

# 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

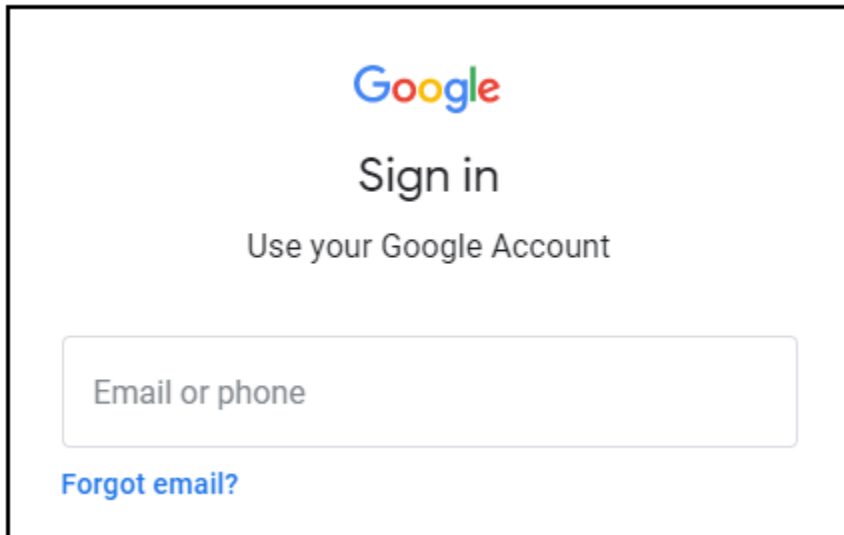
### What you need

To complete this experiment, you need:

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

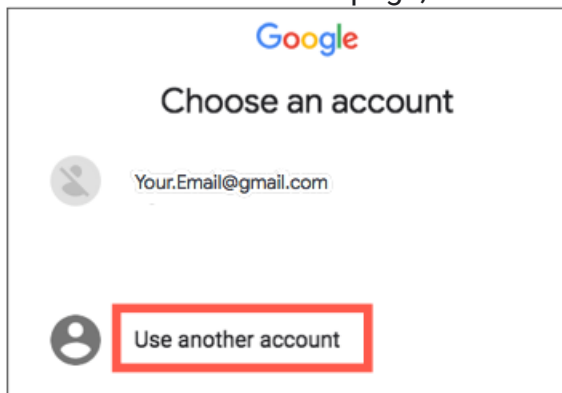
### How to start your experiment and sign in to the Google Cloud Console

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)

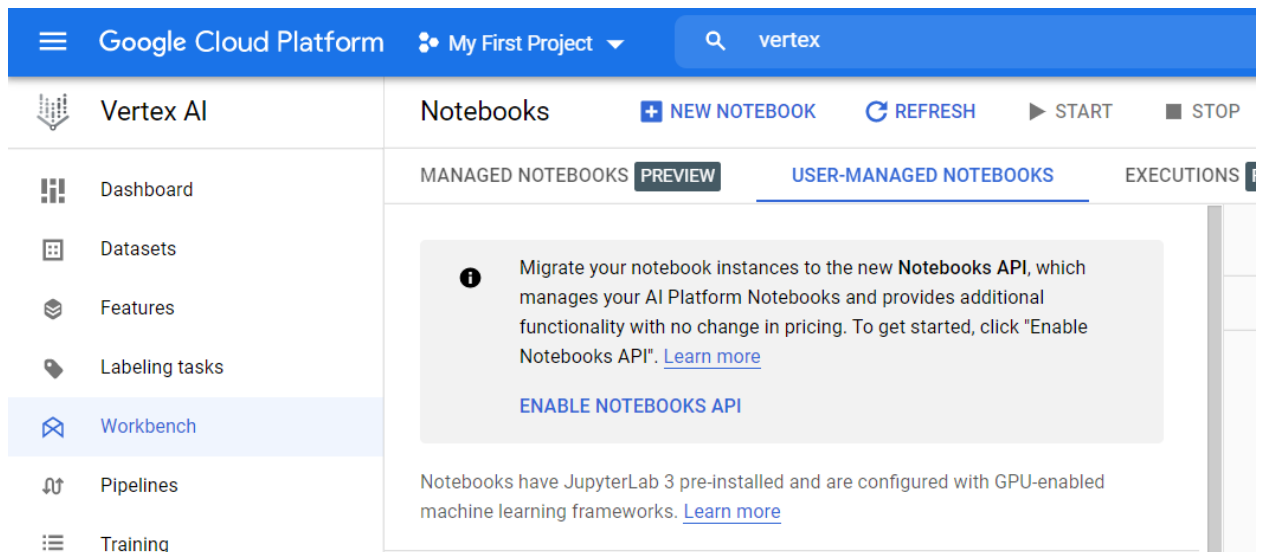
Confirm the recovery options without changing them.

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

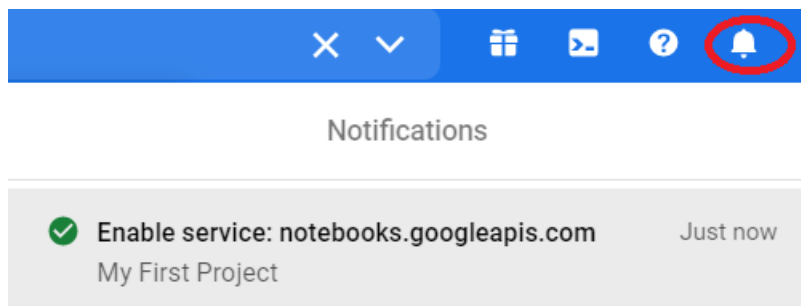
**Note:** You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left.

# Create an Vertex Notebooks instance

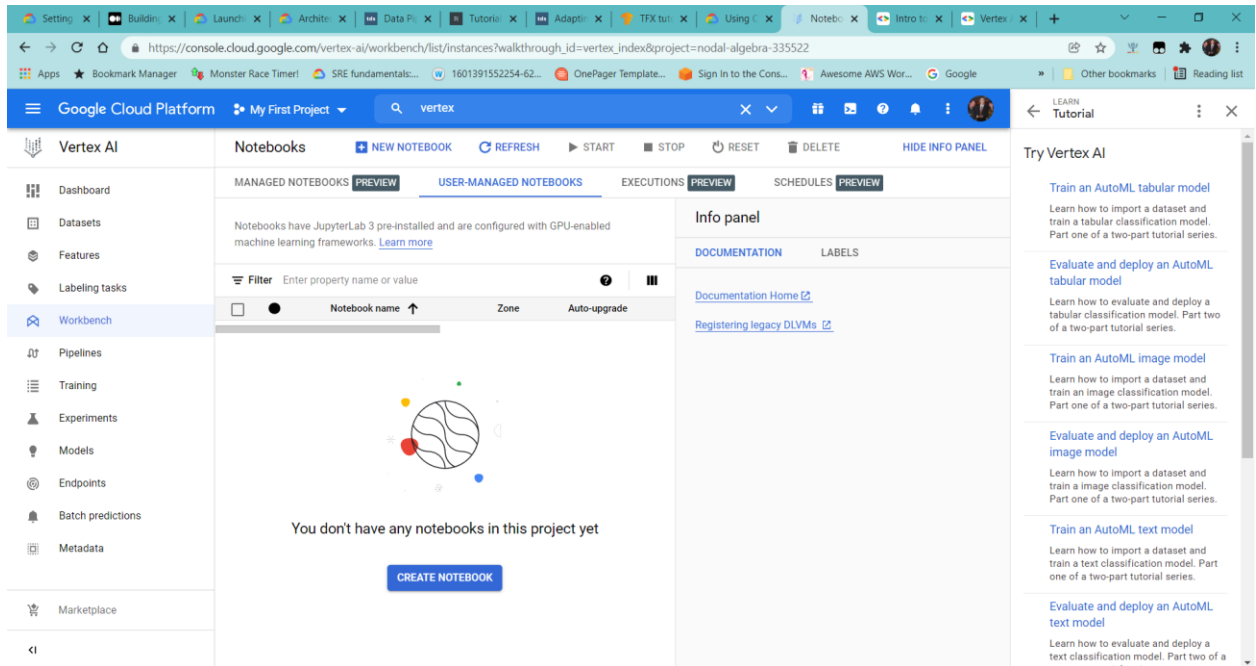
1. Click on the **Navigation Menu**.
2. Navigate to **Vertex AI**, then to **Workbench**.
3. Click **ENABLE NOTEBOOKS API**



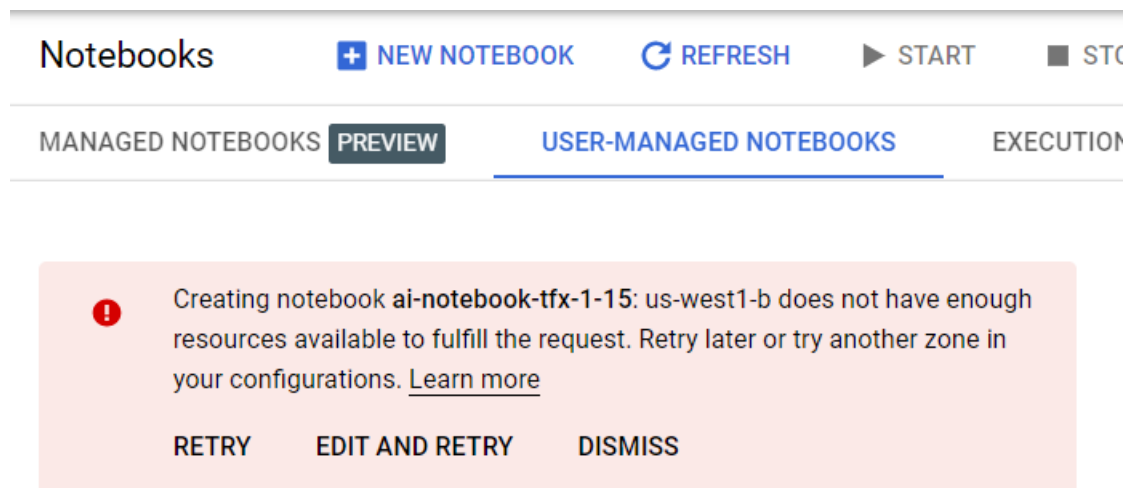
4. Click the Bell icon in the top level navigation on the right to view Notifications
5. Notice that the Notebook Enablement is posted there



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 2.1, 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.



Notebooks have JupyterLab 3 pre-installed and are configured with GPU-enabled machine learning frameworks. [Learn more](#)

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

Google Cloud Platform My First Project compute engine

Compute Engine VM instances CREATE INSTANCE

Virtual machines VM instances Instance templates Sole-tenant nodes Machine images TPUs Committed use discounts

INSTANCES INSTANCE SCHEDULE

VM instances are highly configurable virtual machines for running workloads on Google infrastructure. [Learn more](#)

Filter Enter property name or value

<input type="checkbox"/>	Status	Name ↑	Zone	Recon	Connect
<input type="checkbox"/>	✓	ai-notebook	us-central1-a		SSH

- 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.

INSTANCES INSTANCE SCHEDULE

VM instances are highly configurable virtual machines for running workloads on Google infrastructure. [Learn more](#)

Filter Enter property name or value

<input checked="" type="checkbox"/>	Status	Name ↑	Zone	Recon	Connect
<input checked="" type="checkbox"/>	⌚	ai-notebook	us-central1-a		SSH

Related actions DISMISS

- View billing report
- Monitor VMs
- Explore VM logs
- Set up firewall rules

ai-notebook

PERMISSIONS LABELS MONITORING

Edit or delete permissions below or "Add Principal" to grant new

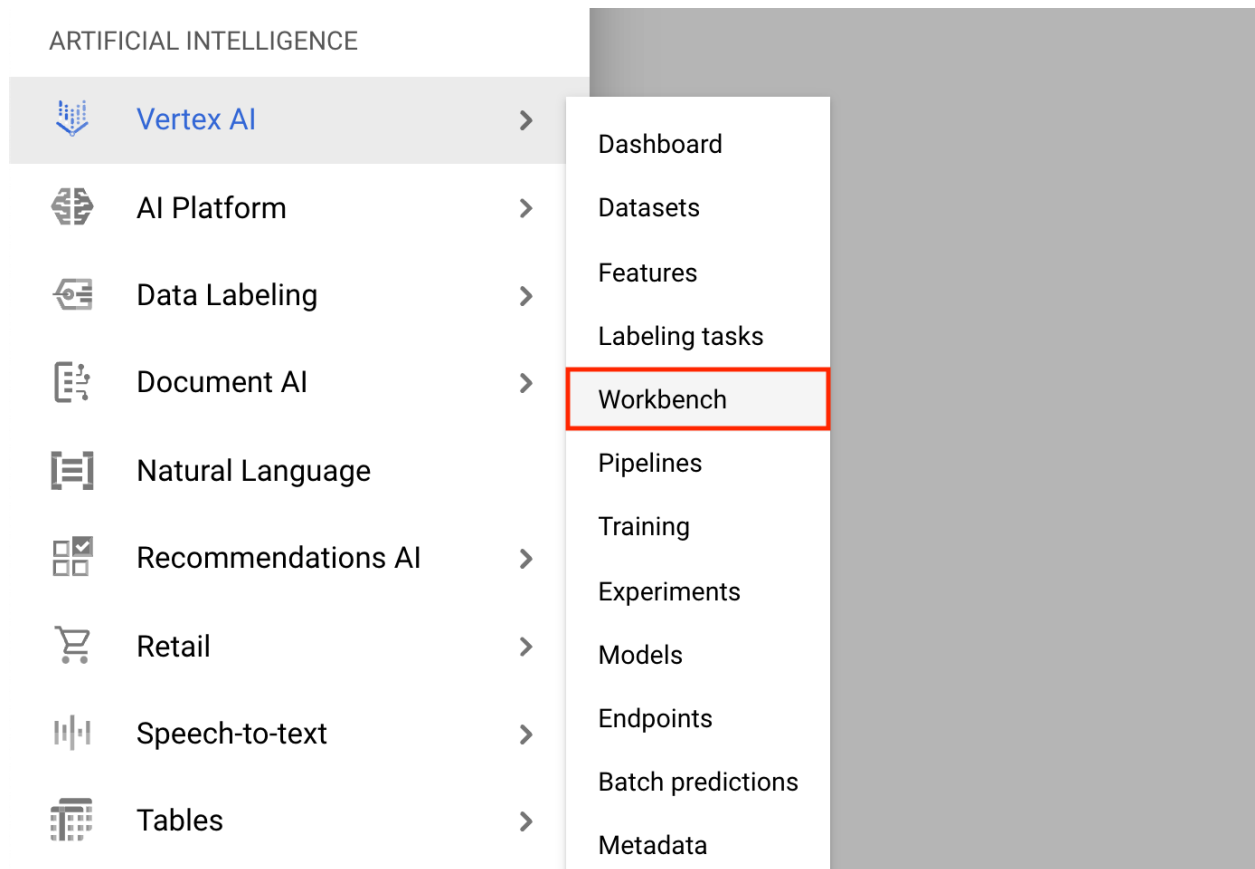
ADD PRINCIPAL

Show inherited permissions

Filter Enter property name or value

Role / Principal ↑	Inheritance
AI Platform Notebooks Service Agent (1)	
Cloud Composer API Service Agent (1)	
Editor (3)	
Kubernetes Engine Service Agent (1)	
Owner (1)	

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.

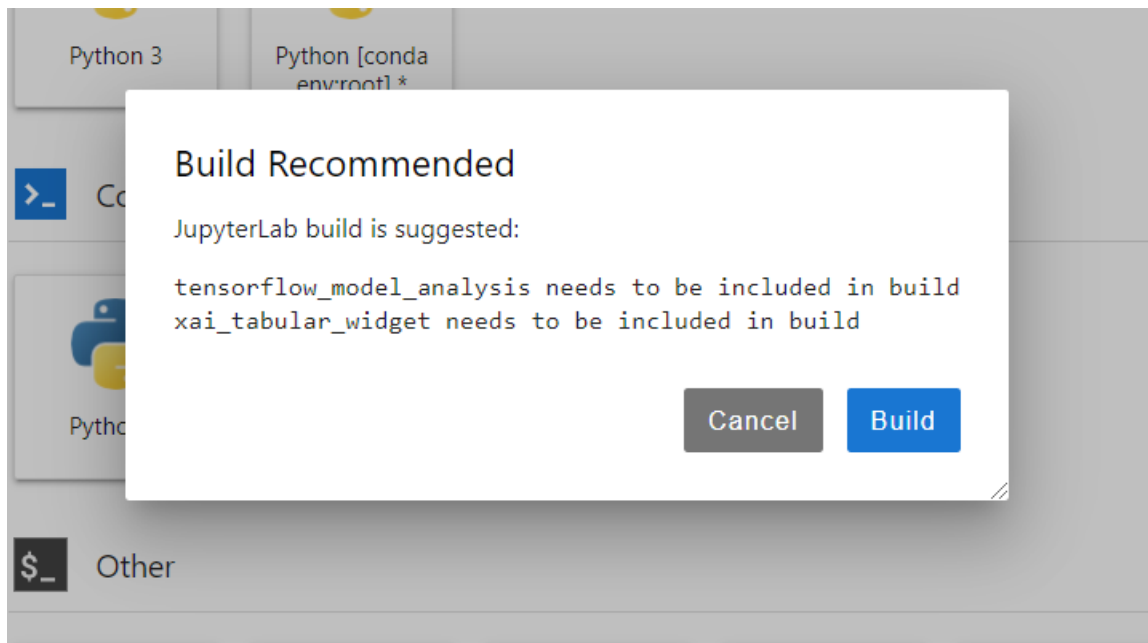
13. Once the instance has been created, select **Open JupyterLab**:

<input type="checkbox"/>	<input checked="" type="checkbox"/>	Instance name ↑	Zone	Auto-upgrade
<input type="checkbox"/>	<input checked="" type="checkbox"/>	ai-notebook	us-central1-a	—

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

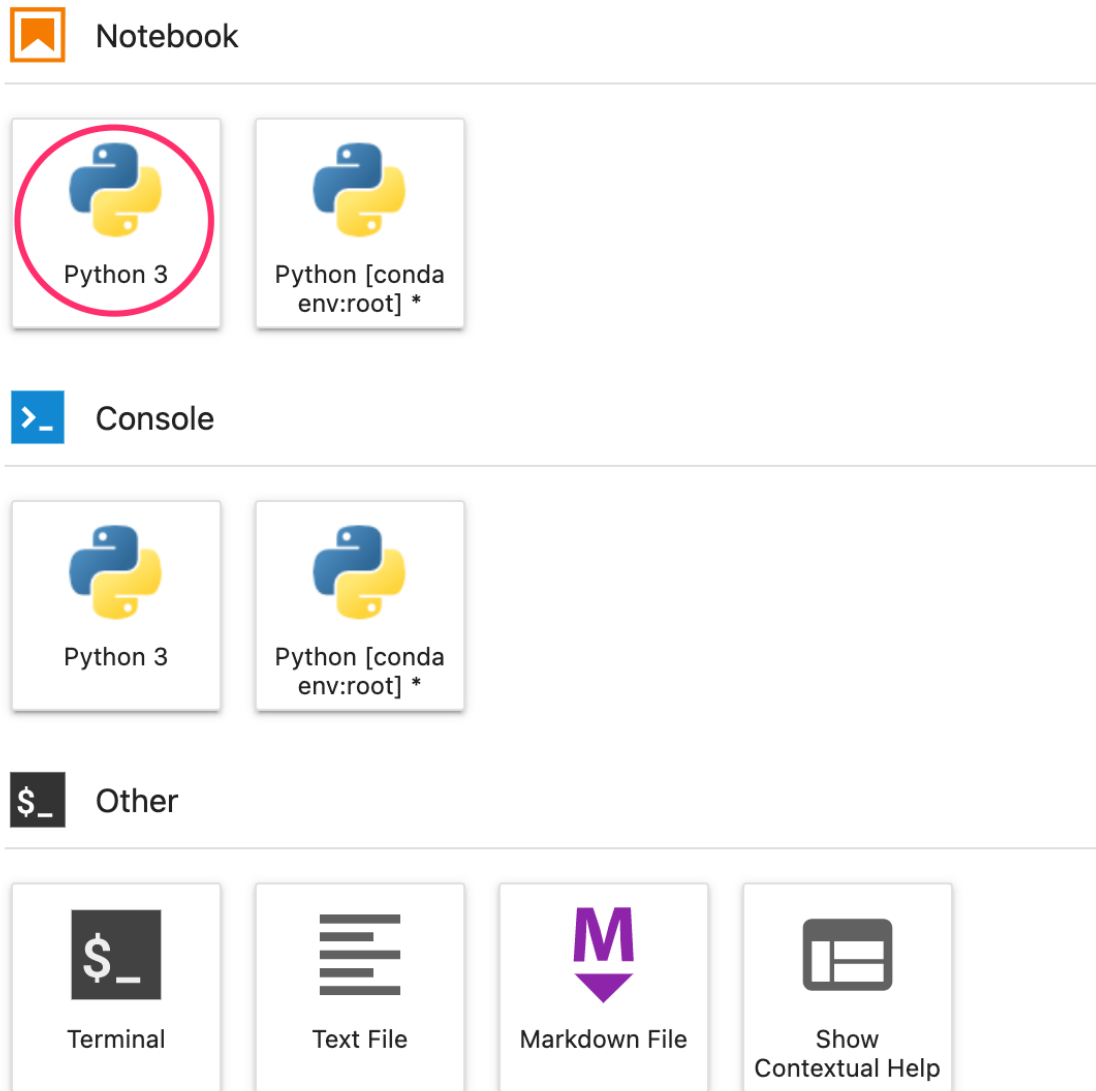
There are a few additional libraries you'll need to install in order to use Vertex Pipelines:

- **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**:



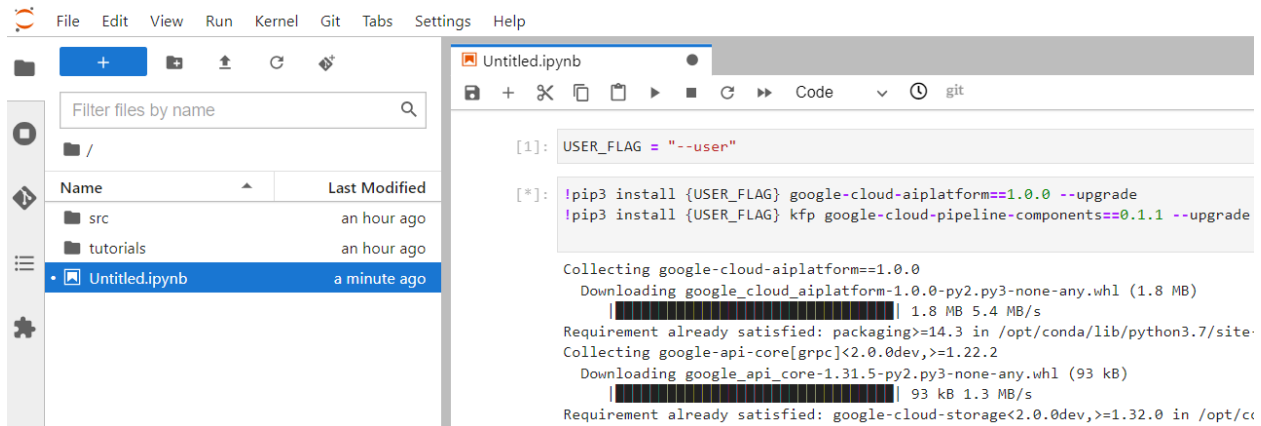
You can access the Launcher menu by clicking on the + sign in the top left of your notebook instance.

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 --upgrade
!pip3 install {USER_FLAG} kfp google-cloud-pipeline-components --upgrade
```



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

```
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 your first pipeline

Create a short pipeline using the KFP SDK. This pipeline doesn't do anything ML related (don't worry, you'll get there!), this exercise is to teach you:

- How to create custom components in the KFP SDK
- How to run and monitor a pipeline in Vertex Pipelines

You'll create a pipeline that prints out a sentence using two outputs: a product name and an emoji description. This pipeline will consist of three components:

- `product_name`: This component will take a product name as input, and return that string as output.
- `emoji`: This component will take the text description of an emoji and convert it to an emoji. For example, the text code for ✨ is "sparkles". This component uses an emoji library to show you how to manage external dependencies in your pipeline.
- `build_sentence`: This final component will consume the output of the previous two to build a sentence that uses the emoji. For example, the resulting output might be "Vertex Pipelines is ✨".

## Step 1: Create a Python function based component

Using the KFP SDK, you can create components based on Python functions. First build the `product_name` component, which simply takes a string as input and returns that string.

Add the following to your notebook:

```
@component(base_image="python:3.9", output_component_file="first-  
component.yaml")  
def product_name(text: str) -> str:  
    return text
```

Take a closer look at the syntax here:

- The `@component` decorator compiles this function to a component when the pipeline is run. You'll use this anytime you write a custom component.
- The `base_image` parameter specifies the container image this component will use.
- The `output_component_file` parameter is optional, and specifies the yaml file to write the compiled component to. After running the cell you should see that file written to your notebook instance. If you wanted to share this component with someone, you could send them the generated yaml file and have them load it with the following:

```
product_name_component = kfp.components.load_component_from_file('./first-  
component.yaml')
```

The `-> str` after the function definition specifies the output type for this component.

## Step 2: Create two additional components

To complete the pipeline, create two more components. The first one takes a string as input, and converts this string to its corresponding emoji if there is one. It returns a tuple with the input text passed, and the resulting emoji:

```

@component(packages_to_install=["emoji"])
def emoji(
    text: str,
) -> NamedTuple(
    "Outputs",
    [
        ("emoji_text", str), # Return parameters
        ("emoji", str),
    ],
):
    import emoji
    emoji_text = text
    emoji_str = emoji.emojize(':' + emoji_text + ':', use_aliases=True)
    print("output one: {}; output_two: {}".format(emoji_text, emoji_str))
    return (emoji_text, emoji_str)

```

This component is a bit more complex than the previous one. Here's what's new:

- The `packages_to_install` parameter tells the component any external library dependencies for this container. In this case, you're using a library called `emoji`.
- This component returns a `NamedTuple` called `Outputs`. Notice that each of the strings in this tuple have keys: `emoji_text` and `emoji`. You'll use these in your next component to access the output.

The final component in this pipeline will consume the output of the first two and combine them to return a string:

```

@component
def build_sentence(
    product: str,
    emoji: str,
    emoji_text: str
) -> str:
    print("We completed the pipeline, hooray!")
    end_str = product + " is "
    if len(emoji) > 0:
        end_str += emoji
    else:
        end_str += emoji_text
    return(end_str)

```

You might be wondering: how does this component know to use the output from the previous steps you defined? Good question! You will tie it all together in the next step.

## Step 3: Putting the components together into a pipeline

The component definitions defined above created factory functions that can be used in a pipeline definition to create steps. To set up a pipeline, use the `@dsl.pipeline` decorator, give the pipeline a name and description, and provide the root path where your pipeline's artifacts should be written. By artifacts, it means any output files generated by your pipeline. This intro pipeline doesn't generate any, but your next pipeline will.

In the next block of code you define an `intro_pipeline` function. This is where you specify the inputs to your initial pipeline steps, and how steps connect to each other:

- `product_task` takes a product name as input. Here you're passing "Vertex Pipelines" but you can change this to whatever you'd like.
- `emoji_task` takes the text code for an emoji as input. You can also change this to whatever you'd like. For example, "party\_face" refers to the 🎉 emoji. Note that since both this and the `product_task` component don't have any steps that feed input into them, you manually specify the input for these when you define your pipeline.
- The last step in the pipeline - `consumer_task` has three input parameters:
  - The output of `product_task`. Since this step only produces one output, you can reference it via `product_task.output`.
  - The emoji output of the `emoji_task` step. See the `emoji` component defined above where you named the output parameters.
  - Similarly, the `emoji_text` named output from the `emoji` component. In case your pipeline is passed text that doesn't correspond with an emoji, it'll use this text to construct a sentence.

```
@dsl.pipeline(  
    name="hello-world",  
    description="An intro pipeline",  
    pipeline_root=PIPELINE_ROOT,  
)  
# You can change the `text` and `emoji_str` parameters here to update the  
pipeline output  
def intro_pipeline(text: str = "Vertex Pipelines", emoji_str: str =  
"sparkles"):  
    product_task = product_name(text)  
    emoji_task = emoji(emoji_str)  
    consumer_task = build_sentence(  
        product_task.output,  
        emoji_task.outputs["emoji"],  
        emoji_task.outputs["emoji_text"],  
    )
```

## Step 4: Compile and run the pipeline

With your pipeline defined, you're ready to compile it. The following will generate a JSON file that you'll use to run the pipeline:

```
compiler.Compiler().compile(  
    pipeline_func=intro_pipeline, package_path="intro_pipeline_job.json"  
)
```

```
/home/jupyter/.local/lib/python3.7/site-packages/kfp/v2/compiler  
/compiler.py:1266: FutureWarning: APIs imported from the v1 name  
space (e.g. kfp.dsl, kfp.components, etc) will not be supported  
by the v2 compiler since v2.0.0  
    category=FutureWarning,
```

Next, instantiate an API client:

```
api_client = AIPlatformClient(  
    project_id=PROJECT_ID,  
    region=REGION,  
)
```

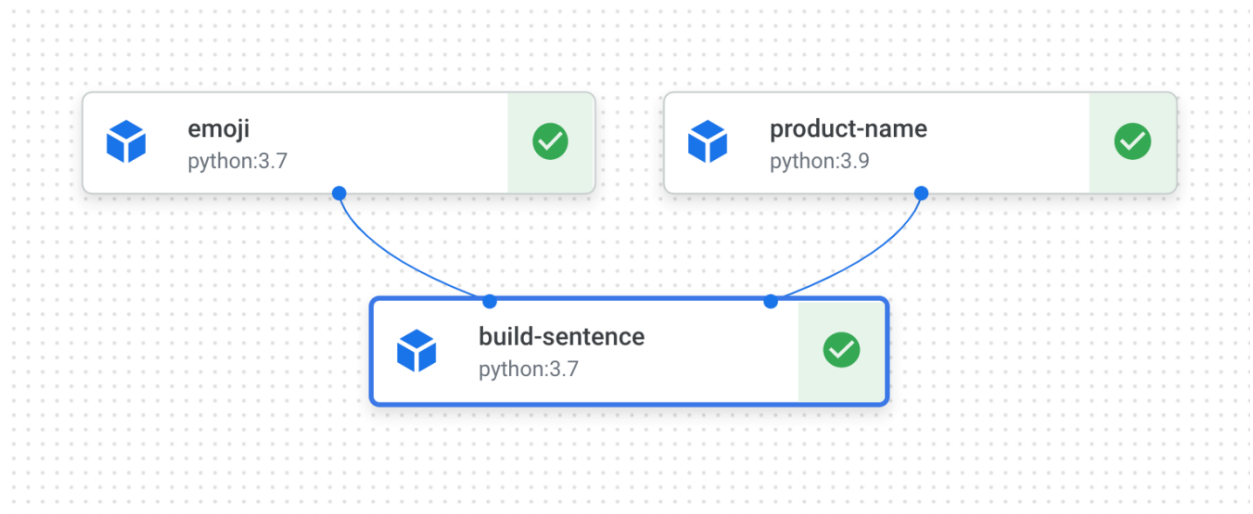
```
/home/jupyter/.local/lib/python3.7/site-packages/kfp/v2/google/c  
lient/client.py:173: FutureWarning: AIPlatformClient will be dep  
recated in v2.0.0. Please use PipelineJob https://googleapis.dev/python/aiplatform/latest/modules/google/cloud/aiplatform/pipeline\_jobs.html in Vertex SDK. Install the SDK using "pip install google-cloud-aiplatform"  
    category=FutureWarning,
```

Finally, run the pipeline:

```
response = api_client.create_run_from_job_spec(  
    job_spec_path="intro_pipeline_job.json",  
    # pipeline_root=PIPELINE_ROOT # this argument is necessary if you did  
    not specify PIPELINE_ROOT as part of the pipeline definition.  
)
```

Running the pipeline should generate a link to view the pipeline run in your console. It should look like this when complete:







This pipeline will take **5-6 minutes** to run. When complete, you can click on the `build-sentence` component to see the final output:

<b>Name</b>	build-sentence
<b>Type</b>	system.ContainerExecution
<b>Duration</b>	0 sec
<b>Started</b>	15 Jun 2021, 14:53:14
<b>Completed</b>	15 Jun 2021, 14:53:14

### Input parameters

Parameter	Type	Value
emoji	string	
emojitext	string	partying_face
product	string	Vertex Pipelines

### Output parameters

Parameter	Type	Value
Output	string	Vertex Pipelines is 

Check if your emoji pipeline has completed

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

## Congratulations!

In this experiment, you created and ran an emoji pipeline.