Chapter 12. Pipelines Part 2: Kubeflow Pipelines

In Chapter 11, we discussed the orchestration of our pipelines with Apache Beam and Apache Airflow. These two orchestration tools have some great benefits: Apache Beam is simple to set up, and Apache Airflow is widely adopted for other ETL tasks.

In this chapter, we want to discuss the orchestration of our pipelines with Kubeflow Pipelines. Kubeflow Pipelines allows us to run machine learning tasks within Kubernetes clusters, which provides a highly scalable pipeline solution. As we discussed in Chapter 11 and show in Figure 12-1, our orchestration tool takes care of the coordination between the pipeline components.

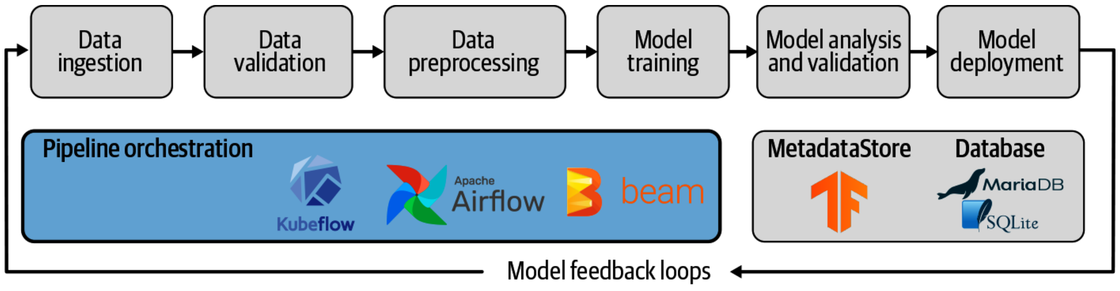


Figure 12-1. Pipeline orchestrators

The setup of Kubeflow Pipelines is more complex than the installation of Apache Airflow or Apache Beam. But, as we will discuss later in this chapter, it provides great features, including *Pipeline Lineage Browser*, *TensorBoard Integration*, and the ability to view TFDV and 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.

This chapter is split into two parts. In the first part, we will discuss how to set up and execute pipelines with Kubeflow Pipelines. The demonstrated setup is independent from the execution environment. It can be a cloud provider offering managed Kubernetes clusters or an on-premise Kubernetes installation.

INTRODUCTION TO KUBERNETES

If Kubernetes concepts and terminology are new to you, check out our appendices. Appendix A provides a brief overview of Kubernetes.

In the second part of the chapter, we will discuss how to run Kubeflow Pipelines with the Google Cloud AI Platform. This is specific to the Google Cloud environment. It takes care of much of the infrastructure and lets you use Dataflow to easily scale data tasks (e.g., the data preprocessing). We recommend this route if you would like to use Kubeflow Pipelines but do not want to spend time managing your Kubernetes infrastructure.

Introduction to Kubeflow Pipelines

Kubeflow Pipelines is a Kubernetes-based orchestration tool with machine learning in mind. While Apache Airflow was designed for ETL processes, Kubeflow Pipelines has the end-to-end execution of machine learning pipelines at its heart.

Kubeflow Pipelines provides a consistent UI to track machine learning pipeline runs, a central place to collaborate between data scientists (as we'll discuss in "Useful Features of Kubeflow Pipelines"), and a way to schedule runs for continuous model builds. In addition, Kubeflow Pipelines provides its own software development kit (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. We think TFX pipelines lead to a higher level of pipeline standardization and, therefore, are less error prone. If you are interested in more details about the Kubeflow Pipelines SDK, we can recommend the suggested reading in "Kubeflow Versus Kubeflow Pipelines".

When we set up Kubeflow Pipelines, as we discuss in "Installation and Initial Setup", Kubeflow Pipelines will install a variety of tools, including the UI, the workflow controller, a MySQL database instance, and the ML MetadataStore we discussed in "What Is ML Metadata?".

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 12-2, 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 from 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.

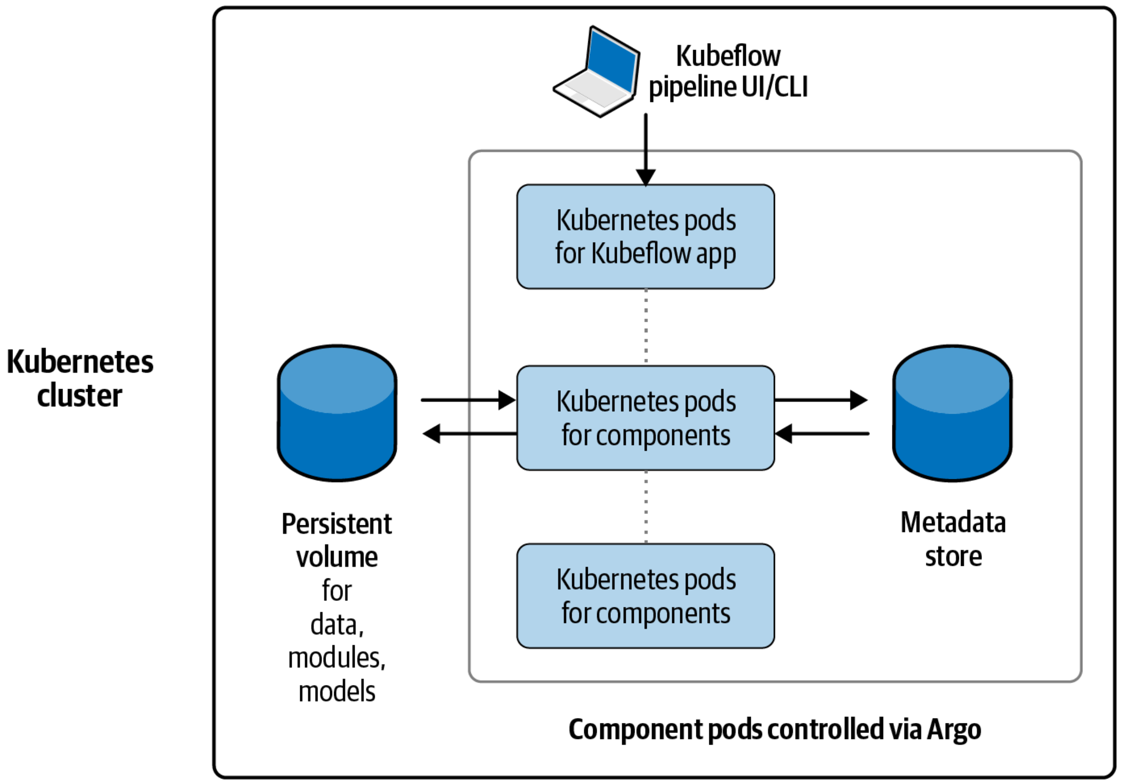

Kubeflow pipeline UI/CLI
Kubernetes pods for Kubeflow app
Kubernetes cluster
Kubernetes pods for components
Persistent volume for data, modules, models
Metadata store
Kubernetes pods for components
Component pods controlled via Argo

Figure 12-2. Overview of Kubeflow Pipelines

KUBEFLOW VERSUS KUBEFLOW PIPELINES

Kubeflow and Kubeflow Pipelines are often mixed up. Kubeflow is a suite of open source projects which encompass a variety of machine learning tools, including TFJob for the training of machine models, Katib for the optimization of model hyperparameters, and KFServing for the deployment of machine learning models. Kubeflow Pipelines is another one of the projects of the Kubeflow suite, and it is focused on deploying and managing end-to-end ML workflows.

In this chapter, we will focus on the installation and the operation of Kubeflow Pipelines only. If you are interested in a deeper introduction to Kubeflow, we recommend the project's documentation.

Furthermore, we can recommend two Kubeflow books:

- *Kubeflow Operations Guide* by Josh Patterson et al. (O'Reilly)
- *Kubeflow for Machine Learning* (forthcoming) by Holden Karau et al. (O'Reilly)

As we will demonstrate in this chapter, Kubeflow Pipelines provides a highly scalable way of running machine learning pipelines. Kubeflow Pipelines is running Argo behind the scenes to orchestrate the individual component dependencies. Due to this orchestration through Argo, our pipeline orchestration will have a different workflow, as we discussed in Chapter 11. We will take a look at the Kubeflow Pipelines orchestration workflow in "Orchestrating TFX Pipelines with Kubeflow Pipelines".

WHAT IS ARGO?

Argo is a collection of tools for managing workflows, rollouts, and continuous delivery tasks. Initially designed to manage DevOps tasks, it is also a great manager for machine learning workflows. Argo manages all tasks as containers within the Kubernetes environment. For more information, please check out the continuously growing documentation.

Installation and Initial Setup

Kubeflow Pipelines is executed inside a Kubernetes cluster. For this section, we will assume that you have a Kubernetes cluster created with at least 16 GB and 8 CPUs across your node pool and that you have configured `kubectl` to connect with your newly created Kubernetes cluster.

CREATE A KUBERNETES CLUSTER

For a basic setup of a Kubernetes cluster on a local machine or a cloud provider like Google Cloud, please check out Appendix A and Appendix B. Due to the resource requirements by Kubeflow Pipelines, the Kubernetes setup with a cloud provider is preferred. Managed Kubernetes services available from cloud providers include:

1. Amazon Elastic Kubernetes Service (Amazon EKS)
2. Google Kubernetes Engine (GKE)
3. Microsoft Azure Kubernetes Service (AKS)
4. IBM's Kubernetes Service

For more details regarding Kubeflow's underlying architecture, Kubernetes, we highly recommend *Kubernetes: Up and Running* by Brendan Burns et al. (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=0.5.0
$ kubectl apply -k "github.com/kubeflow/pipelines/manifests/"\
    "kustomize/cluster-scoped-resources?ref=$PIPELINE_VERSION"
customresourcedefinition.apiextensions.k8s.io/
    applications.app.k8s.io created
...
clusterrolebinding.rbac.authorization.k8s.io/
    kubeflow-pipelines-cache-deployer-clusterrolebinding created
```

```
$ kubectl wait --for condition=established \
              --timeout=60s crd/applications.app.k8s.io
customresourcedefinition.apiextensions.k8s.io/
    applications.app.k8s.io condition met

$ kubectl apply -k "github.com/kubeflow/pipelines/manifests/"\
    "kustomize/env/dev?ref=$PIPELINE_VERSION"
```

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

```
$ kubectl -n kubeflow get pods
NAME                                              READY   STATUS      AGE
cache-deployer-deployment-c6896d66b-62gc5         0/1     Pending     90s
cache-server-8869f945b-4k7qk                      0/1     Pending     89s
controller-manager-5cbdfbc5bd-bnfxx               0/1     Pending     89s
...
```

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                                              READY   STATUS      AGE
cache-deployer-deployment-c6896d66b-62gc5         1/1     Running     4m6s
cache-server-8869f945b-4k7qk                      1/1     Running     4m6s
controller-manager-5cbdfbc5bd-bnfxx               1/1     Running     4m6s
...
```

Individual pods can be investigated with:

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

MANAGED KUBEFLOW PIPELINES INSTALLATIONS

If you would like to experiment with Kubeflow Pipelines, Google Cloud provides managed installations through the Google Cloud AI Platform. In "Pipelines Based on Google Cloud AI Platform", we'll discuss in-depth how to run your TFX pipelines on Google Cloud's AI Platform and how to create setups on Kubeflow Pipelines from Google Cloud's Marketplace.

Accessing Your Kubeflow Pipelines Installation

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:

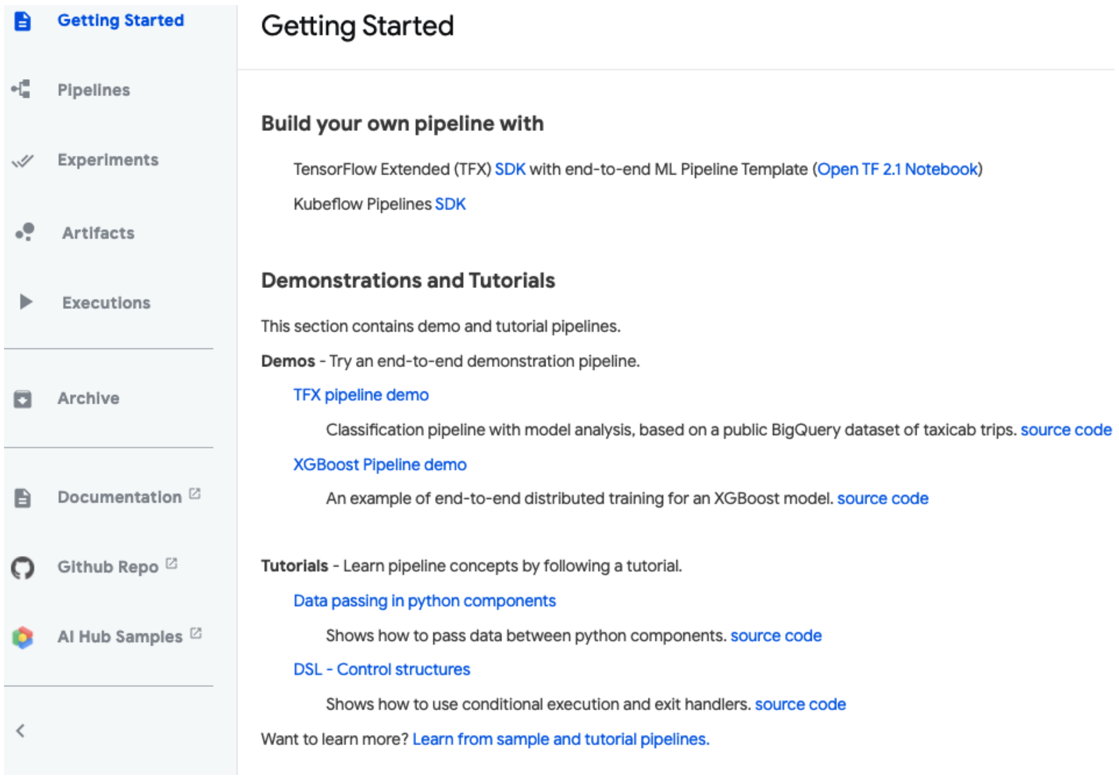
```
$ kubectl port-forward -n kubeflow svc/ml-pipeline-ui 8080:80
```

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.

Google Cloud users can access Kubeflow Pipelines by accessing the public domain created for your Kubeflow installation. You can obtain the URL by executing:

```
$ kubectl describe configmap inverse-proxy-config -n kubeflow \
| grep googleusercontent.com
<id>-dot-<region>.pipelines.googleusercontent.com
```

You can then access the provided URL with your browser of choice. If everything works out, you will see the Kubeflow Pipelines dashboard or the landing page, as shown in Figure 12-3.



Getting Started

Build your own pipeline with

TensorFlow Extended (TFX) SDK with end-to-end ML Pipeline Template (Open TF 2.1 Notebook)

Kubeflow Pipelines SDK

Demonstrations and Tutorials

This section contains demo and tutorial pipelines.

Demos - Try an end-to-end demonstration pipeline.

TFX pipeline demo

Classification pipeline with model analysis, based on a public BigQuery dataset of taxicab trips. source code

XGBoost Pipeline demo

An example of end-to-end distributed training for an XGBoost model. source code

Tutorials - Learn pipeline concepts by following a tutorial.

Data passing in python components

Shows how to pass data between python components. source code

DSL - Control structures

Shows how to use conditional execution and exit handlers. source code

Want to learn more? Learn from sample and tutorial pipelines.

Figure 12-3. Getting started with 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 the pipeline orchestration and the workflow from TFX to Kubeflow Pipelines.

Orchestrating TFX Pipelines with Kubeflow Pipelines

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. In "Pipelines Based on Google Cloud AI Platform", we'll show how we can take advantage of a managed cloud service like GCP's Dataflow to scale your pipelines beyond your Kubernetes cluster.

Before we get into the details of how to orchestrate machine learning 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 what we discussed in Chapter 11, so we will begin with an overview of the full picture. Figure 12-4 shows the overall architecture.

As with Airflow and Beam, we still need a Python script that defines the TFX components in our pipeline. We'll reuse the Example 11-1 script from Chapter 11. In contrast to the execution of the Apache Beam or Airflow TFX runners, the Kubeflow runner won't trigger a pipeline run, but rather generates the configuration files for an execution on the Kubeflow setup.

As shown in Figure 12-4, 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.

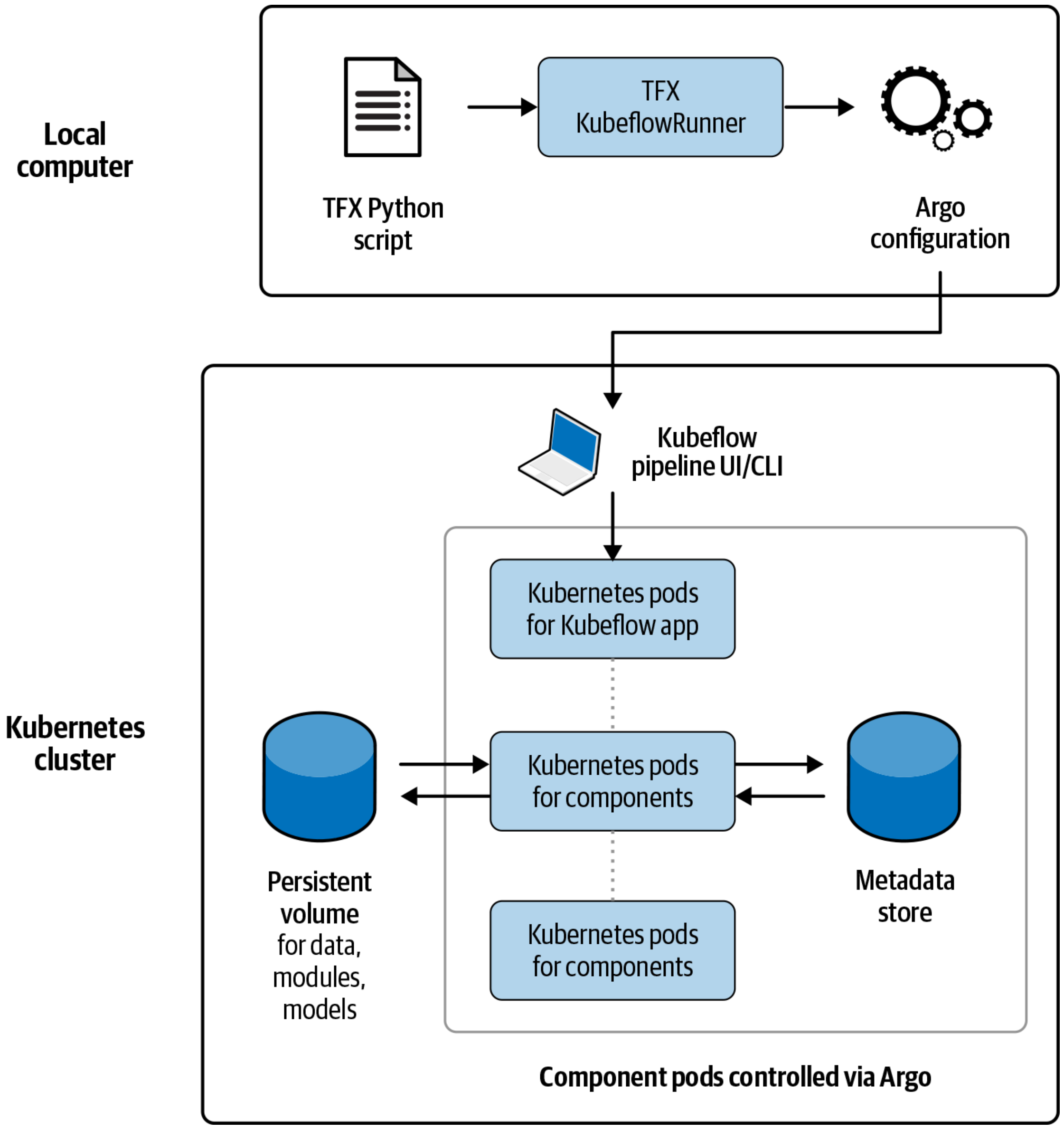

Local computer
TFX Python script
TFX KubeflowRunner
Argo configuration
Kubernetes cluster
Kubeflow pipeline UI/CLI
Kubernetes pods for Kubeflow app
Kubernetes pods for components
Kubernetes pods for components
Persistent volume for data, modules, models
Metadata store
Component pods controlled via Argo

Figure 12-4. Workflow from TFX script to Kubeflow Pipelines

CUSTOM TFX CONTAINER IMAGES

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`. We describe how to do this in Appendix C.

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 hard drives that can be accessed by all components (e.g., in cloud storage buckets or persistent volumes in a Kubernetes cluster). If you are interested in setting up a persistent volume, check out "Exchange Data Through Persistent Volumes" in Appendix C.

Pipeline Setup

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 that 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 our pipeline containers (if required), we can now "assemble" the TFX Runner script to generate the Argo YAML instructions for our Kubeflow Pipelines execution.[1]

As we discussed in Chapter 11, we can reuse the `init_components` function to generate our components. This allows us to focus on the Kubeflow-specific configuration.

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

```
import os

pipeline_name = 'consumer_complaint_pipeline_kubeflow'

persistent_volume_claim = 'tfx-pvc'
persistent_volume = 'tfx-pv'
persistent_volume_mount = '/tfx-data'

# Pipeline inputs
data_dir = os.path.join(persistent_volume_mount, 'data')
module_file = os.path.join(persistent_volume_mount, 'components', 'module.py')
```

```
# Pipeline outputs
output_base = os.path.join(persistent_volume_mount, 'output', pipeline_name)
serving_model_dir = os.path.join(output_base, pipeline_name)
```

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, 'data')
module_file = os.path.join(bucket, 'components', 'module.py')
...
```

With the files paths 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. Calling `get_default_kubeflow_metadata_config()` will return the database information provided by the Kubernetes cluster. 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., *gcr.io/oreilly-book/ml-pipelines-tfx-custom:0.22.0*).

`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,[2] 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_dag_runner

...
PROJECT_ID = 'oreilly-book'
IMAGE_NAME = 'ml-pipelines-tfx-custom'
TFX_VERSION = '0.22.0'

metadata_config = \
    kubeflow_dag_runner.get_default_kubeflow_metadata_config() ❶
pipeline_operator_funcs = \
    kubeflow_dag_runner.get_default_pipeline_operator_funcs() ❷
pipeline_operator_funcs.append( ❸
    onprem.mount_pvc(persistent_volume_claim,
                     persistent_volume,
                     persistent_volume_mount))
runner_config = kubeflow_dag_runner.KubeflowDagRunnerConfig(
    kubeflow_metadata_config=metadata_config,
    tfx_image="gcr.io/{}/{}:{}".format(
        PROJECT_ID, IMAGE_NAME, TFX_VERSION), ❹
    pipeline_operator_funcs=pipeline_operator_funcs
)
```

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.

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 digital subscriber line (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_memory())
```

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:[3].]

```
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()
)
```

The function `use_aws_secret()` assumes that the *AWS_ACCESS_KEY_ID* and *AWS_SECRET_ACCESS_KEY* are registered as base64-encoded Kubernetes secrets.[4] The equivalent function for Google Cloud credentials is called `use_gcp_secrets()`.

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_dag_runner
from pipelines.base_pipeline import init_components, init_pipeline ❶

components = init_components(data_dir, module_file, serving_model_dir,
                             training_steps=50000, eval_steps=15000)
p = init_pipeline(components, output_base, direct_num_workers=0)

output_filename = "{}.yaml".format(pipeline_name)
kubeflow_dag_runner.KubeflowDagRunner(config=runner_config,
                                      output_dir=output_dir, ❷
                                      output_filename=output_filename).run(p)
```

❶ Reuse the base modules for the components.

❷ Optional Argument.

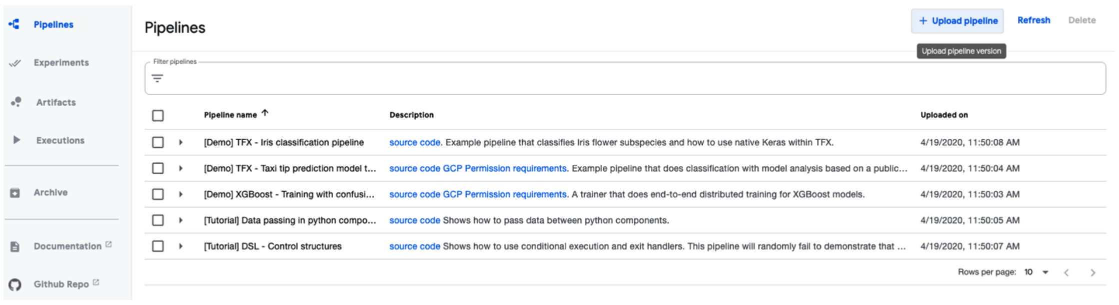
The arguments `output_dir` and `output_filename` are optional. If not provided, the Argo configuration will be provided as a compressed *tar.gz* file in the same directory from which we executed the following python script. For better visibility, we configured the output format to be YAML, and we set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

After running the following command, you will find the Argo configuration *consumer_complaint_pipeline_kubeflow.yaml* in the directory *pipelines/kubeflow_pipelines/argo_pipeline_files/*:

```
$ python pipelines/kubeflow_pipelines/pipeline_kubeflow.py
```

Executing the Pipeline

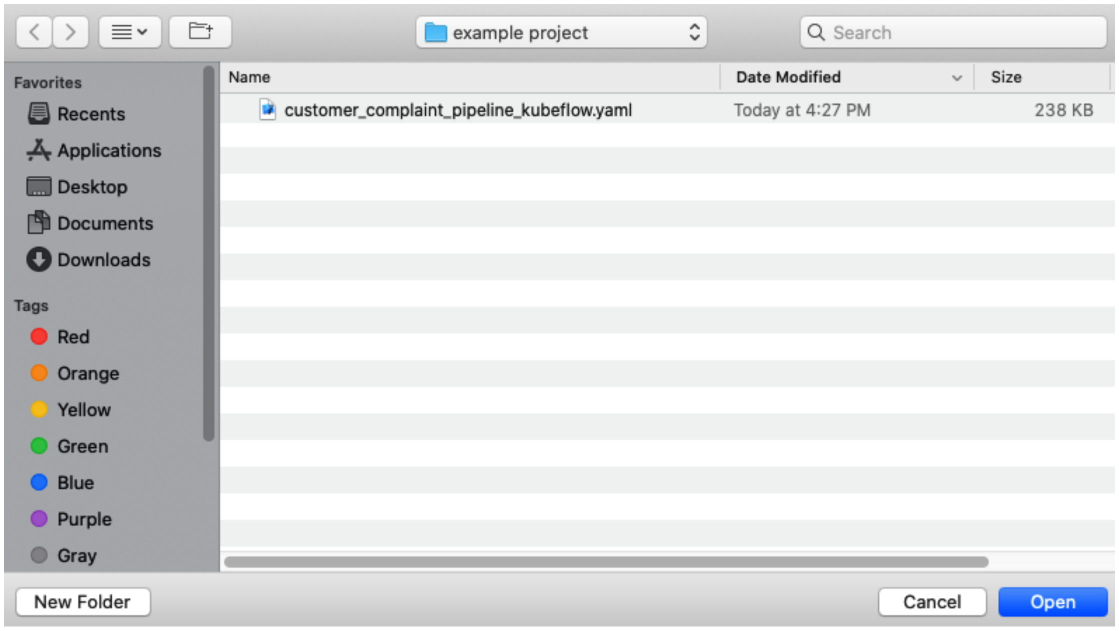
Now it is time to access your Kubeflow Pipelines dashboard. If you want to create a new pipeline, click "Upload pipeline" for uploading, as shown in Figure 12-5. Alternatively, you can select an existing pipeline and upload a new version.



Pipeline name	Description	Uploaded on
[Demo] TFX - Iris classification pipeline	source code. Example pipeline that classifies Iris flower subspecies and how to use native Keras within TFX.	4/19/2020, 11:50:08 AM
[Demo] TFX - Taxi tip prediction model t...	source code GCP Permission requirements. Example pipeline that does classification with model analysis based on a public...	4/19/2020, 11:50:04 AM
[Demo] XGBoost - Training with confusi...	source code GCP Permission requirements. A trainer that does end-to-end distributed training for XGBoost models.	4/19/2020, 11:50:03 AM
[Tutorial] Data passing in python compo...	source code Shows how to pass data between python components.	4/19/2020, 11:50:05 AM
[Tutorial] DSL - Control structures	source code Shows how to use conditional execution and exit handlers. This pipeline will randomly fail to demonstrate that ...	4/19/2020, 11:50:07 AM

Figure 12-5. Overview of loaded pipelines

Select the Argo configuration, as shown in Figure 12-6.

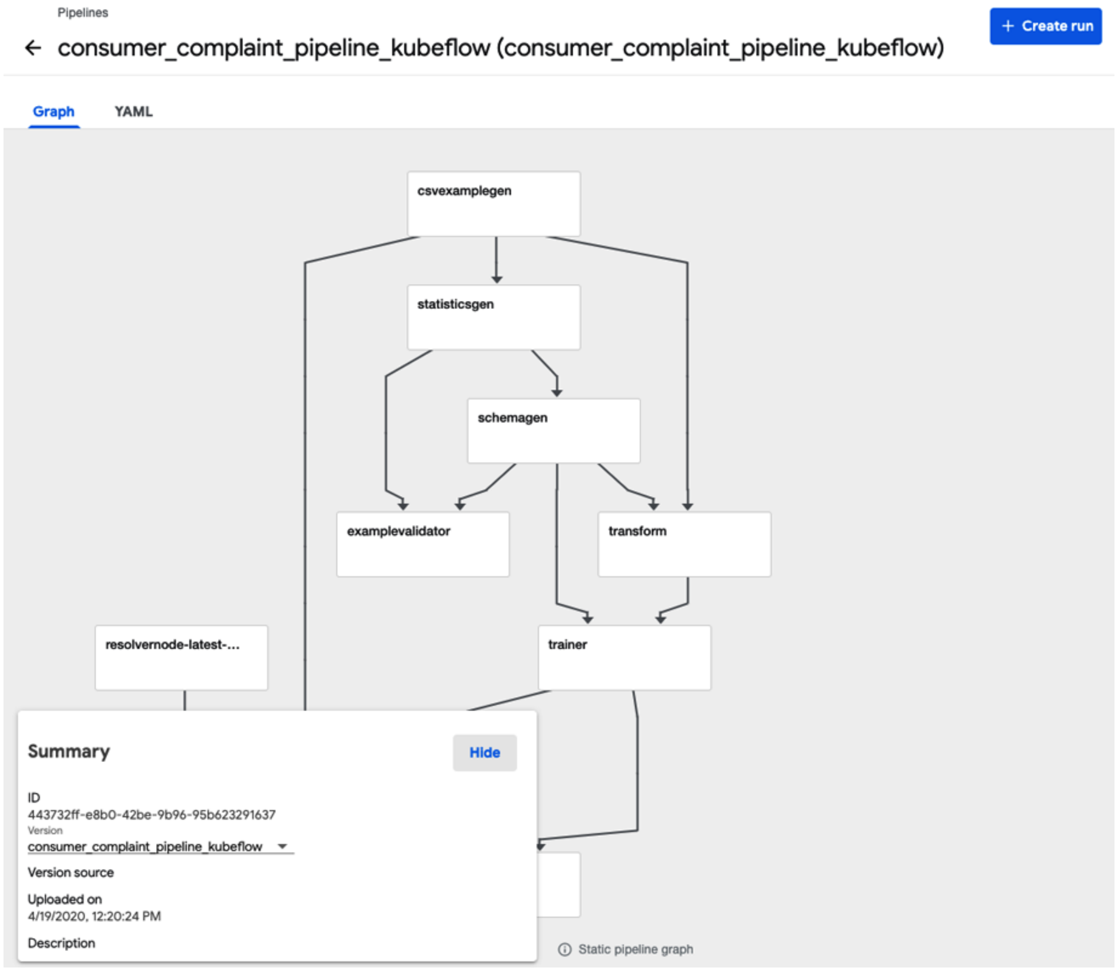


Name	Date Modified	Size
customer_complaint_pipeline_kubeflow.yaml	Today at 4:27 PM	238 KB

Figure 12-6. Selecting your generated Argo configuration file

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 12-7.

You can now configure your pipeline run. Pipelines can be run once or on a reoccurring basis (e.g., with a cron job). Kubeflow Pipelines also allows you to group your pipeline runs in *experiments*.



Pipelines

consumer_complaint_pipeline_kubeflow (consumer_complaint_pipeline_kubeflow)

Summary

ID
443732ff-e8b0-42be-9b96-95b623291637
Version
consumer_complaint_pipeline_kubeflow

Version source

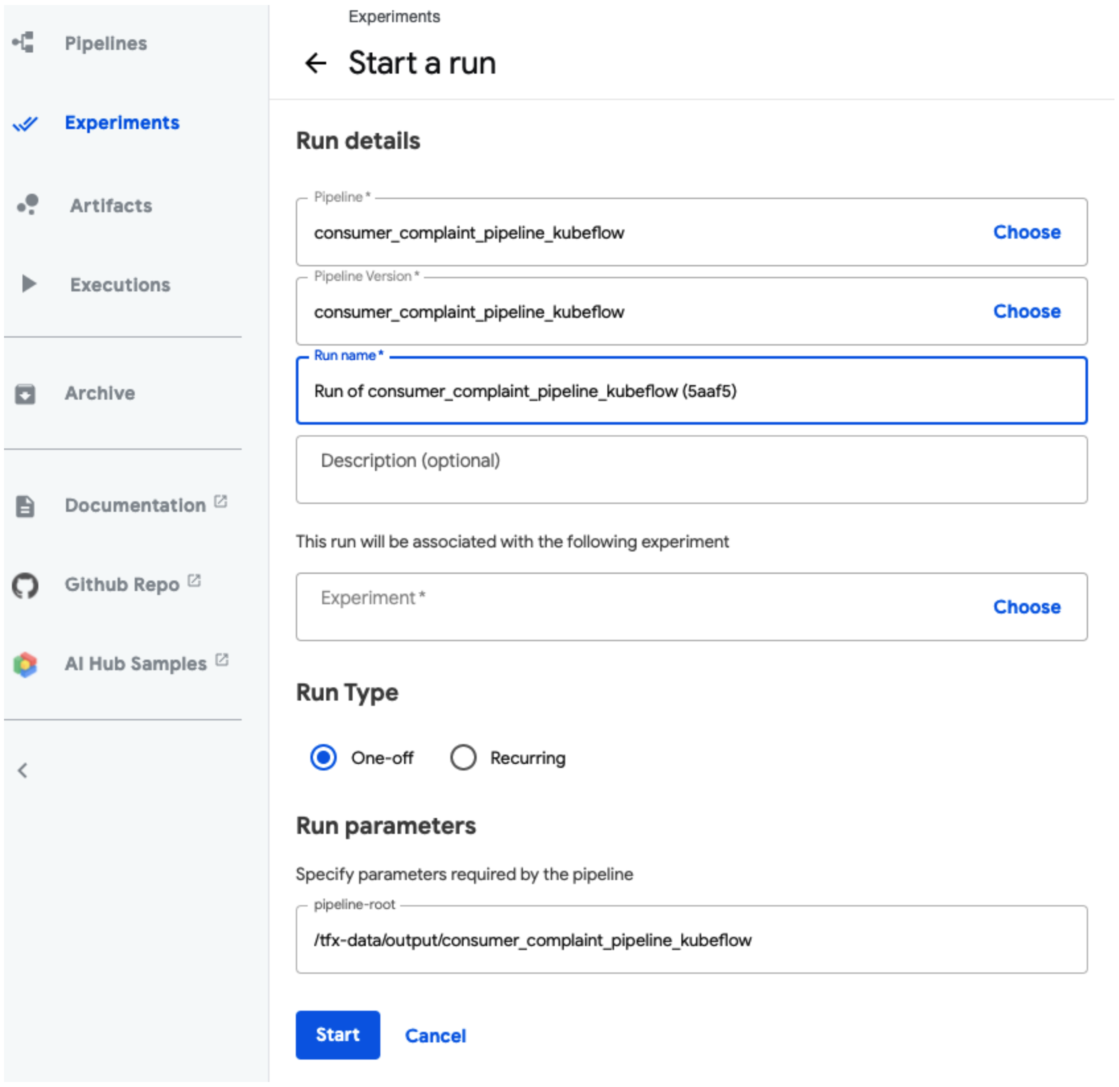
Uploaded on
4/19/2020, 12:20:24 PM

Description

Figure 12-7. Creating a pipeline run

Once you hit Start, as shown in Figure 12-8, 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.

If you want to see the execution details of a run in progress, you can click the "Run name," as shown in Figure 12-9.



Experiments

Start a run

Run details

Pipeline *: consumer_complaint_pipeline_kubeflow — Choose

Pipeline Version *: consumer_complaint_pipeline_kubeflow — Choose

Run name *: Run of consumer_complaint_pipeline_kubeflow (5aaf5)

Description (optional)

This run will be associated with the following experiment

Experiment * — Choose

Run Type

[✓ One-off] Recurring

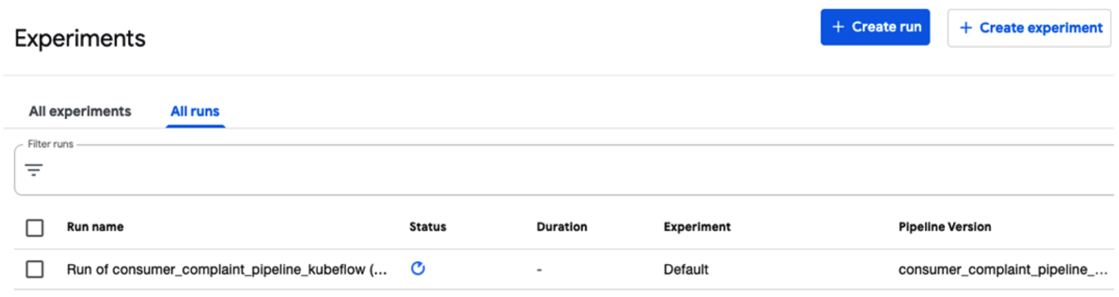
Run parameters

Specify parameters required by the pipeline

pipeline-root: /tfx-data/output/consumer_complaint_pipeline_kubeflow

Start Cancel

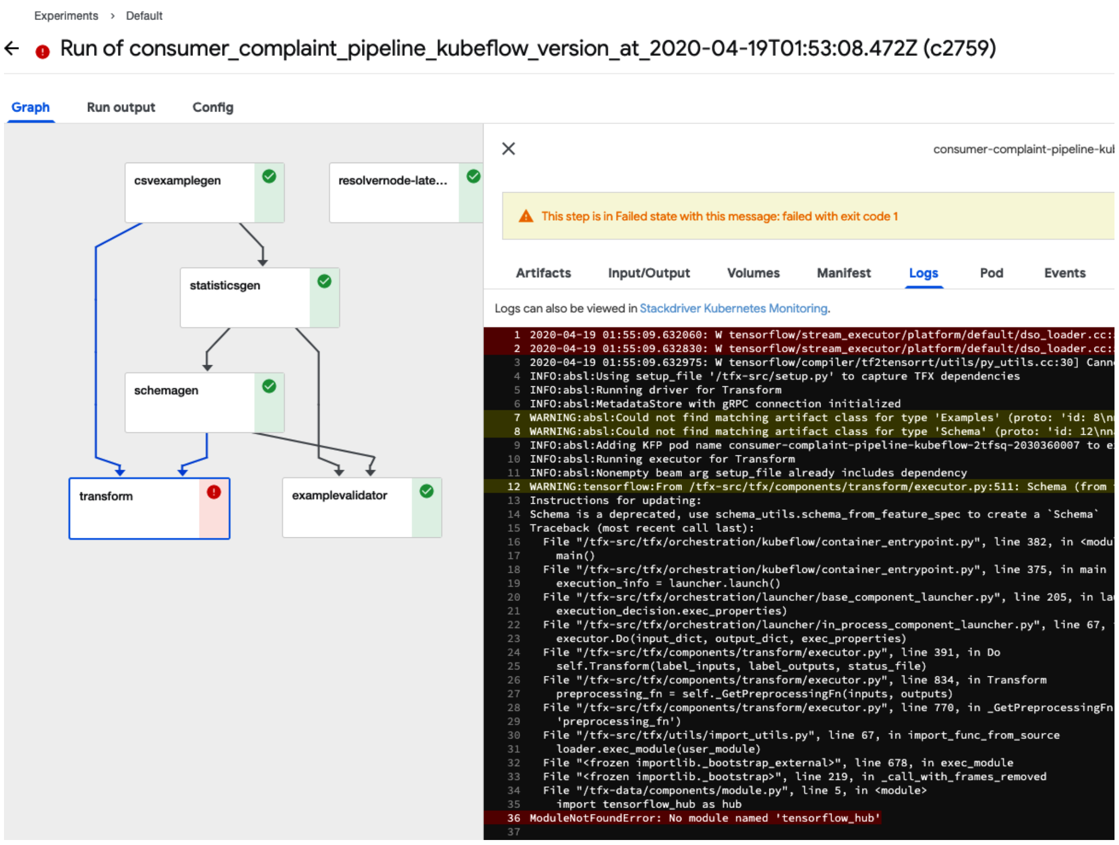
Figure 12-8. Defined pipeline run details



Run name	Status	Duration	Experiment	Pipeline Version
Run of consumer_complaint_pipeline_kubeflow (...		-	Default	consumer_complaint_pipeline_...

Figure 12-9. Pipeline run in progress

You can now inspect the components during or after their execution. For example, you can check the log files from a particular component if the component failed. Figure 12-10 shows an example where the `Transform` component is missing a Python library. Missing libraries can be provided by adding them to a custom TFX container image as discussed in Appendix C.



This step is in Failed state with this message: failed with exit code 1

Logs can also be viewed in Stackdriver Kubernetes Monitoring.

```
2020-04-19 01:55:09.632060: W tensorflow/stream_executor/platform/default/dso_loader.cc:
2020-04-19 01:55:09.632830: W tensorflow/stream_executor/platform/default/dso_loader.cc:
2020-04-19 01:55:09.632975: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:30] Cann
INFO:absl:Using setup_file '/tfx-src/setup.py' to capture TFX dependencies
INFO:absl:Running driver for Transform
INFO:absl:MetadataStore with gRPC connection initialized
WARNING:absl:Could not find matching artifact class for type 'Examples' (proto: 'id: 8\n
WARNING:absl:Could not find matching artifact class for type 'Schema' (proto: 'id: 12\nn
INFO:absl:Adding KFP pod name consumer-complaint-pipeline-kubeflow-2tfsq-2030360007 to e
INFO:absl:Running executor for Transform
INFO:absl:Nonempty beam arg setup_file already includes dependency
WARNING:tensorflow:From /tfx-src/tfx/components/transform/executor.py:511: Schema (from
Instructions for updating:
Schema is a deprecated, use schema_utils.schema_from_feature_spec to create a `Schema`
Traceback (most recent call last):
  File "/tfx-src/tfx/orchestration/kubeflow/container_entrypoint.py", line 382, in <modu
    main()
  File "/tfx-src/tfx/orchestration/kubeflow/container_entrypoint.py", line 375, in main
    execution_info = launcher.launch()
  File "/tfx-src/tfx/orchestration/launcher/base_component_launcher.py", line 205, in la
    execution_decision.exec_properties)
  File "/tfx-src/tfx/orchestration/launcher/in_process_component_launcher.py", line 67,
    executor.Do(input_dict, output_dict, exec_properties)
  File "/tfx-src/tfx/components/transform/executor.py", line 391, in Do
    self.Transform(label_inputs, label_outputs, status_file)
  File "/tfx-src/tfx/components/transform/executor.py", line 834, in Transform
    preprocessing_fn = self._GetPreprocessingFn(inputs, outputs)
  File "/tfx-src/tfx/components/transform/executor.py", line 770, in _GetPreprocessingFn
    'preprocessing_fn')
  File "/tfx-src/tfx/utils/import_utils.py", line 67, in import_func_from_source
    loader.exec_module(user_module)
  File "<frozen importlib._bootstrap_external>", line 678, in exec_module
  File "<frozen importlib._bootstrap>", line 219, in _call_with_frames_removed
  File "/tfx-data/components/module.py", line 5, in <module>
    import tensorflow_hub as hub
ModuleNotFoundError: No module named 'tensorflow_hub'
```

Figure 12-10. Inspecting a component failure

A successful pipeline run is shown in Figure 12-11. After a run completes, you can find the validated and exported machine learning model in the filesystem location set in the `Pusher` component. In our example case, we pushed the model to the path */tfx-data/output/consumer_complaint_pipeline_kubeflow/* on the persistent volume.

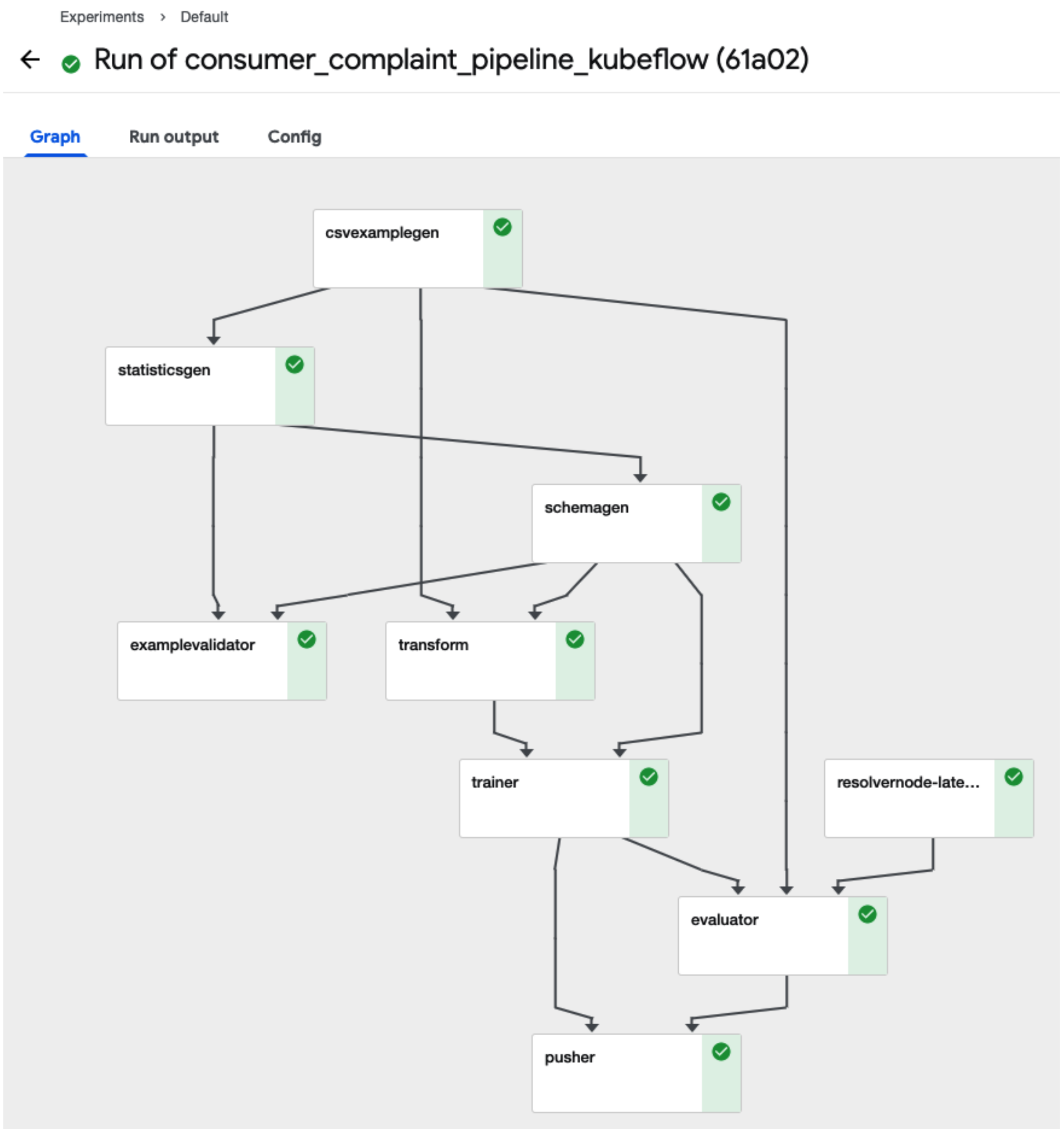


Figure 12-11. Successful pipeline run

You can also inspect the status of your pipeline with `kubectl`. Since every component runs as its own pod, all pods with the pipeline name in the name prefix should be in the state Completed:

```
$ kubectl -n kubeflow get pods
NAME                                                   READY   STATUS      AGE
cache-deployer-deployment-c6896d66b-gmkqf              1/1     Running     28m
cache-server-8869f945b-lb8tb                           1/1     Running     28m
consumer-complaint-pipeline-kubeflow-nmvzb-1111865054  0/2     Completed   10m
consumer-complaint-pipeline-kubeflow-nmvzb-1148904497  0/2     Completed   3m38
consumer-complaint-pipeline-kubeflow-nmvzb-1170114787  0/2     Completed   9m
consumer-complaint-pipeline-kubeflow-nmvzb-1528408999  0/2     Completed   5m43
consumer-complaint-pipeline-kubeflow-nmvzb-2236032954  0/2     Completed   13m
consumer-complaint-pipeline-kubeflow-nmvzb-2253512504  0/2     Completed   13m
consumer-complaint-pipeline-kubeflow-nmvzb-2453066854  0/2     Completed   10m
```

```
consumer-complaint-pipeline-kubeflow-nmvzb-2732473209   0/2    Completed    11m
consumer-complaint-pipeline-kubeflow-nmvzb-997527881    0/2    Completed    10m
...
```

You can also investigate the logs of a specific component through `kubectl` by executing the following command. Logs for specific components can be retrieved through the corresponding pod:

```
$ kubectl logs -n kubeflow podname
```

TFX CLI

Alternative to the UI-based pipeline creation process, you can also create pipelines and kick off pipeline runs programmatically through the TFX CLI. You can find details on how to set up the TFX CLI and how to deploy machine learning pipelines without a UI in "TFX Command-Line Interface" of Appendix C.

Useful Features of Kubeflow Pipelines

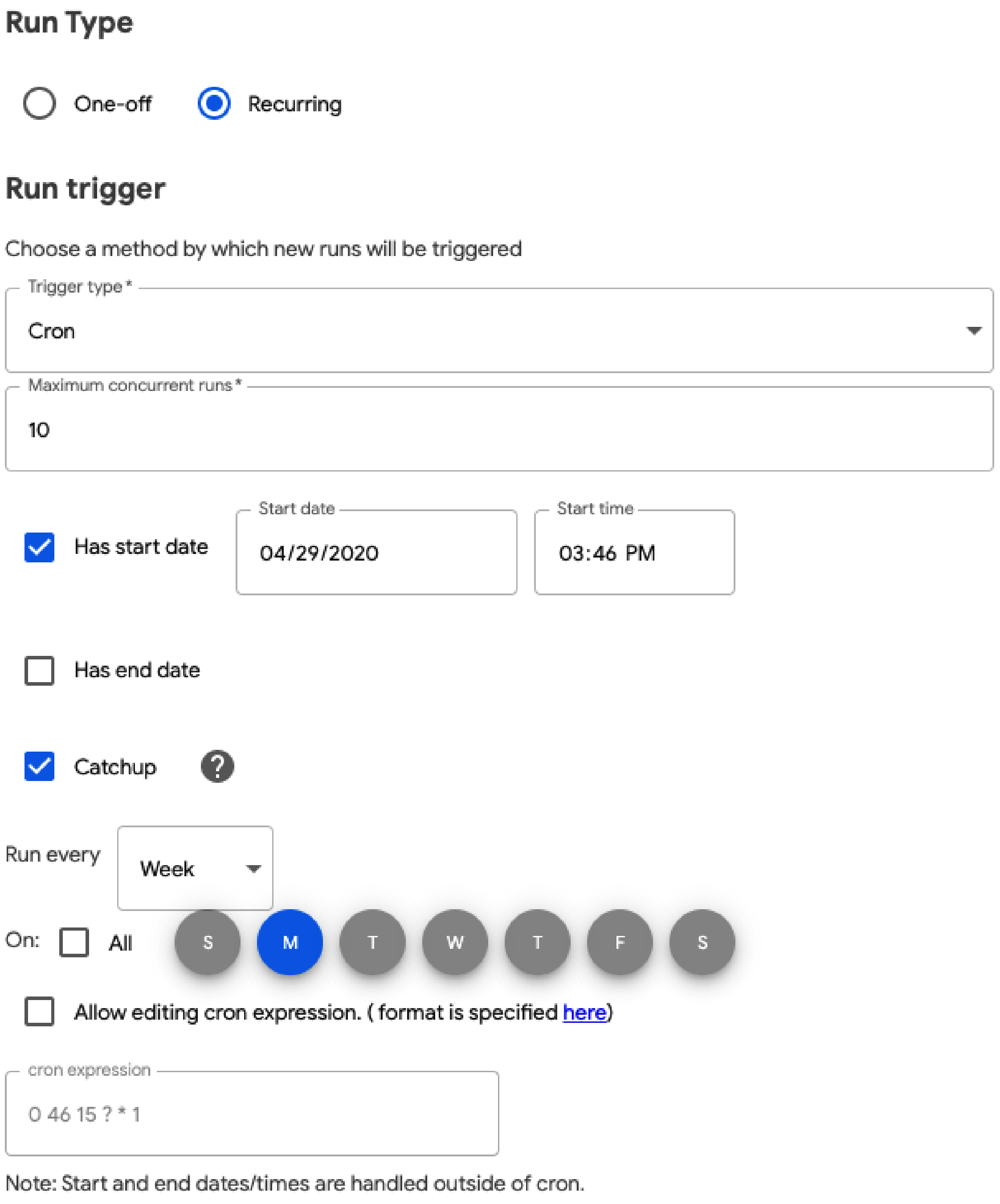
In the following sections, we want to highlight useful features of Kubeflow Pipelines.

Restart failed pipelines

The execution of pipeline runs can take a while, sometimes a matter of hours. TFX stores the state of each component in the ML MetadataStore, and it is possible for Kubeflow Pipelines to track the successfully completed component tasks of a pipeline run. Therefore, it offers the functionality to restart failed pipeline runs from the component of the last failure. This will avoid rerunning successfully completed components and, therefore, save time during the pipeline rerun.

Recurring runs

Besides kicking off individual pipeline runs, Kubeflow Pipelines also lets us run the pipeline according to a schedule. As shown in Figure 12-12, we can schedule runs similar to schedules in Apache Airflow.


Run Type

One-off [✓ Recurring]

Run trigger

Choose a method by which new runs will be triggered

Trigger type *: Cron

Maximum concurrent runs *: 10

[✓] Has start date — Start date: 04/29/2020 — Start time: 03:46 PM

[ ] Has end date

[✓] Catchup

Run every: Week

On: [ ] All — S [✓ M] T W T F S

[ ] Allow editing cron expression. (format is specified here)

cron expression: 0 46 15 ? * 1

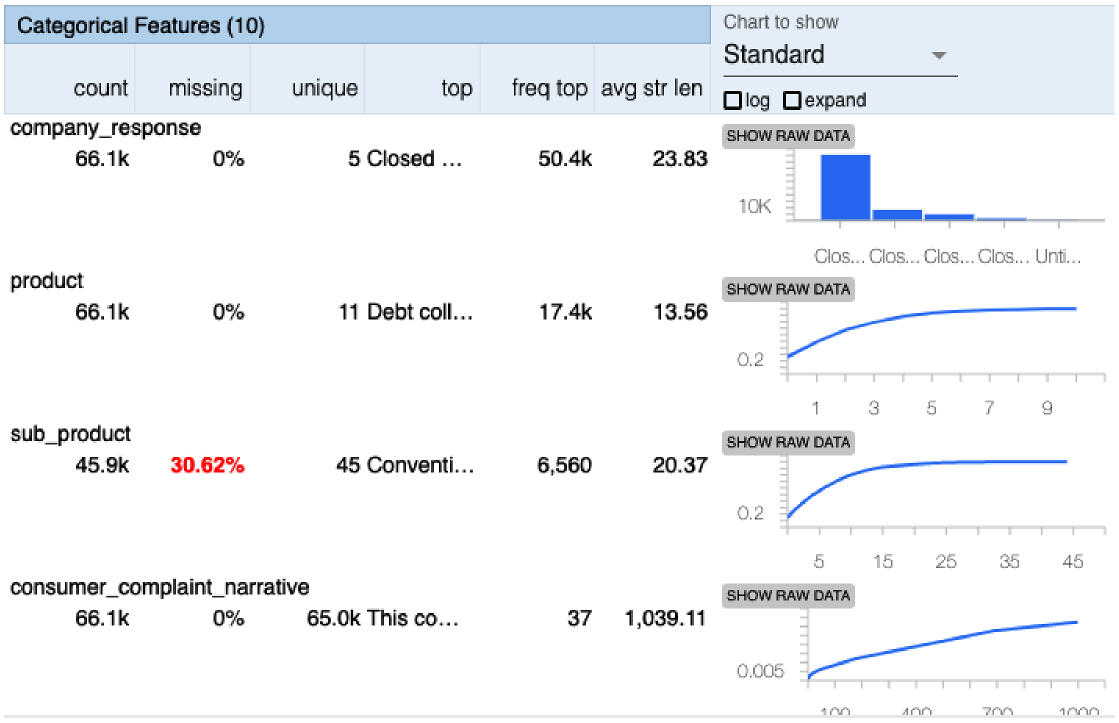
Note: Start and end dates/times are handled outside of cron.

Figure 12-12. Scheduling recurring runs with Kubeflow Pipelines

Collaborating and reviewing pipeline runs

Kubeflow Pipelines provides interfaces for data scientists to collaborate and to review the pipeline runs as a team. In Chapters 4 and 7, we discussed visualizations to show the results of the data or model validation. After the completion of these pipeline components, we can review the results of the components.

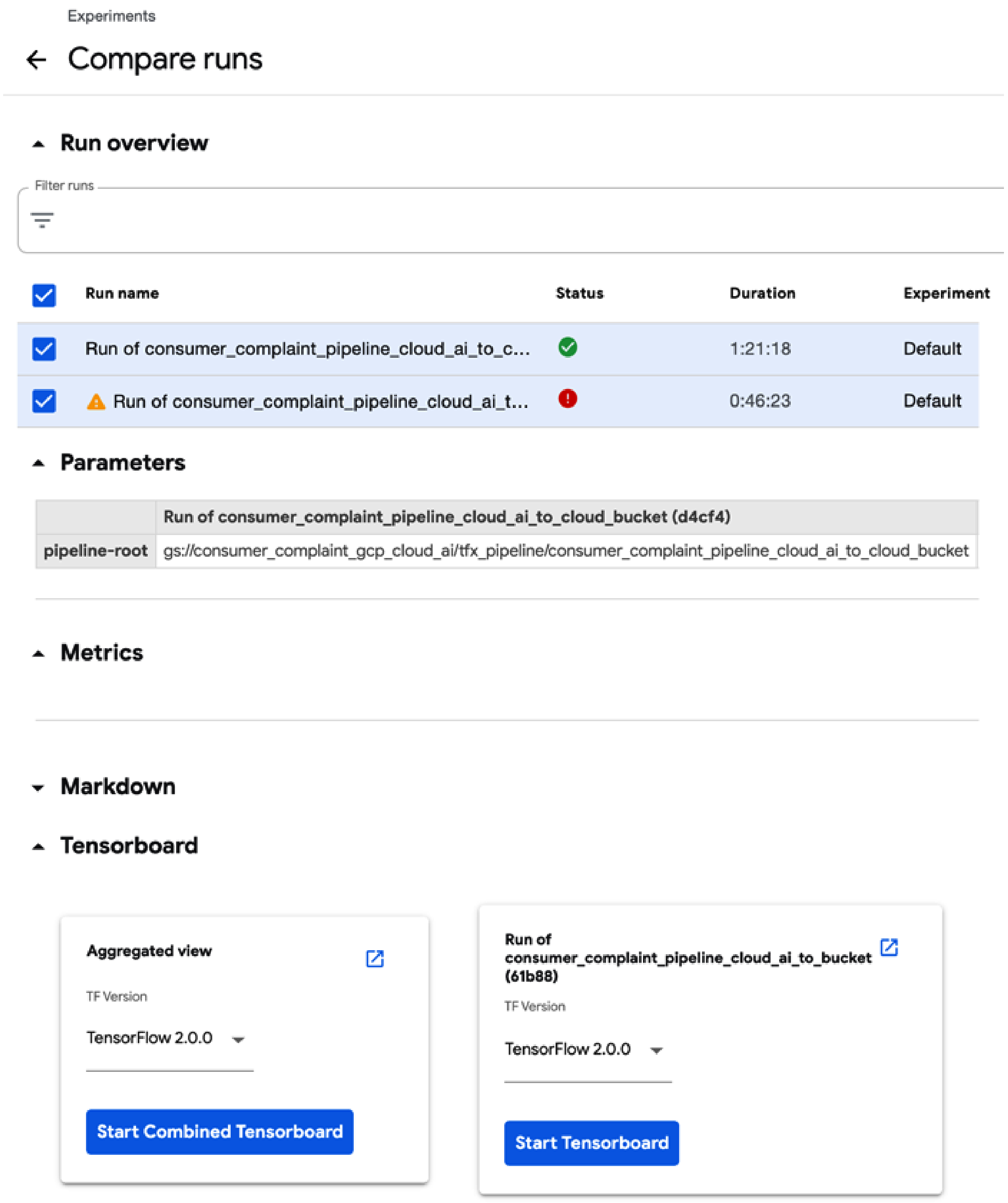
Figure 12-13 shows the results of the data validation step as an example. Since the component output is saved to a disk or to a cloud storage bucket, we can also review the pipeline runs retroactively.


Categorical Features (10)

	count	missing	unique	top	freq top	avg str len
company_response	66.1k	0%	5	Closed …	50.4k	23.83
product	66.1k	0%	11	Debt coll…	17.4k	13.56
sub_product	45.9k	30.62%	45	Conventi…	6,560	20.37
consumer_complaint_narrative	66.1k	0%	65.0k	This co…	37	1,039.11

Figure 12-13. TFDV statistics available in Kubeflow Pipelines

Since the results from every pipeline run and component of those runs are saved in the ML MetadataStore, we can also compare the runs. As shown in Figure 12-14, Kubeflow Pipelines provides a UI to compare pipeline runs.


Experiments

Compare runs

Run overview

Filter runs

Run name	Status	Duration	Experiment
Run of consumer_complaint_pipeline_cloud_ai_to_c...	✓	1:21:18	Default
⚠ Run of consumer_complaint_pipeline_cloud_ai_t...	!	0:46:23	Default

Parameters

	Run of consumer_complaint_pipeline_cloud_ai_to_cloud_bucket (d4cf4)
pipeline-root	gs://consumer_complaint_gcp_cloud_ai/tfx_pipeline/consumer_complaint_pipeline_cloud_ai_to_cloud_bucket

Metrics

Markdown

Tensorboard

Aggregated view
TF Version: TensorFlow 2.0.0
Start Combined Tensorboard

Run of consumer_complaint_pipeline_cloud_ai_to_bucket (61b88)
TF Version: TensorFlow 2.0.0
Start Tensorboard

Figure 12-14. Comparing pipeline runs with Kubeflow Pipelines

Kubeflow Pipelines also integrates TensorFlow's TensorBoard nicely. As you can see in Figure 12-15, we can review the statistics from model training runs with TensorBoard. After the creation of the underlying Kubernetes pod, we can review the statistics from model training runs with TensorBoard.

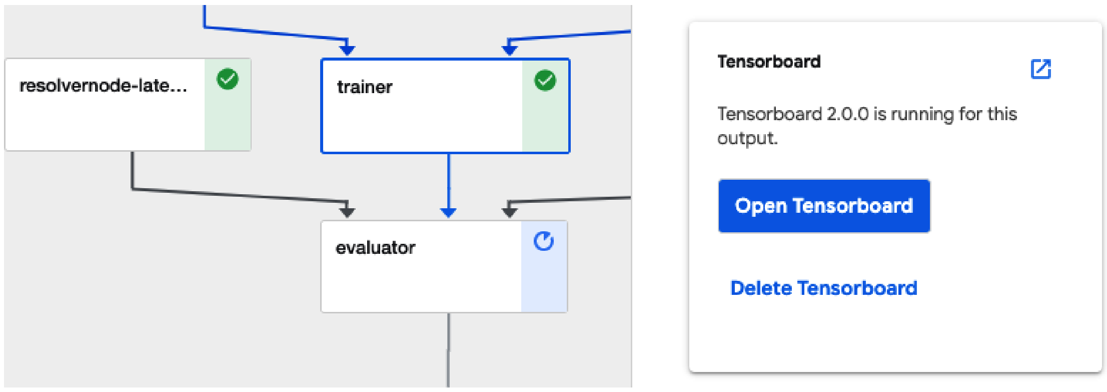


Figure 12-15. Reviewing training runs with TensorFlow's TensorBoard

Auditing the pipeline lineage

For the wider adoption of machine learning, it is critical to review the creation of the model. If, for example, data scientists observe that the trained model is unfair (as we discussed in Chapter 7), it is important to retrace and reproduce the data or hyperparameters that we used. We basically need an audit trail for each machine learning model.

Kubeflow Pipelines offers a solution for such an audit trail with the Kubeflow Lineage Explorer. It creates a UI that can query the ML MetadataStore data easily.

As shown in the lower right corner of Figure 12-16, a machine learning model was pushed to a certain location. The Lineage Explorer allows us to retrace all components and artifacts that contributed to the exported model, all the way back to the initial, raw dataset. We can retrace who signed off on the model if we use the human in the loop component (see "Human in the Loop"), or we can check the data validation results and investigate whether the initial training data started to drift.

As you can see, Kubeflow Pipelines is an incredibly powerful tool for orchestrating our machine learning pipelines. If your infrastructure is based on AWS or Azure, or if you want full control over your setup, we recommend this approach. However, if you are already using GCP, or if you would like a simpler way to use Kubeflow Pipelines, read on.

Figure 12-16. Inspecting the pipeline lineage with Kubeflow Pipelines

Pipelines Based on Google Cloud AI Platform

If you don't want to spend the time administrating your own Kubeflow Pipelines setup or if you would like to integrate with GCP's AI Platform or other GCP services like Dataflow, AI Platform training and serving, etc., this section is for you. In the following section, we will discuss how to set up Kubeflow Pipelines through Google Cloud's AI Platform. Furthermore, we will highlight how you can train your machine learning models with Google Cloud's AI jobs and scale your preprocessing with Google Cloud's Dataflow, which can be used as an Apache Beam runner.

Pipeline Setup

Google's AI Platform Pipelines lets you create a Kubeflow Pipelines setup through a UI. Figure 12-17 shows the front page for AI Platform Pipelines, where you can start creating your setup.

BETA PRODUCT

As you can see in Figure 12-17, at the time of writing, this Google Cloud product is still in beta. The presented workflows might change.

Figure 12-17. Google Cloud AI Platform Pipelines

When you click New Instance (near the top right of the page), it will send you to the Google Marketplace, as shown in Figure 12-18.

Figure 12-18. Google Cloud Marketplace page for Kubeflow Pipelines

After selecting Configure, you'll be asked at the top of the menu to either choose an existing Kubernetes cluster or create a cluster, as shown in Figure 12-19.

NODE SIZES

When creating a new Kubernetes cluster or selecting an existing cluster, consider the available memory of the nodes. Each node instance needs to provide enough memory to hold the entire model. For our demo project, we selected `n1-standard-4` as an instance type. At the time of writing, we could not create a custom cluster while starting Kubeflow Pipelines from Marketplace. If your pipeline setup requires larger instances, we recommend creating the cluster and its nodes first, and then selecting the cluster from the list of existing clusters when creating the Kubeflow Pipelines setup from GCP Marketplace.

Figure 12-19. Configuring your cluster for Kubeflow Pipelines

ACCESS SCOPE

During the Marketplace creation of Kubeflow Pipelines or your custom cluster creation, select "Allow full access to all Cloud APIs" when asked for the access scope of the cluster nodes. Kubeflow Pipelines require access to a variety of Cloud APIs. Granting access to all Cloud APIs simplifies the setup process.

After configuring your Kubernetes cluster, Google Cloud will instantiate your Kubeflow Pipelines setup, as shown in Figure 12-20.

Figure 12-20. Creating your Kubeflow Pipelines setup

After a few minutes, your setup will be ready for use and you can find your Kubeflow Pipelines setup as an instance listed in the AI Platform Pipelines list of deployed Kubeflow setups. If you click Open Pipelines Dashboard, as shown in Figure 12-21, you'll be redirected to your newly deployed Kubeflow Pipelines setup. From here, Kubeflow Pipelines will work as we discussed in the previous section and the UI will look very similar.

Figure 12-21. List of Kubeflow deployments

STEP-BY-STEP INSTALLATIONS AVAILABLE IN THE AI PLATFORM PIPELINES DASHBOARD

If you install Kubeflow Pipelines manually step by step, as discussed in "Accessing Your Kubeflow Pipelines Installation" and Appendix B, your Kubeflow Pipelines setup will also be listed under your AI Platform Pipelines instances.

TFX Pipeline Setup

The configuration of our TFX pipelines is very similar to the configuration of the `KubeflowDagRunner`, which we discussed previously. In fact, if you mount a persistent volume with the required Python module and training data as discussed in "Pipeline Setup", you can run your TFX pipelines on the AI Platform Pipelines.

In the next sections, we will show you a few changes to the earlier Kubeflow Pipelines setup that can either simplify your workflow (e.g., loading data from Google Storage buckets) or assist you with scaling pipelines beyond a Kubernetes cluster (e.g., by training a machine learning model with AI Platform Jobs).

Use Cloud Storage buckets for data exchange

In "Pipeline Setup", we discussed that we can load the data and Python modules required for pipeline execution from a persistent volume that is mounted in the Kubernetes cluster. If you run pipelines within the Google Cloud ecosystem, you can also load data from Google Cloud Storage buckets. This will simplify the workflows, enabling you to upload and review files through the GCP web interface or the `gcloud` SDK.

The bucket paths can be provided in the same way as file paths on a disk, as shown in the following code snippet:

```
input_bucket = 'gs://YOUR_INPUT_BUCKET'
output_bucket = 'gs://YOUR_OUTPUT_BUCKET'
data_dir = os.path.join(input_bucket, 'data')

tfx_root = os.path.join(output_bucket, 'tfx_pipeline')
pipeline_root = os.path.join(tfx_root, pipeline_name)
```

```
serving_model_dir = os.path.join(output_bucket, 'serving_model_dir')
module_file = os.path.join(input_bucket, 'components', 'module.py')
```

It is often beneficial to split the buckets between input (e.g., the Python module and training data) and output data (e.g., trained models), but you could also use the same buckets.

Training models with an AI Platform job

If you want to scale model training through a GPU or TPU, you can configure your pipeline to run the training step of the machine learning model on this hardware:

```
project_id = 'YOUR_PROJECT_ID'
gcp_region = 'GCP_REGION>' ❶

ai_platform_training_args = {
    'project': project_id,
    'region': gcp_region,
    'masterConfig': {
        'imageUri': 'gcr.io/oreilly-book/ml-pipelines-tfx-custom:0.22.0'} ❷
    'scaleTier': 'BASIC_GPU', ❸
}
```

❶ For example, `us-central1`.

❷ Provide a custom image (if required).

❸ Other options include `BASIC_TPU`, `STANDARD_1`, and `PREMIUM_1`.

For the `Trainer` component to observe the AI Platform configuration, you need to configure the component executor and swap out the `GenericExecutor` we have used with our `Trainer` component so far. The following code snippet shows the additional arguments required:

```
from
tfx.extensions.google_cloud_ai_platform.trainer import executor \
as ai_platform_trainer_executor
```

```
trainer = Trainer(
    ...
    custom_executor_spec=executor_spec.ExecutorClassSpec(
        ai_platform_trainer_executor.GenericExecutor),
    custom_config = {
            ai_platform_trainer_executor.TRAINING_ARGS_KEY:
                ai_platform_training_args}
)
```

Instead of training machine learning models inside a Kubernetes cluster, you can distribute the model training using the AI Platform. In addition to the distributed training capabilities, the AI Platform provides access to accelerated training hardware like TPUs.

When the `Trainer` component is triggered in the pipeline, it will kick off a training job in the AI Platform Jobs, as shown in Figure 12-22. There you can inspect log files or the completion status of a training task.

Figure 12-22. AI Platform training jobs

Serving models through AI Platform endpoints

If you run your pipelines within the Google Cloud ecosystem, you can also deploy machine learning models to endpoints of the AI Platform. These endpoints have the option to scale your model in case the model experiences spikes of inferences.

Instead of setting a `push_destination` as we discussed in "TFX Pusher Component", we can overwrite the executor and provide Google Cloud details for the AI Platform deployment. The following code snippet shows the required configuration details:

```
ai_platform_serving_args = {
    'model_name': 'consumer_complaint',
    'project_id': project_id,
    'regions': [gcp_region],
}
```

Similar to the setup of the `Trainer` component, we need to exchange the component's executor and provide the `custom_config` with the deployment details:

```
from tfx.extensions.google_cloud_ai_platform.pusher import executor \
    as ai_platform_pusher_executor

pusher = Pusher(
    ...
    custom_executor_spec=executor_spec.ExecutorClassSpec(
        ai_platform_pusher_executor.Executor),
    custom_config = {
        ai_platform_pusher_executor.SERVING_ARGS_KEY:
            ai_platform_serving_args
    }
)
```

If you provide the configuration of the `Pusher` component, you can avoid setting up and maintaining your instance of TensorFlow Serving by using the AI Platform.

DEPLOYMENT LIMITATIONS

At the time of writing, models were restricted to a maximum size of 512 MB for deployments through the AI Platform. Our demo project is larger than the limit and, therefore, can't be deployed through AI Platform endpoints at the moment.

Scaling with Google's Dataflow

So far, all the components that rely on Apache Beam have executed data processing tasks with the default `DirectRunner`, meaning that the processing tasks will be executed on the same instance where Apache Beam initiated the task run. In this situation, Apache Beam will consume as many CPU cores as possible, but it won't scale beyond the single instance.

One alternative is to execute Apache Beam with Google Cloud's Dataflow. In this situation, TFX will process jobs with Apache Beam and the latter will submit tasks to Dataflow. Depending on each job's requirements, Dataflow will spin up compute instances and distribute job tasks across instances. This is a pretty neat way of scaling data preprocessing jobs like statistics generation or data preprocessing.[5]

In order to use the scaling capabilities of Google Cloud Dataflow, we need to provide a few more Beam configurations, which we'll pass to our pipeline instantiation:

```
tmp_file_location = os.path.join(output_bucket, "tmp")
beam_pipeline_args = [
    "--runner=DataflowRunner",
    "--experiments=shuffle_mode=auto",
    "--project={}".format(project_id),
    "--temp_location={}".format(tmp_file_location),
    "--region={}".format(gcp_region),
    "--disk_size_gb=50",
]
```

Besides configuring `DataflowRunner` as the `runner` type, we also set the `shuffle_mode` to `auto`. This is an interesting feature of Dataflow. Instead of running transformations like `GroupByKey` in the Google Compute Engine's VM, the operation will be processed in the service backend of Dataflow. This reduces the execution time and the CPU/memory costs of the compute instances.

Pipeline Execution

Executing the pipeline with the Google Cloud AI Platform is no different than what we discussed in "Orchestrating TFX Pipelines with Kubeflow Pipelines". The TFX script will generate the Argo configuration. The configuration can then be uploaded to the Kubeflow Pipelines setup on the AI Platform.

During the pipeline execution, you can inspect the training jobs as discussed in "Training models with an AI Platform job" and you can observe the Dataflow jobs in detail, as shown in Figure 12-23.

Figure 12-23. Google Cloud Dataflow job details

The Dataflow dashboard provides valuable insights into your job's progress and scaling requirements.

Summary

Running pipelines with Kubeflow Pipelines provides great benefits that we think offset the additional setup requirements. We see the pipeline lineage browsing, the seamless integration with TensorBoard, and the options for recurring runs as good reasons to choose Kubeflow Pipelines as the pipeline orchestrator.

As we discussed earlier, the current workflow for running TFX pipelines with Kubeflow Pipelines is different than the workflows we discussed for pipelines running on Apache Beam or Apache Airflow in Chapter 11. However, the configuration of the TFX components is the same, as we discussed in the previous chapter.

In this chapter, we walked through two Kubeflow pipeline setups: the first setup can be used with almost any managed Kubernetes service, such as AWS Elastic Kubernetes Service or Microsoft Azure Kubernetes Service. The second setup can be used with Google Cloud's AI Platform.

In the following chapter, we will discuss how you can turn your pipeline into a cycle using feedback loops.

[1] You can follow along with the script that generates the Argo YAML instructions in the book's GitHub repo.

[2] For more information on Kubernetes ConfigMaps, check out "Some Kubernetes Definitions".

[3] Visit Nvidia for more on installing their latest drivers for Kubernetes clusters

[4] Check the documentation for information on Kubernetes secrets and how to set them up.

[5] Dataflow is only available through Google Cloud. Alternative distribution runners are Apache Flink and Apache Spark.