Appendix C. Tips for Operating Kubeflow Pipelines

When you operate your TFX pipelines with Kubeflow Pipelines, you might want to customize the underlying container images of your TFX components. Custom TFX images are required if your components rely on additional Python dependencies outside of the TensorFlow and TFX packages. In the case of our demo pipeline, we have an additional Python dependency, the TensorFlow Hub library, for accessing our language model.

In the second half of this appendix, we want to show you how to transfer data to and from your local computer and your persistent volume. The persistent volume setup is beneficial if you can access your data via a cloud storage provider (e.g., with an on-premise Kubernetes cluster). The presented steps will guide you through the process of copying data to and from your cluster.

Custom TFX Images

In our example project, we use a language model provided by TensorFlow Hub. We use the `tensorflow_hub` library to load the language model efficiently. This particular library isn't part of the original TFX image; therefore, we need to build a custom TFX image with the required library. This is also the case if you plan to use custom components like the ones we discussed in Chapter 10.

Fortunately, as we discussed in Appendix A, Docker images can be built without much trouble. The following *Dockerfile* shows our custom image setup:

```
FROM tensorflow/tfx:0.22.0

RUN python3.6 -m pip install "tensorflow-hub" ❶
RUN ... ❷
```

```
ENTRYPOINT ["python3.6", "/tfx-src/tfx/scripts/run_executor.py"]
❸
```

❶ Install required packages.

❷ Install additional packages if needed.

❸ Don't change the container entry point.

We can easily inherit the standard TFX image as a base for our custom image. To avoid any sudden changes in the TFX API, we highly recommend pinning the version of the base image to a specific build (e.g., *tensorflow/tfx:0.22.0*) instead of the common `latest` tag. The TFX images are built on the Ubuntu Linux distribution and come with Python installed. In our case, we can simply install the additional Python package for the Tensorflow Hub models.

It is very important to provide the same entry point as configured in the base image. Kubeflow Pipelines expects that the entry point will trigger the component's executor.

Once we have defined our Docker image, we can build and push the image to a container registry. This can be AWS Elastic, GCP or Azure Container Registry. It's important to ensure that the running Kubernetes cluster can pull images from the container registry and has permission to do so for private containers. In the following code, we demonstrate those steps for the GCP Container Registry:

```
$ export TFX_VERSION=0.22.0
$ export PROJECT_ID=<your gcp project id>
$ export IMAGE_NAME=ml-pipelines-tfx-custom

$ gcloud auth configure-docker
$ docker build pipelines/kubeflow_pipelines/tfx-docker-image/. \
    -t gcr.io/$PROJECT_ID/$IMAGE_NAME:$TFX_VERSION
$ docker push gcr.io/$PROJECT_ID/$IMAGE_NAME:$TFX_VERSION
```

Once the built image is uploaded, you can see the image available in the cloud provider's container registry, as shown in Figure C-1.

COMPONENT-SPECIFIC IMAGES

At the time of writing, it isn't possible to define custom images for specific component containers. At the moment, the requirements for all components need to be included in the image. However, there are currently proposals being discussed to allow component-specific images in the future.

Figure C-1. Google Cloud's Container Registry

We can now use this container image for all of our TFX components in our Kubeflow Pipelines setup.

Exchange Data Through Persistent Volumes

As we discussed earlier, we need to provide containers to mount a filesystem to read from and write data to locations outside of the container filesystem. In the Kubernetes world, we can mount filesystems through *persistent volumes* (PVs) and *persistent volume claims* (PVCs). In simple terms, we can provision a drive to be available inside of a Kubernetes cluster and then claim that filesystem in its entirety or a portion of its space.

You can set up such PVs through the Kubernetes configurations that we provide in "Persistent Volume Setups for Kubeflow Pipelines". If you would like to use this setup, you will need to create a disk with your cloud provider (e.g., AWS Elastic Block Storage or GCP Block Storage). In the following example, we create a disk drive with a size of 20 GB named *tfx-pv-disk*:

```
$ export GCP_REGION=us-central1-c
$ gcloud compute disks create tfx-pv-disk --size=20Gi --zone=$GCP_REGION
```

We can now provision the disk to be used as a PV in our Kubernetes cluster. The following `kubectl` command will facilitate the provisioning:

```
$ kubectl apply -f "https://github.com/Building-ML-Pipelines/"\
    "building-machine-learning-pipelines/blob/master/pipelines/"\
    "kubeflow_pipelines/kubeflow-config/storage.yaml"
$ kubectl apply -f "https://github.com/Building-ML-Pipelines/"\
    "building-machine-learning-pipelines/blob/master/pipelines/"\
    "kubeflow_pipelines/kubeflow-config/storage-claim.yaml"
```

After the provisioning is completed, we can check if the execution worked by calling `kubectl get pvc`, as shown in the following example:

```
$ kubectl -n kubeflow get pvc
NAME             STATUS   VOLUME    CAPACITY   ACCESS MODES   STORAGECLASS   AG
tfx-pvc          Bound    tfx-pvc   20Gi       RWO            manual         2m
```

Kubernetes' `kubectl` provides a handy `cp` command to copy data from our local machines to the remote PV. In order to copy the pipeline data (e.g., the Python module for the transform and training steps, as well as the training data), we need to mount the volume to a Kubernetes pod. For the copy operations, we created a simple app that basically just idles and allows us to access the PV. You can create the pod with the following *kubectl* command:

```
$ kubectl apply -f "https://github.com/Building-ML-Pipelines/"\
    "building-machine-learning-pipelines/blob/master/pipelines/"\
    "kubeflow_pipelines/kubeflow-config/storage-access-pod.yaml"
```

The pod `data-access` will mount the PV, and then we can create the necessary folders and copy the required data to the volume:

```
$ export DATA_POD=`kubectl -n kubeflow get pods -o name | grep data-access`
$ kubectl -n kubeflow exec $DATA_POD -- mkdir /tfx-data/data
$ kubectl -n kubeflow exec $DATA_POD -- mkdir /tfx-data/components
$ kubectl -n kubeflow exec $DATA_POD -- mkdir /tfx-data/output
```

```
$ kubectl -n kubeflow cp \
    ../building-machine-learning-pipelines/components/module.py \
    ${DATA_POD#*/}:/tfx-data/components/module.py
$ kubectl -n kubeflow cp \
    ../building-machine-learning-pipelines/data/consumer_complaints.csv
    ${DATA_POD#*/}:/tfx-data/data/consumer_complaints.csv
```

After all the data is transferred to the PV, you can delete the `data-access` pod by running the following command:

```
$ kubectl delete -f \
    pipelines/kubeflow_pipelines/kubeflow-config/storage-access-pod.yaml
```

The `cp` command also works in the other direction, in case you want to copy the exported model from your Kubernetes cluster to a different location outside of your cluster.

TFX Command-Line Interface

TFX provides a CLI to manage your TFX projects and their orchestration runs. The CLI tool provides you *TFX Templates*, a predefined folder and file structure. Projects that use the provided folder structure can then be managed through the CLI tool instead of a web UI (in the case of Kubeflow and Airflow). It also incorporated the Skaffold library to automate the creation and publication of custom TFX images.

TFX CLI UNDER ACTIVE DEVELOPMENT

The TFX CLI is under active development at the time of writing this section. The commands might change or more functionality might be added. Also, more TFX templates might become available in the future.

TFX and Its Dependencies

TFX CLI requires the *Kubeflow Pipelines SDK* and the Skaffold, a Python tool for continuously building and deploying Kubernetes applications.

If you haven't installed or updated TFX and the Python SDK from Kubeflow Pipelines, run the two `pip install` commands:

```
$ pip install -U tfx
$ pip install -U kfp
```

The installation of Skaffold depends on your operating system:

Linux

```
$ curl -Lo skaffold \
https://storage.googleapis.com/\
skaffold/releases/latest/skaffold-linux-amd64
$ sudo install skaffold /usr/local/bin/
```

MacOS

```
$ brew install skaffold
```

Windows

```
$ choco install -y skaffold
```

After the installation of Skaffold, make sure the execution path of the tool is added to the `PATH` of the terminal environment where you are executing the TFX CLI tool. The following bash example shows how Linux users can add the Skaffold path to their `PATH` bash variable:

```
$ export PATH=$PATH:/usr/local/bin/
```

Before we discuss how to use the TFX CLI tool, let's discuss TFX templates briefly.

TFX Templates

TFX provides project templates to organize machine learning pipeline projects. The templates provide a predefined folder structure and a blueprint for your feature, model, and preprocessing definitions. The following `tfx template copy` command will download the *taxi cab* example project of the TFX project:

```
$ export PIPELINE_NAME="customer_complaint"
$ export PROJECT_DIR=$PWD/$PIPELINE_NAME
$ tfx template copy --pipeline-name=$PIPELINE_NAME \
                    --destination-path=$PROJECT_DIR \
                    --model=taxi
```

When the copy command completes its execution, you can find a folder structure, as seen in the following bash output:

```
$ tree .
.
├── __init__.py
├── beam_dag_runner.py
├── data
│   └── data.csv
├── data_validation.ipynb
├── kubeflow_dag_runner.py
├── model_analysis.ipynb
├── models
│   ├── __init__.py
│   ├── features.py
│   ├── features_test.py
│   ├── keras
│   │   ├── __init__.py
│   │   ├── constants.py
│   │   ├── model.py
│   │   └── model_test.py
│   ├── preprocessing.py
│   └── preprocessing_test.py
├── pipeline
│   ├── __init__.py
│   ├── configs.py
│   └── pipeline.py
└── template_pipeline_test.tar.gz
```

We have taken the *taxi cab* template[1] and tuned our book example project to match the template. The results can be found in the book's GitHub repository. If you want to follow along with this example, please copy the CSV file *consumer_complaints.csv* into the folder:

```
$pwd/$PIPELINE_NAME/data
```

Also, double check the file *pipelines/config.py*, which defines the GCS bucket and other pipeline details. Update the GCS bucket path with a bucket you created or use the GCS buckets that were created when you created the Kubeflow Pipelines installation through GCP's AI Platform. You can find the path with the following command:

```
$ gsutil -l
```

Publishing Your Pipeline with TFX CLI

We can publish the TFX pipeline, which we created based on the TFX template, to our Kubeflow Pipelines application. To access our Kubeflow Pipelines setup, we need to define our GCP Project, a path for our TFX container image and the URL of our Kubeflow Pipelines endpoint. In "Accessing Your Kubeflow Pipelines Installation", we discussed how to obtain the endpoint URL. Before publishing our pipeline with TFX CLI, let's set up the required environment variables for our example:

```
$ export PIPELINE_NAME="<pipeline name>"
$ export PROJECT_ID="<your gcp project id>"
$ export CUSTOM_TFX_IMAGE=gcr.io/$PROJECT_ID/tfx-pipeline
$ export ENDPOINT="<id>-dot-<region>.pipelines.googleusercontent.com"
```

With the details defined, we can now create the pipeline through the TFX CLI with the following command:

```
$ tfx pipeline create --pipeline-path=kubeflow_dag_runner.py \
                      --endpoint=$ENDPOINT \
                      --build-target-image=$CUSTOM_TFX_IMAGE
```

The `tfx pipeline create` command performs a variety of things. With the assistance of Skaffold, it creates a default docker image and publishes the container image via the Google Cloud Registry. It also runs the Kubeflow Runner, as we discussed in Chapter 12, and uploads the Argo configuration to the pipeline endpoint. After the command completes the execution, you will find two new files in the template folder structure: *Dockerfile* and *build.yaml*.

The *Dockerfile* contains an image definition similar to the *Dockerfile* we discussed in "Custom TFX Images". The *build.yaml* file configures Skaffold and sets the docker images registry details and tag policy.

You will be able to see the pipeline now registered in your Kubeflow Pipelines UI. You can start a pipeline run with the following command:

```
$ tfx run create --pipeline-name=$PIPELINE_NAME \
                 --endpoint=$ENDPOINT

Creating a run for pipeline: customer_complaint_tfx
Detected Kubeflow.
Use --engine flag if you intend to use a different orchestrator.
Run created for pipeline: customer_complaint_tfx
+-----------------------+----------+----------+---------------------------+
| pipeline_name         | run_id   | status   | created_at                |
+=======================+==========+==========+===========================+
| customer_complaint_tfx | <run-id> |          | 2020-05-31T21:30:03+00:00 |
+-----------------------+----------+----------+---------------------------+
```

You can check on the status of the pipeline run with the following command:

```
$ tfx run status --pipeline-name=$PIPELINE_NAME \
                 --endpoint=$ENDPOINT \
                 --run_id <run_id>

Listing all runs of pipeline: customer_complaint_tfx
+-----------------------+----------+------------+---------------------------+
| pipeline_name         | run_id   | status     | created_at                |
+=======================+==========+============+===========================+
```

```
| customer_complaint_tfx | <run-id> | Running    | 2020-05-31T21:30:03+00:00 |
+------------------------+----------+------------+---------------------------+
```

A list of all runs for a given pipeline can be obtained with the following command:

```
$ tfx run list --pipeline-name=$PIPELINE_NAME \
               --endpoint=$ENDPOINT

Listing all runs of pipeline: customer_complaint_tfx
+------------------------+----------+------------+---------------------------+
| pipeline_name          | run_id   | status     | created_at                |
+========================+==========+============+===========================+
| customer_complaint_tfx | <run-id> | Running    | 2020-05-31T21:30:03+00:00 |
+------------------------+----------+------------+---------------------------+
```

STOP AND DELETE PIPELINE RUNS

You can stop a pipeline run with `tfx run terminate`. Pipeline runs can be deleted with `tfx run delete`.

TFX CLI is a very useful tool in the TFX toolchain. It supports not only Kubeflow Pipelines but also Apache Airflow and Apache Beam orchestrators.

1 At the time of writing, this was the only template available.