

Chapter 11. Pipelines Part 1: Apache Beam and Apache Airflow

In the previous chapters, we introduced all the necessary components to build a machine learning pipeline using TFX. In this chapter, we will put all the components together and show how to run the full pipeline with two orchestrators: Apache Beam and Apache Airflow. In [Chapter 12](#), we will also show how to run the pipeline with Kubeflow Pipelines. All of these tools follow similar principles, but we will show how the details differ and provide example code for each.

As we discussed in [Chapter 1](#), the pipeline orchestration tool is vital to abstract the glue code that we would otherwise need to write to automate a machine learning pipeline. As shown in [Figure 11-1](#), the pipeline orchestrators sit underneath the components we have already 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 all this code already exists in the form of these orchestrators!

Figure 11-1. Pipeline orchestrators

We will start this chapter by discussing the use cases for the different tools. Then, we will walk through some common code that is required to move from an interactive pipeline to one that can be orchestrated by these tools. Apache Beam and Apache Airflow are simpler to set up than Kubeflow Pipelines, so we will discuss them in this chapter before moving on to the more powerful Kubeflow Pipelines in [Chapter 12](#).

Which Orchestration Tool to Choose?

In this chapter and in [Chapter 12](#), we discuss three orchestration tools that you could use to run your pipelines: Apache Beam, Apache Airflow, and Kubeflow Pipelines. You need to pick only one of them to run each pipeline. Before we take a deep dive into how to use all of these tools, we

will describe some of the benefits and drawbacks to each of them, which will help you decide what is best for your needs.

Apache Beam

If you are 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 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 Beam as an intermediate step to ensure your pipeline runs correctly before moving to Airflow or 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 Apache Airflow and Kubeflow Pipelines shine.

Apache Airflow

Apache Airflow is often already used in companies for data-loading tasks. Expanding an existing Apache Airflow setup to run your pipeline means you would not need to learn a new tool such as Kubeflow.

If you use Apache Airflow in combination with a production-ready database like PostgreSQL, you can take advantage of executing partial pipelines. This can save a significant amount of time if a time-consuming pipeline fails and you want to avoid rerunning all the previous pipeline steps.

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 is more complicated than the Airflow installation, it opens up a variety of new opportunities, including the ability to view TFDV and TFMA visualizations, model lineage, and the artifact collections.

Kubernetes is also an excellent infrastructure platform to deploy machine learning models. Inference routing through the Kubernetes tool Istio is currently state of the art in the field of machine learning infrastructure.

You can set up a Kubernetes cluster with a variety of cloud providers, so you are not 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 up and down the nodes of your cluster.

Kubeflow Pipelines on AI Platform

It's also possible to run Kubeflow Pipelines on Google's AI Platform, which is part of GCP. This takes care of much of the infrastructure for you and makes it easy to load data from Google Cloud Storage buckets. Also, the integration of Google's Dataflow simplifies the scaling of your pipelines. However, this locks you into one single cloud provider.

If you decide to go with Apache Beam or Airflow, this chapter has the information you will need. If you choose Kubeflow (either via Kubernetes or on Google Cloud's AI Platform), you will only need to read the next section of this chapter. This will show you how to convert your interactive pipeline to a script, and then you can head over to [Chapter 12](#) afterward.

Converting Your Interactive TFX Pipeline to a Production Pipeline

Up to this point, our examples have shown how to run all the different components of a TFX pipeline in a notebook-style environment, or *interactive context*. To run the pipeline in a notebook, each component needs to be triggered manually when the previous one has finished. In order to automate our pipelines, we will need to write a Python script that will run all these components without any input from us.

Fortunately, we already have all the pieces of this script. We'll summarize all the pipeline components that we have discussed so far:

Ingests the new data from the data source we wish to use

([Chapter 3](#))

StatisticsGen

Calculates the summary statistics of the new data ([Chapter 4](#))

SchemaGen

Defines the expected features for the model, as well as their types and ranges ([Chapter 4](#))

ExampleValidator

Checks the data against the schema and flags any anomalies ([Chapter 4](#))

Transform

Preprocesses the data into the correct numerical representation that the model is expecting ([Chapter 5](#))

Trainer

Trains the model on the new data ([Chapter 6](#))

Resolver

Checks for the presence of a previously blessed model and returns it for comparison ([Chapter 7](#))

Evaluator

Evaluates the model's performance on an evaluation dataset and validates the model if it is an improvement on the previous version ([Chapter 7](#))

Pusher

Pushes the model to a serving directory if it passes the validation step ([Chapter 7](#))

The full pipeline is shown in [Example 11-1](#).

Example 11-1. The base pipeline

```

import tensorflow_model_analysis as tfma
from tfx.components import (CsvExampleGen, Evaluator, ExampleValidator, Pusher,
                             ResolverNode, SchemaGen, StatisticsGen, Trainer,
                             Transform)

from tfx.components.base import executor_spec
from tfx.components.trainer.executor import GenericExecutor
from tfx.dsl.experimental import latest_blessed_model_resolver
from tfx.proto import pusher_pb2, trainer_pb2
from tfx.types import Channel
from tfx.types.standard_artifacts import Model, ModelBlessing
from tfx.utils.dsl_utils import external_input

def init_components(data_dir, module_file, serving_model_dir,
                    training_steps=2000, eval_steps=200):

    examples = external_input(data_dir)
    example_gen = CsvExampleGen(...)
    statistics_gen = StatisticsGen(...)
    schema_gen = SchemaGen(...)
    example_validator = ExampleValidator(...)
    transform = Transform(...)
    trainer = Trainer(...)
    model_resolver = ResolverNode(...)
    eval_config=tfma.EvalConfig(...)
    evaluator = Evaluator(...)
    pusher = Pusher(...)

    components = [
        example_gen,
        statistics_gen,
        schema_gen,
        example_validator,
        transform,
        trainer,
        model_resolver,
        evaluator,
        pusher
    ]
    return components

```

In our example project, we have split the component instantiation from the pipeline configuration to focus on the pipeline setup for the different orchestrators.

The `init_components` function instantiates the components. It requires three inputs in addition to the number of training steps and evaluation steps:

data_dir

Path where the training/eval data can be found.

module_file

Python module required by the `Transform` and `Trainer` components. These are described in Chapters [5](#) and [6](#), respectively.

serving_model_dir

Path where the exported model should be stored.

Besides the small tweaks to the Google Cloud setup we will discuss in [Chapter 12](#), the component setup will be identical for each orchestrator platform. Therefore, we'll reuse the component definition across the different example setups for Apache Beam, Apache Airflow, and Kubeflow Pipelines. If you would like to use Kubeflow Pipelines, you may find Beam is useful for debugging your pipeline. But if you would like to jump straight in to Kubeflow Pipelines, turn to the next chapter!

Simple Interactive Pipeline Conversion for Beam and Airflow

If you would like to orchestrate your pipeline using Apache Beam or Airflow, you can also convert a notebook to a pipeline via the following steps. For any cells in your notebook that you don't want to export, use the `%%skip_for_export` Jupyter magic command at the start of each cell.

First, set the pipeline name and the orchestration tool:

```
runner_type = 'beam' ❶  
pipeline_name = 'consumer_complaints_beam'
```

❶ Alternatively, `airflow`.

Then, set all the relevant file paths:

```
notebook_file = os.path.join(os.getcwd(), notebook_filename)

# Pipeline inputs
data_dir = os.path.join(pipeline_dir, 'data')
module_file = os.path.join(pipeline_dir, 'components', 'module.py')
requirement_file = os.path.join(pipeline_dir, 'requirements.txt')

# Pipeline outputs
output_base = os.path.join(pipeline_dir, 'output', pipeline_name)
serving_model_dir = os.path.join(output_base, pipeline_name)
pipeline_root = os.path.join(output_base, 'pipeline_root')
metadata_path = os.path.join(pipeline_root, 'metadata.sqlite')
```

Next, list the components you wish to include in your pipeline:

```
components = [
    example_gen, statistics_gen, schema_gen, example_validator,
    transform, trainer, evaluator, pusher
]
```

And export the pipeline:

```
pipeline_export_file = 'consumer_complaints_beam_export.py'
context.export_to_pipeline(notebook_file path=_notebook_file,
                           export_file path=pipeline_export_file,
                           runner_type=runner_type)
```

This export command will generate a script that can be run using Beam or Airflow, depending on the `runner_type` you choose.

Introduction to Apache Beam

Because Apache Beam is running behind the scenes in many TFX components, we introduced it in [Chapter 2](#). Various TFX components (e.g., TFDV or TensorFlow Transform) use Apache Beam for the abstraction of the internal data processing. But many of the same Beam functions can also

be used to run your pipeline. In the next section, we'll show you how to orchestrate our example project using Beam.

Orchestrating TFX Pipelines with Apache Beam

Apache Beam is already installed as a dependency of TFX, and this makes it very easy to start using it as our pipeline orchestration tool. Beam is very simple and does not have all the functionality of Airflow or Kubeflow Pipelines, like graph visualizations, scheduled executions, etc.

Beam can also be a good way to debug your machine learning pipeline. By using Beam during your pipeline debugging and then moving to Airflow or Kubeflow Pipelines, you can rule out root causes of pipeline errors coming from the more complex Airflow or Kubeflow Pipelines setups.

In this section, we will run through how to set up and execute our example TFX pipeline with Beam. We introduced the `Beam Pipeline` function in [Chapter 2](#). This is what we'll use together with our [Example 11-1](#) script to run the pipeline. We will define a `Beam Pipeline` that accepts the TFX pipeline components as an argument and also connects to the SQLite database holding the ML MetadataStore:

```
import absl
from tfx.orchestration import metadata, pipeline

def init_beam_pipeline(components, pipeline_root, direct_num_workers):

    absl.logging.info("Pipeline root set to: {}".format(pipeline_root))
    beam_arg = [
        "--direct_num_workers={}".format(direct_num_workers), ❶
        "--requirements_file={}".format(requirement_file)
    ]

    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_config(metadata_path),
        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 This is where 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 from [Example 11-1](#), initialize the pipeline as earlier, and run the pipeline using

`BeamDagRunner().run(pipeline)` :

```

from tfx.orchestration.beam.beam_dag_runner import BeamDagRunner

components = init_components(data_dir, module_file, serving_model_dir,
                             training_steps=100, eval_steps=100)
pipeline = init_beam_pipeline(components, pipeline_root, direct_num_workers)
BeamDagRunner().run(pipeline)

```

This is a minimal setup that you can easily integrate with the rest of your infrastructure or schedule using a cron job. You can also scale up this pipeline using [Apache Flink](#) or Spark. An example using Flink is briefly described in [this TFX example](#).

In the next section, we will move on to Apache Airflow, which offers many extra features when we use it to orchestrate our pipelines.

Introduction to Apache Airflow

Airflow is Apache's project for workflow automation. The project was initiated in 2016, and it has gained significant attention from large corporations and the general data science community since then. In December 2018, the project "graduated" from the Apache Incubator and became its own [Apache project](#).

Apache Airflow lets you represent workflow tasks through DAGs represented via Python code. Also, Airflow lets you schedule and monitor workflows. This makes it an ideal orchestration tool for our TFX pipelines.

In this section, we'll go through the basics of setting up Airflow. Then, we'll show how we can use it to run our example project.

Installation and Initial Setup

The basic setup of Apache Airflow is straightforward. If you are using Mac or Linux, define the location for the Airflow data with this command:

```
$ export AIRFLOW_HOME=~/.airflow
```

Once the main data folder for Airflow is defined, you can install Airflow:

```
$ pip install apache-airflow
```

Airflow can be installed with a variety of dependencies. At the time of writing, the list of extensions is PostgreSQL support, Dask, Celery, and Kubernetes.

A complete list of Airflow extensions and how to install them can be found in the [Airflow documentation](#).

With Airflow now installed, you need to create an initial database where all the task status information will be stored. Airflow provides you a command to initialize the Airflow database:

```
$ airflow initdb
```

If you use Airflow out of the box and haven't changed any configurations, Airflow will instantiate an SQLite database. This setup works to execute demo projects and to run smaller workflows. If you want to scale your workflow with Apache Airflow, we highly recommend a deep dive into [the documentation](#).

A minimal Airflow setup consists of the Airflow scheduler, which coordinates the tasks and task dependencies, as well as a web server, which provides a UI to start, stop, and monitor the tasks.

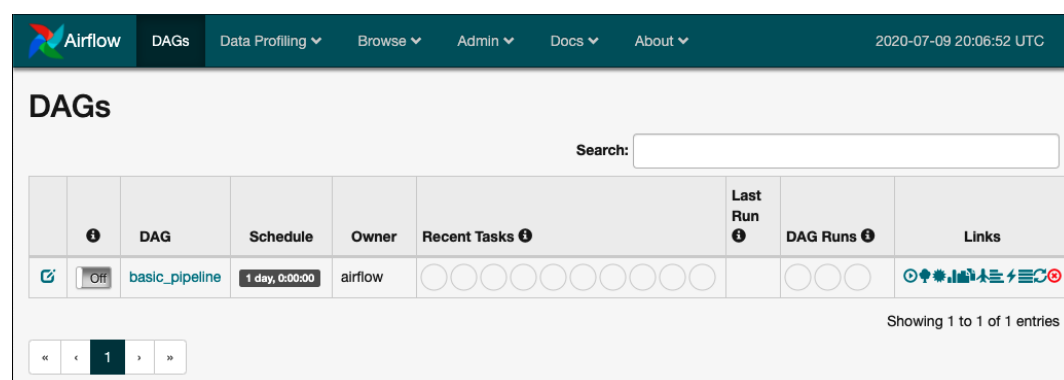
Start the scheduler with the following command:

```
$ airflow scheduler
```

In a different terminal window, start the Airflow web server with this command:

```
$ airflow webserver -p 8081
```

The command argument `-p` sets the port where your web browser can access the Airflow interface. When everything is working, go to <http://127.0.0.1:8081> and you should see the interface shown in [Figure 11-2](#).



The screenshot shows the Apache Airflow web interface. At the top is a navigation bar with the Airflow logo and links for DAGs, Data Profiling, Browse, Admin, Docs, and About. The current date and time are 2020-07-09 20:06:52 UTC. Below the navigation bar is the 'DAGs' section, which includes a search bar. A table lists the DAGs, with one entry visible: 'basic_pipeline'. The table columns are DAG, Schedule, Owner, Recent Tasks, Last Run, DAG Runs, and Links. The 'basic_pipeline' entry shows a schedule of '1 day, 0:00:00', owner 'airflow', and a series of empty circles for recent tasks. The 'DAG Runs' column shows three empty circles. The 'Links' column contains icons for various actions. At the bottom, it says 'Showing 1 to 1 of 1 entries' and a pagination bar with '1' selected.

	DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	basic_pipeline	1 day, 0:00:00	airflow				

Figure 11-2. Apache Airflow UI

The default settings of Airflow can be overwritten by changing the relevant parameters in the Airflow configuration. If you store your graph definitions in a different place than `~/airflow/dags`, you may want to overwrite the default configuration by defining the new locations of the pipeline graphs in `~/airflow/airflow.cfg`.

Basic Airflow Example

With the Airflow installation in place, let's take a look at how to set up a basic Airflow pipeline. In this example, we won't include any TFX components.

Workflow pipelines are defined as Python scripts, and Airflow expects the DAG definitions to be located in `~/airflow/dags`. A basic pipeline consists of project-specific Airflow configurations, task definitions, and the definition of the task dependencies.

Project-specific configurations

Airflow gives you the option to configure project-specific settings, such as when to retry failed workflows or notifying a specific person if a workflow fails. The list of configuration options is extensive. We recommend you reference the [Airflow documentation](#) for an updated overview.

Your Airflow pipeline definitions start with importing the relevant Python modules and project configurations:

```
from airflow import DAG
from datetime import datetime, timedelta

project_cfg = { ❶
    'owner': 'airflow',
    'email': ['your-email@example.com'],
    'email_on_failure': True,
    'start_date': datetime(2019, 8, 1),
    'retries': 1,
    'retry_delay': timedelta(hours=1),
}
```

```
dag = DAG( ❷
    'basic_pipeline',
    default_args=project_cfg,
    schedule_interval=timedelta(days=❶))
```

❶ Location to define the project configuration.

❷ The DAG object will be picked up by Airflow.

Again, Airflow provides a range of configuration options to set up DAG objects.

Task definitions

Once the DAG object is set up, we can create workflow tasks. Airflow provides task operators that execute tasks in a Bash or Python environment. Other predefined operators let you connect to cloud data storage buckets like GCP Storage or AWS S3.

A very basic example of task definitions is shown in the following:

```
from airflow.operators.python_operator import PythonOperator

def example_task(_id, **kwargs):
    print("task {}".format(_id))
    return "completed task {}".format(_id)

task_1 = PythonOperator(
    task_id='task 1',
    provide_context=True,
    python_callable=example_task,
    op_kwargs={'_id': 1},
    dag=dag,
)

task_2 = PythonOperator(
    task_id='task 2',
    provide_context=True,
    python_callable=example_task,
    op_kwargs={'_id': 2},
    dag=dag,
)
```

In a TFX pipeline, you don't need to define these tasks because the TFX library takes care of it for you. But these examples will help you understand what is going on behind the scenes.

Task dependencies

In our machine learning pipelines, tasks depend on each other. For example, our model training tasks require that data validation is performed before training starts. Airflow gives you a variety of options for declaring these dependencies.

Let's assume that our `task_2` depends on `task_1`. You could define the task dependency as follows:

```
task_1.set_downstream(task_2)
```

Airflow also offers a `bit-shift` operator to denote the task dependencies:

```
task_1 >> task_2 >> task_X
```

In the preceding example, we defined a chain of tasks. Each of the tasks will be executed if the previous task is successfully completed. If a task does not complete successfully, dependent tasks will not be executed and Airflow marks them as skipped.

Again, this will be taken care of by the TFX library in a TFX pipeline.

Putting it all together

After explaining all the individual setup steps, let's put it all together. In your `DAG` folder in your `AIRFLOW_HOME` path, usually at `~/airflow/dags`, create a new file `basic_pipeline.py`:

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta
```

```

project_cfg = {
    'owner': 'airflow',
    'email': ['your-email@example.com'],
    'email_on_failure': True,
    'start_date': datetime(2020, 5, 13),
    'retries': 1,
    'retry_delay': timedelta(hours=1),
}

dag = DAG('basic_pipeline',
          default_args=project_cfg,
          schedule_interval=timedelta(days=1))

def example_task(_id, **kwargs):
    print("Task {}".format(_id))
    return "completed task {}".format(_id)

task_1 = PythonOperator(
    task_id='task_1',
    provide_context=True,
    python_callable=example_task,
    op_kwargs={'_id': 1},
    dag=dag,
)

task_2 = PythonOperator(
    task_id='task_2',
    provide_context=True,
    python_callable=example_task,
    op_kwargs={'_id': 2},
    dag=dag,
)

task_1 >> task_2

```

You can test the pipeline setup by executing this command in your terminal:

```
python ~/airflow/dags/basic_pipeline.py
```

Our `print` statement will be printed to Airflow's log files instead of the terminal. You can find the log file at:

```
~/airflow/logs/NAME OF YOUR PIPELINE/TASK NAME/EXECUTION TIME/
```

If we want to inspect the results of the first task from our basic pipeline, we have to investigate the log file:

```
$ cat ../logs/basic_pipeline/task_1/2019-09-07T19\:36\:18.027474+00\:00/1.log

...
[2019-09-07 19:36:25,165] {logging_mixin.py:95} INFO - Task 1 ❶
[2019-09-07 19:36:25,166] {python_operator.py:114} INFO - Done. Returned value w
    completed task 1
[2019-09-07 19:36:26,112] {logging_mixin.py:95} INFO - [2019-09-07 19:36:26,112]
    {local_task_job.py:105} INFO - Task exited with return code 0
```

- ❶ Our print statement
- ❷ Our return message after a successful completion

To test whether Airflow recognized the new pipeline, you can execute:

```
$ airflow list_dags

-----
DAGS
-----

basic_pipeline
```

This shows that the pipeline was recognized successfully.

Now that you have an understanding of the principles behind an Airflow pipeline, let's put it into practice with our example project.

Orchestrating TFX Pipelines with Apache Airflow

In this section, we will demonstrate how we can orchestrate TFX pipelines with Airflow. This lets us use features such as Airflow's UI and its scheduling capabilities, which are very helpful in a production setup.

Pipeline Setup

Setting up a TFX pipeline with Airflow is very similar to the `BeamDagRunner` setup for Beam, except that we have to configure more settings for the Airflow use case.

Instead of importing the `BeamDagRunner`, we will use the `AirflowDAGRunner`. The runner takes an additional argument, which is the configurations of Apache Airflow (the same configurations that we discussed in [“Project-specific configurations”](#)). The `AirflowDagRunner` takes care of all the task definitions and dependencies that we described previously so that we can focus on our pipeline.

As we discussed earlier, the files for an Airflow pipeline need to be located in the `~/airflow/dags` folder. We also discussed some common configurations for Airflow, such as scheduling. We provide these for our pipeline:

```
airflow_config = {
    'schedule_interval': None,
    'start_date': datetime.datetime(2020, 4, 17),
    'pipeline_name': 'your_ml_pipeline',
}
```

Similar to the Beam example, we initialize the components and define the number of workers:

```
from tfx.orchestration import metadata, pipeline

def init_pipeline(components, pipeline_root:Text,
                  direct_num_workers:int) -> pipeline.Pipeline:

    beam_arg = [
        "--direct_num_workers={}".format(direct_num_workers),
    ]
    p = pipeline.Pipeline(pipeline_name=pipeline_name,
                          pipeline_root=pipeline_root,
                          components=components,
                          enable_cache=True,
                          metadata_connection_config=metadata.
                          sqlite_metadata_connection_config(metadata_path),
```

```

        beam_pipeline_args=beam_arg)

    return p

```

Then, we initialize the pipeline and execute it:

```

from tfx.orchestration.airflow.airflow_dag_runner import AirflowDagRunner
from tfx.orchestration.airflow.airflow_dag_runner import AirflowPipelineConfig
from base_pipeline import init_components

components = init_components(data_dir, module_file, serving_model_dir,
                             training_steps=100, eval_steps=100)
pipeline = init_pipeline(components, pipeline_root, 0)
DAG = AirflowDagRunner(AirflowPipelineConfig(airflow_config)).run(pipeline)

```

Again, this code is very similar to the code for the Apache Beam pipeline, but instead of `BeamDagRunner`, we use `AirflowDagRunner` and `AirflowPipelineConfig`. We initialize the components using [Example 11-1](#), and then Airflow looks for a variable named `DAG`.

In this [book's GitHub repo](#), we provide a Docker container that allows you to easily try out the example pipeline using Airflow. It sets up the Airflow web server and scheduler, and moves the files to the correct locations. You can also learn more about Docker in [Appendix A](#).

Pipeline Execution

As we discussed earlier, once we have started our Airflow web server, we can open the UI at the port we define. The view should look very similar to [Figure 11-3](#). To run a pipeline, we need to turn the pipeline on and then trigger it using the Trigger DAG button, indicated by the Play icon.

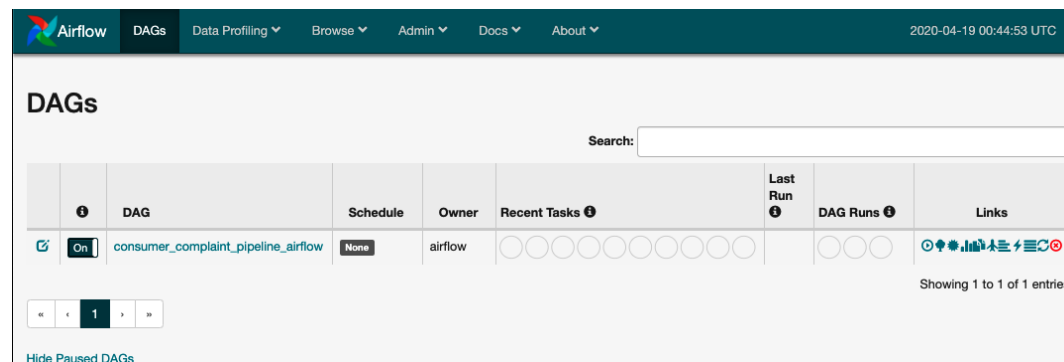


Figure 11-3. Turning on a DAG in Airflow

The graph view in the web server UI ([Figure 11-4](#)) is useful to see the dependencies of the components and the progress of the pipeline execution.

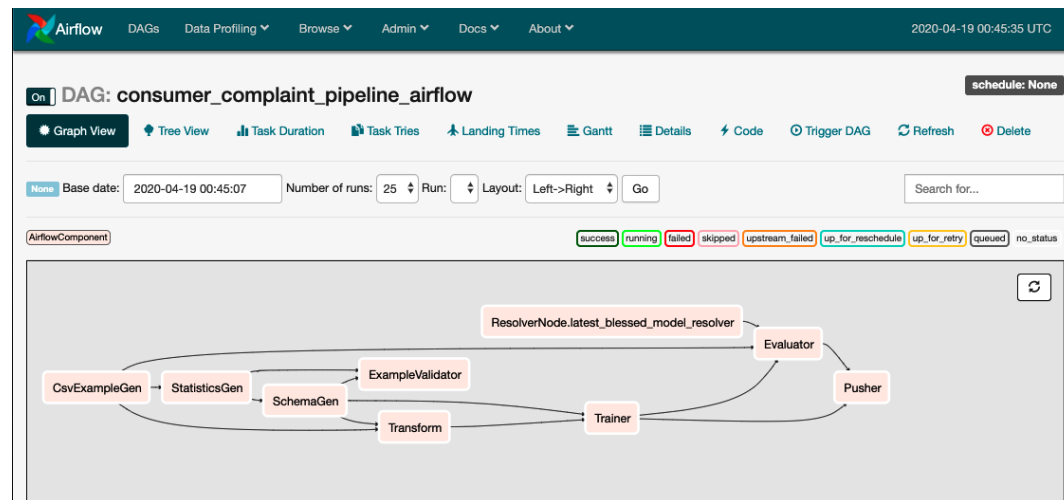


Figure 11-4. Airflow graph view

You will need to refresh the browser page to see the updated progress. As the components finish, they will acquire a green box around the edge, as shown in [Figure 11-5](#). You can view the logs from each component by clicking on them.

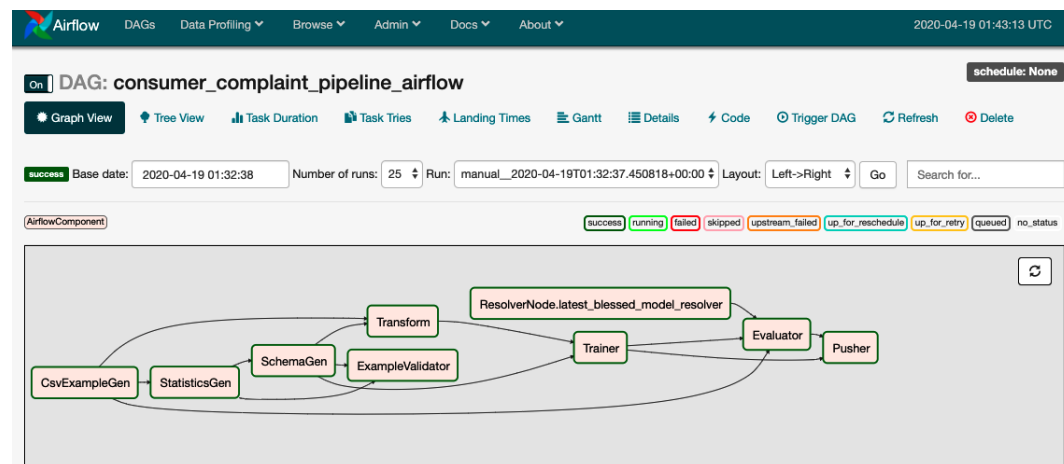


Figure 11-5. Completed pipeline in Airflow

Orchestrating pipelines with Airflow is a good choice if you want a fairly lightweight setup that includes a UI or if your company is already using Airflow. But if your company is already running Kubernetes clusters, the next chapter describes Kubeflow Pipelines, a much better orchestration tool for this situation.

Summary

In this chapter, we discussed the different options for orchestrating your machine learning pipelines. You need to choose the tool that best suits your setup and your use case. We demonstrated how to use Apache Beam to run a pipeline, then introduced Airflow and its principles, and finally showed how to run the complete pipeline with Airflow.

In the next chapter, we will show how to run pipelines using Kubeflow Pipelines and Google's AI Platform. If these do not fit your use case, you can skip straight to [Chapter 13](#) where we will show you how to turn your pipeline into a cycle using feedback loops.

[Support](#) [Sign Out](#)

©2022 O'REILLY MEDIA, INC. [TERMS OF SERVICE](#) [PRIVACY POLICY](#)