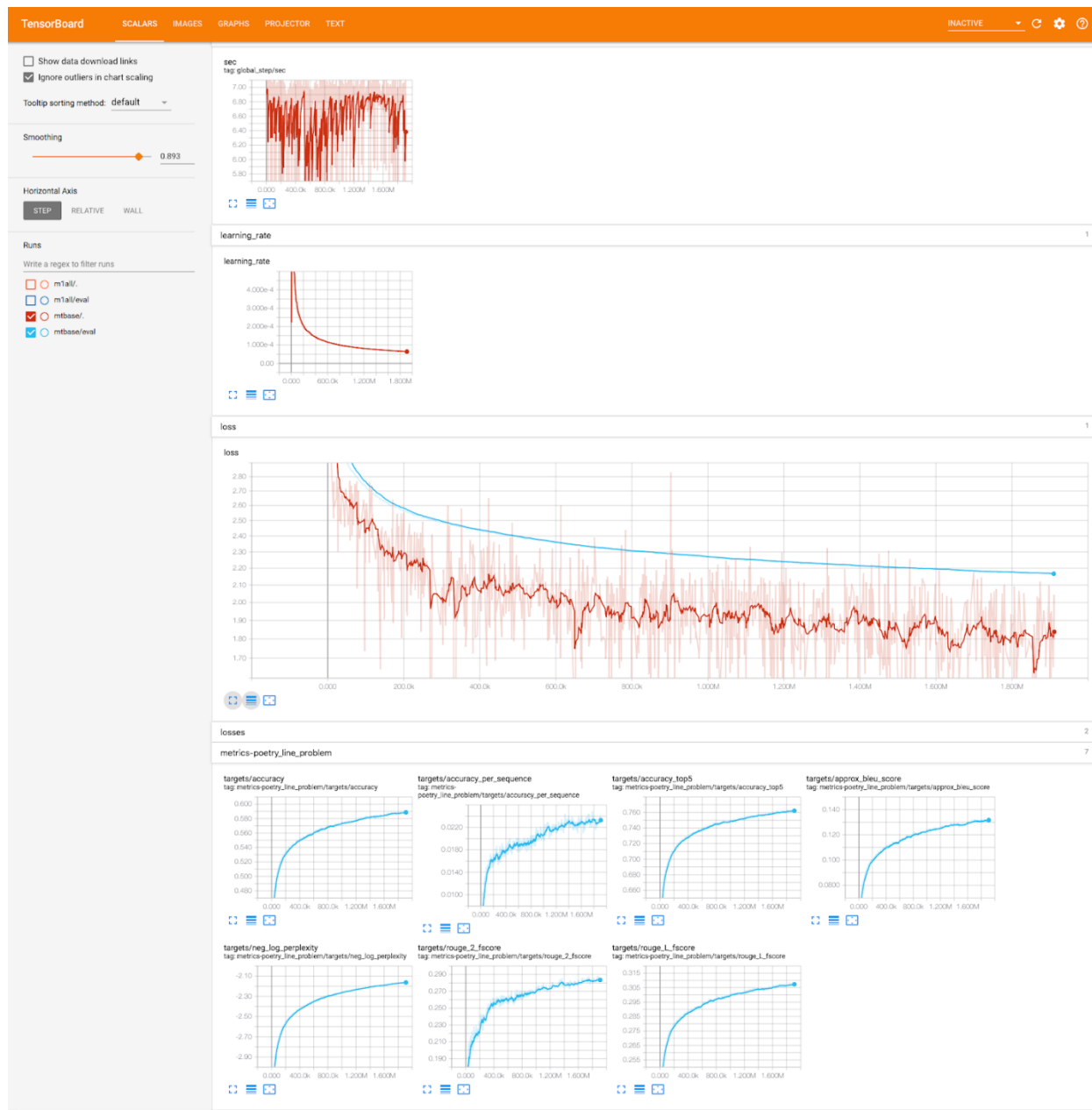
Then, recompile the pipeline, upload the new version (give it a new name), and then run the new version of the pipeline.

View model training information in TensorBoard

Once the training step is complete, select its **Visualizations** tab and click the blue **Start TensorBoard** button, then once it's ready, click **Open Tensorboard**.

The screenshot displays the Google Cloud AI Platform console interface. At the top, it shows the breadcrumb 'Experiments > Default' and the title 'Run of gh_summ_hosted_kfp3 (b6087)'. Below the title are three tabs: 'Graph', 'Run output', and 'Config'. The 'Graph' tab is selected, showing a pipeline diagram with four steps: 'Copy training chec...' (checked), 'Train T2T model' (checked and highlighted with a blue border), 'serve' (checked), and 'webapp' (checked). On the right side, there is a sidebar with three tabs: 'Input/Output', 'Visualizations' (circled in red), and 'ML Metadata'. Under the 'Visualizations' tab, there is a 'Tensorboard' section with a 'TF Version' dropdown set to 'TensorFlow 2.0.0' and a blue 'Start Tensorboard' button (also circled in red). Below this is a 'Visualization Creator' section with a link to 'create visualizations manually'.

Note: If after a minute or so the TensorBoard tab fails to load, try reloading the page.



Explore the Artifacts and Executions dashboard

Kubeflow Pipelines automatically logs metadata about the pipeline steps as a pipeline executes.

Both **Artifact** and **Execution** information is recorded. Click these entries in the left nav bar of the dashboard to explore further.

Pipelines	Artifacts					
Experiments	Filter					
Artifacts						
Executions						
Archive						
Documentation						
Github Repo						
AI Hub Samples						

Pipeline/Workspace ↑	Name	ID	Type	URI	Created at
github-issue-summarization-gd...	copy_output_p...	9	GCSPPath	minio://mlpipeline/artifacts/gith...	9/21/2020, 4:2...
	main-logs	10	NoType	minio://mlpipeline/artifacts/gith...	9/21/2020, 4:2...
	mlpipeline-ui-m...	11	NoType	minio://mlpipeline/artifacts/gith...	9/21/2020, 4:3...
	launch_server	12	String	minio://mlpipeline/artifacts/gith...	9/21/2020, 4:3...
	train_output_p...	13	GCSPPath	minio://mlpipeline/artifacts/gith...	9/21/2020, 4:3...
	main-logs	14	NoType	minio://mlpipeline/artifacts/gith...	9/21/2020, 4:3...
	main-logs	15	NoType	minio://mlpipeline/artifacts/gith...	9/21/2020, 4:3...
	main-logs	16	NoType	minio://mlpipeline/artifacts/gith...	9/21/2020, 4:3...

For Artifacts, you can view both an overview panel and a Lineage Explorer panel.

Artifacts

← launch_server

Overview

Lineage Explorer

Type: String

URI

[minio://mlpipeline/artifacts/github-issue-summarization-gdgks/github-issue-summarization-gdgks-598729998/train-t2t-model-launch_server.tgz](#)

Properties

Custom Properties

```

argo_artifact
{
  "name": "train-t2t-model-launch_server",
  "path": "/tmp/output",
  "s3": {
    "accessKeySecret": {
      "key": "accesskey",
      "name": "mlpipeline-minio-artifact"
    },
    "bucket": "mlpipeline",
    "endpoint": "minio-service.kubeflow:9000",
    "insecure": true,
    "key": "artifacts/github-issue-summarization-gdgks/github-issue-summarization-gdgks-598729998/train-t2t-model-launch_server.tgz",
    "secretKeySecret": {
      "key": "secretkey",
      "name": "mlpipeline-minio-artifact"
    }
  }
}

```

name

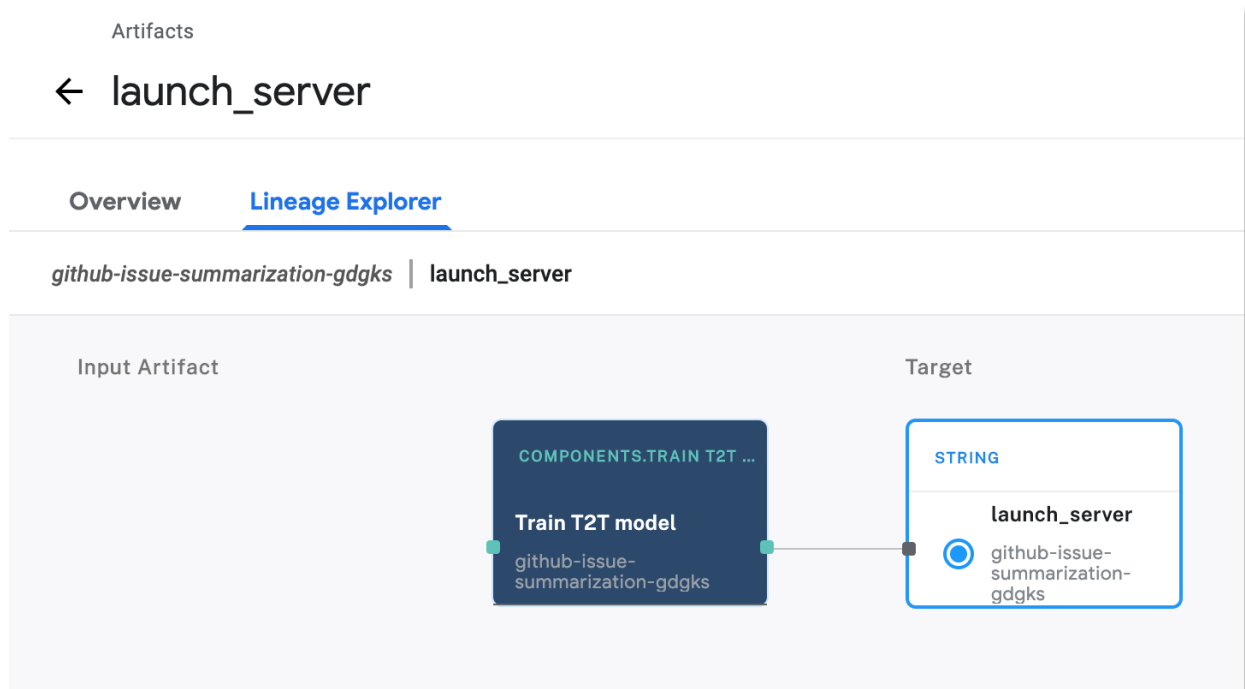
launch_server

pipeline_name

github-issue-summarization-gdgks

run_id

github-issue-summarization-gdgks



Bring up the web app created by the pipeline and make some predictions

The last step in the pipeline deploys a web app, which provides a UI for querying the trained model — served via [TF Serving](#) — to make predictions.

After the pipeline completes, connect to the web app by *port-forwarding* to its service (we're port-forwarding because, for this experiment, the webapp service is not set up to have an external endpoint).

Find the service name by running this command in the Cloud Shell:

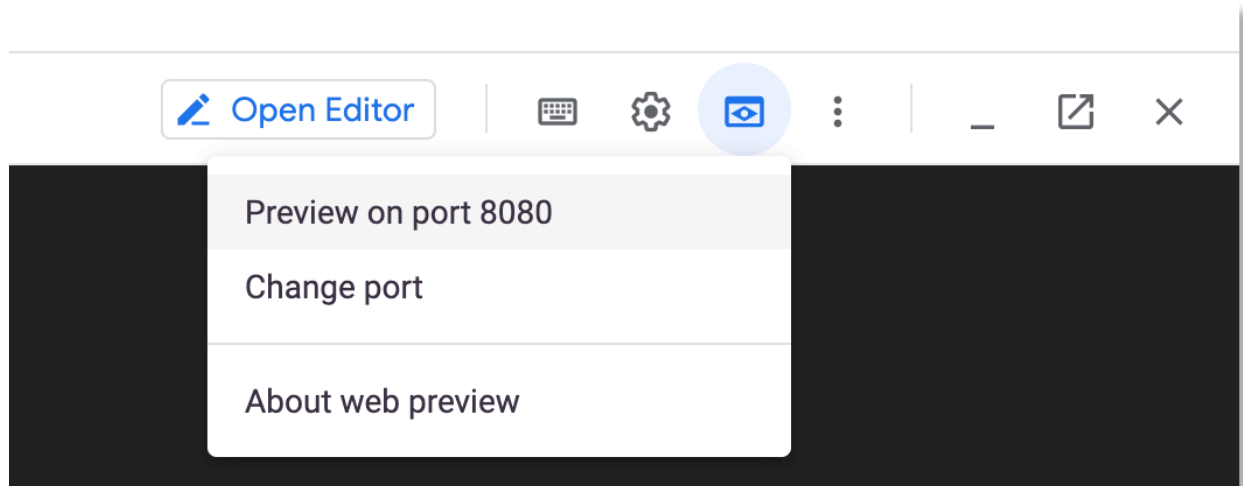
```
kubectl get services
```

Look for a service name like this: `ghsumm-*-webappsvc` in the list.

Then, in the Cloud Shell, port-forward to that service as follows, **changing the following command to use the name of your webappsvc**:

```
kubectl port-forward svc/ghsumm-xxxxx-webappsvc 8080:80
```

Once port-forwarding is running, click on the 'preview' icon above the Cloud Shell pane, and in the dropdown, click "Preview on port 8080".



You should see a page like this come up in a new tab:

Github Issue Summarization

This app takes as input a Github issue body and predicts a title for it. Behind the scenes it uses a [Tensor2Tensor](#) TensorFlow model, served via [TF-Serving](#) .

(Thanks to [Hamel Husain](#) for the original concept and source data.)

Enter the body of a github issue or the url of a github issue and click on Submit. The model then tries to generate a title or summary of the issue.

Enter Github Issue Body

Populate Random Issue

OR Enter Github Issue URL

Generate Title

This demo is run using [Kubeflow](#) - a machine learning toolkit for Kubernetes. Kubeflow is dedicated to making deployment of machine learning on Kubernetes simple, portable and scalable.

Click the **Populate Random Issue** button to retrieve a block of text. Click on **Generate Title** to call the trained model and display a prediction.

Enter Github Issue Body

Populate Random Issue

hey there, i've installed this plugin and i was trying to use. however, i've followed your instruction but didn't work. i selected the line, clicked on right button and i could see the option send to terminal but the submenu is empty. i am using macos with the newest version updated. regards, fabio

OR Enter Github Issue URL

<https://github.com/kubeflow/kubeflow/issues/232>

Generate Title

Machine Generated Title

"not working on macos sierra"

Note: It can take a few seconds to display a summary, especially the first time you make a request— for this workshop we're not using GPUs for the TensorFlow Serving instance.

If your pipeline parameters included a valid GitHub token, you can alternately try entering a GitHub URL in the second field, then clicking "Generate Title". If you did *not* set up a valid GitHub token, use only the "Populate Random Issue" field.

Run a pipeline from an AI Platform Notebook

You can also interactively define and run Kubeflow Pipelines from a Jupyter notebook using the KFP SDK. [AI Platform Notebooks](#), which we'll use for this experiment, makes this very straightforward.

Create a notebook instance

We'll create a notebook instance from the Cloud Shell using its API. (Alternatively, you can create a notebook via the [Cloud Console](#). See the documentation for [more information](#)).

Set the following environment variables in the Cloud Shell:

```
export INSTANCE_NAME="kfp-ghsumm" export
VM_IMAGE_PROJECT="deeplearning-platform-release" export
VM_IMAGE_FAMILY="tf2-2-3-cpu" export
MACHINE_TYPE="n1-standard-4"
export LOCATION="us-central1-c"
```


Then, from the Cloud Shell, run the command to create the notebook instance:

```
gcloud beta notebooks instances create $INSTANCE_NAME \
--vm-image-project=$VM_IMAGE_PROJECT \
--vm-image-family=$VM_IMAGE_FAMILY \
--machine-type=$MACHINE_TYPE --location=$LOCATION
```

When you first run this command, you may be asked to enable the `notebooks` API for your project. Reply 'y' if so.

After a few minutes, your notebook server will be up and running. You can see your Notebook instances listed in the Cloud Console.

<input type="checkbox"/>	<input checked="" type="checkbox"/>	kfp-ghsumm	OPEN JUPYTERLAB	us-central1-c	4 vCPUs, 15 GB RAM
--------------------------	-------------------------------------	------------	---------------------------------	---------------	--------------------

Upload the experiment notebook

After the notebook instance is created, click [this link](#) to upload the experiment's Jupyter notebook. Select the notebook instance to use. The notebook will be automatically opened.

Execute the notebook

Follow the instructions in the notebook for the remainder of the lab. Note that in the "Setup" part of the notebook, you will need to fill in your own values before running the rest of the notebook.

(If you're using your own project, don't forget to return and do the "Clean up" section of this lab).

Clean up

We should do this in our temporary experiment account, but you may wish to take down your Pipeline and Notebook if you're using your own project in the future to be spend efficient.

Take down the Pipelines GKE cluster

You can delete the Pipelines cluster from the [Cloud Console](#). (You have the option of just deleting the Pipelines installation if you want to reuse the GKE cluster).

Delete the AI Notebook instance

If you ran the "Notebook" part of the experiment, you can DELETE or STOP the notebook instance from the [Cloud Console](#).

Optional: Remove the GitHub token

Navigate to <https://github.com/settings/tokens> and remove the generated token.

Congratulations!

In this experiment, you created and ran a data processing pipeline with Kubeflow Pipelines.