Chapter 15

TFX: MLOps and Deploying Models with TensorFlow

Machine learning at scale requires more than building models; it demands systems that automate data handling, training, validation, and deployment. This discipline, often called MLOps, extends DevOps principles to the machine learning domain. While MLOps automates the journey of data to model delivery, productionization is the destination—placing a trained model into a robust service that can handle real-world use cases. At companies like Google, Facebook, and Amazon, this ensures hundreds of constantly evolving models are retrained and validated automatically as new data streams in, protecting against model staleness and failure.

TensorFlow Extended (TFX) provides the ecosystem to implement such pipelines. A complete example involves predicting the severity of forest fires in Portugal's Montesinho park using weather features. The pipeline begins with CsvExampleGen, which loads the raw CSV dataset and splits it into training and evaluation sets. Next, StatisticsGen computes distributions and visualizations of the features, followed by SchemaGen, which infers data types, bounds, and domains. Finally, Transform converts raw inputs into machine-learning-ready features through scaling, bucketizing, or encoding. Figure 1 shows how the pipeline directories evolve as these steps are executed, with separate folders for training and evaluation splits.

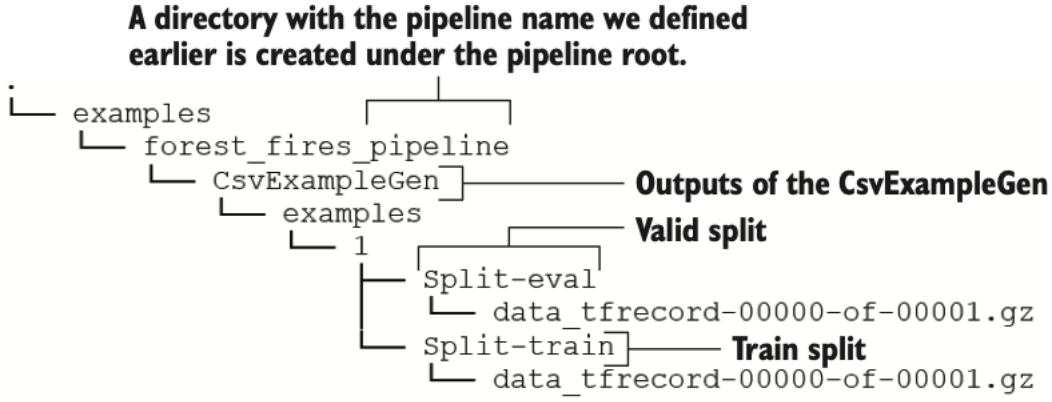

Figure 1 The directory structure after running the CsvExampleGen

Once transformed data is prepared, the Trainer component defines and fits a Keras model. The model architecture combines dense features with categorical embeddings or one-hot vectors, producing a regression network to estimate fire area. TensorFlow Serving requires signatures—special functions annotated with @tf.function—which define how inputs and outputs are exposed through an API. Figure 2 illustrates the interaction: a client sends an HTTP request, the server routes it to the correct signature, the model generates predictions, and results flow back to the client.

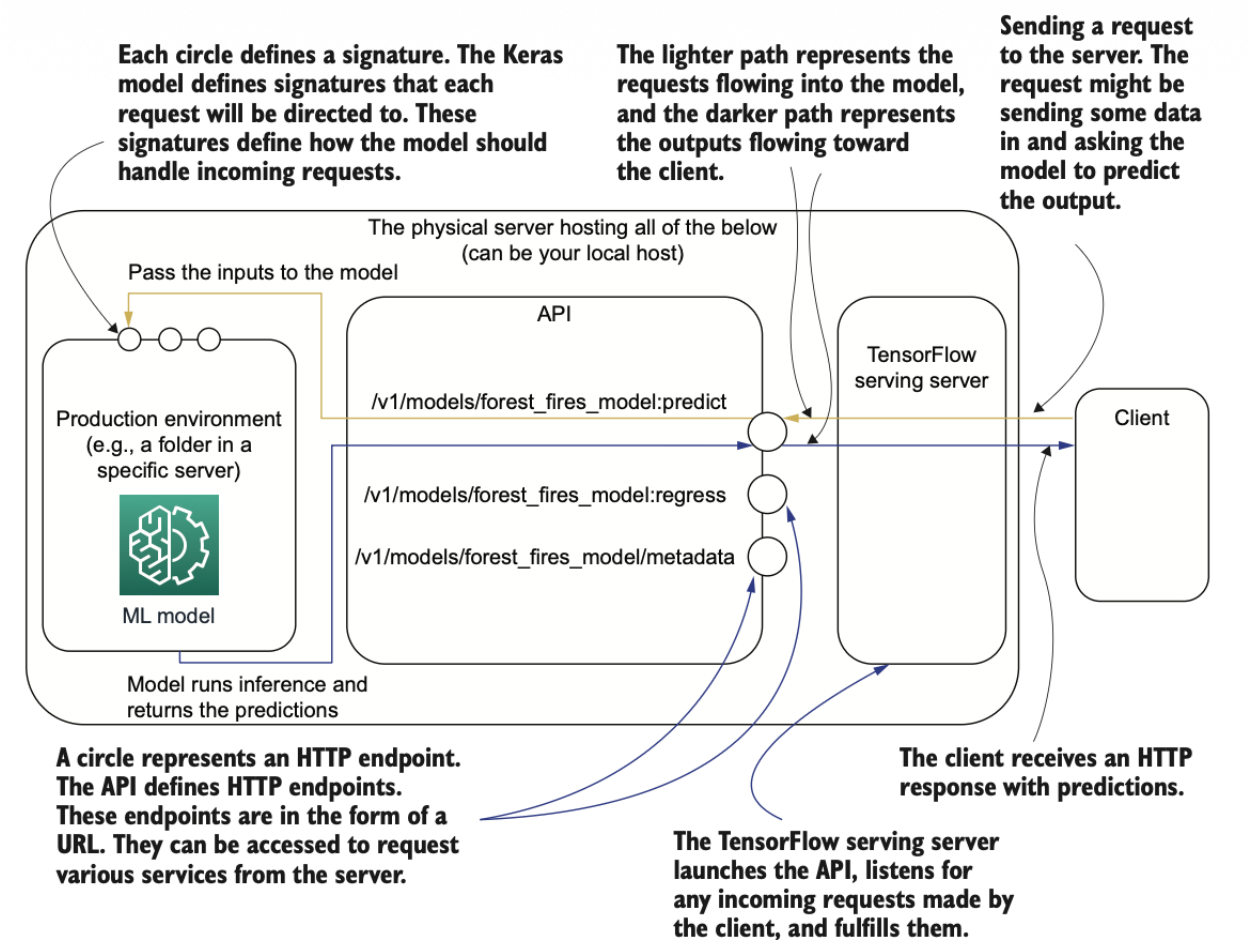

Figure 2 How the model interacts with the API, the TensorFlow server, and the client

Training logs reveal the pipeline's complexity: Python wheel packages are generated for distribution, model checkpoints are written to directories, and validation losses are reported across epochs. Outliers in the forest fire dataset, particularly skewed variables like FFMC, are handled using tensorflow_data_validation (tfdv). With this, anomalies can be flagged, thresholds edited, and cleaned data revalidated. Figure 3 displays a summary statistics chart where heavy skewness in some columns highlights the need for anomaly detection.

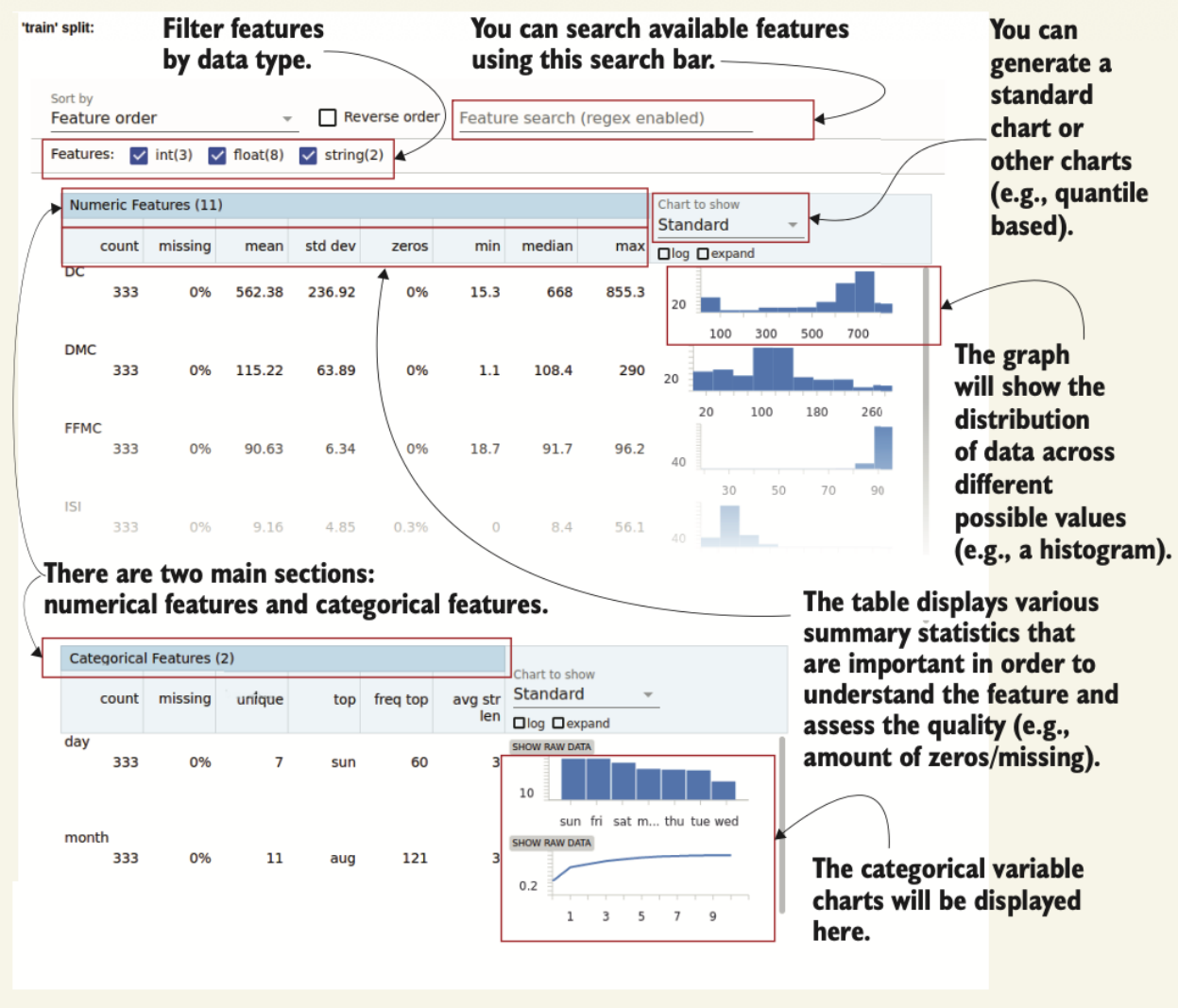

Figure 3 The summary statistics graphs generated for the data by the StatisticsGen component

To move from experimentation to production, models are wrapped in Docker containers, which encapsulate the model, its serving binaries, and all dependencies in an isolated unit. Docker images (such as tensorflow/serving) provide REST endpoints through which clients can request predictions. Figure 4 presents the architecture: the ML model sits behind a TensorFlow Serving API, exposed on a server port, where endpoints like /v1/models/forest_fires_model:predict deliver predictions. The pipeline also integrates validators—InfraValidator checks that the container is healthy, while Evaluator compares new models against baselines. Only when both infrastructure and accuracy checks pass will the Pusher component release the model to production.

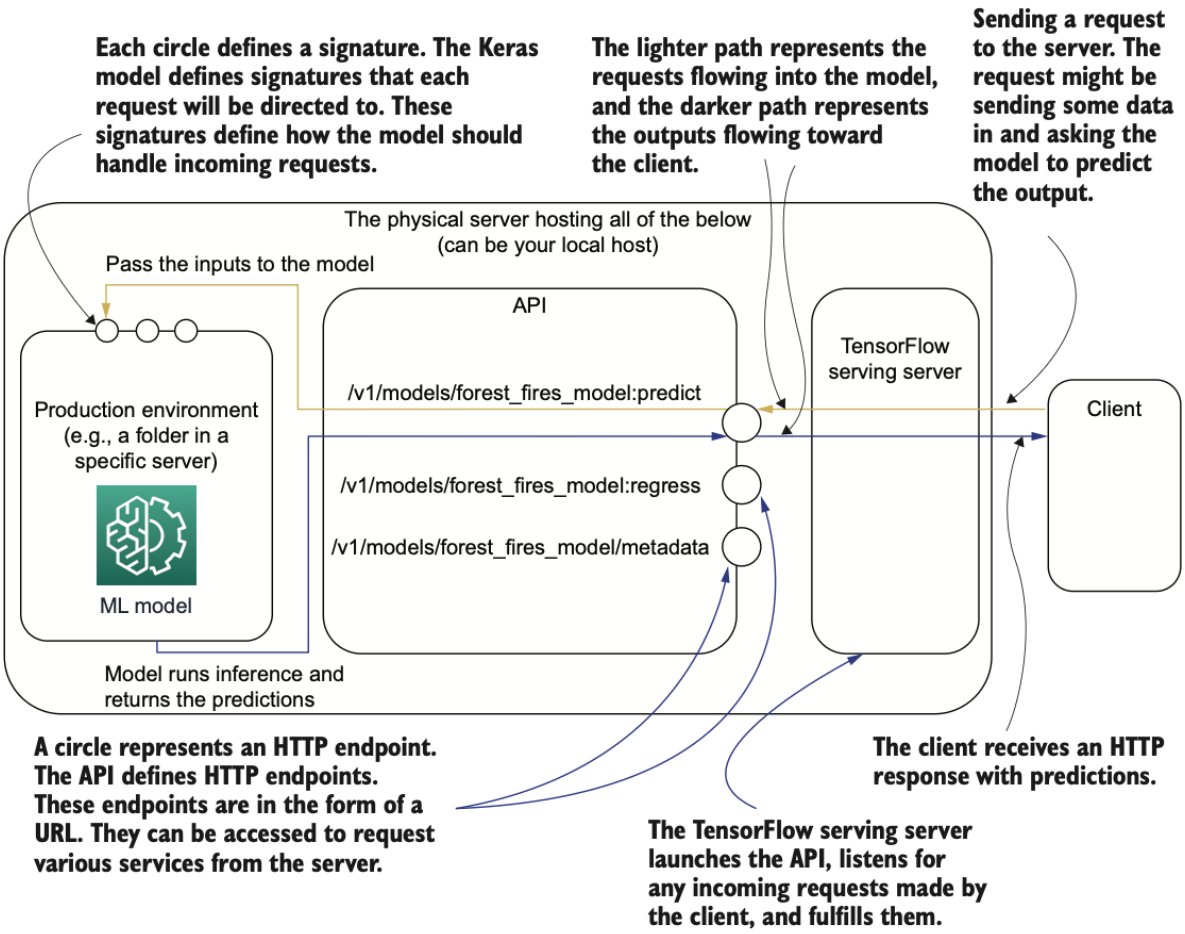

Figure 4 How the model interacts with the API, the TensorFlow server, and the client

The final pipeline integrates all TFX components: data ingestion, statistics, schema inference, transformations, model training, anomaly detection, validation, and deployment. Once deployed, clients interact with the model purely through APIs, receiving real-time predictions as JSON responses.

In conclusion, TFX provides a structured and automated route to MLOps. It standardizes machine learning development with pipelines that not only train models but also enforce validation, guarantee reproducibility, and simplify deployment through Docker and TensorFlow Serving. This transforms models from fragile experiments into scalable, production-ready systems that organizations can rely on for continuous learning and decision-making.

Reproduced code for forest_fires_transform.py:

```
import tensorflow as tf

import tensorflow_transform as tft
```

```
import forest_fires_constants


_DENSE_FLOAT_FEATURE_KEYS                                        =
forest_fires_constants.DENSE_FLOAT_FEATURE_KEYS

_VOCAB_FEATURE_KEYS = forest_fires_constants.VOCAB_FEATURE_KEYS

_BUCKET_FEATURE_KEYS = forest_fires_constants.BUCKET_FEATURE_KEYS

_BUCKET_FEATURE_BOUNDARIES                                       =
forest_fires_constants.BUCKET_FEATURE_BOUNDARIES

_LABEL_KEY = forest_fires_constants.LABEL_KEY

_transformed_name = forest_fires_constants.transformed_name


def preprocessing_fn(inputs):

    outputs = {}

    for key in _DENSE_FLOAT_FEATURE_KEYS:

        outputs[_transformed_name(key)] = tft.scale_to_z_score(

            sparse_to_dense(inputs[key])

        )

    for key in _VOCAB_FEATURE_KEYS:

        outputs[_transformed_name(key)]                          =
tft.compute_and_apply_vocabulary(

            sparse_to_dense(inputs[key]),

            num_oov_buckets=1

        )

    for     key,     boundary     in     zip(_BUCKET_FEATURE_KEYS,
_BUCKET_FEATURE_BOUNDARIES):

        outputs[_transformed_name(key)] = tft.apply_buckets(

            sparse_to_dense(inputs[key]),

            bucket_boundaries=[boundary]

        )
```

```
    outputs[_transformed_name(_LABEL_KEY)]                          =
sparse_to_dense(inputs[_LABEL_KEY])


    return outputs


def sparse_to_dense(x):
    return tf.squeeze(
        tf.sparse.to_dense(
            tf.SparseTensor(x.indices,                    x.values,
[x.dense_shape[0], 1])
        ),
        axis=1
    )
```