Chapter 19. Advanced TFX

In the preceding chapter, we showed you how to orchestrate your ML pipelines using standard TFX components. In this chapter, we'll introduce advanced concepts of ML pipelines and show you how to extend your portfolio of components by quickly writing your own custom components. We will also show you different ways of writing your own components and explain when to use which option.

Advanced Pipeline Practices

In this section, we will discuss additional concepts to advance your pipeline setups. So far, all the pipeline concepts we've discussed comprised linear graphs with one entry and one exit point. In the preceding chapter, we discussed the fundamentals of directed acyclic graphs (DAGs). As long as our pipeline graph is directed and doesn't create any circular connections, we can be creative with our setup. In the following subsections, we will highlight a few concepts to increase the productivity of pipelines.

WARNING

Some of the concepts in this chapter are part of the *v1* TFX API, but they're still in an experimental stage. That means the specific API is still subject to change, though that is highly unlikely at this stage.

Configure Your Components

Sometimes you'll need to set up a component of the same type twice. For example, you might do this if you want to evaluate your model twice. TFX will complain that it already is set up with an Evaluator component. In those cases, we highly recommend giving your components custom identifiers. You can simply do this by calling:

```
evaluator = Evaluator(…).with_id("My Very Special
```

By assigning the custom identifier, TFX will assign the name to the component and use it while the graph is being built.

If you are running your TFX pipelines on Vertex or Kubeflow Pipelines, you can also specify resources per component. That is very handy, but only in cases when a single component requires lots of resources and when the remaining pipeline is not very resource intensive. For those cases, TFX provides the component method `with_platform_config` to specify the resources for the specific component.

In the following example, we request five CPU cores and 10 GB of memory for the component execution:

```
from kfp.pipeline_spec import pipeline_spec_pb2 a
…
```

```
evaluator = Evaluator(…).with_platform_config(
    pipeline_pb2.PipelineDeploymentConfig.Pipeli
.ResourceSpec(cpu_limit=5.0, memory_limit=10.0))
```

You can even assign GPUs as resources:

```
accelerator = \
pipeline_spec_pb2.PipelineDeploymentConfig.Pipel
.AcceleratorConfig(
    count=1, type="NVIDIA_TESLA_V100"
)
platform_config = \
    pipeline_spec_pb2.PipelineDeploymentConfig.P
    .ResourceSpec(
        cpu_limit=5.0, memory_limit=10.0, accele
)
evaluator = Evaluator(…).with_platform_config(pla
```

The component execution will fail if the resource requests can't be met.

Import Artifacts

While most artifacts are produced by components, you can bring your ready-made files into a TFX pipeline by importing them. `Importer` is a

system node that creates an `Artifact` from the remotely located payload directory as a desired artifact type:

```
  hparams_importer = tfx.dsl.Importer(
    source_uri='...',
    artifact_type=HyperParameters,
    custom_properties={
         'version': ‘new’,
     },
    properties={
         "test_property": "property_content",

     },
            ).with_id('hparams_importer')

            trainer = Trainer(
    ...,
    hyperparameters=hparams_importer.outputs['resu
            )
```

As `Importer` is a node of the pipeline, it should be included in the `Pipeline(components=[...])` when creating a `Pipeline` instance.

In the preceding example, `properties` and `custom_properties` attributes instruct the Importer to attach specified information to the created

Artifacts (`properties` is used for those properties declared for the artifact type being imported, while `custom_properties` is used for information that is not artifact type dependent). When an Artifact is created using an Importer node, subsequent components can access it from the Importer's `outputs` dictionary using the result key by default. The outputs key can be customized via the `Importer` 's `output_key` attribute.

The definitions of `properties` and `custom_properties` are as follows:

`properties`

A dictionary of properties for the imported `Artifact` . These properties should be ones declared for the given `artifact_type` .

`custom_properties`

A dictionary of custom properties for the imported `Artifact` . These properties should be of type `Text` or `int` .

Use Resolver Node

The `Resolver` node is a special TFX node that handles special artifact resolution logics that will be used as inputs for downstream nodes. To use `Resolver` , pass the following to the Resolver constructor:

- The name of the `Resolver` instance
- A subclass of `ResolverStrategy`
- Configs that will be used to construct an instance of `ResolverStrategy`
- Channels to resolve with their tag

Here is an example:

```
    example_gen = ImportExampleGen(...)
    examples_resolver = Resolver(
          strategy_class=tfx.dsl.experimental.SpanRa
          config={'range_config': range_config},
          examples=Channel(type=Examples, producer_c
          ).with_id('Resolver.span_resolver')
    trainer = Trainer(
        examples=examples_resolver.outputs['examples
        ...)
```

A resolver strategy defines a type behavior used for input selection, passed as `strategy_class` when initializing the resolver. A `ResolverStrategy` subclass must override the `resolve_artifacts()` function, which takes a `Dict[str, List[Artifact]]` as parameters and returns the resolved `dict` of the same type.

You can find experimental `ResolverStrategy` classes under the `tfx.v1.dsl.experimental` module, including `LatestArtifactStrategy`, `LatestBlessedModelStrategy`, `SpanRangeStrategy`, and so forth. Each strategy specifies a distinct approach to retrieve the artifacts:

`LatestArtifactStrategy`

Queries the latest *n* artifacts in the channel. Number *n* is configured by users on `desired_num_of_artifacts`.

`LatestBlessedModelStrategy`

Identifies the most recent `Model` artifact within the ML Metadata (MLMD) store, and selects the latest blessed model. This strategy is often used to select a blessed model as the baseline for validation.

`SpanRangeStrategy`

Queries the `Examples` artifact within a range of span configurations.

The resolver strategy also allows users to define their custom strategy for artifact selection.

Execute a Conditional Pipeline

TFX allows you to skip components if a condition isn't met. This is a useful feature if you want to skip the execution of the Model Pusher or skip the generation of a Model Card if the model didn't pass the model evaluation.

You can nest the `Pusher` component in a condition block as follows:

```
evaluator = Evaluator(
    examples=example_gen.outputs['examples'],
    model=trainer.outputs['model'],
    eval_config=EvalConfig(...)
)
with Cond(evaluator.outputs['blessing'].future()
          .custom_property('blessed') == 1):
    pusher = Pusher(
        model=trainer.outputs['model'],
        push_destination=PushDestination(...)
)
```

Python provides the context manager `with` that allows us to skip an entire section of the pipeline if an artifact attribute doesn't meet the condition (e.g., is not being blessed). You need to call the attribute's `future()` method; that way, the attribute will be evaluated during the pipeline execution.

Export TF Lite Models

Mobile deployments have become an increasingly important platform for ML models. ML pipelines can help with consistent exports for mobile deployments. Very few changes are required for mobile deployment compared to deployment to model servers. This helps keep the mobile and the server models updated consistently and helps the consumers of the model to have a consistent experience across different devices.

NOTE

Because of hardware limitations of mobile and edge devices, TensorFlow Lite (TF Lite) doesn't support all TensorFlow operations. Therefore, not every model can be converted to a TF Lite–compatible model with the default operators. However, you can add additional TensorFlow operators, or even your own custom operators. For more information on which TensorFlow operations are supported, visit the TF Lite website.

In the TensorFlow ecosystem, TF Lite is the solution for mobile deployments. TF Lite is a version of TensorFlow that can be run on edge or mobile devices. After the model training, we can export the model to TF Lite through the `rewrite_saved_model` operation:

```
from tfx.components.trainer.executor import Trai
from tfx.components.trainer.rewriting import conv
from tfx.components.trainer.rewriting import rewr
from tfx.components.trainer.rewriting import rewr
```

```
def run_fn(fn_args: TrainerFnArgs):
    ...
    temp_saving_model_dir = os.path.join(fn_args
    model.save(temp_saving_model_dir,
               save_format='tf',
               signatures=signatures) ❶
    tfrw = rewriter_factory.create_rewriter(
        rewriter_factory.TFLITE_REWRITER,
        name='tflite_rewriter',
        enable_experimental_new_converter=True
    ) ❷
    converters.rewrite_saved_model(temp_saving_mo
                                   fn_args.servi
                                   tfrw,
                                   rewriter.Model
```

❶ Export the model as a saved model.

❷ Instantiate the TF Lite rewriter.

❸ Convert the model to TF Lite format.

Instead of exporting a saved model after the training, we convert the saved model to a TF Lite–compatible model. Our Trainer component then exports and registers the TF Lite model with the metadata store. The downstream components, such as the Evaluator or the Pusher, can then consume the TF Lite–compliant model. The following example shows how we can evaluate

the TF Lite model, which is helpful in detecting whether the model optimizations (e.g., quantization) have led to a degradation of the model's performance:

```
eval_config = tfma.EvalConfig(
    model_specs=[tfma.ModelSpec(label_key='my_lab
    ...
)
evaluator = Evaluator(
    examples=example_gen.outputs['examples'],
    model=trainer_mobile_model.outputs['model'],
    eval_config=eval_config,
    instance_name='tflite_model')
```

With this pipeline setup, we can now produce models for mobile deployment automatically and push them in the artifact stores for model deployment in mobile apps. For example, a Pusher component could ship the produced TF Lite model to a cloud bucket where a mobile developer could pick up the model and deploy it with Google's ML Kit in an iOS or Android mobile app.

TIP

The `rewriter_factory` can also convert TensorFlow models to TensorFlow.js models. This conversion allows the deployment of models to web browsers and Node.js runtime environments. You can use this new functionality by replacing the `rewriter_factory` name with `rewriter_factory.TFJS_REWRITER` and set the `rewriter.ModelType` to `rewriter.ModelType.TFJS_MODEL` in our earlier example.

Warm-Starting Model Training

In some situations, we may not want to start training a model from scratch. Warm starting is the process of beginning our model training from a checkpoint of a previous training run, which is particularly useful if the model is large and training is time-consuming. This may also be useful in situations under the GDPR, the EU privacy law that states that a user of a product can withdraw their consent for the use of their data at any time. By using warm-start training, we can remove only the data belonging to this particular user and fine-tune the model rather than needing to begin training again from scratch.

In a TFX pipeline, warm-start training requires the Resolver component that we introduced in “Use Resolver Node ”. The Resolver picks up the details of the latest trained model and passes them on to the Trainer component:

```
latest_model_resolver = ResolverNode(
    instance_name='latest_model_resolver',
```

```
        resolver_class=latest_artifacts_resolver.Late
        latest_model=Channel(type=Model)
    )
```

The latest model is then passed to the Trainer using the `base_model` argument:

```
    trainer = Trainer(
        module_file=trainer_file,
        transformed_examples=transform.outputs['trans
        custom_executor_spec=executor_spec.ExecutorCl
        schema=schema_gen.outputs['schema'],
        base_model=latest_model_resolver.outputs['lat
        transform_graph=transform.outputs['transform_
        train_args=trainer_pb2.TrainArgs(num_steps=TR
        eval_args=trainer_pb2.EvalArgs(num_steps=EVAL
```

In your code for your Trainer component you can access the `base_model` reference, load the model, and fine-tune the loaded model with the data found in your `train_args`.

Use Exit Handlers

Sometimes it is quite handy to trigger tasks or messages when a pipeline completes. For example, you could send off a Slack message if a pipeline

failed or ask for a human review if it succeeded. The TFX concept to provide this functionality is called *exit handlers*.

TFX provides a function decorator `exit_handler` that triggers any function to be executed after the component finishes into an exit handler. Your exit handler function needs to accept one function argument of `tfx.dsl.components.Parameter[str]` that contains the pipeline status when the exit handler is called:

```
from kfp.pipeline_spec import pipeline_spec_pb2
from tfx import v1 as tfx
from tfx.utils import proto_utils

@tfx.orchestration.experimental.exit_handler
def customer_exit_handler(final_status: tfx.dsl.c
    pipeline_task_status = pipeline_pb2.Pipeline
    proto_utils.json_to_proto(final_status, pipe
    print(pipeline_task_status)
```

The `pipeline_task_status` contains a bunch of useful information. For example, you can access the state of the pipeline, the error message, or the `pipeline_job_resource_name`. You can access the details via the parsed `final_status` as follows:

```
job_id = status["pipelineJobResourceName"].split(
if status["state"] == "SUCCEEDED":
    print(f"Pipeline job *{job_id}* completed suc
```

TFX provides a number of states. However, the exit handler will only provide a subset of states, since it is always called at the end of a pipeline. Notable states are:

- Succeeded
- Canceled
- Failed

All available states can be found in the `PipelineStateEnum.PipelineTaskState` protobuffer definition.

Once you have declared the function with all the functionality you want to execute after the pipeline completes its run, you need to enable the exit handler in your pipeline runner as follows:

```
my_exit_handler = customer_exit_handler(
    final_status=tfx.dsl.experimental.FinalStatus
)
dsl_pipeline = tfx.dsl.Pipeline(...)
runner = tfx.orchestration.experimental.KubeflowV
```

```
runner.set_exit_handler([my_exit_handler])
runner.run(pipeline=dsl_pipeline)
```

Once your pipeline completes its run, whether by completing all components or due to a failure of one component, the exit handler will be triggered and the status of the pipeline will be available to the handler function.

WARNING

The exit handler functionality is currently only available when running TFX in Vertex Pipelines.

Trigger Messages from TFX

An example of an exit handler is the `MessageExitHandler` component. It allows you to send messages to Slack users, but it can easily be extended to handle any message provider (e.g., sending emails or sending text messages via the Twilio API).

The component is part of TFX-Addons, a collection of useful third-party TFX components (for more information, check out "TFX-Addons"). You can install the library of components with the following:

```
$ pip install tfx-addons
```

Once the library is installed, you can instantiate the `MessageExitHandler` and provide a Slack token of the user submitting the message (e.g., a bot) and the ID of the channel where you want to send the message to:

```
  exit_handler = MessageExitHandler(
      final_status=tfx.orchestration.experimental.F
      message_type="slack",
      slack_credentials=json.dumps({
        "slack_token": "YOUR_SLACK_TOKEN",
        "slack_channel_id": "YOUR_SLACK_CHANNEL_ID"
      })
  )
```

TIP

We don't recommend storing credentials in plain text. The `MessageExitHandler` supports the handling of encrypted credentials. However, the user needs to provide a function for decrypting the credentials. You can set the reference to the decryption function as follows:

```
exit_handler = MessageExitHandler(
    final_status=tfx.orchestration.experimental.
                          FinalStatusStr(),
    message_type="slack",
    slack_credentials=json.dumps({
      "slack_token": "YOUR_SLACK_TOKEN",
      "slack_channel_id": "YOUR_SLACK_CHANNEL_ID"
    }),
    decrypt_fn='path.to.your.decrypt.function'
)
```

The rest of the setup follows the generic exit handler setup we discussed in the preceding section:

```
from tfx_addons.message_exit_handler.component ir
...
dsl_pipeline = pipeline.create_pipeline(...)
runner = kubeflow_v2_dag_runner.KubeflowV2DagRunn
exit_handler = MessageExitHandler(...)
runner.set_exit_handler([exit_handler])
runner.run(pipeline=dsl_pipeline)
```

With this additional component, you can easily integrate your TFX pipelines into your Slack setup, or modify it for any other messaging service.

Custom TFX Components: Architecture and Use Cases

In this chapter, we are discussing TFX components, their architecture, and how to write your own custom components. In this section, we give quick overviews of the architecture of TFX components and discuss situations for using custom components.

Architecture of TFX Components

Except for ExampleGen components, all TFX pipeline components read from a channel to get input artifacts from the metadata store. The data is then loaded from the path provided by the metadata store and processed. The output of the component, the processed data, is then written to the metadata store to be provided to the next pipeline components. The generic internals of a component are always:

- Receive the component input.
- Execute an action.
- Store the final result.

In TFX terms, the three internal parts of the component are called the *driver*, *executor*, and *publisher*. The driver handles the querying of the metadata store. The executor performs the actions of the component. And the publisher manages the saving of the output metadata in the MetadataStore component. The driver and the publisher aren't moving any data. Instead, they read and write references from the MetadataStore. Figure 19-1 shows the generic structure of a TFX component.

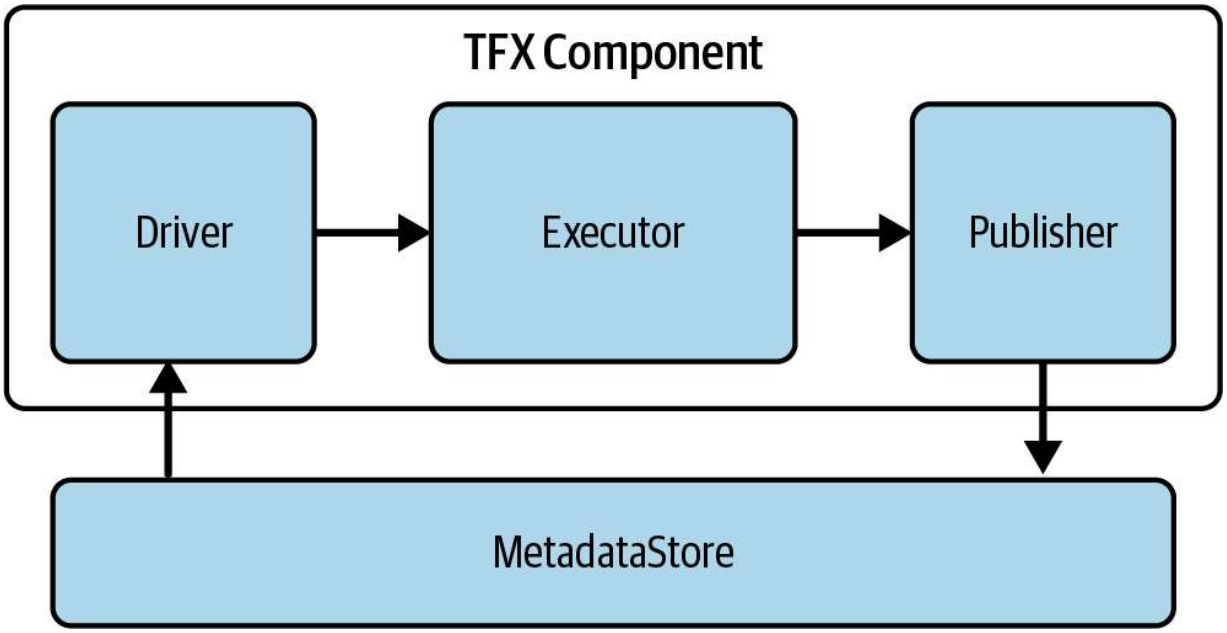

Figure 19-1. TFX component overview

The inputs and outputs of the components are called *artifacts*. Examples of artifacts include raw input data, preprocessed data, and trained models. Each artifact is associated with metadata stored in the MetadataStore. The artifact metadata consists of an artifact type as well as artifact properties. This artifact setup guarantees that the components can exchange data

effectively. TFX currently provides 20 different types of artifacts; however, you can also write custom artifacts if existing components are not fitting your needs.

Use Cases of Custom Components

Custom components could be applied anywhere along your ML pipeline. They give you the flexibility to customize your ML pipelines to your needs. Custom components can be used for actions such as:

- Ingesting data from your custom database
- Sending an email with the generated data statistics to the data science team
- Notifying the DevOps team if a new model was exported
- Kicking off a post-export build process for Docker containers
- Tracking additional information in your ML audit trail

Many production environments and use cases have unique needs, and it’s important to build strong processes that meet those needs. By developing custom components, you can include any tasks, integrations, or processes that you need, and include them in well-defined pipeline flows that follow strong MLOps best practices.

Now let’s look at four ways to write your own custom TFX components.

Using Function-Based Custom Components

The simplest way to implement a TFX component is by using the concept of function-based custom components. Here, we can simply write a Python function and apply it to our pipeline data or model.

You can turn any Python function into a custom TFX component via the following steps:

1. Decorate your Python function with the TFX `@tfx.dsl.components.component` decorator.
2. Add type annotations so that TFX knows which arguments are *inputs*, *outputs*, and *execution parameters*. Note that inputs, outputs, and execution parameters don't need to be "unpacked." You can directly access your artifact attributes.
3. Set the output values through TFX's `set_custom_property` methods; for example, `output_object.set_string_custom_property()`.

For our example of a function-based custom component, we are reusing our function `convert_image_to_TFExample` to do the core of the work. The following example shows the setup of the remaining component:

```
import os
import tensorflow as tf
from tfx import v1 as tfx
from tfx.types.experimental.simple_artifacts impo

@tfx.dsl.components.component
def MyComponent(data: tfx.dsl.components.InputArt
                examples: tfx.dsl.components.Outpu
                ):

  image_files = tf.io.gfile.listdir(data.uri) ❸
  tfrecord_filename = os.path.join(examples.uri,
  options = tf.io.TFRecordOptions(compression_typ
  writer = tf.io.TFRecordWriter(tfrecord_filename

  for image in image_files:
    convert_image_to_TFExample(image, writer, dat
```

❶ TFX provides custom annotation types for function-based components.

❷ TFX requires proper type annotations to understand which argument is the input, output, or a parameter.

❸ Attributes are directly accessible.

The custom component can now be consumed like any other TFX component. Here is an example of how to use the component in the interactive TFX context in Jupyter Notebooks:

```
ingestion = MyComponent()
context.run(ingestion)
```

Writing a Custom Component from Scratch

In the previous sections, we discussed the implementation of Python-based components. While the implementation is fast, it comes with a few constraints. The goal with the option was implementation speed rather than parallelization and reusability. If you want to focus on those goals, we recommend writing a custom TFX component.

In this section, we will develop a custom component for ingesting JPEG images and their labels in a pipeline. You can see the workflow in Figure 19-2. We will load all images from a provided folder and determine the label based on the filename. In our example project, which you can find in Chapter 20, we want to train an ML model to classify cats and dogs. The filenames of our images include the content of the image (e.g., *dog-1.jpeg*) so that we can determine the label from the filename itself. As part of the

custom component, we want to load each image, convert it to tf.Example format, and save all converted images together as TFRecord files for consumption by downstream components.

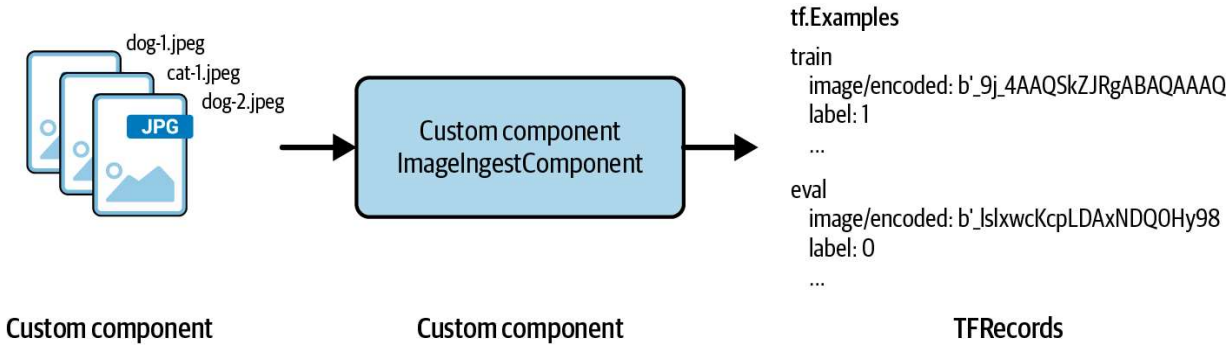

Figure 19-2. Functionality of our demo custom component

We must first define the inputs and outputs of our component as a `ComponentSpec`. Then, we can create our component Executor, which defines how the input data should be processed and how the output data is generated. If the component requires inputs that aren't added in the metadata store, we'll need to write a custom component driver. This is the case when, for example, we want to register an image path in the component and the artifact type hasn't been registered in the metadata store previously.

The parts shown in Figure 19-3 might seem complicated, but we will discuss them each in turn in the following subsections.

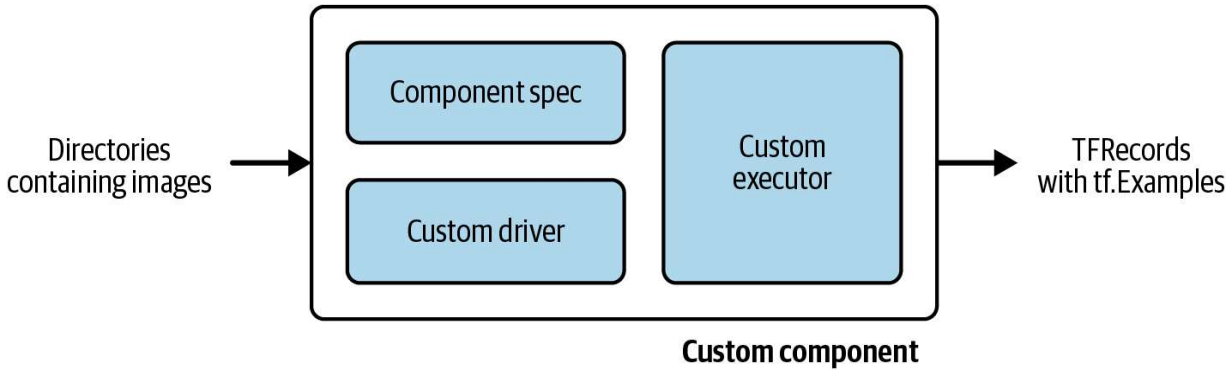

Figure 19-3. Parts of a component from scratch

TIP

If an existing component comes close to meeting your needs, consider forking and reusing it by changing the Executor instead of starting from scratch, as we will discuss in "Reusing Existing Components".

Defining Component Specifications

The component specifications, or `ComponentSpec`, define how components communicate with each other. They describe three important details of each component:

- The component inputs
- The component outputs
- Potential component parameters required during the component execution

Components communicate through *channels*, which are inputs and outputs. These channels have types, as we will see in the following example. The component inputs define the artifacts the component will receive from previously executed components or new artifacts such as filepaths. The component outputs define which artifacts will be written to the metadata store.

The component parameters define options that are required for execution but aren't available in the metadata store, so they are provided when the component is called. This could be the `push_destination` in the case of the Pusher component or the `train_args` in the Trainer component. The following example shows a definition of our component specifications for our image ingestion component:

```
from tfx.types.component_spec import ChannelPara
from tfx.types.component_spec import ExecutionPa
from tfx.types import standard_artifacts
class ImageIngestComponentSpec(types.ComponentSp
    """ComponentSpec for a Custom TFX Image Inge
    PARAMETERS = {
        'name': ExecutionParameter(type=Text),
    }
    INPUTS = {
        'input': ChannelParameter(type=standard_
    }
    OUTPUTS = {
```

```
            'examples': ChannelParameter(type=standa
        }
```

1. Using `ExternalArtifact` to allow new input paths
2. Exporting `Examples`

In our example implementation of `ImageIngestComponentSpec`, we are ingesting an input path through the input argument `input`. The generated TFRecord files with the converted images will be stored in the path passed to the downstream components via the `examples` argument. In addition, we are defining a parameter for the component called `name`.

Defining Component Channels

In our example `ComponentSpec`, we introduced two types of component channels: `ExternalArtifact` and `Examples`. This is a particular pattern used for ingestion components since they are usually the first component in a pipeline and no upstream component is available from which we could have received already-processed `Examples`. If you develop a component further downstream in the pipeline, you would usually want to ingest `Examples`. Therefore, the channel type needs to be `standard_artifacts.Examples`. But we aren't limited to only two types. TFX provides a variety of types. The following is a small list of available types:

- `ExampleStatistics`
- `Model`
- `ModelBlessing`
- `Bytes`
- `String`
- `Integer`
- `Float`

With our `ComponentSpec` now set up, let's take a look at the component executor.

Writing the Custom Executor

The component executor defines the processes inside the component, including how the inputs are used to generate the component outputs. Even though we will write this basic component from scratch, we can rely on TFX classes to inherit function patterns. As part of the `Executor` object, TFX will look for a function called `Do` for the execution details of our component. We will implement our component functionality in this function:

```
from tfx.components.base import base_executor
class Executor(base_executor.BaseExecutor):
    """Executor for Image Ingestion Component."""
    def Do(self, input_dict: Dict[Text, List[type
```

```
            output_dict: Dict[Text, List[types.Ar
            exec_properties: Dict[Text, Any]) -> N
        ...
```

The code snippet shows that the `Do` function of our Executor expects three arguments: `input_dict`, `output_dict`, and `exec_properties`. These Python dictionaries contain the artifact references that we pass to and from the component as well as the execution properties.

TFX expects tf.Example data structures. Therefore, we need to write a function that reads our images, converts the images to a base64 representation, and generates a label. In our case, the images are already sorted by cats or dogs and we can use the filepath to extract the label:

```
def convert_image_to_TFExample(image_filename, t
    image_path = os.path.join(input_base_uri, ima
    lowered_filename = image_path.lower() ❷
    if "dog" in lowered_filename:
        label = 0
    elif "cat" in lowered_filename:
        label = 1
    else:
        raise NotImplementedError("Found unknown
    raw_file = tf.io.read_file(image_path) ❸
    example = tf.train.Example(features=tf.train
```

```
            'image_raw': _bytes_feature(raw_file.nump
            'label': _int64_feature(label)
        }))
        writer.write(example.SerializeToString()) ❺
```

❶ Assemble the complete image path.

❷ Determine the label for each image based on the filepath.

❸ Read the image from a disk.

❹ Create the TensorFlow `Example` data structure.

❺ Write the tf.Example to TFRecord files.

With the completed generic function of reading an image file and storing it in files containing the TFRecord data structures, we can now focus on custom component-specific code.

We want our very basic component to load our images, convert them to tf.Examples, and return two image sets for training and evaluation. For the simplicity of our example, we are hardcoding the number of evaluation examples. In a production-grade component, this parameter should be dynamically set through an execution parameter in the `ComponentSpecs`. The input to our component will be the path to the folder containing all the images. The output of our component will be the

path where we'll store the training and evaluation datasets. The path will contain two subdirectories (*train* and *eval*) that contain the TFRecord files:

```
class ImageIngestExecutor(base_executor.BaseExecu
    def Do(self, input_dict: Dict[Text, List[type
           output_dict: Dict[Text, List[types.Ar
           exec_properties: Dict[Text, Any]) ->
        self._log_startup(input_dict, output_dict
        input_base_uri = artifact_utils.get_singl
        image_files = tf.io.gfile.listdir(input_b
        random.shuffle(image_files)

        for images in splits:
            output_dir = artifact_utils.get_split
                output_dict['examples'], split_na
            tfrecord_filename = os.path.join(outp
            options = tf.io.TFRecordOptions(compr
            writer = tf.io.TFRecordWriter(tfrecor
            for image in images:
                convert_image_to_TFExample(image,
```

❶ Log arguments.

❷ Get the folder path from the artifact.

❸ Obtain all the filenames.

❹ Set the split URI.

❺ Create a TFRecord writer instance with options.

❻ Write an image to a file containing the TFRecord data structures.

Our basic `Do` method receives `input_dict`, `output_dict`, and `exec_properties` as arguments to the method. The first argument contains the artifact references from the metadata store stored as a Python dictionary, the second argument receives the references we want to export from the component, and the last method argument contains additional execution parameters like, in our case, the component name. TFX provides the very useful `artifact_utils` function that lets us process our artifact information. For example, we can use the following code to extract the data input path:

```
  artifact_utils.get_single_uri(input_dict['input'
```

We can also set the name of the output path based on the split name:

```
  artifact_utils.get_split_uri(output_dict['exampl
```

The previous function brings up a good point. For simplicity of the example, we have ignored the options to dynamically set data splits. In fact,

in our example, we are hardcoding the split names and quantity:

```
def get_splits(images: List, num_eval_samples=100
    """ Split the list of image filenames into tr
    train_images = images[num_test_samples:]
    eval_images = images[:num_test_samples]
    splits = [('train', train_images), ('eval', e
    return splits
```

Such functionality wouldn't be desirable for a component in production, but a full-blown implementation would go beyond the scope of this chapter. (In the next section, we will discuss how you can reuse existing component functions and simplify your implementations.)

Writing the Custom Driver

If we would run the component with the executor that we have defined so far, we would encounter a TFX error that the input isn't registered with the metadata store and that we need to execute the previous component before running our custom component. But in our case, we don't have an upstream component, since we are ingesting the data into our pipeline. The data ingestion step is the start of every pipeline. So what is going on?

As we discussed previously, components in TFX communicate with each other via the metadata store, and the components expect that the input

artifacts are already registered in the metadata store. In our case, we want to ingest data from a disk, and we are reading the data for the first time in our pipeline; therefore, the data isn't passed down from a different component, and we need to register the data sources in the metadata store.

NOTE

Normally, TFX components ingest inputs from ExampleGen, including custom ExampleGen components (see "Components of an Orchestrated Workflow"). Therefore, it is extremely rare that you need to implement custom drivers. If you can reuse the input/output architecture of an existing TFX component, you won't need to write a custom driver, and you can skip this step.

Similar to our custom executor, we can reuse a `BaseDriver` class provided by TFX to write a custom driver. We need to overwrite the standard behavior of the component, and we can do that by overriding the `resolve_input_artifacts` method of the `BaseDriver`. A bare-bones driver will *register* our inputs, which is straightforward. We need to *unpack* the channel to obtain the `input_dict`. By looping over all the values of the `input_dict`, we can access each list of inputs. By looping again over each list, we can obtain each input and then register it at the metadata store by passing it to the function `publish_artifacts`. The `publish_artifacts` function will call the metadata store, publish the artifact, and set the state of the artifact as ready to be published:

```
class ImageIngestDriver(base_driver.BaseDriver):
  """Custom driver for ImageIngest."""
  def resolve_input_artifacts(
      self,
      input_channels: Dict[Text, types.Channel],
      exec_properties: Dict[Text, Any],
      driver_args: data_types.DriverArgs,
      pipeline_info: data_types.PipelineInfo) ->
      Dict[Text, List[types.Artifact]]:
    """Overrides BaseDriver.resolve_input_artifac
    del driver_args ❶
    del pipeline_info
    input_dict = channel_utils.unwrap_channel_dic
    for input_list in input_dict.values():
        for single_input in input_list:
            self._metadata_handler.publish_artifa
            absl.logging.debug("Registered input
            absl.logging.debug("single_input.mlmd
                               "{}".format(single
    return input_dict
```

❶ Delete unused arguments.

❷ Unwrap the channel to obtain the input dictionary.

❸ Publish the artifact.

4. Print artifact information.

While we loop over each input, we can print additional information:

```
  print("Registered new input: {}".format(single_i
  print("Artifact URI: {}".format(single_input.uri
  print("MLMD Artifact Info: {}".format(single_inp
```

With the custom driver now in place, we need to assemble our custom component.

Assembling the Custom Component

With our `ImageIngestComponentSpec` defined, the `ImageIngestExecutor` completed, and the `ImageIngestDriver` set up, let's tie it all together in our `ImageIngestComponent`. We could then, for example, load the component in a pipeline that trains image classification models.

To define the actual component, we need to define the specification, executor, and driver classes. We can do this by setting `SPEC_CLASS`, `EXECUTOR_SPEC`, and `DRIVER_CLASS`, as shown in the following example code. As the final step, we need to instantiate our `ComponentSpecs` with the component's arguments (e.g., input and

output examples, and the provided name) and pass it to the instantiated `ImageIngestComponent`.

In the unlikely case that we don't provide an output artifact, we can set our default output artifact to be of type tf.Example, define our hardcoded split names, and set it up as a channel:

```
from tfx.components.base import base_component
from tfx import types
from tfx.types import channel_utils
class ImageIngestComponent(base_component.BaseCo
    """Custom ImageIngestWorld Component."""
    SPEC_CLASS = ImageIngestComponentSpec
    EXECUTOR_SPEC = executor_spec.ExecutorClassSp
    DRIVER_CLASS = ImageIngestDriver
    def __init__(self, input, output_data=None, n
        if not output_data:
            examples_artifact = standard_artifact
            examples_artifact.split_names = \
                artifact_utils.encode_split_names
            output_data = channel_utils.as_channe
        spec = ImageIngestComponentSpec(input=inp
                                        examples=
                                        name=name
        super(ImageIngestComponent, self).__init_
```

By assembling our `ImageIngestComponent`, we have tied together the individual pieces of our basic custom component. In the next section, we'll take a look at how we can execute our basic component.

Using Our Basic Custom Component

After implementing the entire basic component to ingest images and turning these images into TFRecord files, we can use the component like any other component in our pipeline. The following code example shows how. Notice that it looks exactly like the setup of other ingestion components. The only difference is that we need to import our newly created component and then run the initialized component:

```
import os
from tfx.utils.dsl_utils import external_input
from tfx.orchestration.experimental.interactive.i
    InteractiveContext
from image_ingestion_component.component import I
context = InteractiveContext()
image_file_path = "/path/to/files"
examples = external_input(dataimage_file_path_roo
example_gen = ImageIngestComponent(input=examples
                                   name=u'ImageIn
context.run(example_gen)
```

The output from the component can then be consumed by downstream components such as StatisticsGen:

```
from tfx.components import StatisticsGen
statistics_gen = StatisticsGen(examples=example_
context.run(statistics_gen)
context.show(statistics_gen.outputs['statistics'
```

WARNING

The discussed implementation provides only basic functionality and is not production ready. The next two sections cover the missing functionality and updated component for a product-ready implementation.

Implementation Review

In the previous sections, we walked through a basic component implementation. While the component is functioning, it is missing some key functionality (e.g., dynamic split names or split ratios)—and we would expect such functionality from our ingestion component. The basic implementation also required a lot of boilerplate code (e.g., the setup of the component driver). The ingestion of the images in our basic implementation example lacks ingestion efficiency and isn't the most scalable implementation. We can improve the ingestion scalability by using Apache

Beam. To avoid reinventing the wheel, we highly recommend reusing existing components and their Apache Beam support.

In the next section, we will discuss how we could simplify the implementations and adopt the more scalable patterns. By reusing common functionality, such as the component drivers, and reusing existing components, we can speed up implementation and reduce code bugs.

Reusing Existing Components

Instead of writing a component for TFX entirely from scratch, we can inherit an existing component and customize it by overwriting the executor functionality. As shown in Figure 19-4, this is generally the preferred approach when a component is reusing an existing component architecture. In the case of our demo component, the architecture is equivalent with a file base ingestion component (e.g., `CsvExampleGen`). Such components receive a directory path as a component input, load the data from the provided directory, turn the data into tf.Examples, and return the data structures in TFRecord files as output from the component.

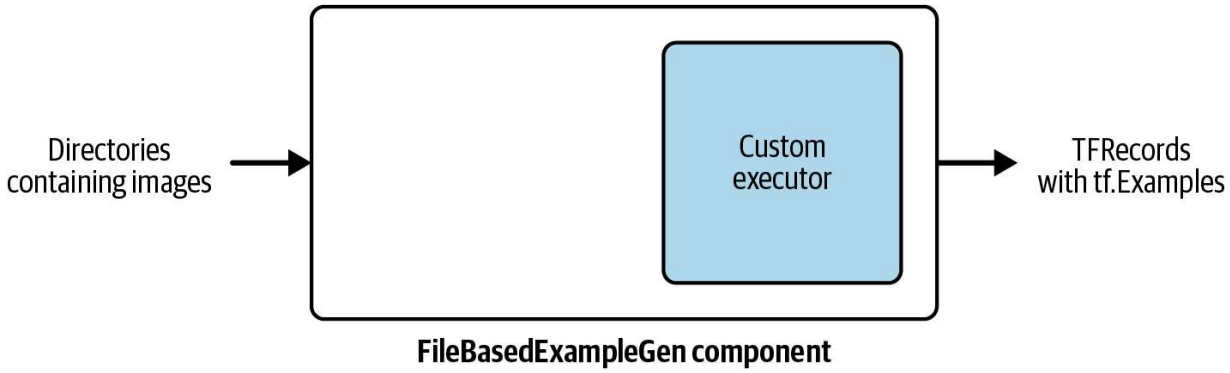

Figure 19-4. Extending existing components

TFX provides the FileBasedExampleGen component for this purpose. Since we are going to reuse an existing component, we can simply focus on developing our custom executor and making it more flexible than our previous basic component.

By reusing an existing component architecture for ingesting data into our pipelines, we can also reuse setups to ingest data efficiently with Apache Beam. TFX and Apache Beam provide classes (e.g., `GetInputSourceToExamplePTransform`) and function decorators (e.g., `@beam.ptransform_fn`) to ingest the data via Apache Beam pipelines. In our example, we use the function decorator `@beam.ptransform_fn`, which allows us to define Apache Beam transformation (`PTransform`). The decorator accepts an Apache Beam pipeline, runs a given transformation (in our case, the loading of the images and their conversion to tf.Examples), and returns the Apache Beam `PCollection` with the transformation results.

The conversion functionality is handled by a function very similar to our previous implementation. The updated conversion implementation has one major difference: we don’t need to instantiate and use a TFRecord writer; instead, we can fully focus on loading images and converting them to tf.Examples. We don’t need to implement any functions to write the tf.Examples to TFRecord data structures, because we did it in our previous implementation. Instead, we return the generated tf.Examples and let the underlying TFX/Apache Beam code handle the writing of the TFRecord files. The following code example shows the updated conversion function:

```
def convert_image_to_TFExample(image_path): ❶
    # Determine the label for each image based or
    lowered_filename = image_path.lower()
    print(lowered_filename)
    if "dog" in lowered_filename:
        label = 0
    elif "cat" in lowered_filename:
        label = 1
    else:
        raise NotImplementedError("Found unknown
    # Read the image.
    raw_file = tf.io.read_file(image_path)
    # Create the TensorFlow Example data structu
    example = tf.train.Example(features=tf.train
        'image_raw': _bytes_feature(raw_file.nump
        'label': _int64_feature(label)
```

```
        }))
        return example ❷
```

❶ Only the filepath is needed.

❷ The function returns examples instead of writing them to a disk.

With the updated conversion function in place, we can now focus on implementing the core executor functionality. Since we are customizing an existing component architecture, we can reuse the same arguments, such as split patterns. Our `image_to_example` function in the following code example takes four input arguments: an Apache Beam pipeline object, an `input_dict` with artifact information, a dictionary with execution properties, and split patterns for ingestion. In the function, we generate a list of available files in the given directories and pass the list of images to an Apache Beam pipeline to convert each image found in the ingestion directories to tf.Examples:

```
@beam.ptransform_fn
def image_to_example(
    pipeline: beam.Pipeline,
    input_dict: Dict[Text, List[types.Artifact]],
    exec_properties: Dict[Text, Any],
    split_pattern: Text) -> beam.pvalue.PCollecti
    input_base_uri = artifact_utils.get_single_ur
    image_pattern = os.path.join(input_base_uri,
```

```
        absl.logging.info(
            "Processing input image data {} "
            "to tf.Example.".format(image_pattern))
        image_files = tf.io.gfile.glob(image_pattern)
        if not image_files:
            raise RuntimeError(
                "Split pattern {} did not match any f
                "".format(image_pattern))
        p_collection = (
            pipeline
            | beam.Create(image_files) ❷
            | 'ConvertImagesToTFRecords' >> beam.Map(
                lambda image: convert_image_to_TFExam
        )
        return p_collection
```

❶ Generate a list of files present in the ingestion paths.

❷ Convert the list to a Beam `PCollection`.

❸ Apply the conversion to every image.

The final step in our custom executor is to overwrite the `GetInputSourceToExamplePTransform` of the `BaseExampleGenExecutor` with our `image_to_example`:

```
class ImageExampleGenExecutor(BaseExampleGenExecu
    @beam.ptransform_fn
    def image_to_example(...):
        ...
    def GetInputSourceToExamplePTransform(self)
        return image_to_example
```

Our custom image ingestion component is now complete!

Since we are reusing an ingestion component and swapping out the processing executor, we can now specify a `custom_executor_spec`. By reusing the FileBasedExampleGen component and overwriting the executor, we can use the entire functionality of ingestion components, like defining the input split patterns or the output train/eval splits. The following code snippet gives a complete example of using our custom component:

```
from tfx.components import FileBasedExampleGen
from tfx.utils.dsl_utils import external_input
from image_ingestion_component.executor import Im
input_config = example_gen_pb2.Input(splits=[
    example_gen_pb2.Input.Split(name='images',
                                pattern='sub-dire
])
output = example_gen_pb2.Output(
    split_config=example_gen_pb2.SplitConfig(spli
        example_gen_pb2.SplitConfig.Split(
```

```
                name='train', hash_buckets=4),
            example_gen_pb2.SplitConfig.Split(
                name='eval', hash_buckets=1)
        ])
    )
    example_gen = FileBasedExampleGen(
        input=external_input("/path/to/images/"),
        input_config=input_config,
        output_config=output,
        custom_executor_spec=executor_spec.ExecutorCl
            ImageExampleGenExecutor)
    )
```

As we have discussed in this section, extending the component executor will always be a simpler and faster implementation than writing a custom component from scratch. Therefore, we recommend this process if you are able to reuse existing component architectures.

TIP

If you would like to see the component in action and follow along with a complete end-to-end example, head over to Chapter 20.

Creating Container-Based Custom

Components

Sometimes you want to reuse tools that can’t be easily integrated in your Python project. For example, if you have a Rust or C++ setup to perform inference testing on your ML model, it would be impractical to integrate the functionality as a function-based custom component. For those cases, TFX provides container-based components.

TFX allows you to express components as entire container images. You can access the functionality by calling the `create_container_component` function.

The `create_container_component` function requires a number of arguments to be set up:

`name`

: This sets the name of your container component (required).

`image`

: This sets the container image (required).

`inputs`

: TFX will pass artifact references to the container during its execution; therefore, TFX expects a dictionary of keys and artifact

types as values (required).

`outputs`

If you would like to pass data to downstream components, you can define output artifacts here. TFX expects the same dictionary as for the inputs.

`parameters`

If you want to pass additional parameters for execution to the container, you can set a dictionary with names and types.

`command`

The container needs a command that will be triggered during the execution. The command can call an entry point script that is available in the container, or you can define your entry point steps directly in the component definition.

WARNING

The container needs to read and write artifacts from outside your container. You need to provide the dependencies, credentials (if needed), and functionality to read artifacts from cloud storage locations.

The *command* can access the artifacts through *placeholders*. The placeholders are evaluated during the runtime of the container. At the time of this writing, TFX supports four different types of placeholders:

`InputValuePlaceholder`

For simple values, you can pass them as value placeholders. They will be passed to the container as strings.

`InputUriPlaceholder`

For more complex data structures, you'll need to store the artifacts in your file storage system and pass the reference as a URI to the container.

`OutputUriPlaceholder`

Similar to `InputUriPlaceholder`, the placeholder is replaced with the URI where the component should store the output artifact's data.

`ConcatPlaceholder`

The placeholder allows you to concatenate different parts; for example, strings with `InputValuePlaceholders`.

Here is an example of how to assemble the container-based component:

```
import tfx.v1 as tfx
list_file_filesystem_component = tfx.dsl.componer
    name=ListFileSystemComponent,
    inputs={
        'path': tfx.standard_artifacts.ExternalA
```

```
        },
        outputs={},
        parameters={},
        image='ubuntu:jammy', ❷
        command=[
            'sh', '-exc',
            '''
              path_value="$1" ❸
            ls "$path_value"
            ''',
            '--path, tfx.dsl.placeholders.InputValueP
        ],
    )
```

❶ Define your inputs, outputs, and parameters.

❷ Use a base image that contains all your dependencies.

❸ You can access the values or URI through the position of the placeholders defined in 4.

❹ Define your placeholder types.

This simple example shows nicely how we use a non-Python-based way of processing data in our pipeline.

Which Custom Component Is Right for You?

In the previous sections, we introduced various options to create custom TFX components for your ML pipelines. You might wonder which option is right for your pipeline. Here are some aspects to consider:

- Function-based components will get you easily up and running. However, those components won’t scale as nicely as Apache Beam–based components will.
- Components written from scratch also can support scalable Apache Beam executions, but they require a larger setup, as demonstrated.
- Reusing existing components often supports the execution on Apache Beam by default. That means your component will scale very well if you change your Apache Beam runner from a DirectRunner to high-throughput setups like Dataflow.
- Container-based components are a good option if you want to integrate non-Python components into your pipeline. However, the setup requires that you manage the artifact download and upload to your storage location outside the container’s filesystem.

TFX-Addons

Most ML problems are repeat problems, and therefore, the TFX community has built a forum to share custom components. As shown in Figure 19-5, the project is called *TFX-Addons*. Through this project, an active community comprising members from companies using TFX, such as Spotify, Twitter, Apple, and Digits, open sources a number of useful TFX components. Check out the project. Maybe your problem has already been solved. If that isn't the case, join the group, participate in monthly calls, and consider making your custom TFX component open source.

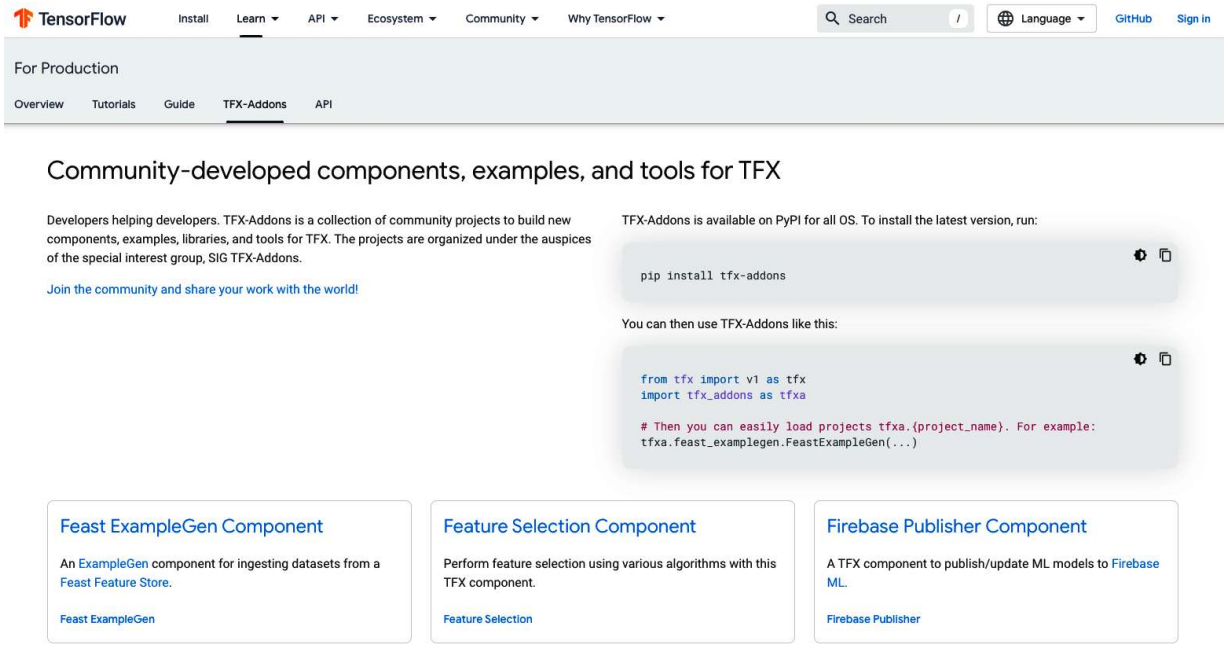

Figure 19-5. The TFX-Addons project

Conclusion

In this chapter, we introduced advanced TFX concepts such as conditional component execution. We also discussed advanced settings for a training setup, such as branching pipeline graphs to produce multiple models from the same pipeline execution. This functionality can be used to produce TF Lite models for deployments in mobile apps. We also discussed warm-starting the training process to continuously train ML models. Warm-starting model training is a great way to shorten the training steps for continuously trained models.

We also showed how writing custom components gives us the flexibility to extend existing TFX components and tailor them for our pipeline needs. Custom components allow us to integrate more steps into our ML pipelines. By adding more components to our pipeline, we can guarantee that all models produced by the pipeline have gone through the same steps. Since the implementation of custom components can be complex, we reviewed a basic implementation of a component from scratch and highlighted an implementation of a new component executor by inheriting existing component functionality.

In the next two chapters, we will take a look at two ML pipelines in depth.