# Vertex Pipelines P1

## Overview

Pipelines help you automate and reproduce your ML workflow. Vertex AI integrates the ML offerings across Google Cloud into a seamless development experience. Previously, models trained with AutoML and custom models were accessible via separate services. Vertex AI combines both into a single API, along with other new products. Vertex AI also includes a variety of MLOps products, like Vertex Pipelines. In this experiment, you will learn how to create and run ML pipelines with Vertex Pipelines.

### Why are ML pipelines useful?

Before diving in, first understand why you would want to use a pipeline. Imagine you're building out a ML workflow that includes processing data, training a model, hyperparameter tuning, evaluation, and model deployment. Each of these steps may have different dependencies, which may become unwieldy if you treat the entire workflow as a monolith. As you begin to scale your ML process, you might want to share your ML workflow with others on your team so they can run it and contribute code. Without a reliable, reproducible process, this can become difficult. With pipelines, each step in your ML process is its own container. This lets you develop steps independently and track the input and output from each step in a reproducible way. You can also schedule or trigger runs of your pipeline based on other events in your Cloud environment, like when new training data is available.

**What you'll learn**

- Use the Kubeflow Pipelines SDK to build scalable ML pipelines
- Create and run a 3-step intro pipeline that takes text input

- Create and run a pipeline that trains, evaluates, and deploys an AutoML classification model

- Use pre-built components for interacting with Vertex AI services, provided through the google_cloud_pipeline_components library

- Schedule a pipeline job with Cloud Scheduler

# Setup and Requirements

**What you need**

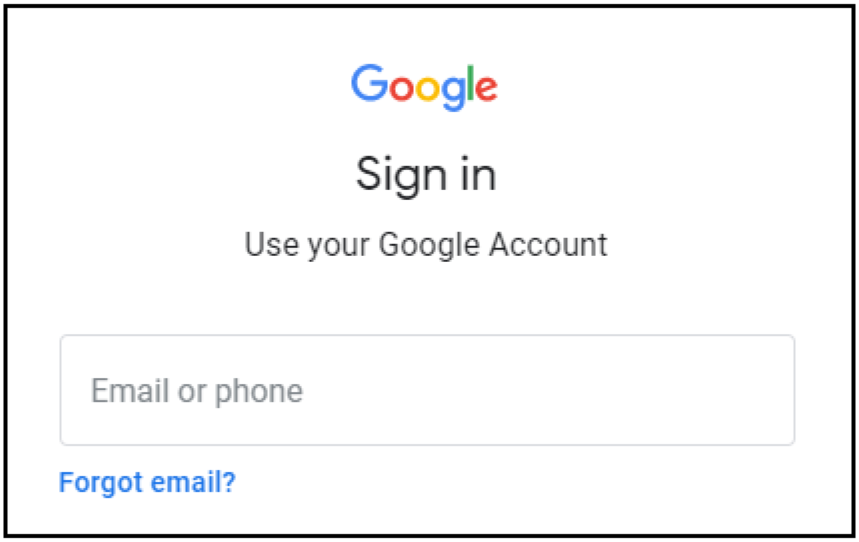
To complete this experiment, you need:

- Access to a standard internet browser (Chrome browser recommended).
- The experiment account provided by the instructor

**How to start your experiment and sign in to the Google Cloud Console**

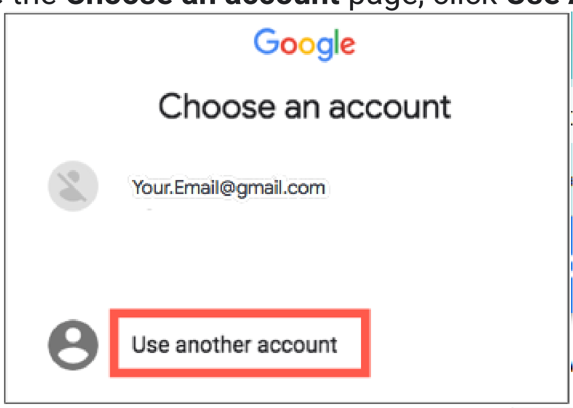
1. Sign into the console.cloud.google.com GCP environment with the username and credentials provided by the instructor

2. Copy the username, and paste into the **Sign in** page.




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

If you see the **Choose an account** page, click **Use Another Account**.




3. In the **Sign in** page, paste the username that you were provided. Then copy and paste the password.

4. Click through the subsequent pages:

    - For recovery email enter [jason@innovationinsoftware.com](mailto:jason@innovationinsoftware.com)

Confirm the recovery options without changing them.

After a few moments, the Cloud Console opens in this tab.

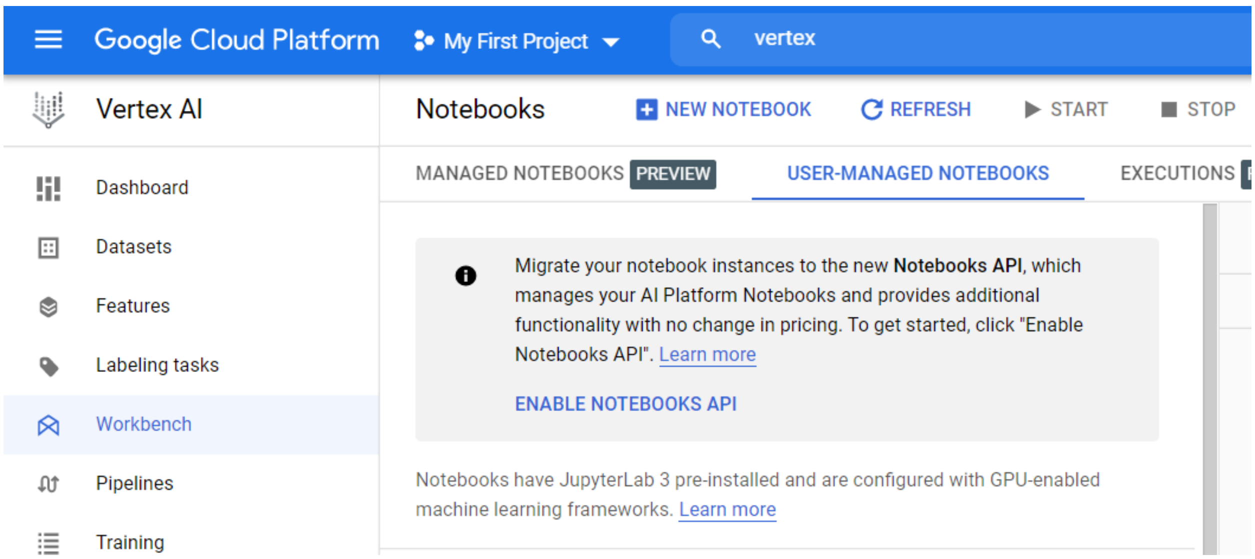
**Note:** You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left.

# Create an Vertex Notebooks instance

1. Click on the **Navigation Menu**.

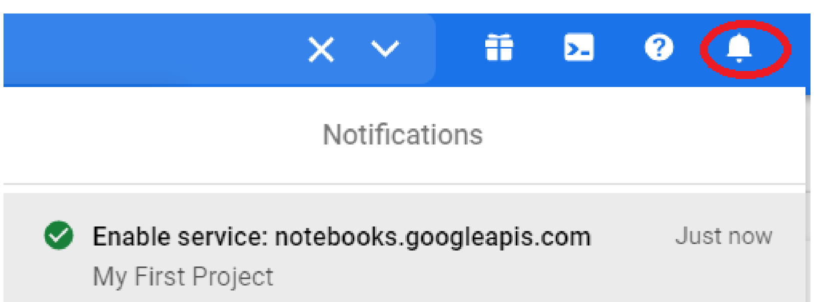
2. Navigate to **Vertex AI**, then to **Workbench**.

3. Click **ENABLE NOTEBOOKS API**

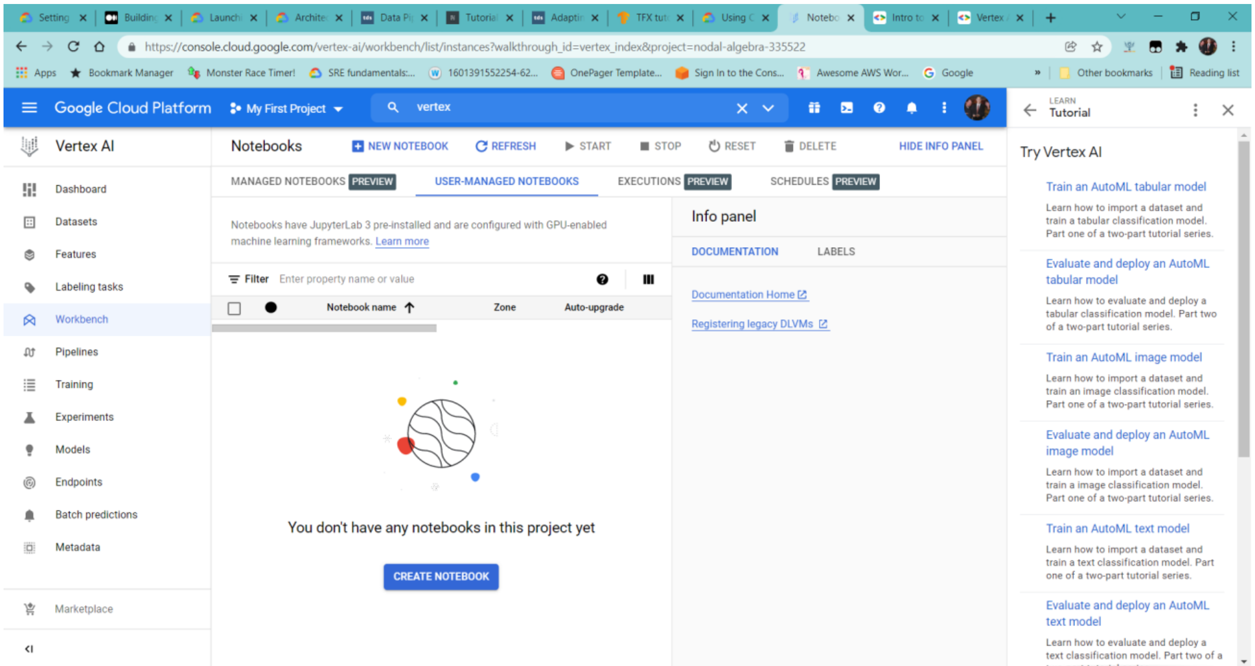



4. Click the Bell icon in the top level navigation on the right to view Notifications

5. Notice that the Notebook Enablement is posted there

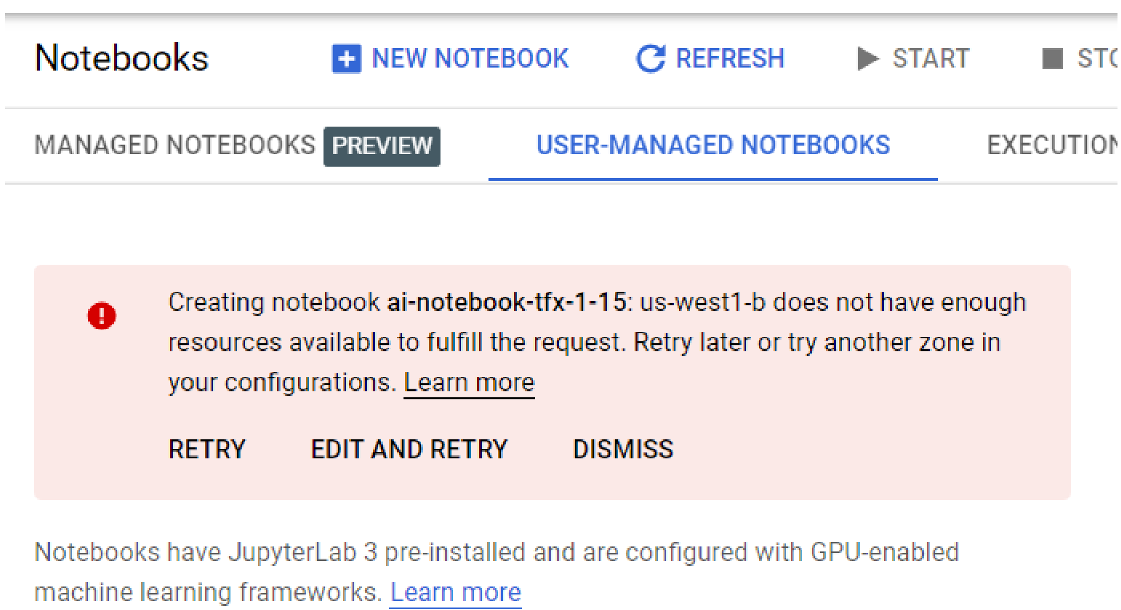



6. Click to **CREATE NOTEBOOK**



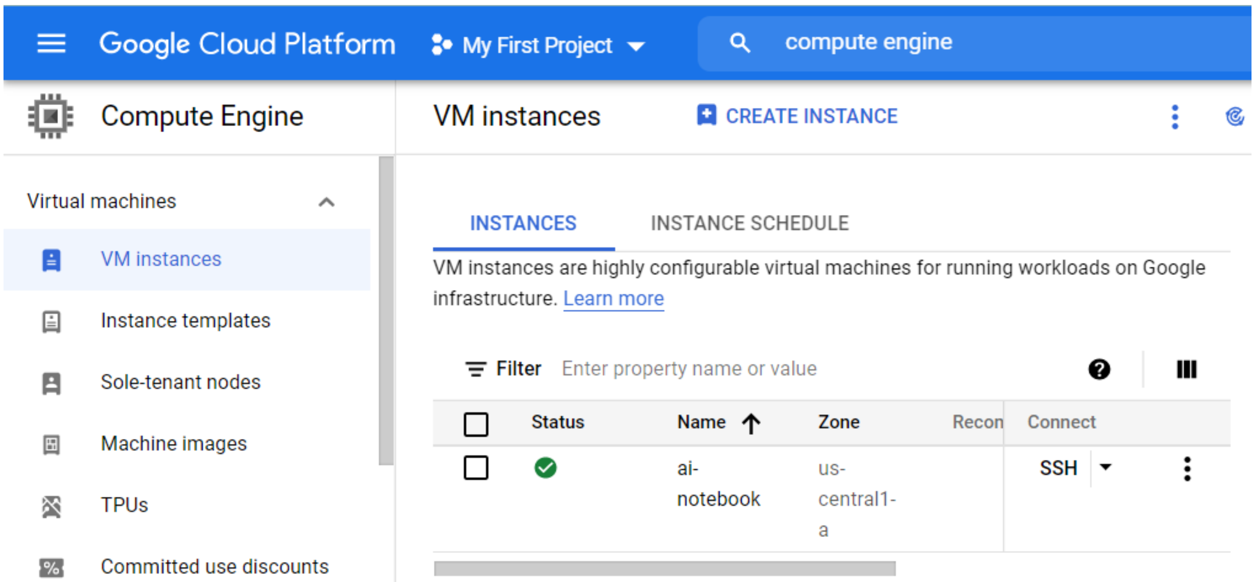
7. Enter a notebook name **ai-notebook**, for my notebook I selected **US-CENTRAL (Iowa)** but any region listed should be fine.  We'll select **Tensorflow 2.3, Debian Linux**, if not already selected and leave the remaining as default.  Click **CREATE**

8. **Note:** Occasionally you'll get ICE'd in which case you'll want to try again with a different region.  Although the Notebook creation defaults to US-WEST Oregon, I normally change that to Central or East regions.

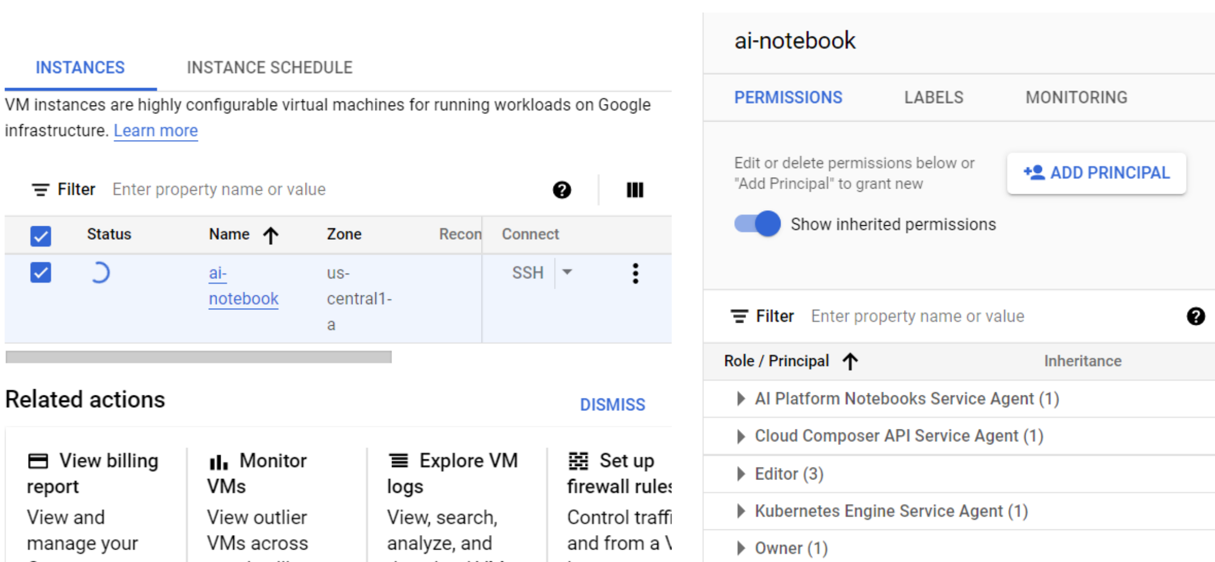


9. This will load the Notebook instances page, defaulting to **User-Managed Notebooks** tab and our new `ai-notebook` should show as being created, then later as connecting to JupyterLab for our access.

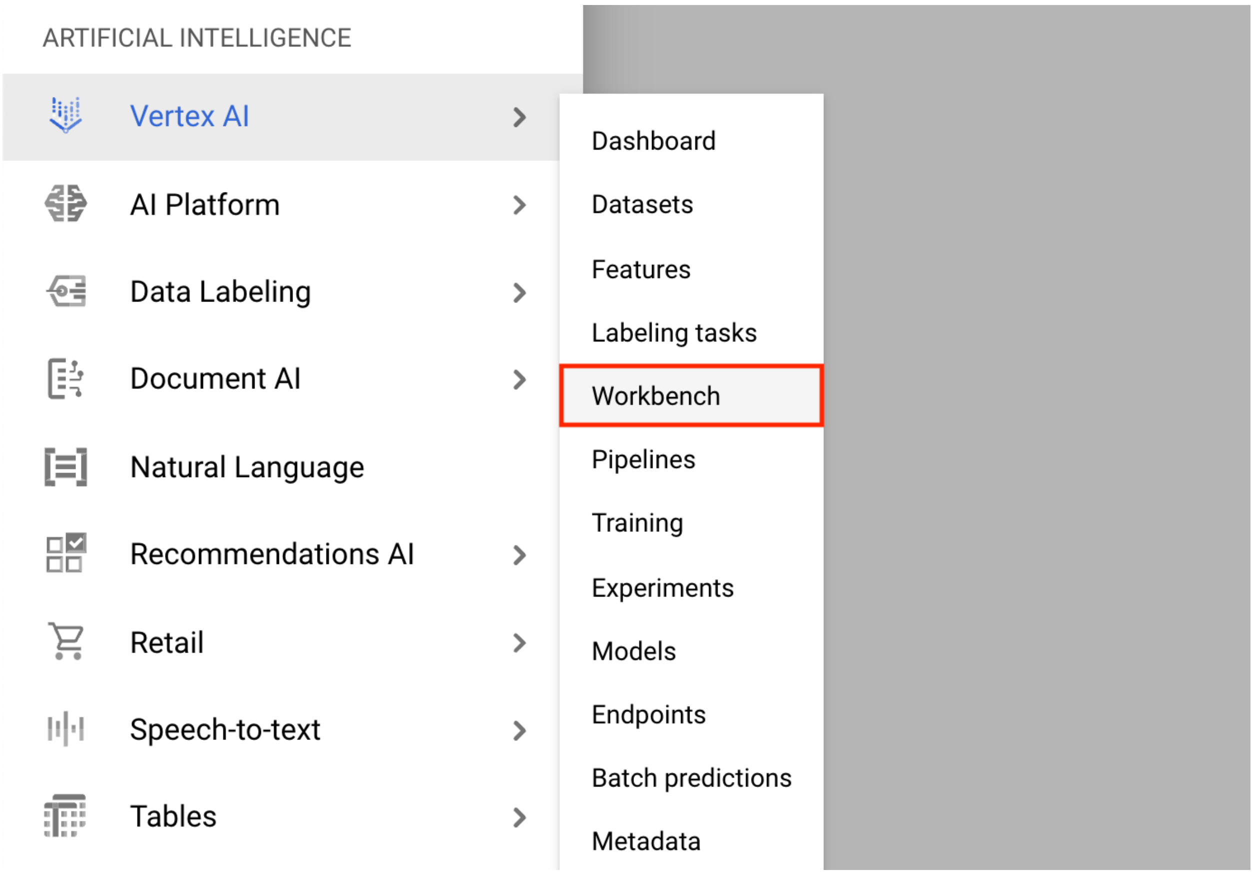
10. Navigate to the Compute Engine service while we wait, and note that we now have a Virtual Machine created name **ai-notebook**.




11. Select the checkbox next to our instance and note that there are a number of IAM permissions that have been assigned for using our Notebook.

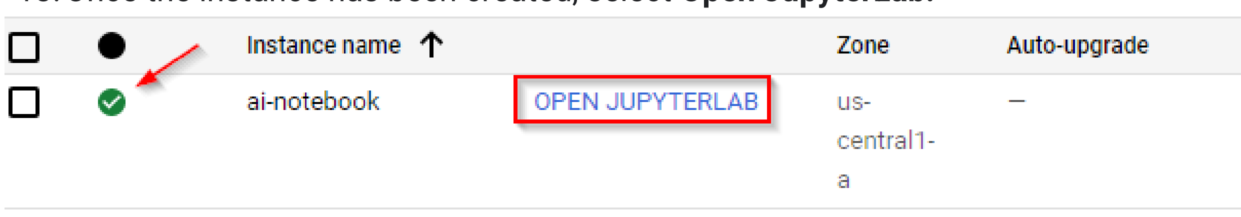


12. Navigate back to our Vertex AI -> Workbench. On the Notebook instances page, navigate to the **User-Managed Notebooks** tab and wait until `ai-notebook` is fully created.




It should take a few minutes for the notebook to be fully created.

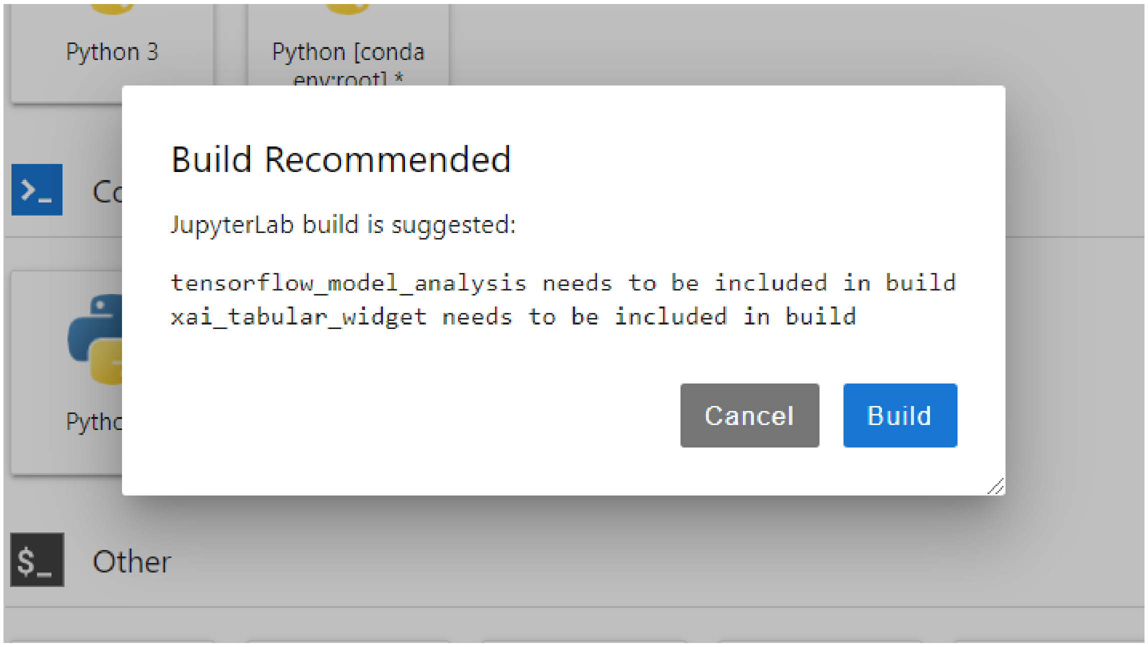
13. Once the instance has been created, select **Open JupyterLab:**




Check if the notebook is created

Check my progress

14. The default environment will suggest some additional build options, and we'll skip those, choose **Cancel**



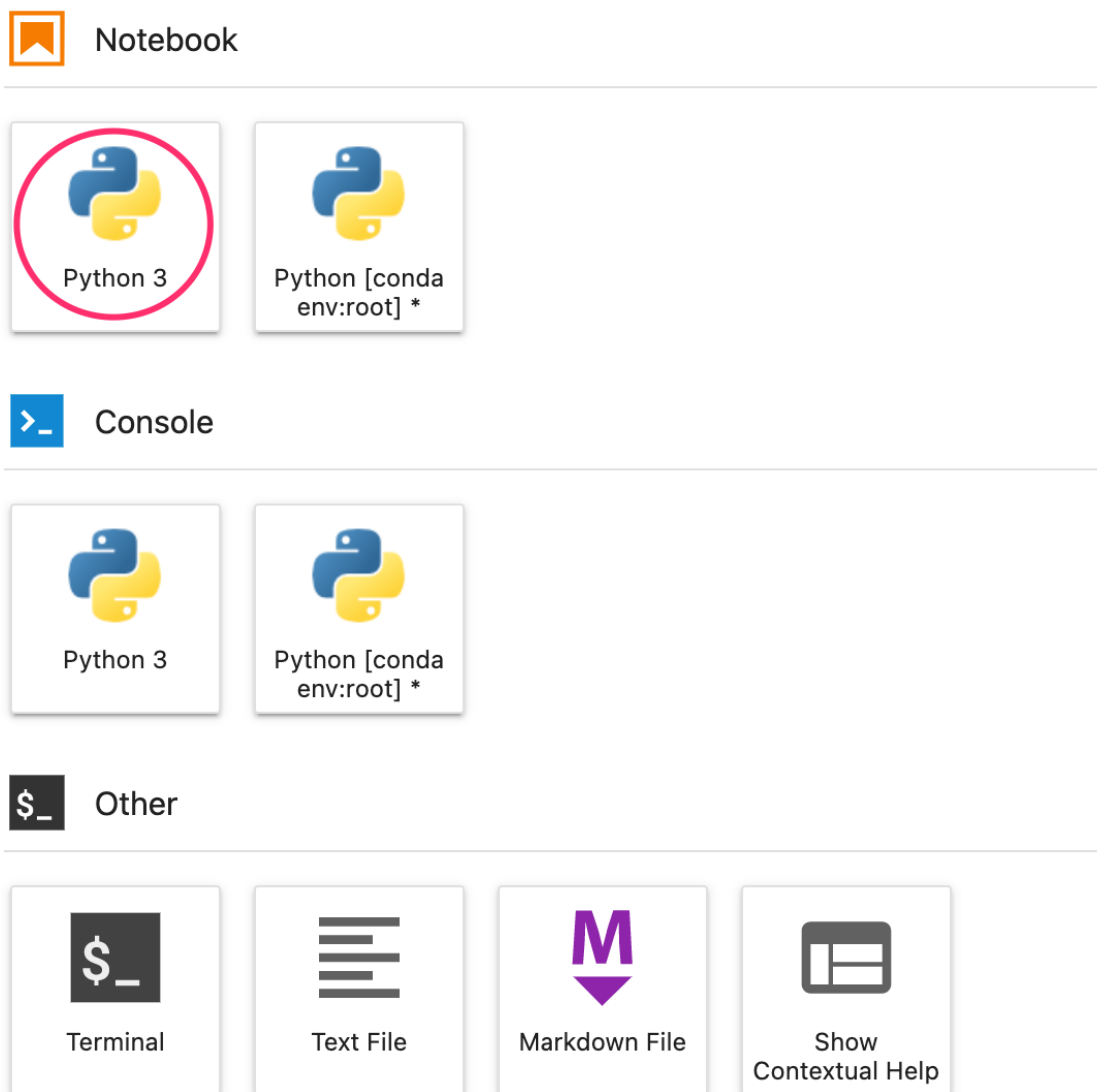

# Vertex Pipelines setup

There are a few additional libraries you'll need to install in order to use Vertex Pipelines:

- **Kubeflow Pipelines**: This is the SDK used to build the pipeline. Vertex Pipelines supports running pipelines built with both Kubeflow Pipelines or TFX.
- **Google Cloud Pipeline Components**: This library provides pre-built components that make it easier to interact with Vertex AI services from your pipeline steps.

## Step 1: Create Python notebook and install libraries

From the Launcher menu in your Notebook instance, create a notebook by selecting **Python 3**:



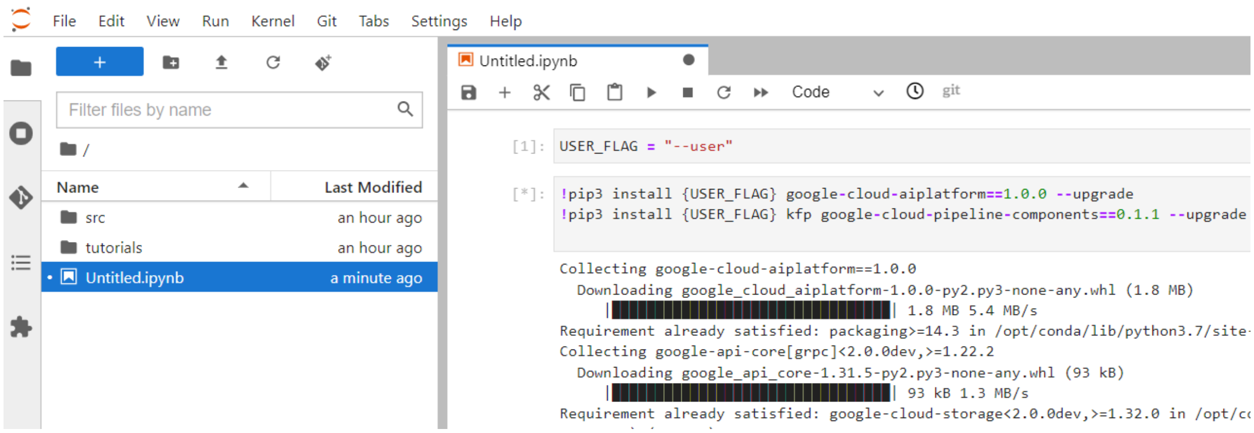

You can access the Launcher menu by clicking on the **+** sign in the top left of your notebook instance.

To install both services needed for this experiment, first set the user flag in a notebook cell:

```
USER_FLAG = "--user"
```

Then run the following from your notebook:

```
!pip3 install {USER_FLAG} google-cloud-aiplatform==1.7.0 --upgrade
!pip3 install {USER_FLAG} kfp==1.8.9 google-cloud-pipeline-components==0.2.0
```




After installing these packages you'll need to restart the kernel:

```python
import os
if not os.getenv("IS_TESTING"):
    # Automatically restart kernel after installs
    import IPython
    app = IPython.Application.instance()
    app.kernel.do_shutdown(True)
```

Finally, check that you have correctly installed the packages. The KFP SDK version should be >=1.6:

```
!python3 -c "import kfp; print('KFP SDK version: {}'.format(kfp.__version__))"
!python3 -c "import google_cloud_pipeline_components; print('google_cloud_pipeline_components version: {}'.format(google_cloud_pipeline_components.__version__))"
```

```
KFP SDK version: 1.8.10
google_cloud_pipeline_components version: 0.2.1
```

# Step 2: Set your project ID and bucket

Throughout this experiment you'll reference your Cloud Project ID and the bucket you created earlier. Next you'll create variables for each of those.

If you don't know your project ID you may be able to get it by running the following:

```
import os
PROJECT_ID = ""
# Get your Google Cloud project ID from gcloud
if not os.getenv("IS_TESTING"):
    shell_output=!gcloud config list --format 'value(core.project)'
2>/dev/null
    PROJECT_ID = shell_output[0]
    print("Project ID: ", PROJECT_ID)
```

```
Project ID:  datapipe-20220110-student10xin
```

Then create a variable to store your bucket name.

```
BUCKET_NAME="gs://" + PROJECT_ID + "-bucket"
print("Bucket: ", BUCKET_NAME)
```

```
Bucket: gs://datapipe-20220110-student10xin-bucket
```

## Step 3: Import libraries

Add the following to import the libraries you'll be using throughout this experiment:

```
from typing import NamedTuple
import kfp
from kfp import dsl
from kfp.v2 import compiler
from kfp.v2.dsl import (Artifact, Dataset, Input, InputPath, Model,
Output,
                        OutputPath, ClassificationMetrics, Metrics,
component)
from kfp.v2.google.client import AIPlatformClient
from google.cloud import aiplatform
from google_cloud_pipeline_components import aiplatform as gcc_aip
```

## Step 4: Define constants

The last thing you need to do before building the pipeline is define some constant variables. `PIPELINE_ROOT` is the Cloud Storage path where the artifacts created by your pipeline will be written. You're using `us-central1` as the region here, but if you used a different `region` when you created your bucket, update the REGION variable in the code below:

```
PATH=%env PATH
%env PATH={PATH}:/home/jupyter/.local/bin
REGION="us-central1"
PIPELINE_ROOT = f"{BUCKET_NAME}/pipeline_root/"
PIPELINE_ROOT
```

After running the code above, you should see the root directory for your pipeline printed. This is the Cloud Storage location where the artifacts from your pipeline will be written. It will be in the format of `gs://<bucket_name>/pipeline_root/`

# Creating an end-to-end ML pipeline

It's time to build your first ML pipeline. In this pipeline, you'll use the UCI Machine Learning Dry Beans dataset, from: KOKLU, M. and OZKAN, I.A., (2020), "Multiclass Classification of Dry Beans Using Computer Vision and Machine Learning Techniques." In Computers and Electronics in Agriculture, 174, 105507. DOI.

This pipeline will take over an hour to complete. So you will not need to wait the entire duration of the pipeline to complete the lab. Follow the steps, until your pipeline job has started.
This is a tabular dataset, and in your pipeline you'll use the dataset to train, evaluate, and deploy an AutoML model that classifies beans into one of 7 types based on their characteristics.

This pipeline will:

- Create a Dataset in Vertex AI
- Train a tabular classification model with AutoML

- Get evaluation metrics on this model
- Based on the evaluation metrics, decide whether to deploy the model using conditional logic in Vertex Pipelines
- Deploy the model to an endpoint using Vertex Prediction

Each of the steps outlined will be a component. Most of the pipeline steps will use pre-built components for Vertex AI services via the `google_cloud_pipeline_components` library yoy imported earlier in this codelab. In this section, we'll define one custom component first. Then, we'll define the rest of the pipeline steps using pre-built components. Pre-built components make it easier to access Vertex AI services, like model training and deployment.

The majority of time for this step is for the AutoML training piece of this pipeline, which will take about an hour.

## Step 1: A custom component for model evaluation

The custom component you'll define will be used towards the end of the pipeline once model training has completed. This component will do a few things:

- Get the evaluation metrics from the trained AutoML classification model
- Parse the metrics and render them in the Vertex Pipelines UI
- Compare the metrics to a threshold to determine whether the model should be deployed

Before defining the component, understand its input and output parameters. As input, this pipeline takes some metadata on your Cloud project, the resulting trained model (you'll define this component later), the model's evaluation metrics, and a `thresholds_dict_str`. The `thresholds_dict_str` is something you'll define when you run your pipeline. In the case of this classification model, this will be the area under the ROC curve value for which you should deploy the model. For example, if you pass in 0.95, that means you'd only like your pipeline to deploy the model if this metric is above 95%.

The evaluation component returns a string indicating whether or not to deploy the model. Add the following in a notebook cell to create this custom component:

```
@component(
    base_image="gcr.io/deeplearning-platform-release/tf2-cpu.2-3:latest",
    output_component_file="tables_eval_component.yaml", # Optional: you
can use this to load the component later
    packages_to_install=["google-cloud-aiplatform"],
```

```
)
def classif_model_eval_metrics(
    project: str,
    location: str,  # "us-central1",
    api_endpoint: str,  # "us-central1-aiplatform.googleapis.com",
    thresholds_dict_str: str,
    model: Input[Model],
    metrics: Output[Metrics],
    metricsc: Output[ClassificationMetrics],
) -> NamedTuple("Outputs", [("dep_decision", str)]):  # Return parameter.
    """This function renders evaluation metrics for an AutoML Tabular
classification model.
    It retrieves the classification model evaluation generated by the
AutoML Tabular training
    process, does some parsing, and uses that info to render the ROC curve
and confusion matrix
    for the model. It also uses given metrics threshold information and
compares that to the
    evaluation results to determine whether the model is sufficiently
accurate to deploy.
    """
    import json
    import logging
    from google.cloud import aiplatform
    # Fetch model eval info
    def get_eval_info(client, model_name):
        from google.protobuf.json_format import MessageToDict
        response = client.list_model_evaluations(parent=model_name)
        metrics_list = []
        metrics_string_list = []
        for evaluation in response:
            print("model_evaluation")
            print(" name:", evaluation.name)
            print(" metrics_schema_uri:", evaluation.metrics_schema_uri)
            metrics = MessageToDict(evaluation._pb.metrics)
            for metric in metrics.keys():
                logging.info("metric: %s, value: %s", metric,
metrics[metric])
            metrics_str = json.dumps(metrics)
            metrics_list.append(metrics)
            metrics_string_list.append(metrics_str)
        return (
            evaluation.name,
            metrics_list,
            metrics_string_list,
        )
    # Use the given metrics threshold(s) to determine whether the model is
    # accurate enough to deploy.
    def classification_thresholds_check(metrics_dict, thresholds_dict):
        for k, v in thresholds_dict.items():
            logging.info("k {}, v {}".format(k, v))
            if k in ["auRoc", "auPrc"]:  # higher is better
                if metrics_dict[k] < v:  # if under threshold, don't
deploy
```

```
                    logging.info(
                        "{} < {}; returning False".format(metrics_dict[k],
v)
                    )
                    return False
        logging.info("threshold checks passed.")
        return True
    def log_metrics(metrics_list, metricsc):
        test_confusion_matrix = metrics_list[0]["confusionMatrix"]
        logging.info("rows: %s", test_confusion_matrix["rows"])
        # log the ROC curve
        fpr = []
        tpr = []
        thresholds = []
        for item in metrics_list[0]["confidenceMetrics"]:
            fpr.append(item.get("falsePositiveRate", 0.0))
            tpr.append(item.get("recall", 0.0))
            thresholds.append(item.get("confidenceThreshold", 0.0))
        print(f"fpr: {fpr}")
        print(f"tpr: {tpr}")
        print(f"thresholds: {thresholds}")
        metricsc.log_roc_curve(fpr, tpr, thresholds)
        # log the confusion matrix
        annotations = []
        for item in test_confusion_matrix["annotationSpecs"]:
            annotations.append(item["displayName"])
        logging.info("confusion matrix annotations: %s", annotations)
        metricsc.log_confusion_matrix(
            annotations,
            test_confusion_matrix["rows"],
        )
        # log textual metrics info as well
        for metric in metrics_list[0].keys():
            if metric != "confidenceMetrics":
                val_string = json.dumps(metrics_list[0][metric])
                metrics.log_metric(metric, val_string)
        # metrics.metadata["model_type"] = "AutoML Tabular classification"
    logging.getLogger().setLevel(logging.INFO)
    aiplatform.init(project=project)
    # extract the model resource name from the input Model Artifact
    model_resource_path = model.uri.replace("aiplatform://v1/", "")
    logging.info("model path: %s", model_resource_path)
    client_options = {"api_endpoint": api_endpoint}
    # Initialize client that will be used to create and send requests.
    client =
aiplatform.gapic.ModelServiceClient(client_options=client_options)
    eval_name, metrics_list, metrics_str_list = get_eval_info(
        client, model_resource_path
    )
    logging.info("got evaluation name: %s", eval_name)
    logging.info("got metrics list: %s", metrics_list)
    log_metrics(metrics_list, metricsc)
    thresholds_dict = json.loads(thresholds_dict_str)
```

```
    deploy = classification_thresholds_check(metrics_list[0],
thresholds_dict)
    if deploy:
        dep_decision = "true"
    else:
        dep_decision = "false"
    logging.info("deployment decision is %s", dep_decision)
    return (dep_decision,)
```

## Step 2: Adding Google Cloud pre-built components

In this step you'll define the rest of your pipeline components and see how they all fit together.

First, define the display name for your pipeline run using a timestamp:

```
import time
DISPLAY_NAME = 'automl-beans{}'.format(str(int(time.time())))
print(DISPLAY_NAME)
```

```
automl-beans1641841035
```

Then copy the following into a new notebook cell:

```
@kfp.dsl.pipeline(name="automl-tab-beans-training-v2",
                  pipeline_root=PIPELINE_ROOT)
def pipeline(
    bq_source: str = "bq://aju-dev-demos.beans.beans1",
    display_name: str = DISPLAY_NAME,
    project: str = PROJECT_ID,
    gcp_region: str = "us-central1",
    api_endpoint: str = "us-central1-aiplatform.googleapis.com",
    thresholds_dict_str: str = '{"auRoc": 0.95}',
):
    dataset_create_op = gcc_aip.TabularDatasetCreateOp(
        project=project, display_name=display_name, bq_source=bq_source
    )
    training_op = gcc_aip.AutoMLTabularTrainingJobRunOp(
        project=project,
        display_name=display_name,
        optimization_prediction_type="classification",
        budget_milli_node_hours=1000,
```

```
        column_transformations=[
            {"numeric": {"column_name": "Area"}},
            {"numeric": {"column_name": "Perimeter"}},
            {"numeric": {"column_name": "MajorAxisLength"}},
            {"numeric": {"column_name": "MinorAxisLength"}},
            {"numeric": {"column_name": "AspectRation"}},
            {"numeric": {"column_name": "Eccentricity"}},
            {"numeric": {"column_name": "ConvexArea"}},
            {"numeric": {"column_name": "EquivDiameter"}},
            {"numeric": {"column_name": "Extent"}},
            {"numeric": {"column_name": "Solidity"}},
            {"numeric": {"column_name": "roundness"}},
            {"numeric": {"column_name": "Compactness"}},
            {"numeric": {"column_name": "ShapeFactor1"}},
            {"numeric": {"column_name": "ShapeFactor2"}},
            {"numeric": {"column_name": "ShapeFactor3"}},
            {"numeric": {"column_name": "ShapeFactor4"}},
            {"categorical": {"column_name": "Class"}},
        ],
        dataset=dataset_create_op.outputs["dataset"],
        target_column="Class",
    )
    model_eval_task = classif_model_eval_metrics(
        project,
        gcp_region,
        api_endpoint,
        thresholds_dict_str,
        training_op.outputs["model"],
    )
    with dsl.Condition(
        model_eval_task.outputs["dep_decision"] == "true",
        name="deploy_decision",
    ):
        deploy_op = gcc_aip.ModelDeployOp(  # noqa: F841
            model=training_op.outputs["model"],
            project=project,
            machine_type="n1-standard-4",
        )
```

What's happening in this code:

- First, just as in the previous pipeline, you define the input parameters this pipeline takes. You need to set these manually since they don't depend on the output of other steps in the pipeline.

- The rest of the pipeline uses a few pre-built components for interacting with Vertex AI services:

    - `TabularDatasetCreateOp` creates a tabular dataset in Vertex AI given a dataset source either in Cloud Storage or BigQuery. In this pipeline, you're passing the data via a BigQuery table URL.
    - `AutoMLTabularTrainingJobRunOp` kicks off an AutoML training job for a tabular dataset. You pass a few configuration parameters to this component, including the model type (in this case, classification), some data on the columns, how long you'd like to run training for, and a pointer to the dataset. Notice that to pass in the dataset to this component, you're providing the output of the previous component via `dataset_create_op.outputs["dataset"]`.
    - `ModelDeployOp` deploys a given model to an endpoint in Vertex AI. There are additional configuration options available, but here you're providing the endpoint machine type, project, and model you'd like to deploy. You're passing in the model by accessing the outputs of the training step in your pipeline.
- This pipeline also makes use of **conditional logic**, a feature of Vertex Pipelines that lets you define a condition, along with different branches based on the result of that condition. Remember that when you defined the pipeline you passed a `thresholds_dict_str` parameter. This is the accuracy threshold you're using to determine whether to deploy your model to an endpoint. To implement this, make use of the `Condition` class from the KFP SDK. The condition passed in is the output of the custom eval component you defined earlier in this lab. If this condition is true, the pipeline will continue to execute the `deploy_op` component. If accuracy doesn't meet the predefined threshold, the pipeline will stop here and won't deploy a model.

## Step 3: Compile and run the end-to-end ML pipeline

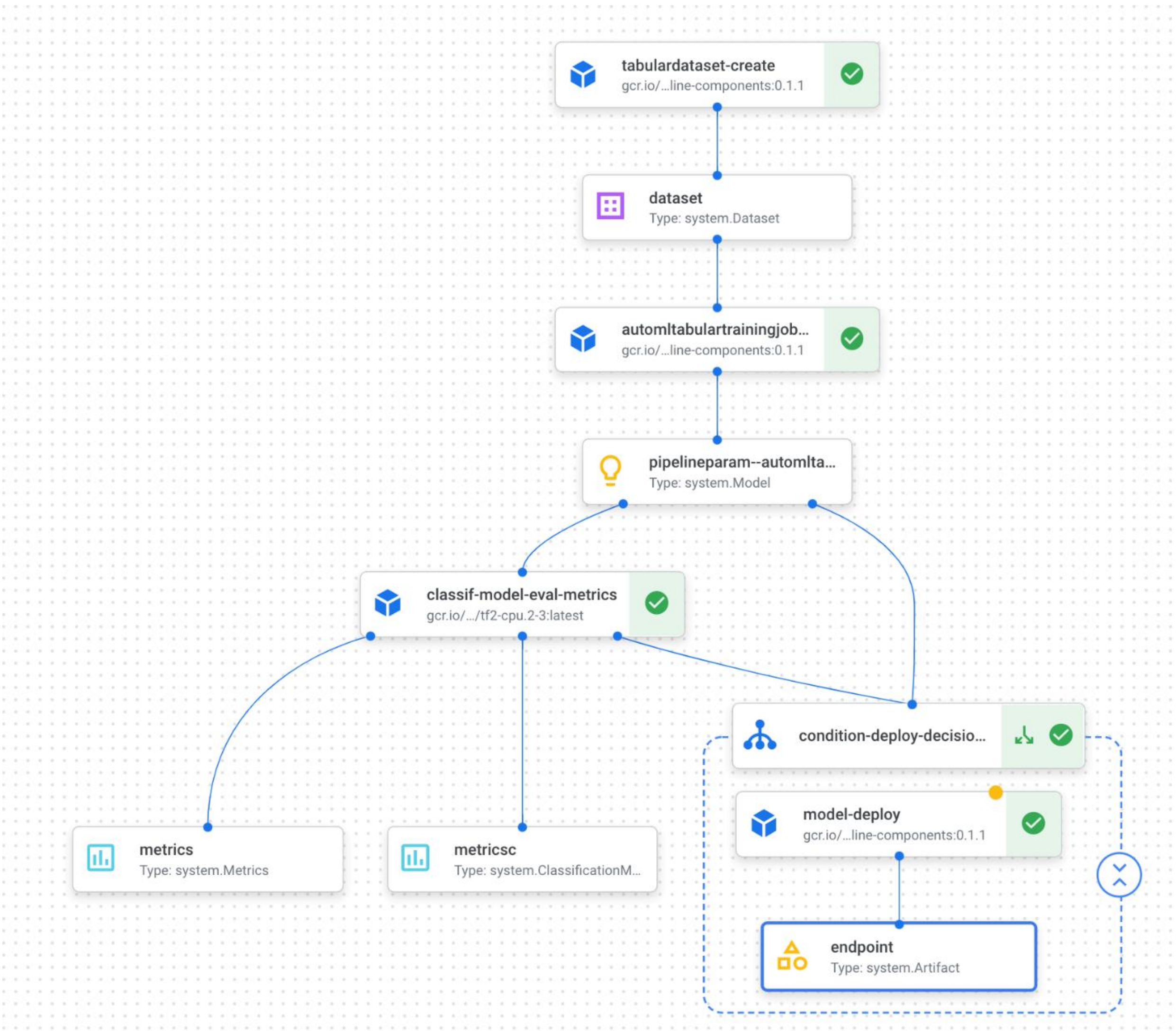
With the full pipeline defined, it's time to compile it:

```
compiler.Compiler().compile(
    pipeline_func=pipeline, package_path="tab_classif_pipeline.json"
)
```

Next, kick off a pipeline run:

```
response = api_client.create_run_from_job_spec(
    "tab_classif_pipeline.json", pipeline_root=PIPELINE_ROOT,
    parameter_values={"project": PROJECT_ID,
                      "display_name": DISPLAY_NAME}
)
```

Click on the link shown after running the cell above to see your pipeline in the console. **This pipeline should take a little over an hour to run**. Most of the time is spent in the AutoML training step. The completed pipeline will look something like this:



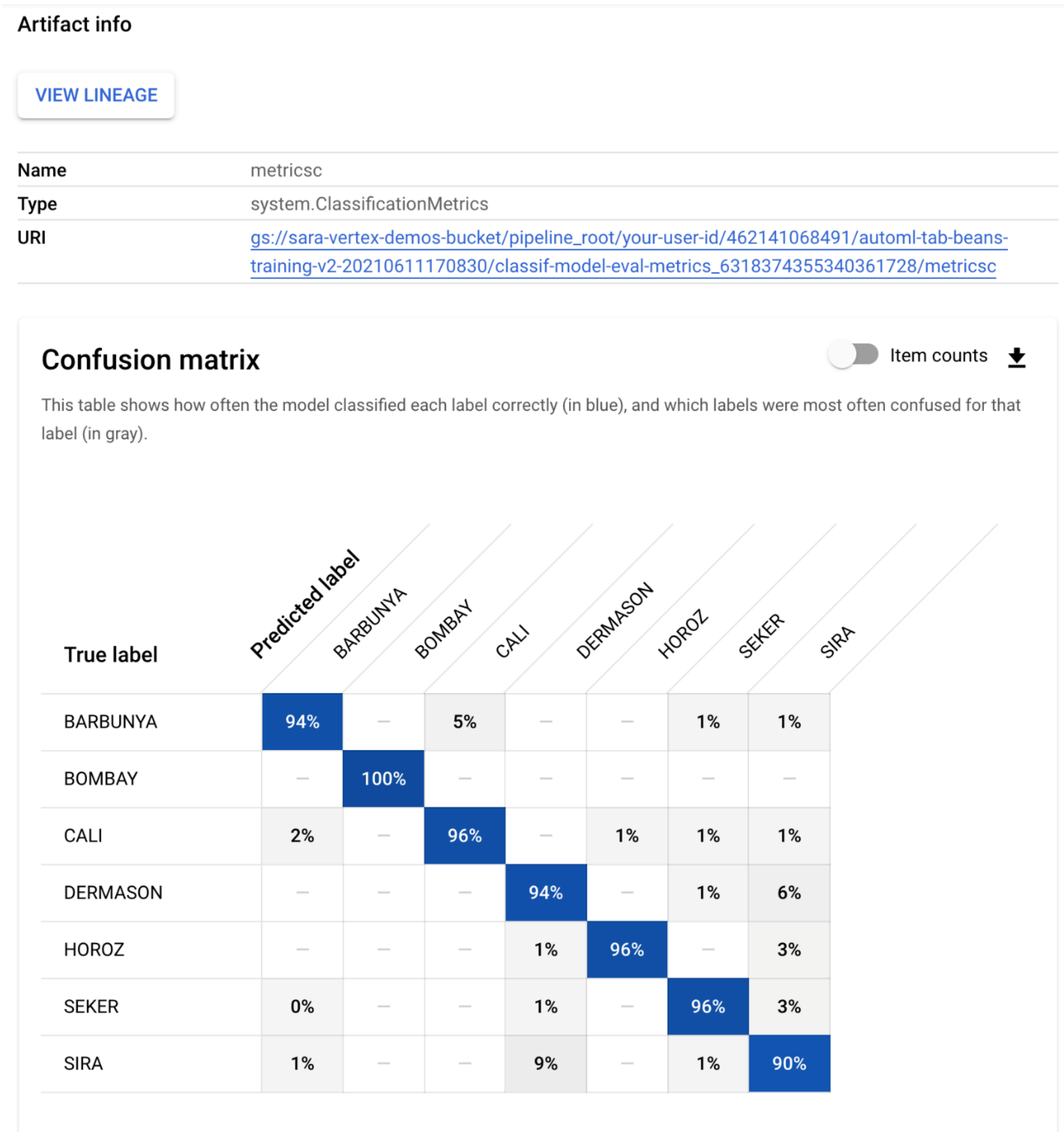

If you toggle the "Expand artifacts" button at the top, you'll be able to see details for the different artifacts created from your pipeline. For example, if you click on the `dataset` artifact, you'll see details on the Vertex AI dataset that was created. You can click the link here to go to the page for that dataset:

### Artifact info

VIEW LINEAGE

| | |
|---|---|
| **Name** | dataset |
| **Type** | system.Dataset |
| **URI** | aiplatform://v1/projects/462141068491/locations/us-central1/datasets/4607129642241888416 |

Similarly, to see the resulting metric visualizations from your custom evaluation component, click on the artifact called **metricsc**. On the right side of your dashboard, you'll be able to see the confusion matrix for this model:



Artifact info

VIEW LINEAGE

| | |
|---|---|
| Name | metricsc |
| Type | system.ClassificationMetrics |
| URI | gs://sara-vertex-demos-bucket/pipeline_root/your-user-id/462141068491/automl-tab-beans-training-v2-20210611170830/classif-model-eval-metrics_6318374355340361728/metricsc |

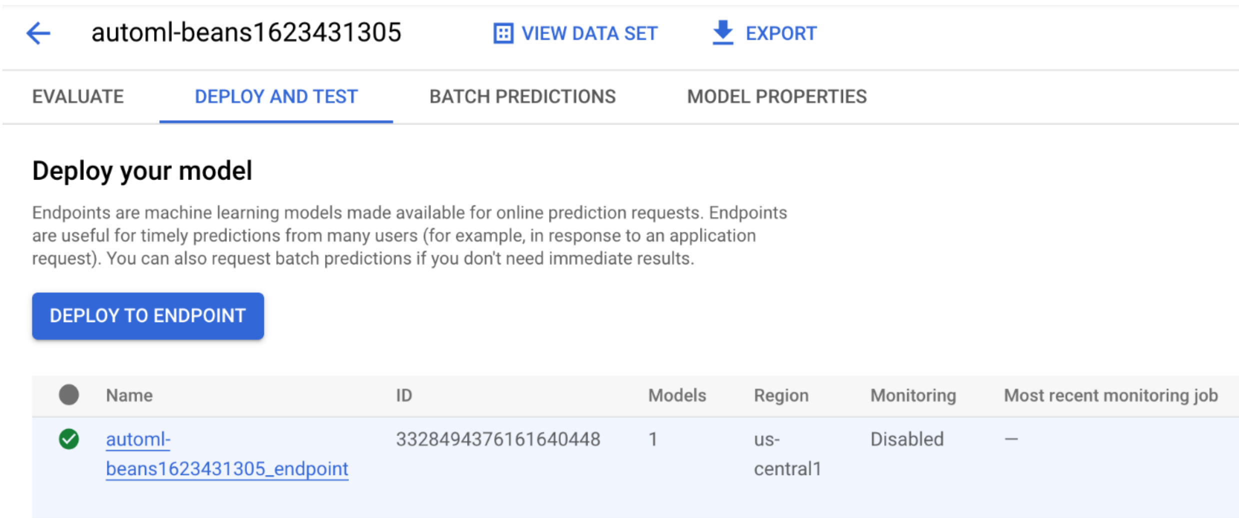
**Confusion matrix**

Item counts

This table shows how often the model classified each label correctly (in blue), and which labels were most often confused for that label (in gray).

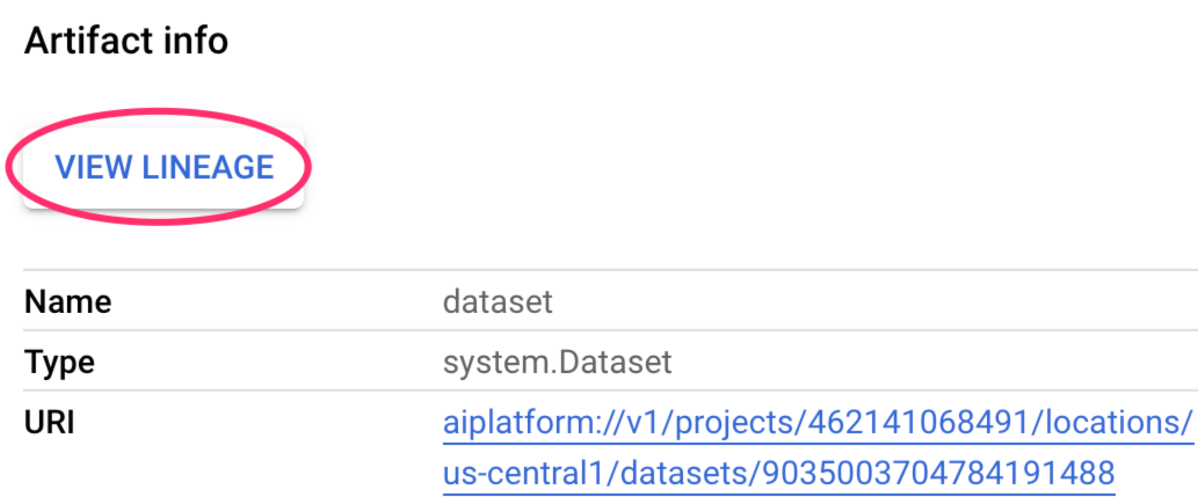
| True label \ Predicted label | BARBUNYA | BOMBAY | CALI | DERMASON | HOROZ | SEKER | SIRA |
|---|---|---|---|---|---|---|---|
| BARBUNYA | 94% | – | 5% | – | – | 1% | 1% |
| BOMBAY | – | 100% | – | – | – | – | – |
| CALI | 2% | – | 96% | – | 1% | 1% | 1% |
| DERMASON | – | – | – | 94% | – | 1% | 6% |
| HOROZ | – | – | – | 1% | 96% | – | 3% |
| SEKER | 0% | – | – | 1% | – | 96% | 3% |
| SIRA | 1% | – | – | 9% | – | 1% | 90% |

To see the model and endpoint created from this pipeline run, go to the models section and click on the model named `automl-beans`. There you should see this model deployed to an endpoint:

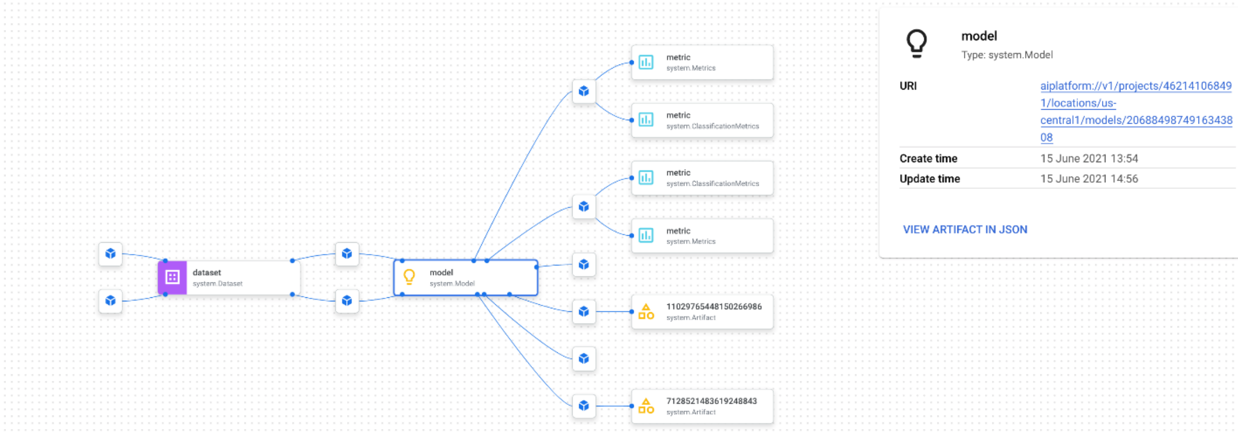



You can also access this page by clicking on the **endpoint** artifact in your pipeline graph.

In addition to looking at the pipeline graph in the console, you can also use Vertex Pipelines for **Lineage Tracking**. Lineage tracking means tracking artifacts created throughout your pipeline. This can help you understand where artifacts were created and how they are being used throughout an ML workflow. For example, to see the lineage tracking for the dataset created in this pipeline, click on the dataset artifact and then **View Lineage**:




This shows all the places this artifact is being used:




Wait until the training job in your pipeline has started and then check your progress below.

Check if your end-to-end ML pipeline training job has started

Check my progress

## Step 4: Comparing metrics across pipeline runs

If you run this pipeline multiple times, you may want to compare metrics across runs. You can use the `aiplatform.get_pipeline_df()` method to access run metadata. Here, we'll get metadata for all runs of this pipeline and load it into a Pandas DataFrame:

```
pipeline_df = aiplatform.get_pipeline_df(pipeline="automl-tab-beans-
training-v2")
small_pipeline_df = pipeline_df.head(2)
small_pipeline_df
```

Copied!

content_copy

You've now learned how to build, run, and get metadata for an end-to-end ML pipeline on Vertex Pipelines.

# Congratulations!

You have learned how to build, run, and get metadata for an end-to-end ML pipeline on Vertex Pipelines.