Chapter 18. Orchestrating Machine Learning Pipelines

In Chapter 1, we introduced ML pipelines and why we need them to produce reproducible and repeatable ML models. In the chapters that followed, we took a deep dive into the individual aspects of ML pipelines, ranging from data ingestion, data validation, model training, and model evaluation, all the way to model deployments. Now it's time to close the loop and focus on how to assemble the individual components into production pipelines.

All the components of an ML pipeline described in the previous chapters need to be executed in a coordinated way or, as we say, *orchestrated*. Inputs to a component must be computed before a given component is executed. The orchestration of these steps is performed by orchestration tools such as Apache Beam or Kubeflow Pipelines, or on Google Cloud's Vertex Pipelines.

In this chapter, we focus on orchestration of the ML components, introducing different orchestration tools and how to pick the best tool for your project.

An Introduction to Pipeline Orchestration

Pipeline orchestration is the “glue” between your pipeline components, such as data ingestion, preprocessing, model training, and model evaluation. Before diving into the details on the different orchestration options, let’s review why we need pipeline orchestration in the first place and introduce the concept of directed acyclic graphs.

Why Pipeline Orchestration?

Pipeline orchestration connects the pipeline components and ensures that they are executed in a specific order. For example, the orchestration tool guarantees that data preprocessing runs before the model training step. At the same time, it is tracking the state of the component execution and caches the state if needed. That ensures that long-running components (e.g., the data preprocessing) don’t need to be rerun in case of a pipeline failure. The orchestrator makes sure that only the failing component reruns.

Orchestration tools like Apache Beam or Kubeflow Pipelines manage ML pipelines in conjunction with a metadata store to track all pipeline artifacts. The pipeline orchestrator executes the components we mentioned in previous chapters. Without one of these orchestration tools, we would need to write code that checks when one component has finished, starts the next

component, schedules runs of the pipeline, and so on. Fortunately, the orchestrator tool takes care of it.

Directed Acyclic Graphs

Pipeline tools like Apache Beam, Kubeflow Pipelines, or Google Cloud Vertex (which is using Kubeflow Pipelines behind the scenes) manage the flow of tasks through a graph representation of the task dependencies.

As Figure 18-1 shows, the pipeline steps are directed. This means the pipeline starts with Task A and ends with Task E, which guarantees that the path of execution is clearly defined by the tasks' dependencies. Directed graphs avoid situations where some tasks start without all dependencies fully computed. Since we know we must preprocess our training data before training a model, the representation as a directed graph prevents the training task from being executed before the preprocessing step is completed.

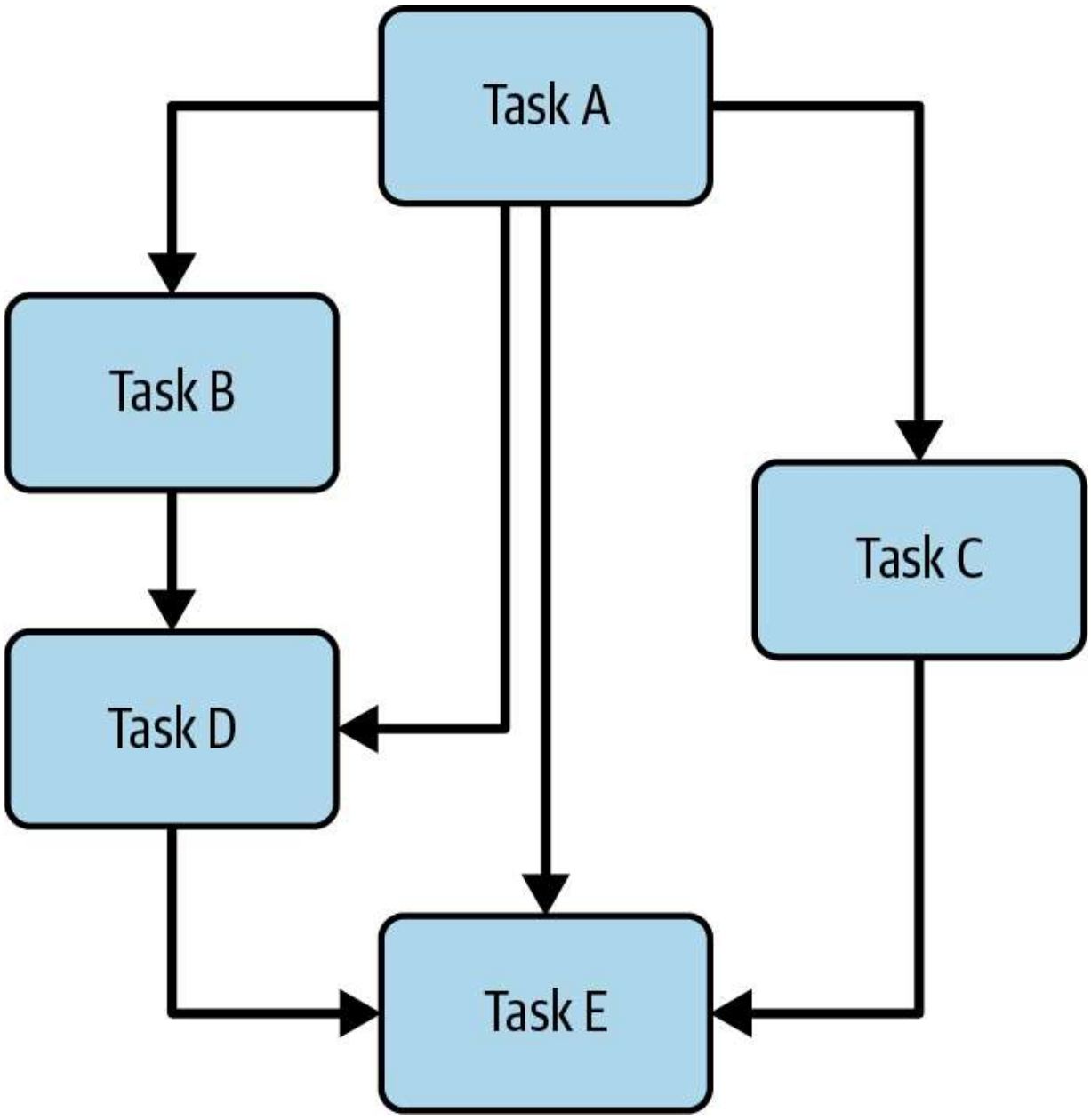

Figure 18-1. Example of a directed acyclic graph

Pipeline graphs must also be *acyclic*, meaning that a graph isn't linking to a previously completed task. This would mean the pipeline could run endlessly and therefore wouldn't finish the workflow.

Because of the two conditions (being directed and acyclic), pipeline graphs are an example of the mathematical concept of directed acyclic graphs (DAGs), and DAGs are a central concept behind orchestration tools.

In the next sections, we'll dive into how to orchestrate your ML pipelines. We'll review the following orchestrator options:

- Interactive pipelines using TFX in conjunction with Jupyter Notebooks
- TFX with Apache Beam as the orchestrator
- TFX with Kubeflow Pipelines as the orchestrator
- TFX with Google Cloud Vertex Pipelines as the orchestrator

Pipeline Orchestration with TFX

TFX supports a number of orchestration tools, and it even supports custom orchestrators. In this section, we are introducing the three most popular options for orchestrating your ML pipelines. But before we dive into the orchestration tools, we'll show you how to test your ML pipelines in Jupyter Notebooks.

Interactive TFX Pipelines

Interactive TFX pipelines are a great way of developing and testing your TFX pipelines. TFX lets you execute your pipeline components in a Jupyter

Notebook, and you can inspect the component output right in your notebook.

In the case of interactive pipelines, you are the orchestrator. TFX won't define a graph for you, so you have to make sure the components are executed in order. The setup is fairly easy; first, you need a simple import:

```
from tfx.orchestration.experimental.interactive.i
import InteractiveContext
```

Then, after instantiating a *context* object with:

```
context = InteractiveContext()
```

you can use the context to execute pipeline components as follows:

```
example_gen = tfx.components.CsvExampleGen(input_
context.run(example_gen)
```

NOTE

One note about `InteractiveContext`: it can only be executed in the context of a Jupyter Notebook. There aren't a lot of configuration options, but you can turn caching on or off for individual components, and you can set your `pipeline_root` (a place where the pipeline artifacts are stored) and your `metadata_connection_config` configuration. The latter lets you connect to a database for your metadata tracking. If it isn't configured, it will default to an SQLite database.

Executing the data ingestion component, called ExampleGen, the notebook will render the output of the component, as shown in Figure 18-2.

Figure 18-2. Example of a TFX pipeline with an interactive context

You'll then have to execute every component in your notebook, and you can consume the output of the previous components as follows:

```
statistics_gen = \
tfx.components.StatisticsGen(examples=example_ge
```

```
context.run(statistics_gen)
```

The context object also allows you to render any output from the component in your notebook directly, as shown in Figure 18-3. This is very handy for components like TFX's StatisticsGen and Evaluator:

```
context.show(statistics_gen.outputs['statistics'])
```

WARNING

You can execute all TFX components in your notebook; however, this limits you to the resources of your notebook hosting server. We don't recommend deploying production ML models through interactive pipelines, since hidden states of Jupyter Notebooks can interfere with the reproducibility of your ML pipeline. We highly recommend you convert the pipeline to use a pipeline orchestrator.

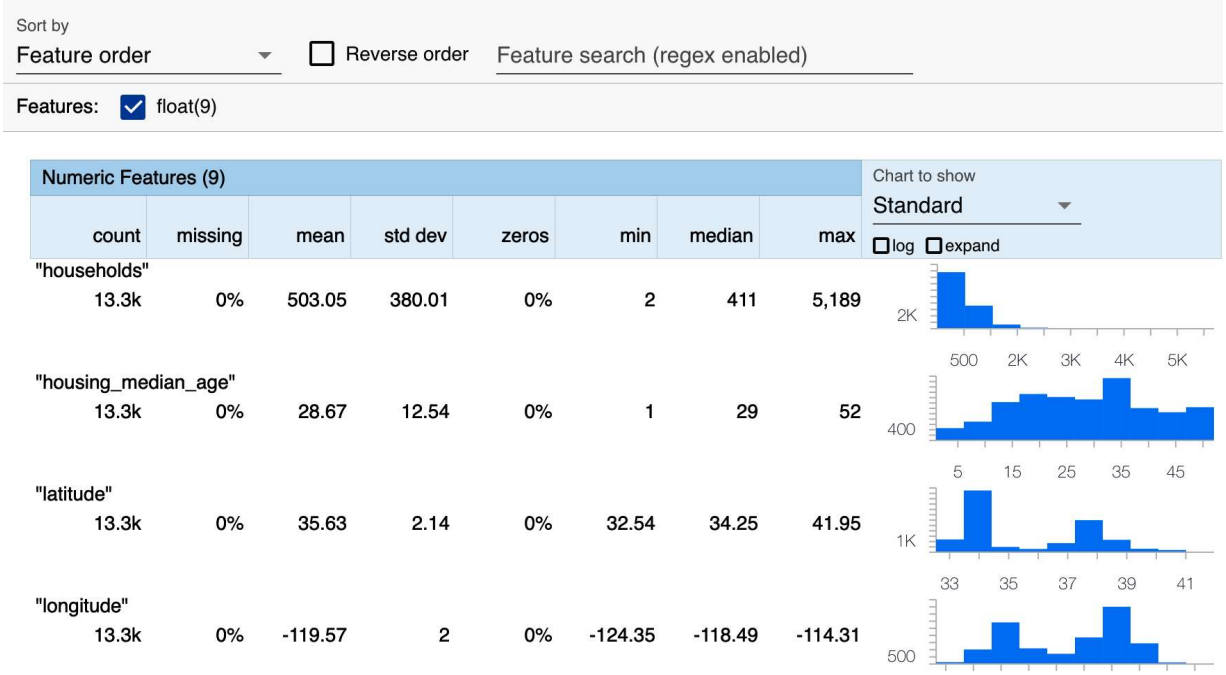

Figure 18-3. Example statistics of a dataset produced from a TFX pipeline in the interactive context

Converting Your Interactive Pipeline for Production

While interactive pipelines are great for experimentation and debugging of TFX pipelines, they aren't recommended for your final deployment of production pipelines. However, TFX provides you an easy way of converting your interactive pipelines to Kubeflow or Google Cloud Vertex Pipelines.

You can convert the interactive pipeline by calling `export_to_pipeline`. When `export_to_pipeline` is executed, it converts the notebook located at `notebook_filepath` to a Python script and saves it to `pipeline_export_filepath`:

```
context.export_to_pipeline(
    notebook_filepath=notebook_filepath,
    export_filepath=pipeline_export_filepath,
    runner_type='kubeflow'
)
```

If your notebook contains cells that you don't want to convert, you can mark them with the magic function `%%skip_for_export`. The conversion process will skip those notebook cells during the conversion.

Orchestrating TFX Pipelines with Apache Beam

If you're using TFX for your pipeline tasks, Apache Beam is already installed since it is one of the core dependencies. Therefore, if you are looking for a minimal installation, reusing Beam to orchestrate is a logical choice. It is straightforward to set up, and it also allows you to use any existing distributed data processing infrastructure you might already be familiar with (e.g., Google Cloud Dataflow). You can also use Apache Beam as an intermediate step to ensure that your pipeline runs correctly and to debug potential pipeline bugs before moving to more complex orchestrators like Kubeflow Pipelines.

However, Apache Beam is missing a variety of tools for scheduling your model updates or monitoring the process of a pipeline job. That's where orchestrators like Kubeflow Pipelines and Google Cloud Vertex Pipelines shine.

Various TFX components (e.g., TensorFlow Data Validation [TFDV] or TensorFlow Transform [TF Transform]) use Apache Beam for the abstraction of distributed data processing. Many of the same Beam functions can also be used to run your pipeline.

In this section, we will run through how to set up and execute our example TFX pipeline with Beam. TFX provides a `Pipeline` object, which lets you set all important configurations. In the following example, we are defining a Beam pipeline that accepts the TFX pipeline components as an argument and also connects to the SQLite database holding the ML Metadata (MLMD) store:

```
from tfx.orchestration import metadata, pipeline
def init_beam_pipeline(components, pipeline_root,
    beam_arg = [
        "--direct_num_workers={}".format(direct_n
        "--requirements_file={}".format(requireme
    ]
    p = pipeline.Pipeline( ❷
        pipeline_name=pipeline_name,
        pipeline_root=pipeline_root,
```

```
            components=components,
            enable_cache=False, ❸
            metadata_connection_config=\
                metadata.sqlite_metadata_connection_c
            beam_pipeline_args=beam_arg)
        return p
```

❶ Beam lets you specify the number of workers. A sensible default is half the number of CPUs (if there is more than one CPU).

❷ You define your pipeline object with a configuration.

❸ We can set the cache to `True` if we would like to avoid rerunning components that have already finished. If we set this flag to `False`, everything gets recomputed every time we run the pipeline.

The Beam pipeline configuration needs to include the name of the pipeline, the path to the root of the pipeline directory, and a list of components to be executed as part of the pipeline.

Next, we will initialize the components and the pipeline, and then run the pipeline using `BeamDagRunner().run(pipeline)`:

```
from tfx.orchestration.beam.beam_dag_runner impor
components = init_components(data_dir, module_fi
                             training_steps=10
```

```
pipeline = init_beam_pipeline(components, pipeli
BeamDagRunner().run(pipeline)
```

This is a very minimal setup. It can be easily integrated into existing workflows, or scheduled using a cron job.

Apache Beam offers a very simple and elegant orchestration setup, but it lacks a number of features. For example, it doesn't visualize the pipeline graph, and it doesn't allow you to schedule a pipeline run.

In the next section, we'll discuss the orchestration of our pipelines with Kubeflow.

Orchestrating TFX Pipelines with Kubeflow Pipelines

Kubeflow Pipelines allows us to run ML tasks within Kubernetes clusters, which provides a highly scalable pipeline solution.

The setup of Kubeflow Pipelines is more complex than the installation of Apache Beam. However, it provides great features, including a pipeline lineage browser, TensorBoard integration, and the ability to view TFDV and TensorFlow Model Analysis (TFMA) visualizations. Furthermore, it

leverages the advantages of Kubernetes, such as autoscaling of computation pods, persistent volume, resource requests, and limits, to name just a few.

Introduction to Kubeflow Pipelines

Kubeflow Pipelines is a Kubernetes-based orchestration tool designed for ML. While Apache Beam was designed for ETL processes, Kubeflow Pipelines has the end-to-end execution of ML pipelines at its heart.

Kubeflow Pipelines provides a consistent UI to track ML pipeline runs, a central place for data scientists to collaborate with one another, and a way to schedule runs for continuous model builds. In addition, Kubeflow Pipelines provides its own SDK to build Docker containers for pipeline runs or to orchestrate containers. The Kubeflow Pipeline domain-specific language (DSL) allows more flexibility in setting up pipeline steps but also requires more coordination between the components.

When we set up Kubeflow Pipelines, it will install a variety of tools, including the UI, the workflow controller, a MySQL database instance, and the MLMD Store.

When we run our TFX pipeline with Kubeflow Pipelines, you will notice that every component is run as its own Kubernetes pod. As shown in Figure 18-4, each component connects with the central metadata store in the cluster and can load artifacts from either a persistent storage volume of a Kubernetes cluster or a cloud storage bucket. All the outputs of the

components (e.g., data statistics from the TFDV execution or the exported models) are registered with the metadata store and stored as artifacts on a persistent volume or a cloud storage bucket.

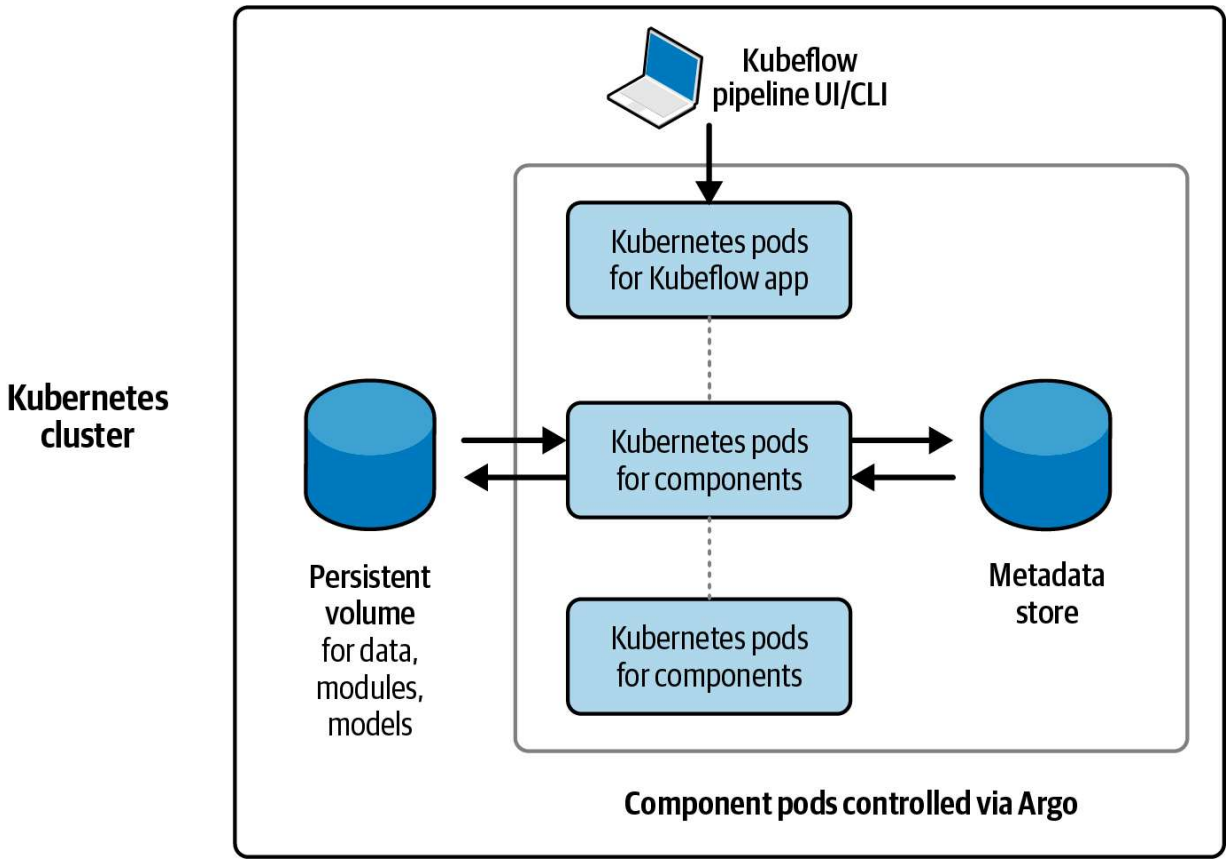

Figure 18-4. An overview of Kubeflow Pipelines

Kubeflow Pipelines relies on another common Kubernetes tool called Argo. Argo allows the scheduling of Kubernetes workflows. It handles the orchestration for your Kubeflow-based ML pipelines.

TIP

This brief section can't serve as a holistic introduction to Kubeflow Pipelines, but the following are two great introductions to Kubeflow and Kubeflow Pipelines:

- *Kubeflow Operations Guide* by Josh Patterson, Michael Katzenellenbogen, and Austin Harris (O'Reilly)
- *Kubeflow for Machine Learning* by Holden Karau, Boris Lublinsky, Richard Liu, and Ilan Filonenko (O'Reilly)

Installation and Initial Setup

Kubeflow Pipelines are executed inside a Kubernetes cluster. For this section, we will assume that you have a Kubernetes cluster created with at least 16 GB and eight CPUs across your node pool and that you have configured `kubectl` to connect with your newly created Kubernetes cluster.

TIP

Due to the resource requirements of Kubeflow Pipelines, using a cloud provider for your Kubernetes setup is preferred. Managed Kubernetes services available from cloud providers include:

- Amazon Elastic Kubernetes Service (Amazon EKS)
- Google Kubernetes Engine (GKE)
- Microsoft Azure Kubernetes Service (AKS)
- IBM's Kubernetes Service

For more details regarding Kubeflow's underlying architecture, Kubernetes, we highly recommend *Kubernetes: Up and Running* by Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson (O'Reilly).

For the orchestration of our pipeline, we are installing Kubeflow Pipelines as a standalone application and without all the other tools that are part of the Kubeflow project. With the following bash commands, we can set up our standalone Kubeflow Pipelines installation. The complete setup might take five minutes to fully spin up correctly:

```
$ export PIPELINE_VERSION=sdk-2.4.0
$ kubectl apply -k \
"github.com/kubeflow/pipelines/manifests/kustomiz
ref=$PIPELINE_VERSION"
$ kubectl wait --for condition=established --time
crd/applications.app.k8s.io
```

```
$ kubectl apply -k \
"github.com/kubeflow/pipelines/manifests/kustomiz
```

You can check the progress of the installation by printing the information about the created pods:

```
$ kubectl -n kubeflow get pods
NAME
cache-deployer-deployment-c6896d66b-62gc5
cache-server-8869f945b-4k7qk
controller-manager-5cbdfbc5bd-bnfxx
...
```

After a few minutes, the status of all the pods should turn to Running. If your pipeline is experiencing any issues (e.g., not enough compute resources), the pods' status would indicate the error:

```
$ kubectl -n kubeflow get pods
NAME
cache-deployer-deployment-c6896d66b-62gc5
cache-server-8869f945b-4k7qk
controller-manager-5cbdfbc5bd-bnfxx
...
```

Individual pods can be investigated with:

```
kubectl -n kubeflow describe pod <pod name>
```

Accessing Kubeflow Pipelines

If the installation completed successfully, regardless of your cloud provider or Kubernetes service, you can access the installed Kubeflow Pipelines UI by creating a port forward with Kubernetes:

```
$ kubectl port-forward -n kubeflow svc/ml-pipeli
```

With the port forward running, you can access Kubeflow Pipelines in your browser by accessing *http://localhost:8080*. For production use cases, a load balancer should be created for the Kubernetes service.

If everything works out, you will see the Kubeflow Pipelines dashboard or the landing page, as shown in Figure 18-5.

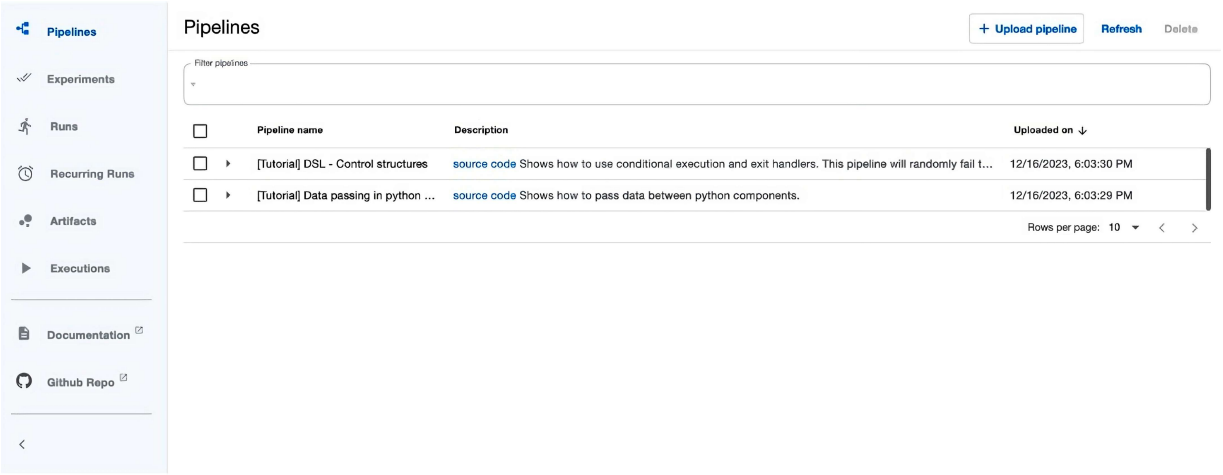

Figure 18-5. The initial screen when you access Kubeflow Pipelines

With the Kubeflow Pipelines setup up and running, we can focus on how to run pipelines. In the next section, we will discuss pipeline orchestration and the workflow from TFX to Kubeflow Pipelines.

The Workflow from TFX to Kubeflow

In earlier sections, we discussed how to set up the Kubeflow Pipelines application on Kubernetes. In this section, we will describe how to run your pipelines on the Kubeflow Pipelines setup, and we'll focus on execution only within your Kubernetes clusters. This guarantees that the pipeline execution can be performed on clusters independent from the cloud service provider.

Before we get into the details of how to orchestrate ML pipelines with Kubeflow Pipelines, we want to step back for a moment. The workflow from TFX code to your pipeline execution is a little more complex than

previously shown in the Apache Beam example, so we will begin with an overview of the full picture. Figure 18-6 shows the overall architecture.

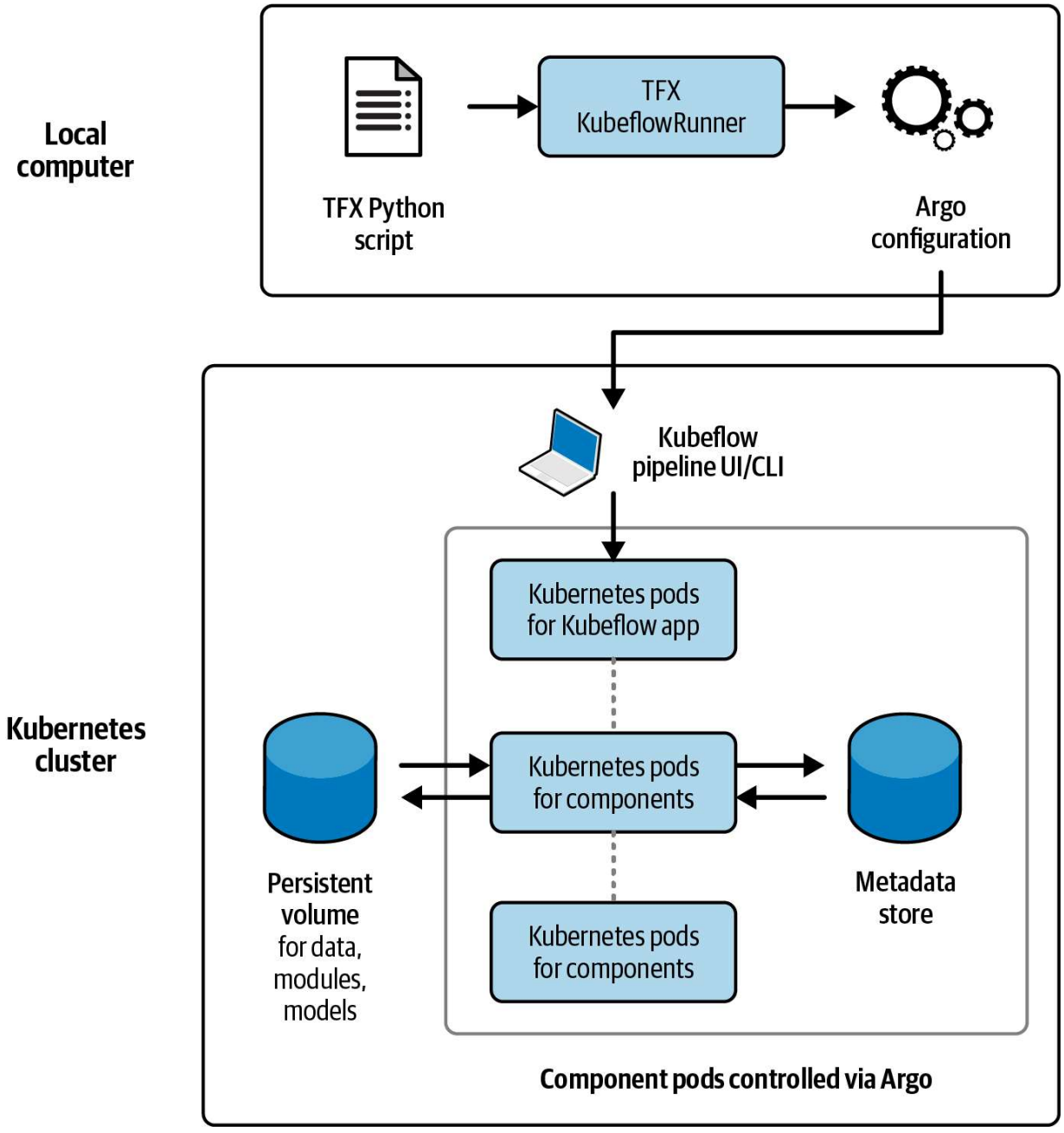

Figure 18-6. Workflow from a TFX script to Kubeflow Pipelines

As shown in Figure 18-6, the TFX KubeflowRunner will convert our Python TFX scripts with all the component specifications to Argo instructions, which can then be executed with Kubeflow Pipelines. Argo will spin up each TFX component as its own Kubernetes pod and run the TFX Executor for the specific component in the container.

NOTE

The TFX image used for all component containers needs to include all required Python packages. The default TFX image provides a recent TensorFlow version and basic packages. If your pipeline requires additional packages, you will need to build a custom TFX container image and specify it in the `KubeflowDagRunnerConfig`.

All components need to read or write to a filesystem outside of the Executor container itself. For example, the data ingestion component needs to read the data from a filesystem, or the final model needs to be pushed by the Pusher to a particular location. It would be impractical to read and write only within the component container; therefore, we recommend storing artifacts in file stores that can be accessed by all components (e.g., in cloud storage buckets or persistent volumes in a Kubernetes cluster).

You can store your training data, Python module, and pipeline artifacts in a cloud storage bucket or in a persistent volume; that is up to you. Your pipeline just needs access to the files. If you choose to read or write data to

and from cloud storage buckets, make sure your TFX components have the necessary cloud credentials when running in your Kubernetes cluster.

With all files in place, and a custom TFX image for your pipeline containers uploaded to the container registry of your choice (if required), we can now "assemble" the TFX Runner script to generate the Argo YAML instructions for our Kubeflow Pipelines execution.

First, let's configure the filepath for our Python module code required to run the Transform and Trainer components. In addition, we will set the folder locations for our raw training data, the pipeline artifacts, and the location where our trained model should be stored. In the following example, we show you how to mount a persistent volume with TFX:

```
import os

pipeline_name = 'cats-and-dogs-classification'
persistent_volume_claim = 'tfx-pvc'
persistent_volume = 'tfx-pv'
persistent_volume_mount = '/tfx-data'
# Pipeline inputs
data_dir = os.path.join(persistent_volume_mount,
# Pipeline outputs
serving_model_dir = os.path.join(
    persistent_volume_mount, 'output', pipeline_r
```

If you decide to use a cloud storage provider, the root of the folder structure can be a bucket, as shown in the following example:

```
import os
...
bucket = 'gs://tfx-demo-pipeline'
# Pipeline inputs
data_dir = os.path.join(bucket, 'PetImages')
…
```

With the filepaths defined, we can now configure our `KubeflowDagRunnerConfig`. Three arguments are important to configure the TFX setup in our Kubeflow Pipelines setup:

`kubeflow_metadata_config`

Kubeflow runs a MySQL database inside the Kubernetes cluster. You can return the database information provided by the Kubernetes cluster by calling `get_default_kubeflow_metadata_config()`. If you want to use a managed database (e.g., AWS RDS or Google Cloud Databases), you can overwrite the connection details through the argument.

`tfx_image`

The image URI is optional. If no URI is defined, TFX will set the image corresponding to the TFX version executing the runner. In our example demonstration, we set the URI to the path of the image in the container registry (e.g., `<region>-docker.pkg.dev/<project_id>/<repo_name>/<image_name>:<image_tag>`).

`pipeline_operator_funcs`

This argument accesses a list of configuration information that is needed to run TFX inside Kubeflow Pipelines (e.g., the service name and port of the gRPC server). Since this information can be provided through the Kubernetes ConfigMap, the `get_default_pipeline_operator_funcs` function will read the ConfigMap and provide the details to the `pipeline_operator_funcs` argument.

In our example project, we will be manually mounting a persistent volume with our project data; therefore, we need to append the list with this information:

```
from kfp import onprem
from tfx.orchestration.kubeflow import kubeflow_d
...
cpu_container_image_uri = \
"<region>-docker.pkg.dev/<project_id>" + \
```

```
    "/<repo_name>/<image_name>:<image_tag>"
    metadata_config = \
        kubeflow_dag_runner.get_default_kubeflow_meta
    pipeline_operator_funcs = \
        kubeflow_dag_runner.get_default_pipeline_oper
    pipeline_operator_funcs.append( ❸
        onprem.mount_pvc(persistent_volume_claim,
                         persistent_volume,
                         persistent_volume_mount))
    runner_config = kubeflow_dag_runner.KubeflowDagRu
        kubeflow_metadata_config=metadata_config,
        tfx_image=cpu_container_image_uri, ❹
        pipeline_operator_funcs=pipeline_operator_fu
    )
```

❶ Obtain the default metadata configuration.

❷ Obtain the default `OpFunc` functions.

❸ Mount volumes by adding them to the `OpFunc` functions.

❹ Add a custom TFX image if required.

WARNING

When we import `from kfp import onprem`, we rely on the Kubeflow Pipeline SDK 1.x, not 2.x.

OpFunc Functions

OpFunc functions allow us to set cluster-specific details, which are important for the execution of our pipeline. These functions allow us to interact with the underlying DSL objects in Kubeflow Pipelines. The OpFunc functions take the Kubeflow Pipelines DSL object `dsl.ContainerOp` as an input, apply the additional functionality, and return the same object.

Two common use cases for adding OpFunc functions to your `pipeline_operator_funcs` are requesting a memory minimum or specifying GPUs for the container execution. But OpFunc functions also allow setting cloud provider–specific credentials or requesting TPUs (in the case of Google Cloud).

Let's look at the two most common use cases of OpFunc functions: setting the minimum memory limit to run your TFX component containers and requesting GPUs for executing all the TFX components. The following example sets the minimum memory resources required to run each component container to 4 GB:

```
def request_min_4G_memory():
    def _set_memory_spec(container_op):
        container_op.set_memory_request('4G')
    return _set_memory_spec
```

```
...
pipeline_operator_funcs.append(request_min_4G_mer
```

The function receives the `container_op` object, sets the limit, and returns the function itself.

We can request a GPU for the execution of our TFX component containers in the same way, as shown in the following example. If you require GPUs for your container execution, your pipeline will only run if GPUs are available and fully configured in your Kubernetes cluster:

```
def request_gpu():
    def _set_gpu_limit(container_op):
        container_op.set_gpu_limit('1')
    return _set_gpu_limit
...
pipeline_op_funcs.append(request_gpu())
```

The Kubeflow Pipelines SDK provides common OpFunc functions for each major cloud provider. The following example shows how to add AWS credentials to TFX component containers:

```
from kfp import aws
...
pipeline_op_funcs.append(
```

```
        aws.use_aws_secret()
    )
```

NOTE

The function `use_aws_secret()` assumes that the credentials `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` are registered as base64-encoded Kubernetes secrets. The equivalent function for Google Cloud credentials is called `use_gcp_secret()`.

With the `runner_config` in place, we can now initialize the components and execute the `KubeflowDagRunner`. But instead of kicking off a pipeline run, the runner will output the Argo configuration, which we will upload in Kubeflow Pipelines in the next section:

```
from tfx.orchestration.kubeflow import kubeflow_d
local_output_dir = "/tmp"
pipeline_definition_file = constants.PIPELINE_NAM
p = create_pipeline()
runner = kubeflow_dag_runner.KubeflowDagRunner(
    config=runner_config, ❶
    output_dir=local_output_dir,
    output_filename=pipeline_definition_file)
runner.run(p)
```

❶ Earlier generated pipeline config

The arguments `output_dir` and `output_filename` are optional. If they are not provided, the Argo configuration will be provided as a compressed *tar.gz* file in the same directory from which we executed the following Python script. For better visibility, we configured the output format to be YAML, and we set a specific output path.

Orchestrating Kubeflow Pipelines

Now it is time to access your Kubeflow Pipelines dashboard. If you want to create a new pipeline, click "Upload pipeline" for uploading, as shown in Figure 18-7. Alternatively, you can select an existing pipeline and upload a new version.

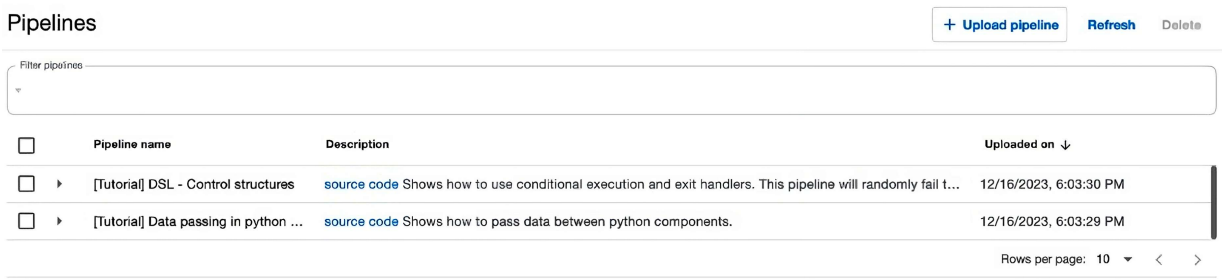

Figure 18-7. An overview of loaded pipelines

Kubeflow Pipelines will now visualize your component dependencies. If you want to kick off a new run of your pipeline, select "Create run," as shown in Figure 18-8.

Once you hit Start, as shown in Figure 18-9, Kubeflow Pipelines, with the help of Argo, will kick into action and spin up a pod for each container,

depending on your direct component graph. When all conditions for a component are met, a pod for a component will be spun up and run the component's executor.

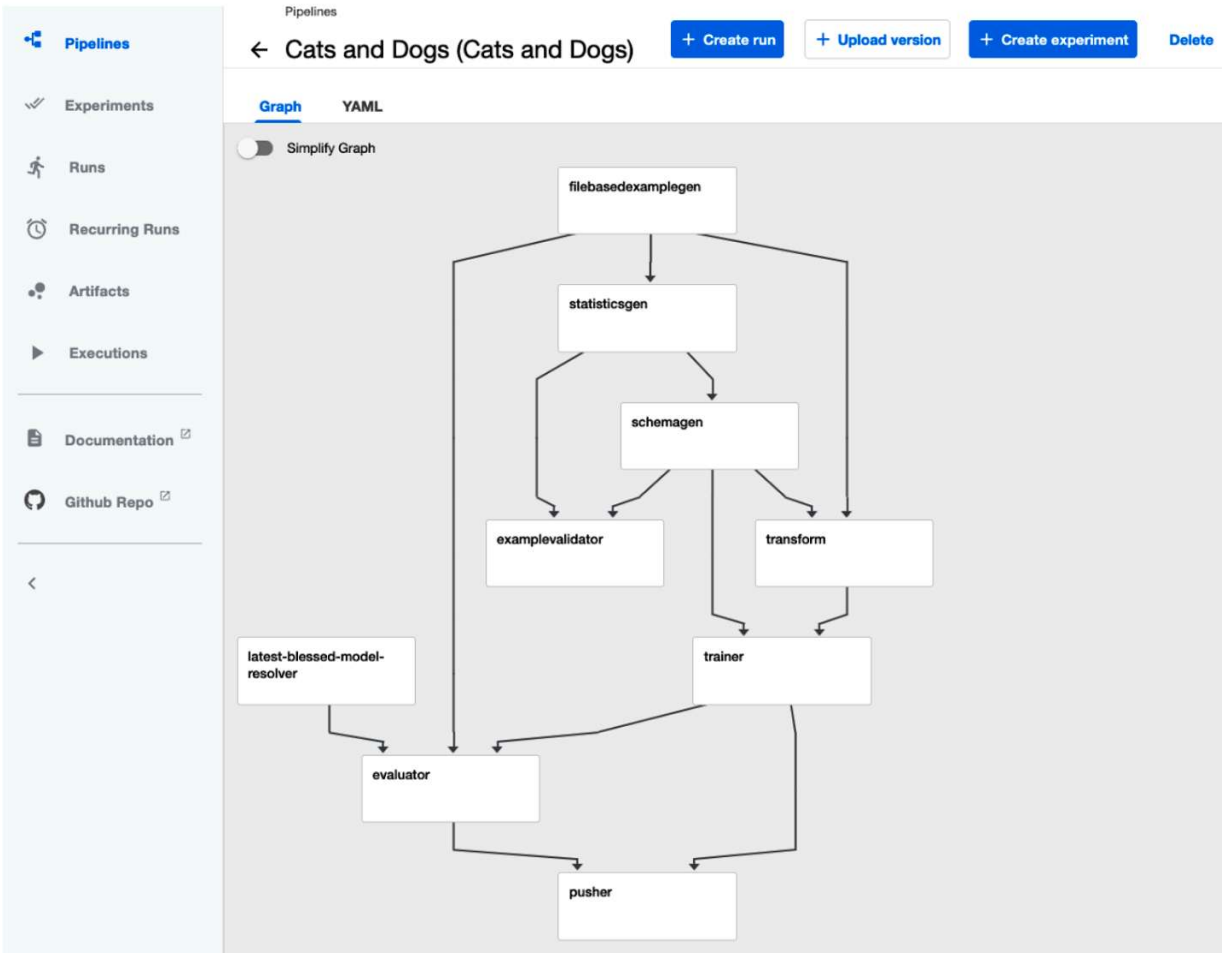

Figure 18-8. Creating a Kubeflow Pipeline run

If you want to see the execution details of a run in progress, you can click "Run name." After a run completes, you can find the validated and exported ML model in the filesystem location set in the Pusher component. In our example case, we pushed the model to the path */tfx-data/output/<pipeline_name>/* on the persistent volume.

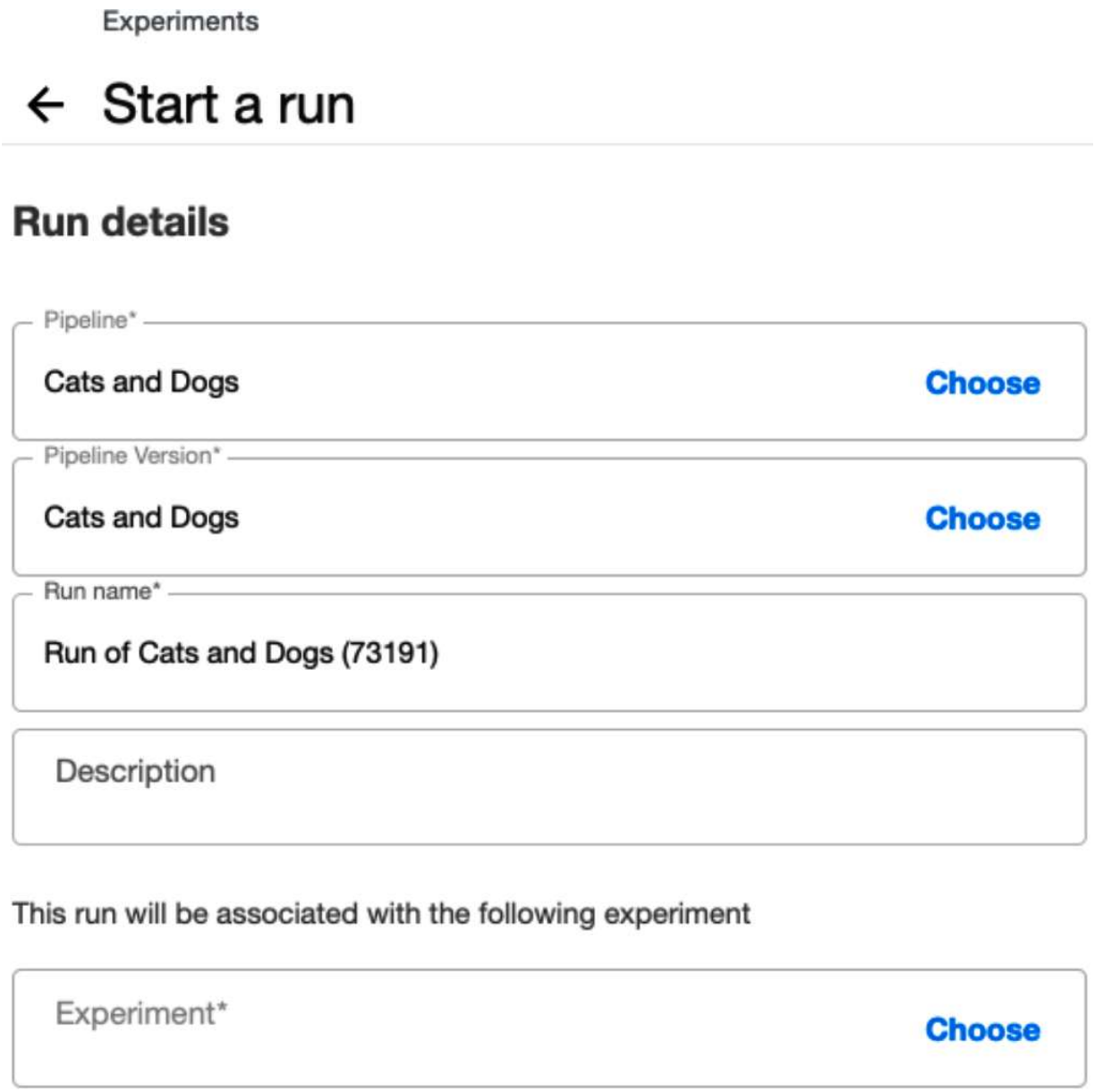

Figure 18-9. Defined Kubeflow Pipeline run details

Kubeflow Pipelines is a great option if you want to orchestrate ML pipelines independent from your infrastructure. You can host Kubeflow anywhere where you can host Kubernetes. This can be on premises or via most cloud providers like Google Cloud, AWS, or Microsoft Azure. If

you’re looking for a fully managed solution to avoid the Kubernetes overhead, Google Cloud Vertex Pipelines is a wonderful option.

Google Cloud Vertex Pipelines

Google Cloud offers a managed service to run your ML pipelines for you, called Vertex Pipelines. Since it is a managed service, you don’t need to set up the infrastructure, and you can parallelize your runs. In addition, you aren’t limited to the available node infrastructure in your cluster. Therefore, Vertex Pipelines is a good alternative if you don’t want to bother with setting up the pipeline infrastructure and want to pay for your pipelines only when you use them. The service is well integrated with other Vertex or Google Cloud products. You can take advantage of the ML training products, you have access to state-of-the-art GPUs, and you can run your components on Google Dataflow for maximum scalability.

A great benefit of TFX is that the pipeline definition doesn’t change with the orchestrator. You can decide to run initially on Apache Beam or Kubeflow Pipelines and then scale your pipelines through Vertex Pipelines when your data volume increases.

In this section, we’ll discuss how you can run your ML pipeline on Vertex Pipelines.

Setting Up Google Cloud and Vertex Pipelines

If it is your first time using Vertex Pipelines, you need to sign up for Google Cloud, create a new Google Cloud project (Figure 18-10), and enable the Vertex AI API for your new project, as shown in Figure 18-11.

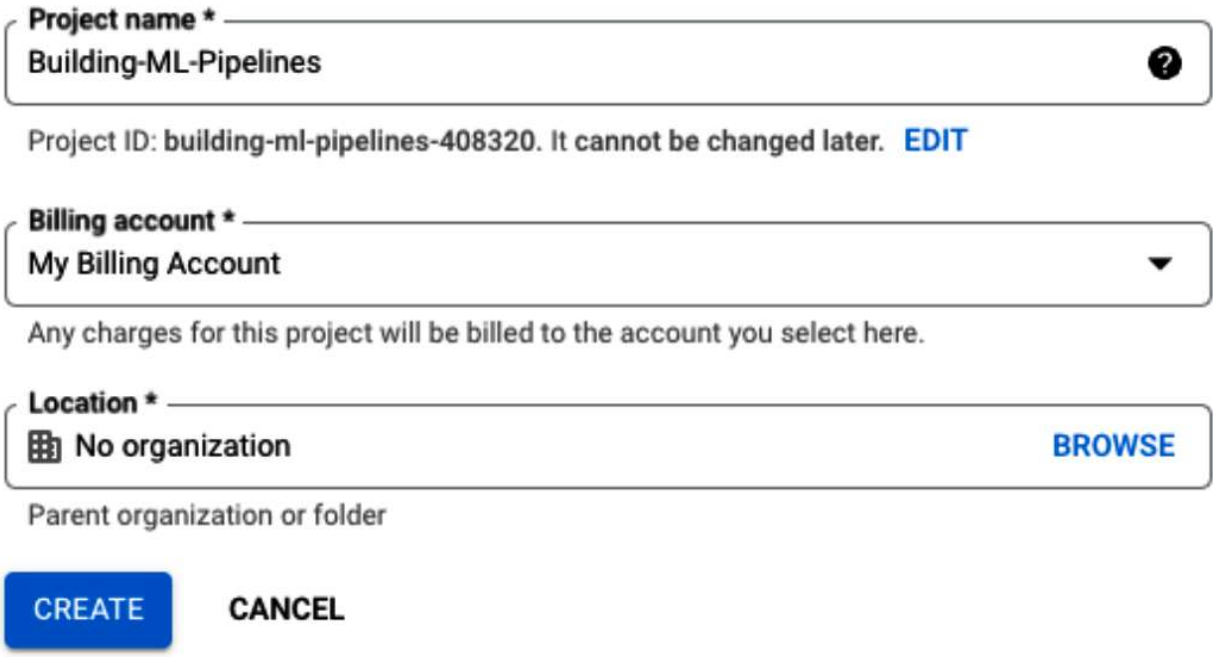

Figure 18-10. Create a new Google Cloud project

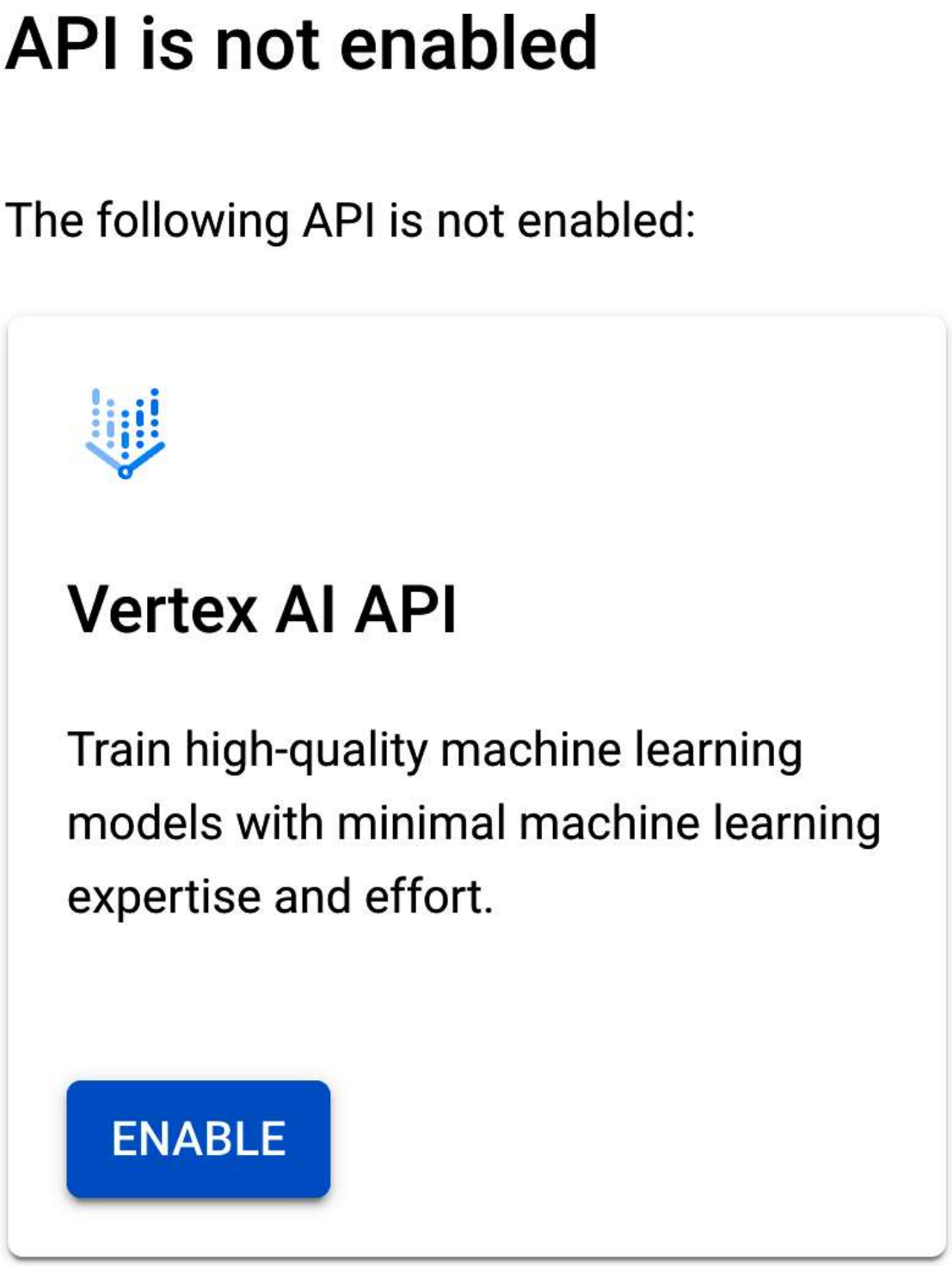

Figure 18-11. Enable the Vertex AI API

Once you're done with the initial setup, make a note of the Google Cloud project ID, as shown in Figure 18-12.

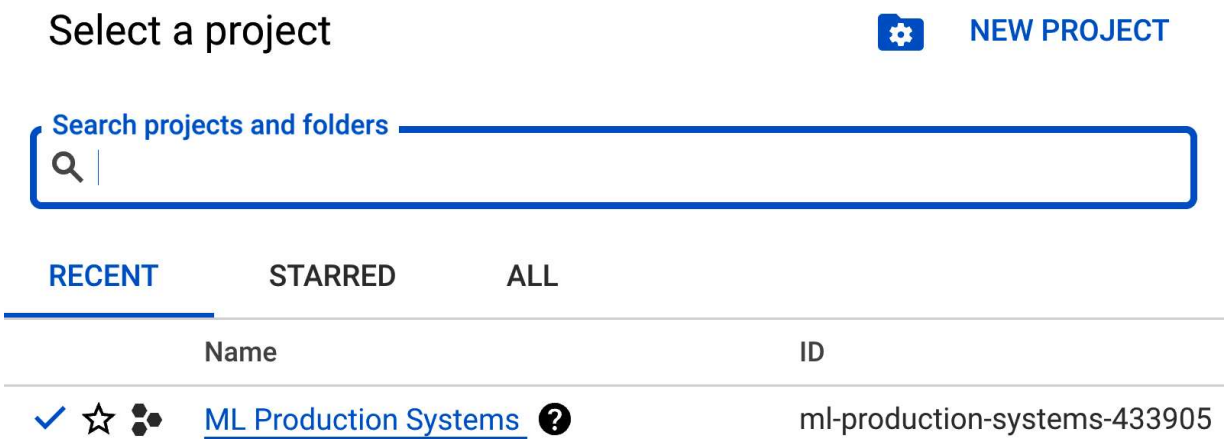

Figure 18-12. An overview of your Google Cloud projects

Furthermore, you need to set up your Google Cloud CLI. Follow the setup steps for your operating system provided by the Google Cloud documentation. Once you have installed the CLI tool, you need to instantiate the tool by running `gcloud init` in the terminal of your operating system. If you are working on a remote machine, you'll need to run the command with the `no-launch-browser` argument: `gcloud init --no-launch-browser`.

More information about the initialization step is available in the Google Cloud documentation.

Lastly, you'll need to create a Google Cloud Storage bucket to use for your pipeline artifacts and pipeline output. To create a new bucket, select Cloud Storage, as shown in Figure 18-13.

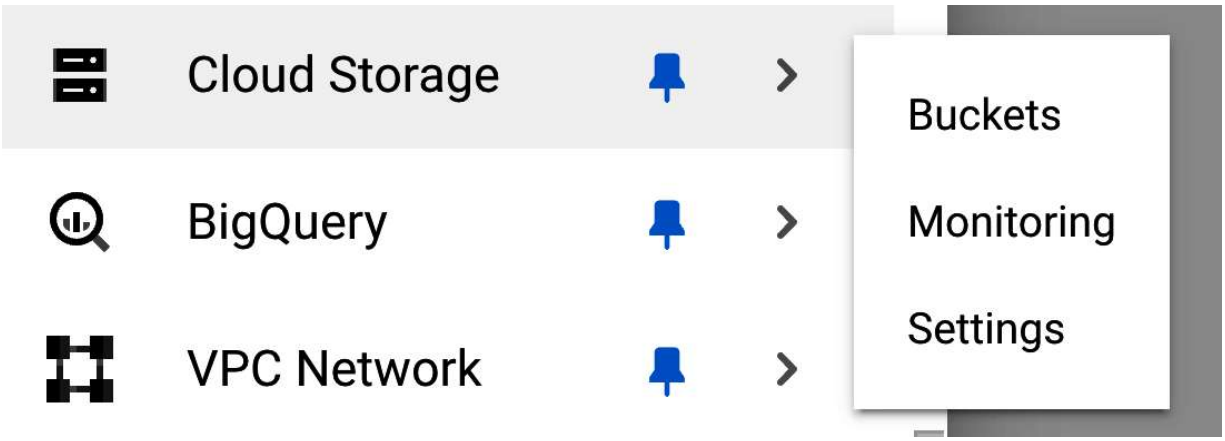

Figure 18-13. Select Buckets to create a new Google Cloud Storage bucket

If you don’t have an existing storage bucket, Google Cloud will ask you to create one, as shown in Figure 18-14.

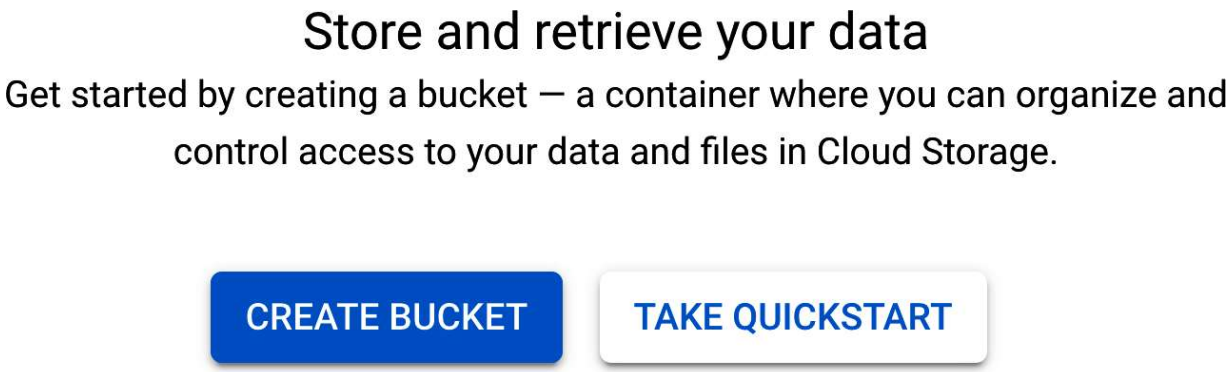

Figure 18-14. Create your first storage bucket

If you already have existing storage buckets in your project, you can create a new bucket by clicking on Create, as shown in Figure 18-15.

Figure 18-15. Create a new bucket

When you create a new bucket, Google Cloud guides you through a number of setup questions, as shown in Figure 18-16.

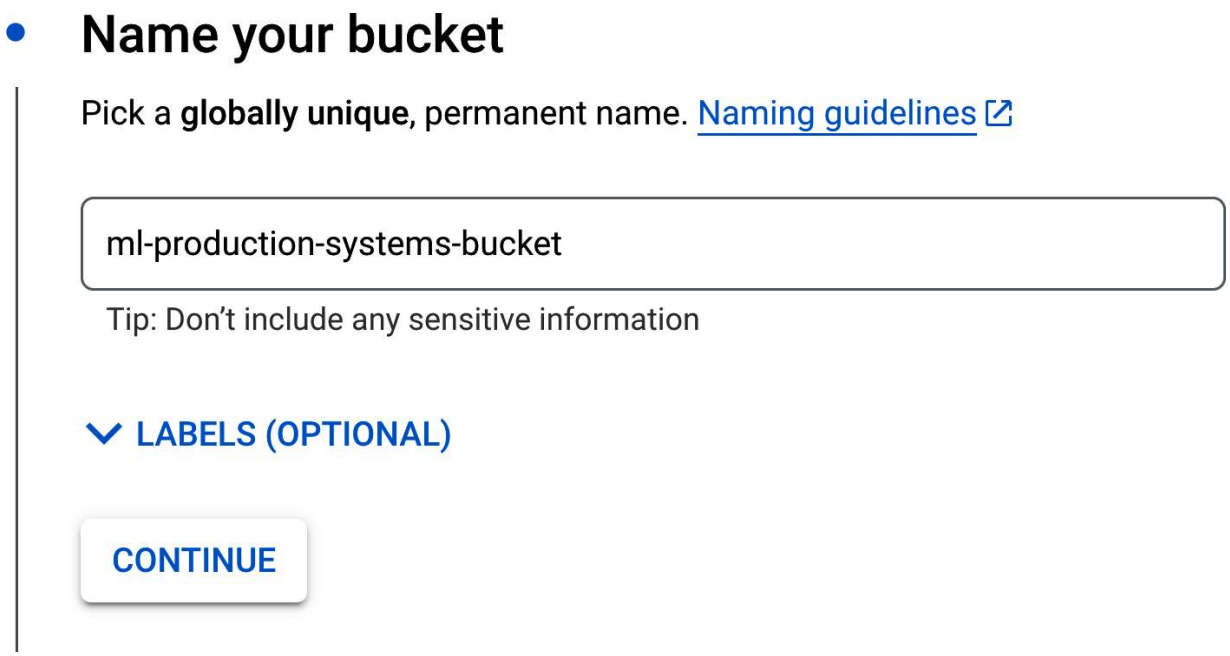

Figure 18-16. Setting up your Google Cloud Storage bucket

There are a multitude of setup options, and we highly recommend consulting the Google Cloud documentation for more details. For the simplest setup, we recommend the following options:

- Pick a region closest to your user location, as shown in Figure 18-17. A Region type is totally sufficient for your initial project.
- Pick Standard as your storage class.
- Check "Enforce public access prevention on this bucket" and use a uniform access control.
- Select None when asked about protection tools.

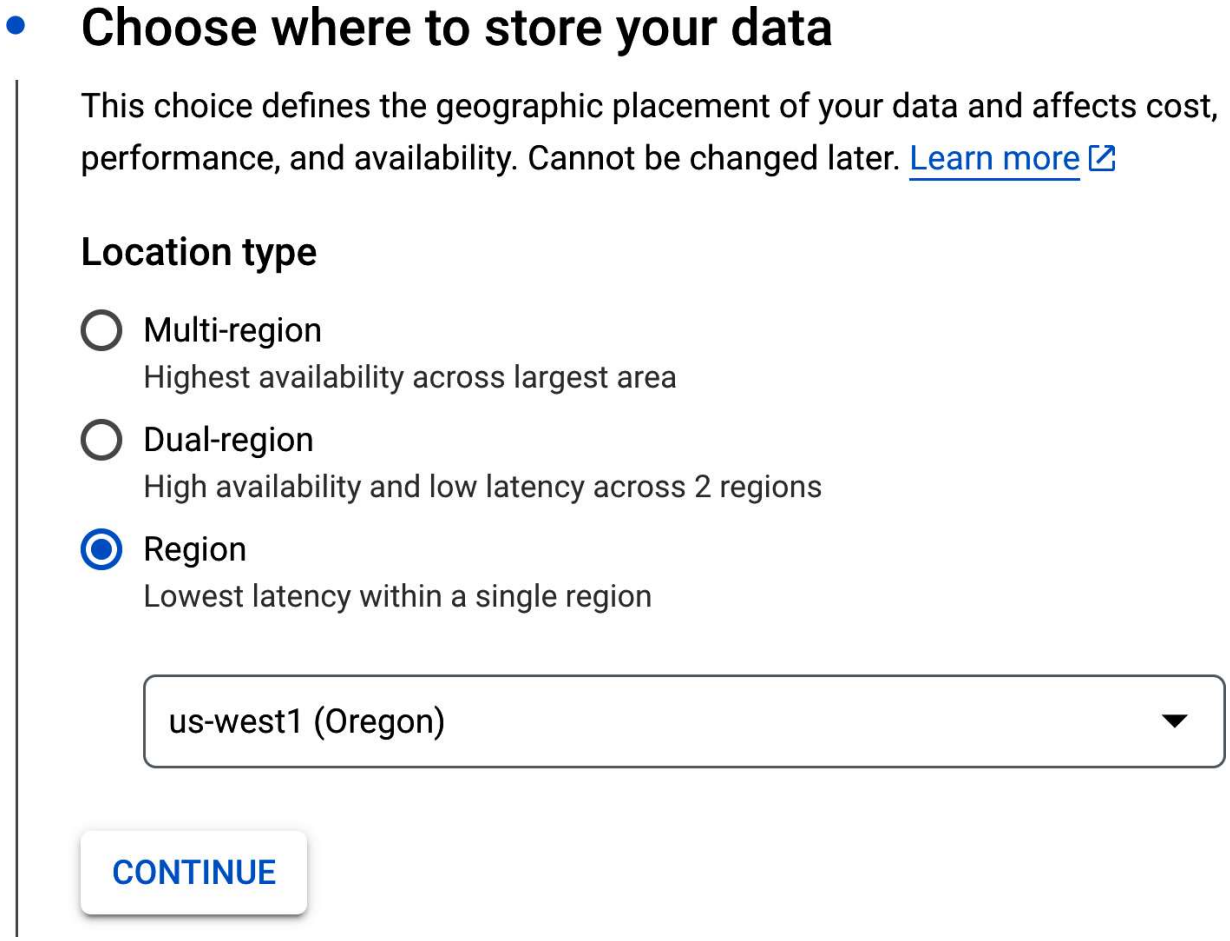

Figure 18-17. Choose your location and type

Before we can focus on the Vertex Pipelines, we need to set up a Google Service Account for our pipeline runs.

Setting Up a Google Cloud Service Account

To run your ML pipeline in Google Cloud Vertex Pipelines, you'll need a service account. This is basically a user account with specific permissions for your pipeline to use during the execution. For production scenarios, it isn't recommended to assign broad permissions for your service account.

Therefore, avoid assigning broad permissions such as Project Owner or Project Editor, since it would allow a number of extra permissions that would lead to security issues in production scenarios.

For the most minimal setup, your service account needs two permission roles:

- Storage Object User
- Vertex AI User

If you want to create a new service account, head over to IAM & Admin > Service Account and then click Create Service Account, as shown in Figure 18-18.

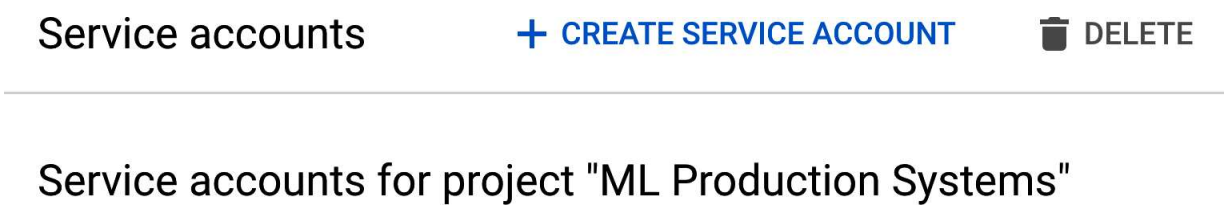

Figure 18-18. Create a new service account

As the first step, you'll be asked for the account name and an account description, as shown in Figure 18-19. Based on the account name, Google creates a service account in the shape of an email address, in our case *machine-learning-production-systems@machine-learning-production-systems-408320.iam.gserviceaccount.com*.

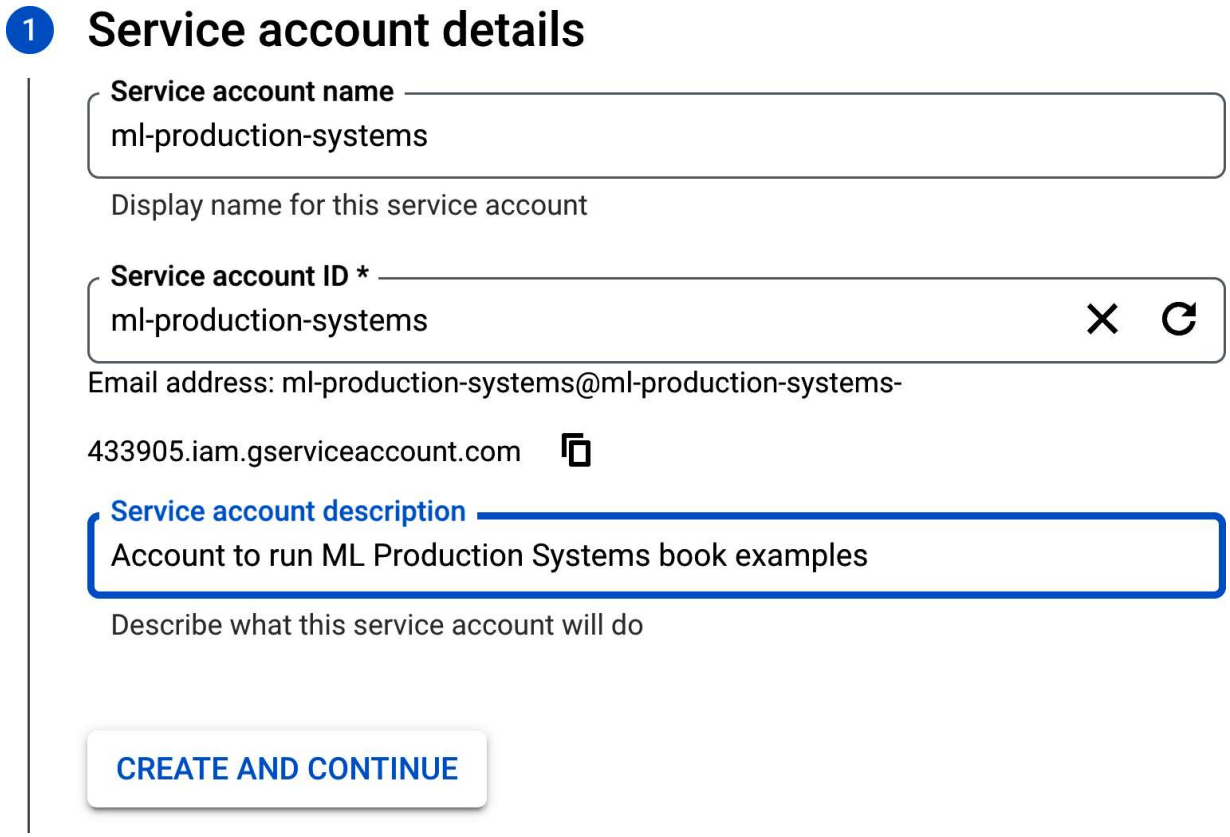

Figure 18-19. Define the account details

With the next step, you can assign the required roles to the service account, as shown in Figure 18-20.

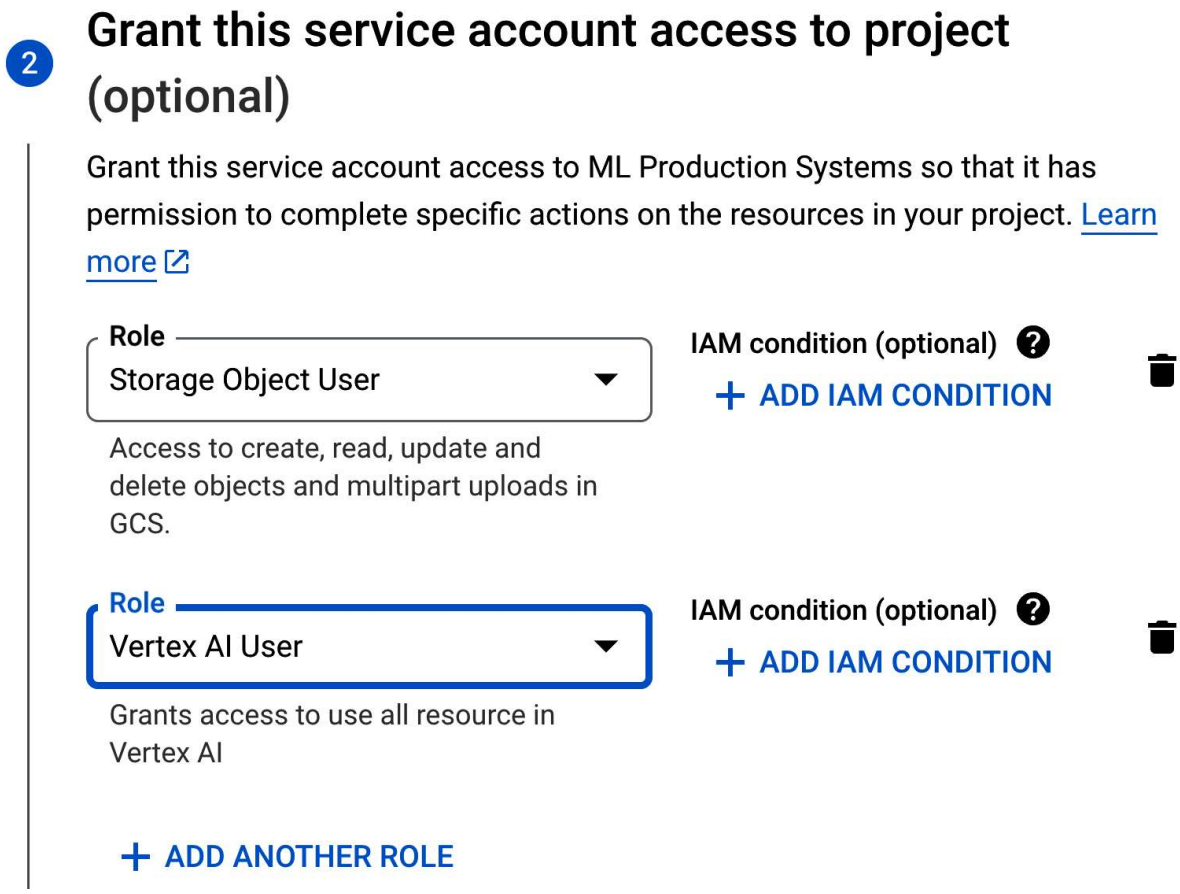

Figure 18-20. Assign required roles for the service account

Once the roles are set, you can complete the step by clicking Done. Note the service account email. You'll need it later, when you kick off your Vertex jobs.

Now with your Google Cloud setup in place, your storage bucket created, and the service account set up, you can use Vertex Pipelines.

ASSIGN THE SERVICE ACCOUNT TO THE BUCKETS

It is considered a best practice to assign a read/write permission to the storage bucket, and not to give service accounts broad read/write permissions. That way, if a service account is compromised, only the buckets with the individual permissions are compromised, not all system buckets.

You can provide bucket read/write access to your newly created service account by heading over to the Google Cloud Storage bucket, choosing Permission, and then granting the role Storage Object User to the service account.

Orchestrating Pipelines with Vertex Pipelines

When you have all your TFX components defined, it is time to create your pipeline definition. First, you need to create a list of your pipeline components:

```
components = [
    example_gen,
    ...
]
```

In our example projects, we created a helper function to assist with the creation of the components.

Next, you'll need to define how Apache Beam should be executed. Since you're running Google Cloud, you could execute any component on Google Dataflow. This is beneficial if you want to distribute large data loads. For simpler workloads (e.g., the entire dataset can fit into your computer memory), we recommend using Apache Beam's `DirectRunner` mode. In that case, every component will run on Vertex Pipelines in its own instance and it is limited by the CPU and memory setup. Check Chapter 19 to learn how to configure component-specific instance configurations.

Here is an example Apache Beam configuration for the `DirectRunner` mode:

```
beam_pipeline_args = [
    "--runner=DirectRunner", ❶
    "--project=" + constants.GCP_PROJECT_ID, ❷
    "--temp_location=" + f"gs://{constants.GCS_BU
    "--direct_running_mode=multi_processing", ❹
    "--direct_num_workers=0", ❺
    "--sdk_container_image=
"<region>-docker.pkg.dev/tfx-oss-public/tfx:{}".f
]
```

❶ Request the `DirectRunner` mode.

❷ Add your Google project ID.

❸ Add your Google Cloud Storage location.

❹ We chose `multi_processing` for true parallelism. Other options are `in_memory` and `multi_threading`.

❺ You can set the maximum number for CPU cores used. Zero will allow all available cores.

❻ Each Apache Beam worker runs its own image. Here we use the base container image from TFX.

If you need to parallelize your ML pipeline, you can switch to executing the pipeline on Google Dataflow. Dataflow is a managed service for running Apache Beam pipelines on Google Cloud. It is highly scalable, and it allows you to process terabytes of data.

A configuration for Dataflow could look like this:

```
beam_pipeline_args = [
    "--runner=DataflowRunner", ❶
    "--region=us-central1", ❷
    "--service_account_email=<your Service accou
    "--machine_type=n1-highmem-4", ❹
```

```
    "--max_num_workers=10",
    "--disk_size_gb=100",
    "--experiments=use_runner_v2",
    "--sdk_container_image=
"<region>-docker.pkg.dev/tfx-oss-public/tfx:{}".
]
```

❶ Request `DataflowRunner`.

❷ Enter your preferred Google Cloud region.

❸ Enter your Google Cloud Service Account email with the permission to run Dataflow.

❹ Set up your preferred Google Cloud Instance type, maximum number of workers, and disk size.

With the Apache Beam configuration set up, we can now create a pipeline object in TFX:

```
from tfx.orchestration import pipeline
my_pipeline = pipeline.Pipeline(
      components=components,
      pipeline_name=constants.PIPELINE_NAME,
      pipeline_root=constants.GCS_PIPELINE_ROOT,
```

```
        beam_pipeline_args=beam_pipeline_args
    )
```

The `pipeline_name` is the name of your pipeline, and the `pipeline_root` refers to the Google Cloud Storage bucket we had created for this pipeline.

Lastly, we need to export the pipeline definition for Vertex Pipelines. Very similar to the previous setups we discussed, we need to define a pipeline runner in TFX. In contrast to the orchestration with Apache Beam, the runner won't start the pipeline, but rather will create a Vertex pipeline definition that we can submit to Vertex Pipelines:

```
from tfx.orchestration.kubeflow.v2 import kubeflo
cpu_container_image_uri = \
"<region>-docker.pkg.dev/tfx-oss-public/tfx:{}".f
    tfx.__version__)
runner_config = kubeflow_v2_dag_runner.KubeflowV2
    default_image=cpu_container_image_uri)
pipeline_definition_file = constants.PIPELINE_NAM
runner = kubeflow_v2_dag_runner.KubeflowV2DagRunn
    config=runner_config,
    output_filename=pipeline_definition_file
)
runner.run(pipeline=create_pipeline(), write_out=
```

Executing the runner will create a JSON definition file named after our pipeline. In the next step, we will demonstrate how to execute the pipeline in Vertex.

Executing Vertex Pipelines

You'll have two options to kick off your Vertex Pipeline runs. You can choose between the user interface or a programmatic way through the Vertex SDK. Here, we will be focusing on the programmatic way.

First, you need to install the Google Cloud AIPlatform SDK. You can do that via `pip`:

```
$ pip install google-cloud-aiplatform
```

Initialize your AIPlatform client as follows:

```
aiplatform.init(
    project=constants.GCP_PROJECT_ID,
    location=constants.VERTEX_REGION,
)
```

We can create a pipeline job object as follows. The job object allows us to control our pipeline runs:

```
job = aiplatform.PipelineJob(
    display_name=constants.PIPELINE_NAME + "-pipe
    template_path=pipeline_definition_file,
    pipeline_root=constants.GCS_PIPELINE_ROOT,
    enable_caching=True,
)
```

If you set `enable_caching` to `True`, Vertex will cache successfully run pipeline steps. If you need to rerun the pipeline—for example, after a pipeline failure—the successfully completed previous steps won't be rerun.

You can now submit the `job` object to Vertex Pipelines with `job.submit`:

```
job.submit(service_account=constants.GCP_SERVICE_
```

Once the job is successfully submitted, the `job` object contains the link to the active pipeline job:

```
Creating PipelineJob
INFO:google.cloud.aiplatform.pipeline_jobs:Creati
PipelineJob created. Resource name:
projects/123/locations/us-central1/pipelineJobs/
cats-and-dog-classification-20231217000838
INFO:google.cloud.aiplatform.pipeline_jobs:
```

```
PipelineJob created. Resource name:
projects/123/locations/us-central1/pipelineJobs/
cats-and-dog-classification-20231217000838
INFO:google.cloud.aiplatform.pipeline_jobs:pipel
aiplatform.PipelineJob.get(
'projects/123/locations/us-central1/
pipelineJobs/cats-and-dog-classification-2023121
View Pipeline Job:
https://console.cloud.google.com/vertex-ai/locat
pipelines/runs/cats-and-dog-classification-20231
?project=123
```

Heading over to Vertex Pipelines, you can now inspect the progress of the pipeline run (shown in Figure 18-21).

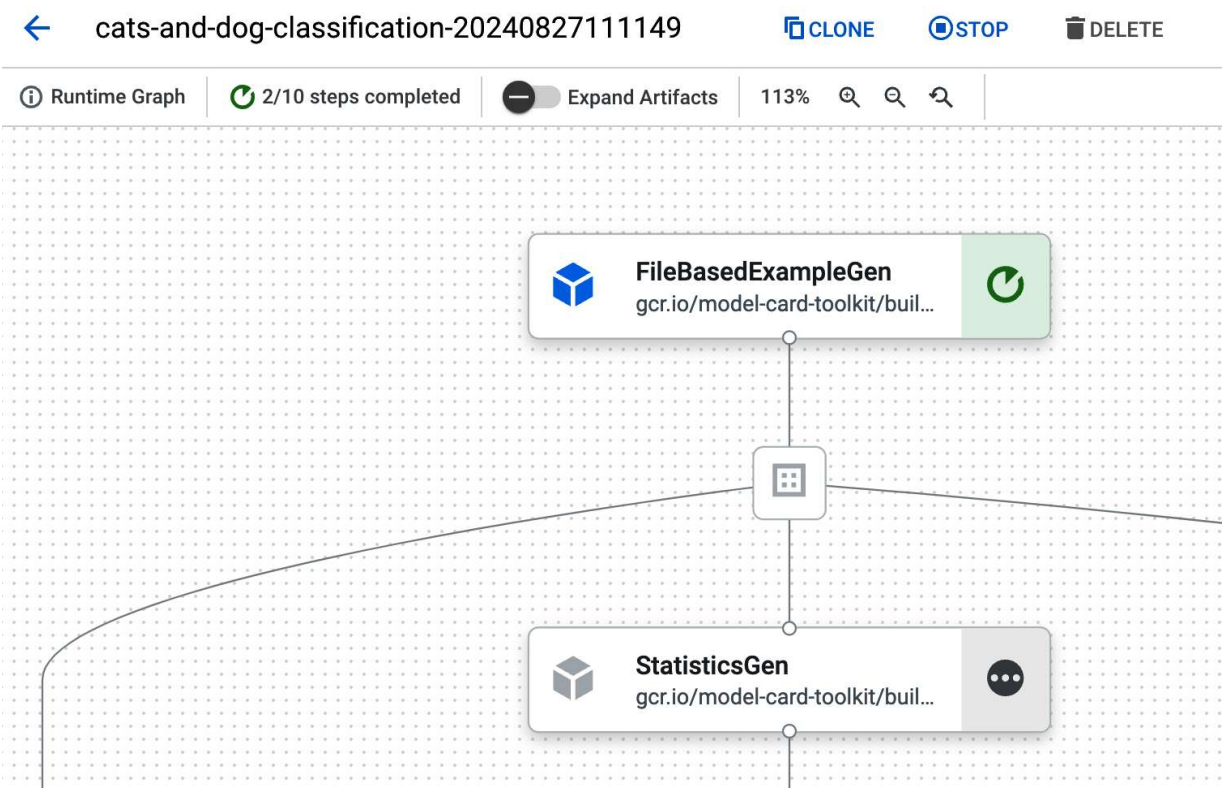

Figure 18-21. Pipeline run in Google Cloud Vertex Pipelines

Choosing Your Orchestrator

In this chapter, we've discussed four orchestration tools that you can use to run your pipelines: Interactive TFX, Apache Beam, Kubeflow Pipelines, and Google Cloud Vertex Pipelines. You need to pick only one of them to run each pipeline, but it is fairly easy to move from one orchestrator to another—usually just a few lines of code. In this section, we will summarize some of the benefits and drawbacks to each of them. It will help you decide what is best for your needs.

Interactive TFX

The Interactive TFX orchestrator (`InteractiveContext`) is only appropriate for use during development of your pipelines or for modification of existing pipelines. It should never be used for production deployment of your pipelines. However, it is usually fairly easy to take a deployed pipeline that is running with a different orchestrator and move it to Interactive TFX for development of modifications or investigation of issues. Since it is an interactive environment, it is usually much easier to work with for development.

Apache Beam

If you're using TFX for your pipeline tasks, you have already installed Apache Beam. Therefore, if you are looking for a minimal installation, reusing Beam to orchestrate is a logical choice. It is straightforward to set up, and it allows you to use any existing distributed data processing infrastructure you might already be familiar with (e.g., Google Cloud Dataflow) either on your own systems or in a managed service.

Kubeflow Pipelines

If you already have experience with Kubernetes and access to a Kubernetes cluster, it makes sense to consider Kubeflow Pipelines. While the setup of Kubeflow isn't as straightforward as the orchestration with Apache Beam, it

opens up a variety of new opportunities, including the ability to view TFDV and TFMA visualizations, track the model lineage, and view the artifact collections.

You can set up a Kubernetes cluster on your own systems, or create a cluster using one of the managed service offerings that are available from a variety of cloud providers, so you aren't limited to a single vendor. Kubeflow Pipelines also lets you take advantage of state-of-the-art training hardware supplied by cloud providers. You can run your pipeline efficiently and scale the nodes of your cluster up and down.

Google Cloud Vertex Pipelines

If you don't want to deal with the setup of Kubernetes clusters, or with Kubeflow, we highly recommend a managed pipeline service like Google Cloud Vertex Pipelines. The service manages the hardware behind the scenes and lets you scale seamlessly. It is a good option if you don't want to be limited by the resource allocation to your Kubernetes cluster or simply don't want to deal with DevOps at all.

Alternatives to TFX

During the past few years, a few alternatives to TFX have been released. Here are five notable alternatives:

MetaFlow

Initially developed by Netflix, this open source project allows bringing data science projects to production. Due to the Netflix origins, the project supports AWS deployments very well.

MLflow

Created by Databricks, this is an open source platform that manages the end-to-end ML lifecycle, including experimentation, reproducibility, and deployment.

ZenML

Originally built on top of TFX, this open source framework supports its own abstraction definitions and orchestration.

Iguazio ML Run

Iguazio's open source project to manage ML pipelines supports all major ML frameworks and provides serverless deployment endpoints.

Ray for ML Infrastructure

Ray's ML platform provides its own infrastructure tooling that integrates Apache Airflow for scheduling, KubeRay, and other libraries.

Conclusion

In this chapter, we discussed how to assemble your ML pipelines and introduced the core principles of pipeline orchestration. To bring your ML into a production setup, we introduced four different options of orchestrating your ML pipelines.

In the next chapter, we will discuss a number of advanced TFX concepts. Furthermore, we'll be introducing four different ways of writing custom TFX components. That way, your production pipeline will be able to handle any use case you might have in mind.