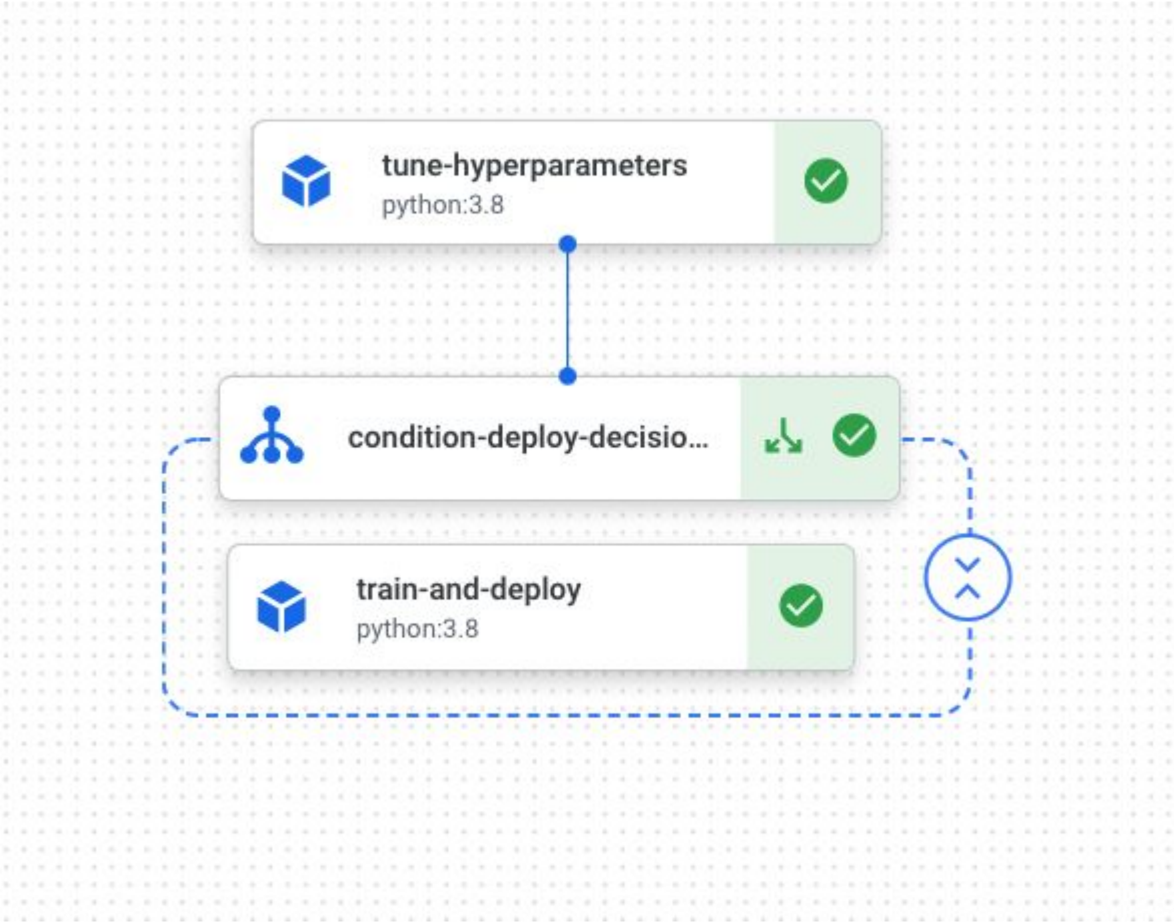
Kubeflow Pipelines on Vertex AI

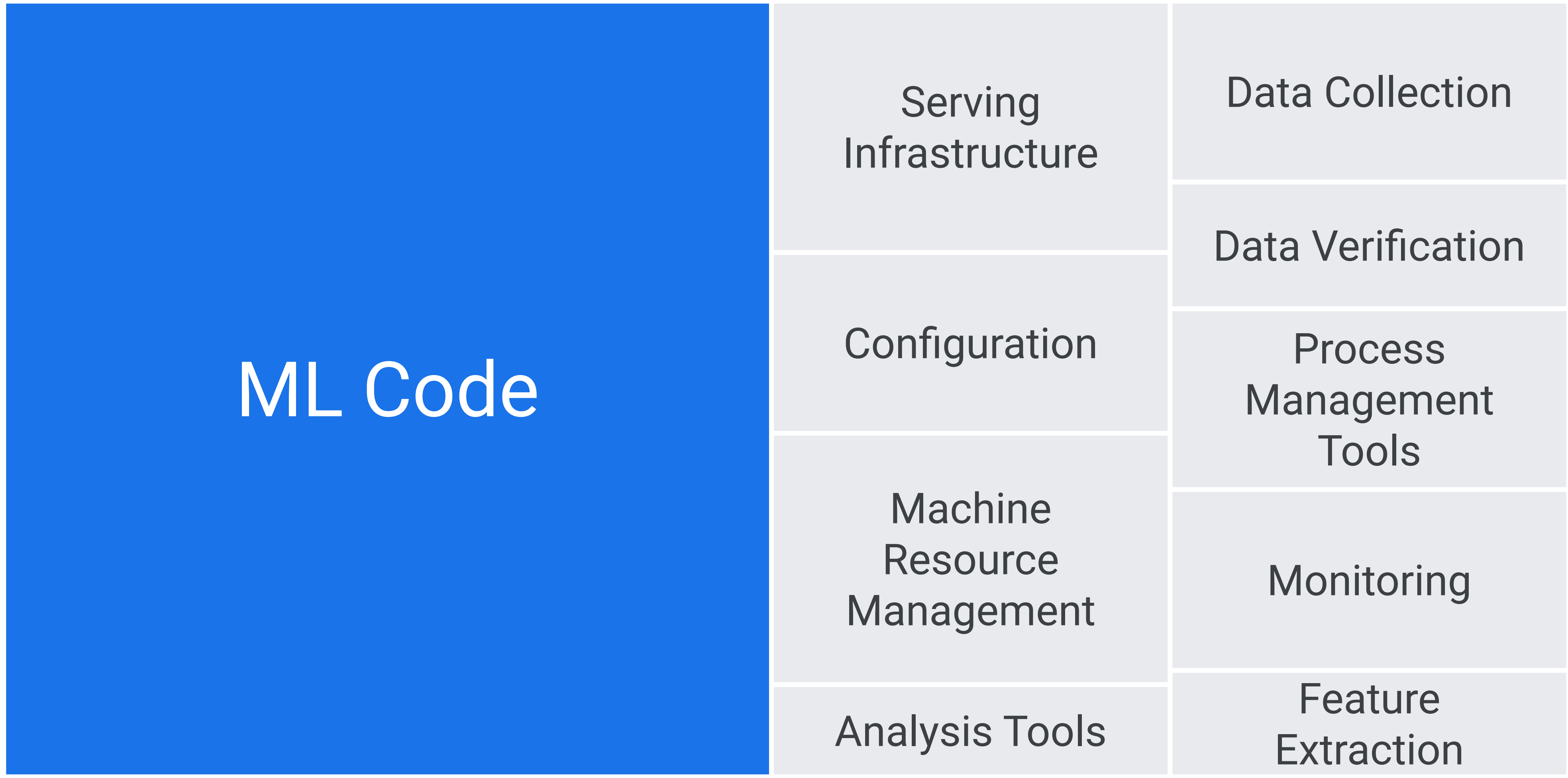


Level 0: All in the notebook

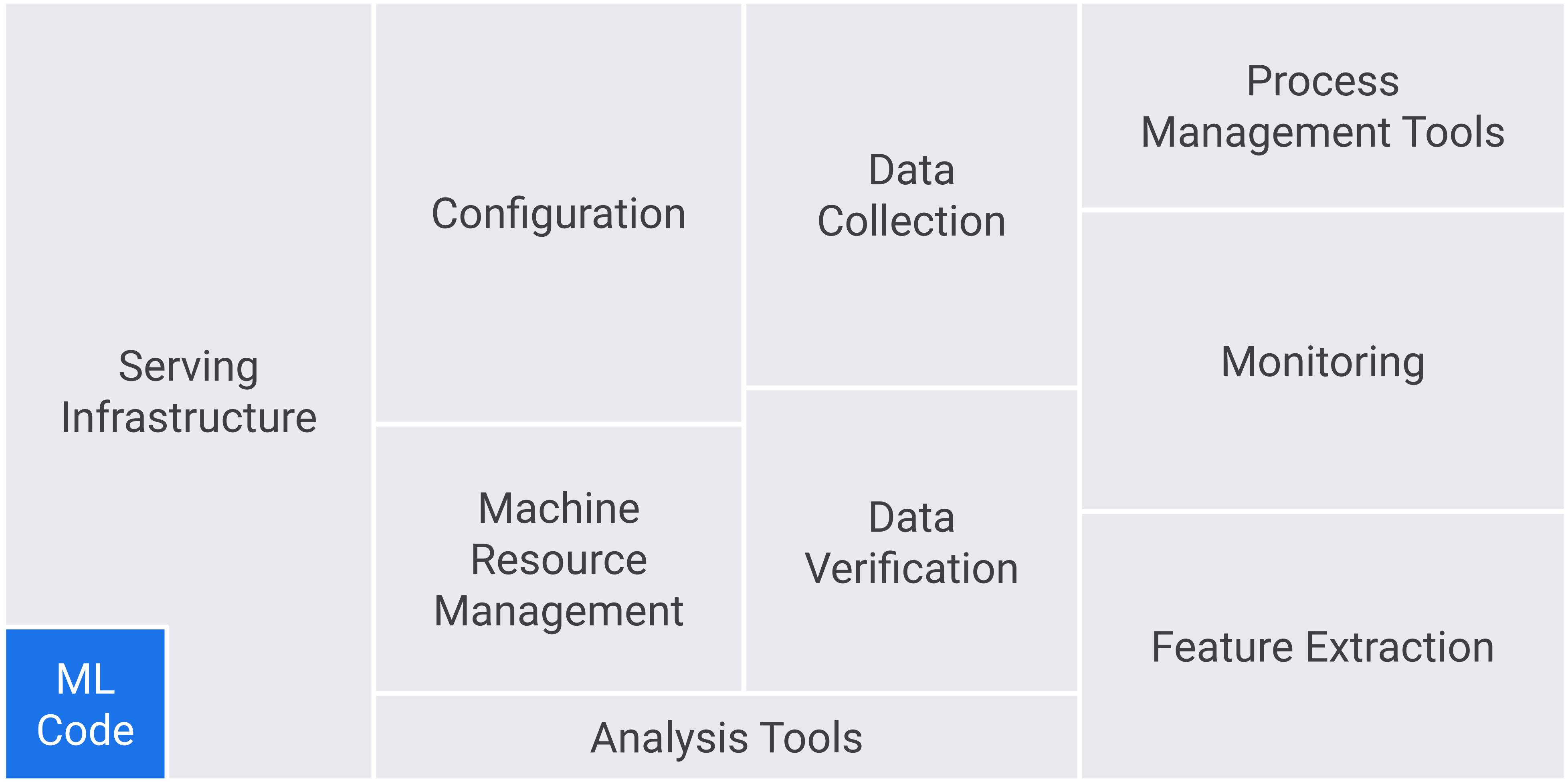
Level 1: Containerized Training

Level 2: ML Pipelines

Perception: ML products are mostly about ML

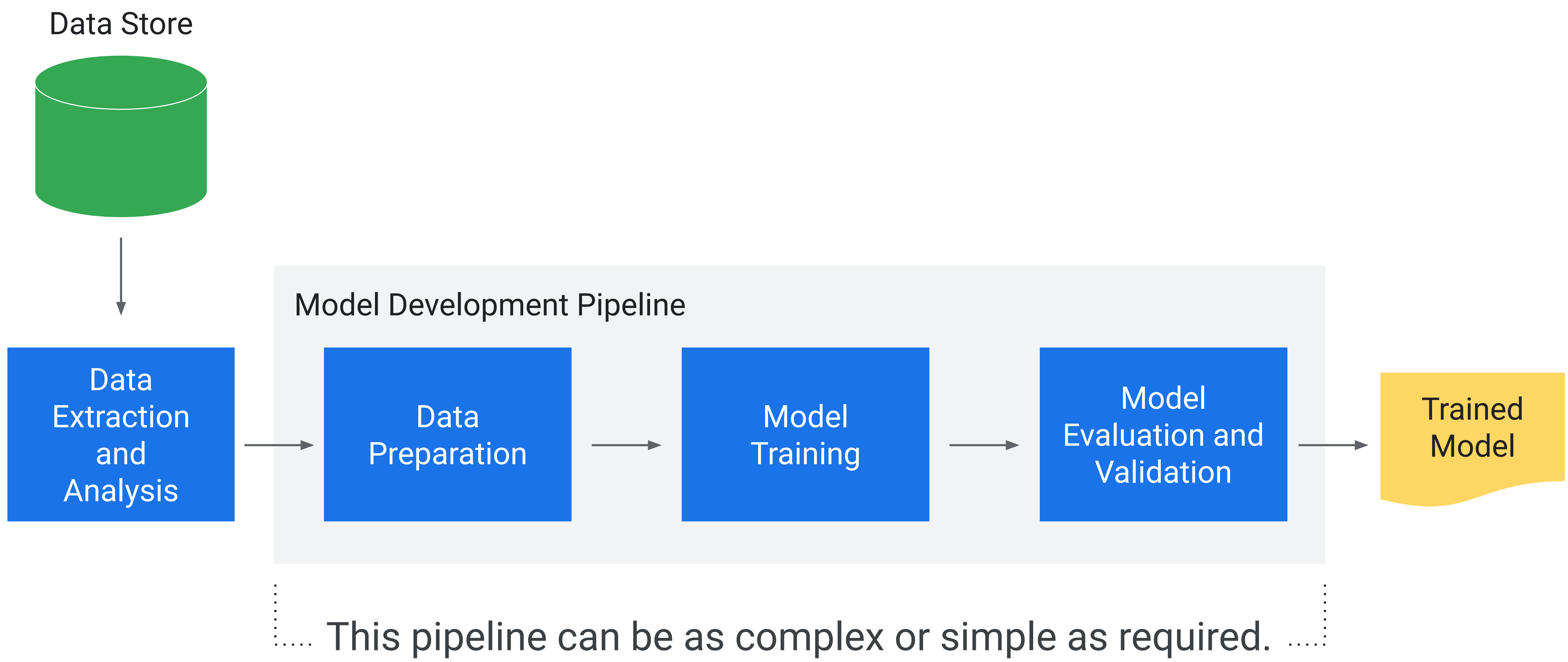


Reality: ML Requires lots of DevOps

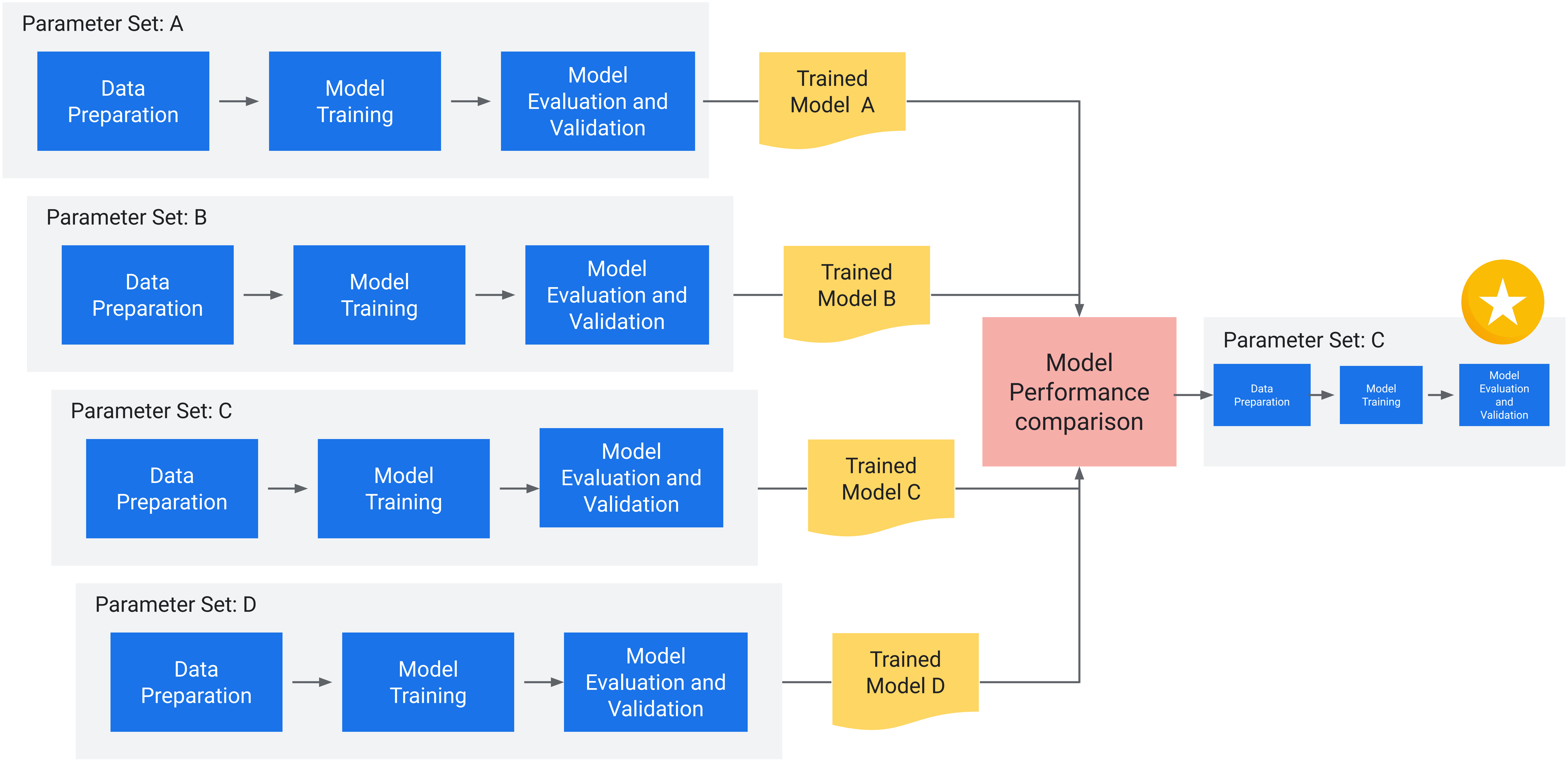


Source: Sculley et al.: Hidden Technical Debt in Machine Learning Systems

The ML process



Machine learning is all about experimentation!



Kubeflow provides a standardized platform for building ML pipelines

- Leverage containers and Kubernetes so that in ML pipelines can be run on a cloud or on-premises with Anthos on GKE.
- Kubeflow is a cloud-native, multi-cloud solution for ML.
- Kubeflow provides a platform for composable, portable, and scalable ML pipelines.
- If you have a Kubernetes-conformant cluster, you can run Kubeflow.

Kubeflow pipelines enable:

ML workflow orchestration

Share, re-use, and compose

Rapid, reliable experimentation

What constitutes a Kubeflow pipeline?

Containerized implementations of ML tasks

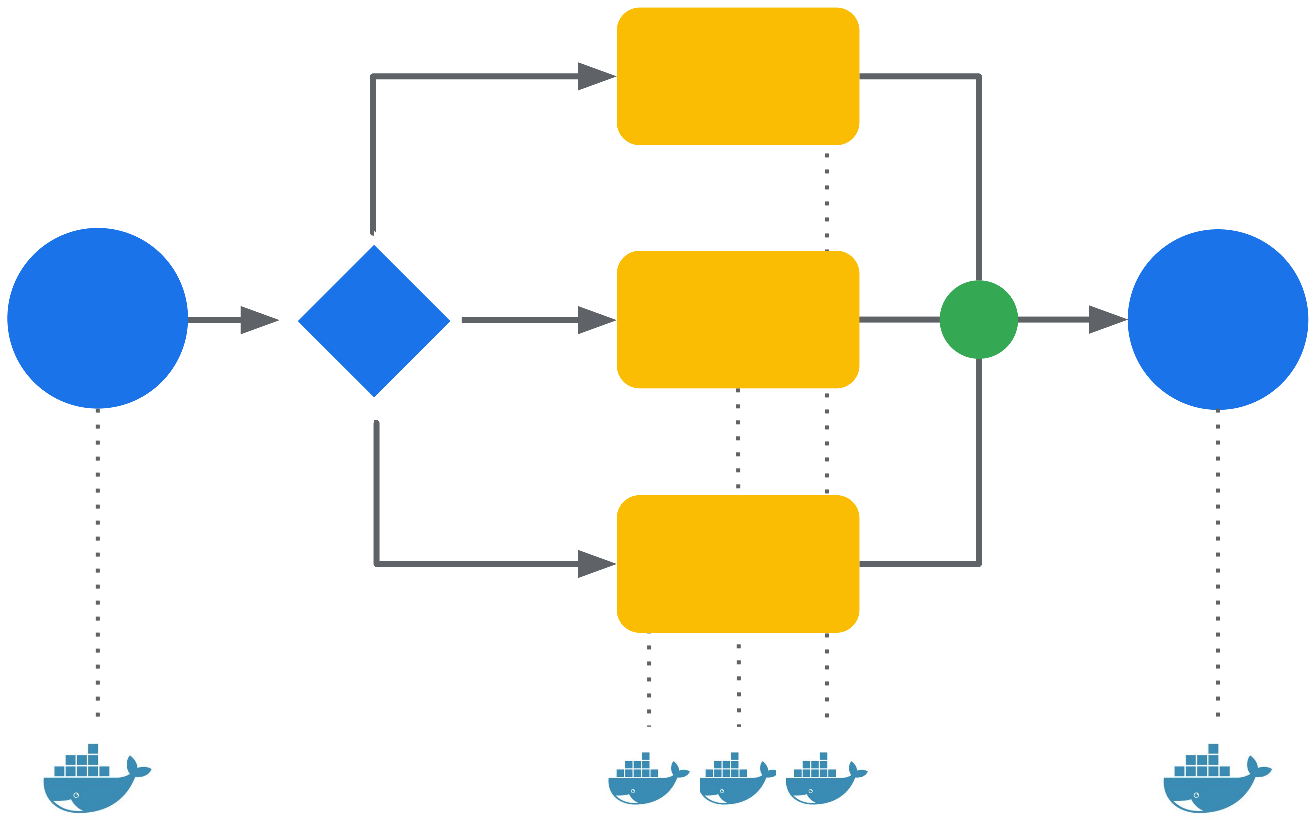
- Example of ML tasks: Data import, training, serving, model evaluation
- Containers provide portability, repeatability, and encapsulation.
- A containerized task can invoke other services, such as Vertex AI, Dataflow, or Dataproc.

Specification of the sequence of steps

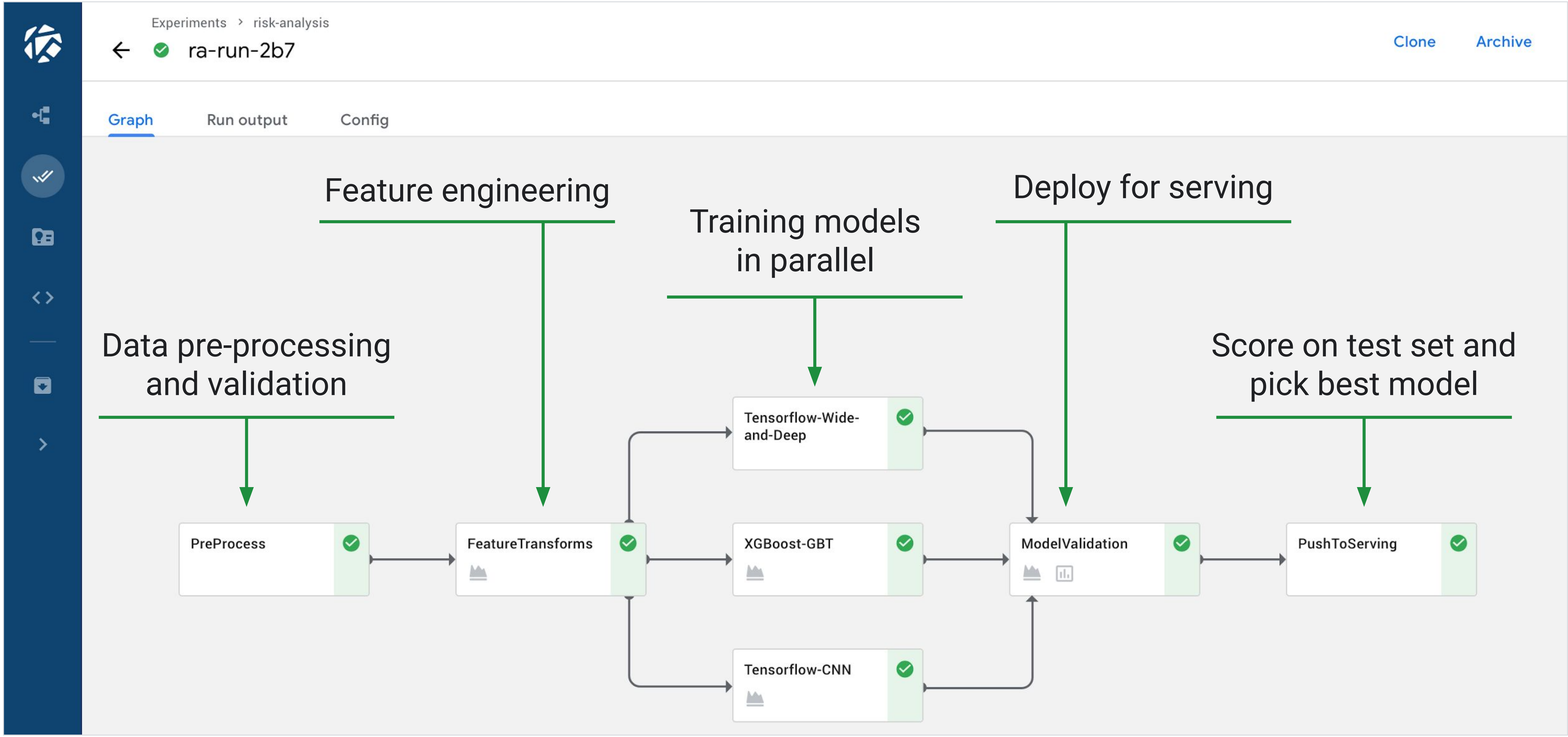
- Specified via Python SDK

Input parameters

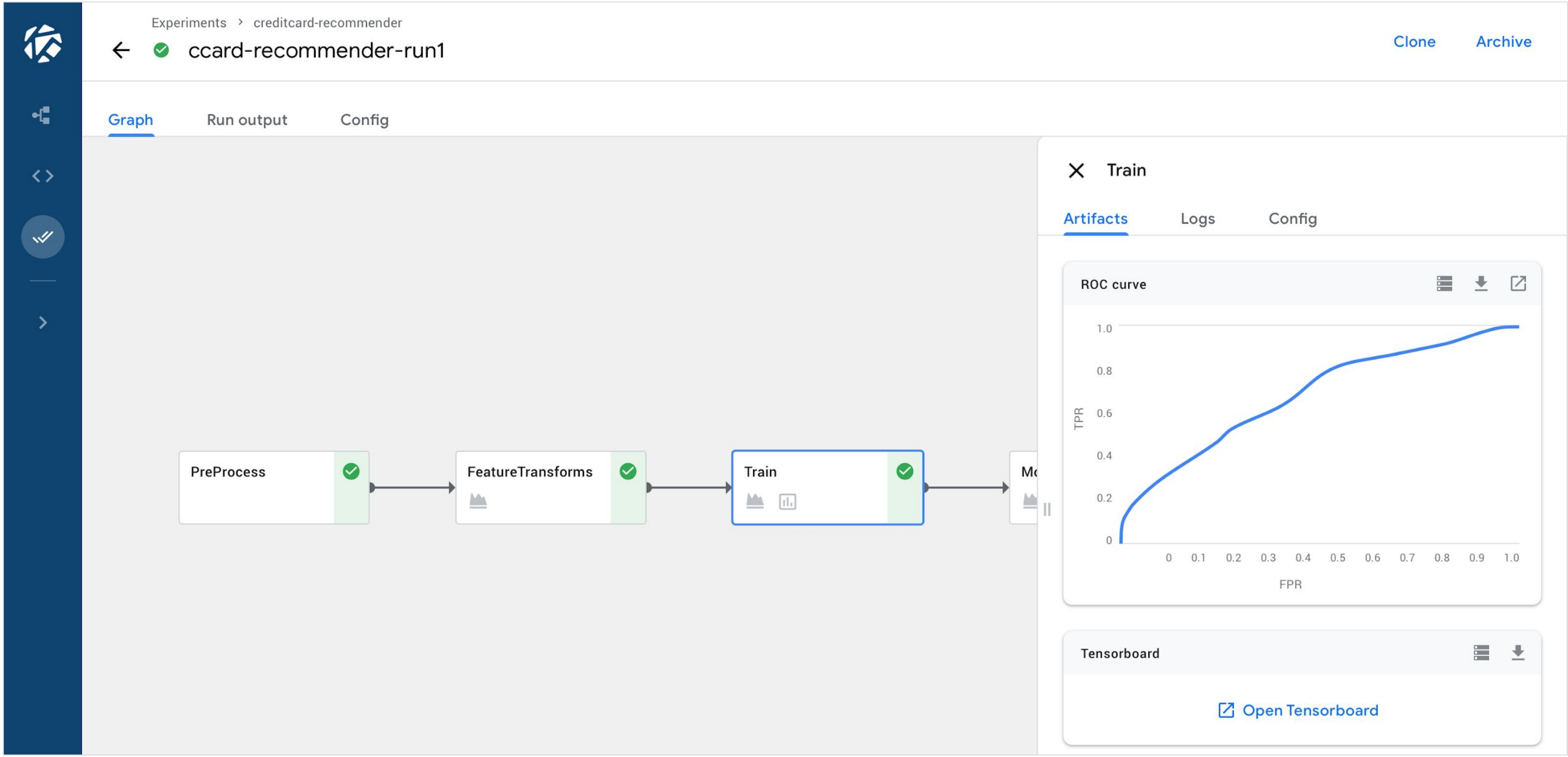
- A “Job” is a pipeline invoked w/specific parameters



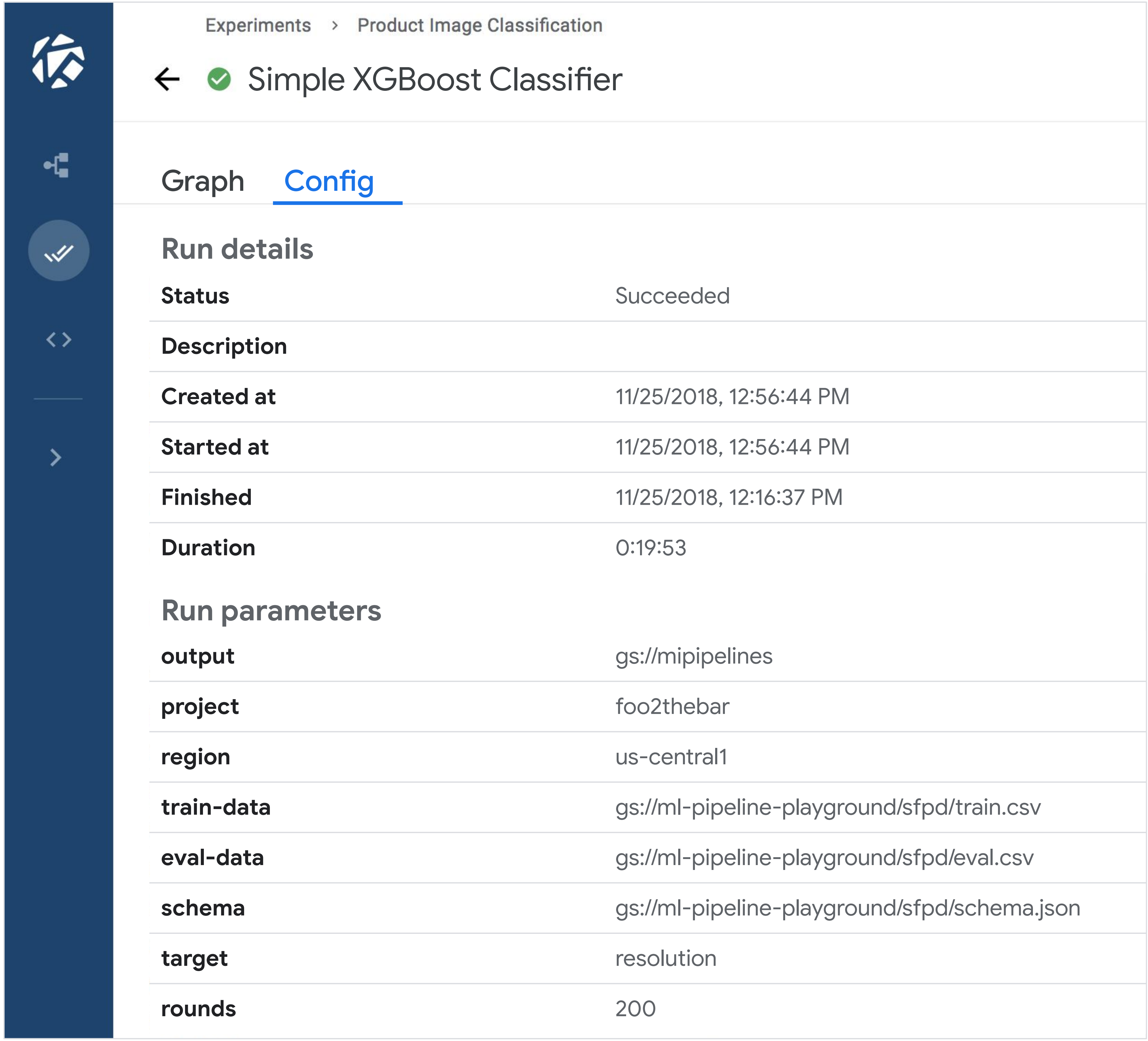
Visual depiction of pipeline topology



Rich visualization of metrics



View all configs, inputs, and outputs



Author pipelines with an intuitive Python SDK

jupyter TFX Pipeline Notebook Last Checkpoint: 10 minutes ago (unsaved changes)

Logout Control Panel

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3

Run Code

Define a multi-step Pipeline

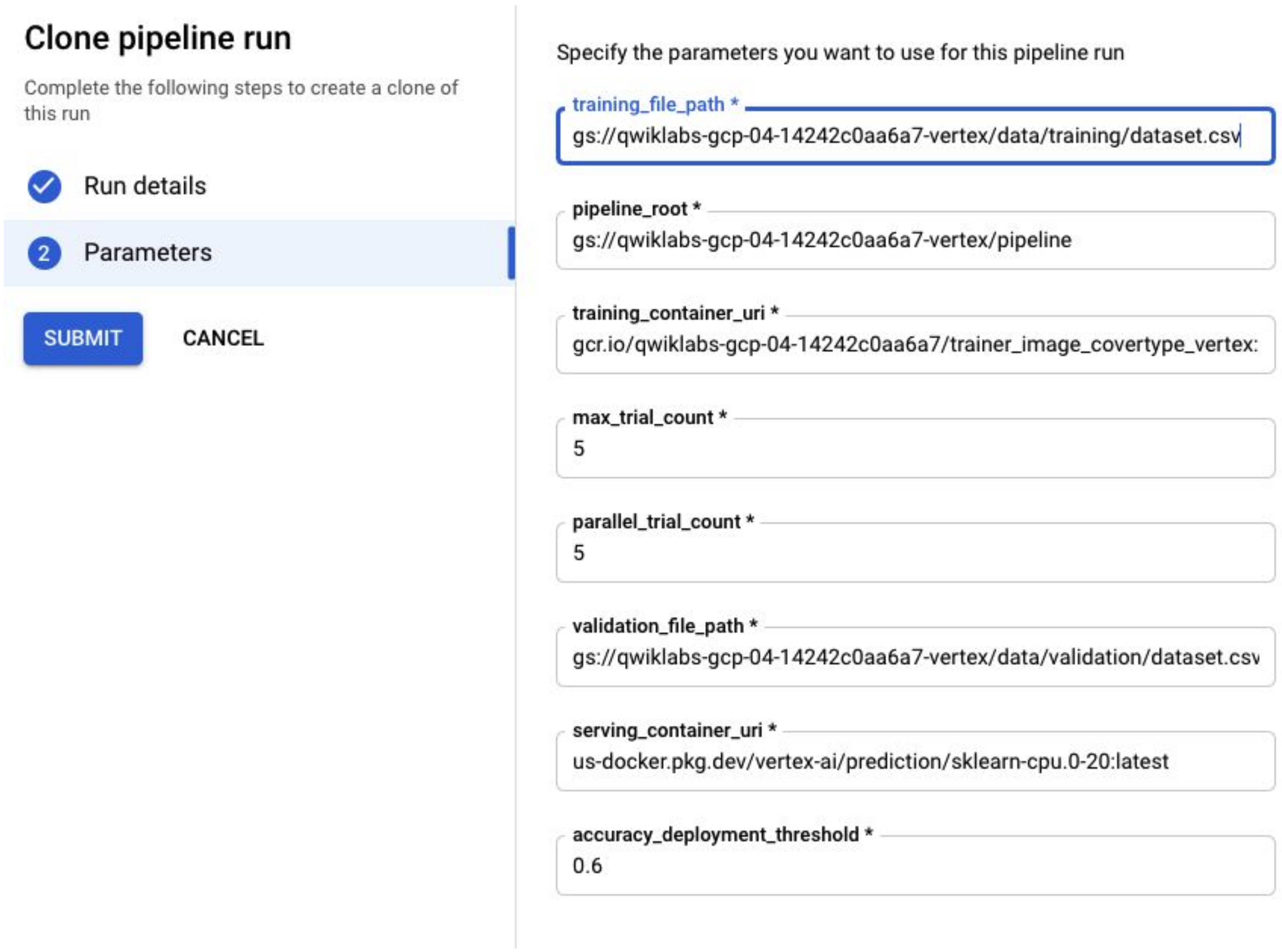
In [3]:
```
import kfp.dsl as dsl

@dsl.pipeline(
  name='TFX Taxi Cab Classification Pipeline Example',
  description='Example pipeline that does classification with model analysis based on a public BigQuery dataset.'
)
def taxi_cab_classification(
    output,
    column_names=dsl.PipelineParam(
        name='column-names',
        value='gs://ml-pipeline-playground/tfx/taxi-cab-classification/column-names.json'),
    key_columns=dsl.PipelineParam(name='key-columns', value='trip_start_timestamp'),
    ...
    analyze_slice_column=dsl.PipelineParam(name='analyze-slice-column', value='trip_start_hour')):

    ...

    preprocess = dataflow_tf_transform_op(train, evaluation, schema, project, preprocess_mode, preprocess_module, trans
    training = tf_train_op(preprocess.output, schema, learning_rate, hidden_layer_size, steps, target, preprocess_modul
    analysis = dataflow_tf_model_analyze_op(training.output, evaluation, schema, project, analyze_mode, analyze_slice_c
    prediction = dataflow_tf_predict_op(evaluation, schema, target, training.output, predict_mode, project, prediction_
    deploy = kubeflow_deploy_op(training.output, tf_server_name)
```

Submit the run

In [16]:
```
import kfp
from kfp import compiler

compiler.Compiler().compile(taxi_cab_classification,  'tfx.tar.gz')

run = client.run_pipeline(exp.id, 'tfx', 'tfx.tar.gz',
                          params={'output': 'gs://bradley-playground',
                                  'project': 'bradley-playground'})
```

Package and share pipelines with pipeline artifacts

- Upload and execute pipelines via UI (in addition to API/SDK).
- Pipeline steps can be authored as reusable components.



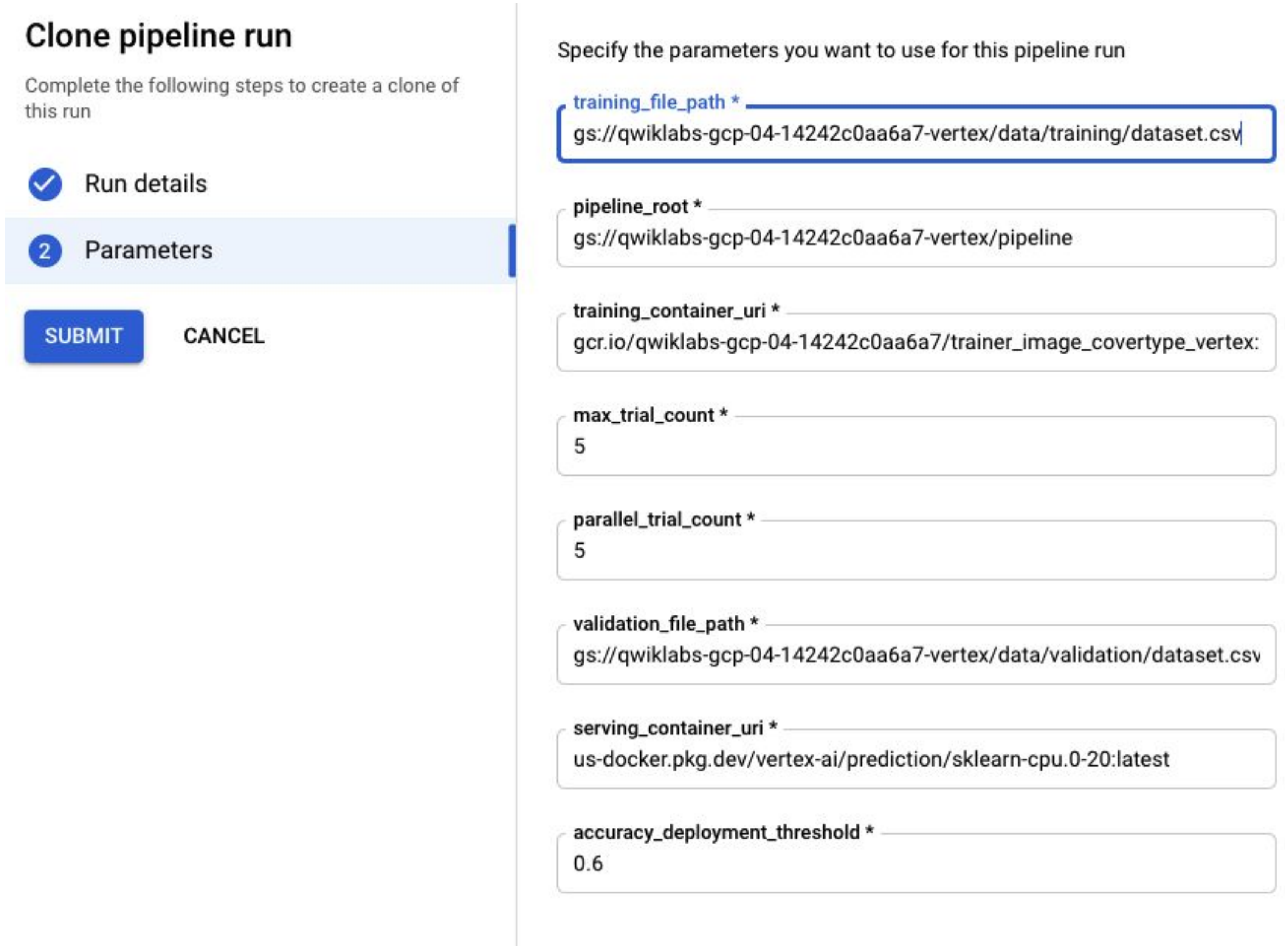
Kubeflow offers a Domain Specific Language (DSL) in Python that allows you to use Python code to describe Kubeflow tasks as they organize themselves in a Directed Acyclic Graph (DAG).

We describe this DSL next...

```python
import kfp

@kfp.dsl.pipeline(
    name="covertype-kfp-pipeline",
    description="The Covertype Classifier KFP Pipeline",
    pipeline_root=PIPELINE_ROOT,

)
def covertype_train(
    training_container_uri: str = TRAINING_CONTAINER_IMAGE_URI,
    serving_container_uri: str = SERVING_CONTAINER_IMAGE_URI,
    training_file_path: str = TRAINING_FILE_PATH,
    validation_file_path: str = VALIDATION_FILE_PATH,
    accuracy_deployment_threshold: float = THRESHOLD,
    max_trial_count: int = MAX_TRIAL_COUNT,
    parallel_trial_count: int = PARALLEL_TRIAL_COUNT,
    pipeline_root: str = PIPELINE_ROOT,
):
```

Pipeline Decorator

Pipeline Run Parameters




```python
def covertype_train(
    training_container_uri: str = TRAINING_CONTAINER_IMAGE_URI,
    serving_container_uri: str = SERVING_CONTAINER_IMAGE_URI,
    training_file_path: str = TRAINING_FILE_PATH,
    validation_file_path: str = VALIDATION_FILE_PATH,
    accuracy_deployment_threshold: float = THRESHOLD,
    max_trial_count: int = MAX_TRIAL_COUNT,
    parallel_trial_count: int = PARALLEL_TRIAL_COUNT,
    pipeline_root: str = PIPELINE_ROOT,
):
```

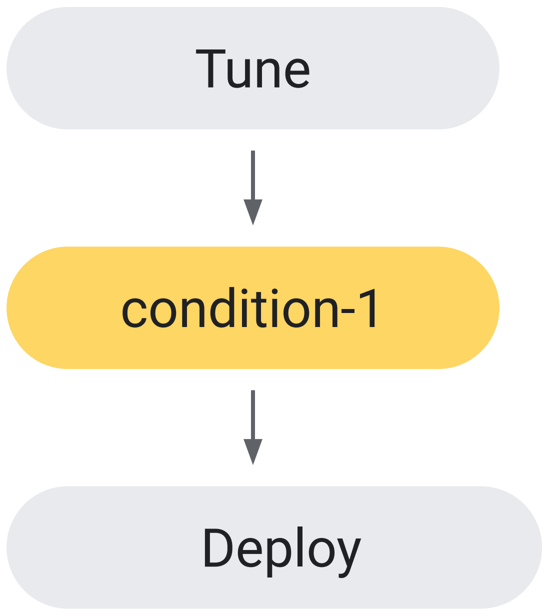
The Run Parameters are supplied at run time.

Define the task DAG within the pipeline function body

```
@kfp.dsl.pipeline(...)
def covertype_train(...):
    # Task DAG defined here
```

1. Create the “ops.”
2. Compose them into a DAG.

(OPs = components)



Creation and composition of ops

```
tuning_op = tune_hyperparameters_component(
    project=PROJECT_ID,
    location=REGION,
    container_uri=training_container_uri,
    # etc.
)
```

```
train_and_deploy_op = train_and_deploy_component(
    project=PROJECT_ID,
    location=REGION,
    alpha=tuning_op.outputs['best_alpha'],
    max_iter=tuning_op.outputs['best_max_iter'],
    # etc.
)
```

1. Ops creation
2. Ops composition

Some ops can be triggered conditionally to other ops output

```
# Deploy the model if the primary metric is higher than a given threshold

accuracy = tuning_op.outputs['best_accuracy']

with dsl.Condition(accuracy >= accuracy_deployment_threshold, name="deploy_decision"):
    train_and_deploy_op = train_and_deploy_component(
        project=PROJECT_ID,
        location=REGION,
        container_uri=training_container_uri,
        serving_container_uri=serving_container_uri,
        training_file_path=training_file_path,
        validation_file_path=validation_file_path,
        staging_bucket=staging_bucket,
        alpha=tuning_op.outputs['best_alpha'],
        max_iter=tuning_op.outputs['best_max_iter'],
    )
```

Compile the Kubeflow pipeline

```
dsl-compile-v2 --py pipeline_vertex/pipeline.py --output PIPELINE_JSON
```

```
[12]: !head {PIPELINE_JSON}
```


```
{
  "pipelineSpec": {
    "components": {
      "comp-condition-deploy-decision-1": {
        "dag": {
          "tasks": {
            "train-and-deploy": {
              "cachingOptions": {
                "enableCache": true
              },
```


- The compilation produces a JSON description of the pipeline
- This JSON version will ultimately be converted by Vertex into a Kubeflow YAML argo file after upload to Vertex

Upload and run on Vertex AI Pipeline

```
from google.cloud import aiplatform

aiplatform.init(project=PROJECT_ID, location=REGION)

pipeline = aiplatform.PipelineJob(
    display_name='covertype_kfp_pipeline',
    template_path=PIPELINE_JSON,
    enable_caching=False,
)

pipeline.run()
```

3 main types of Kubeflow components

01

Pre-built components

- Just load the component from its description and compose.

02

Lightweight Python components

- Implement the component code.

03

Custom components

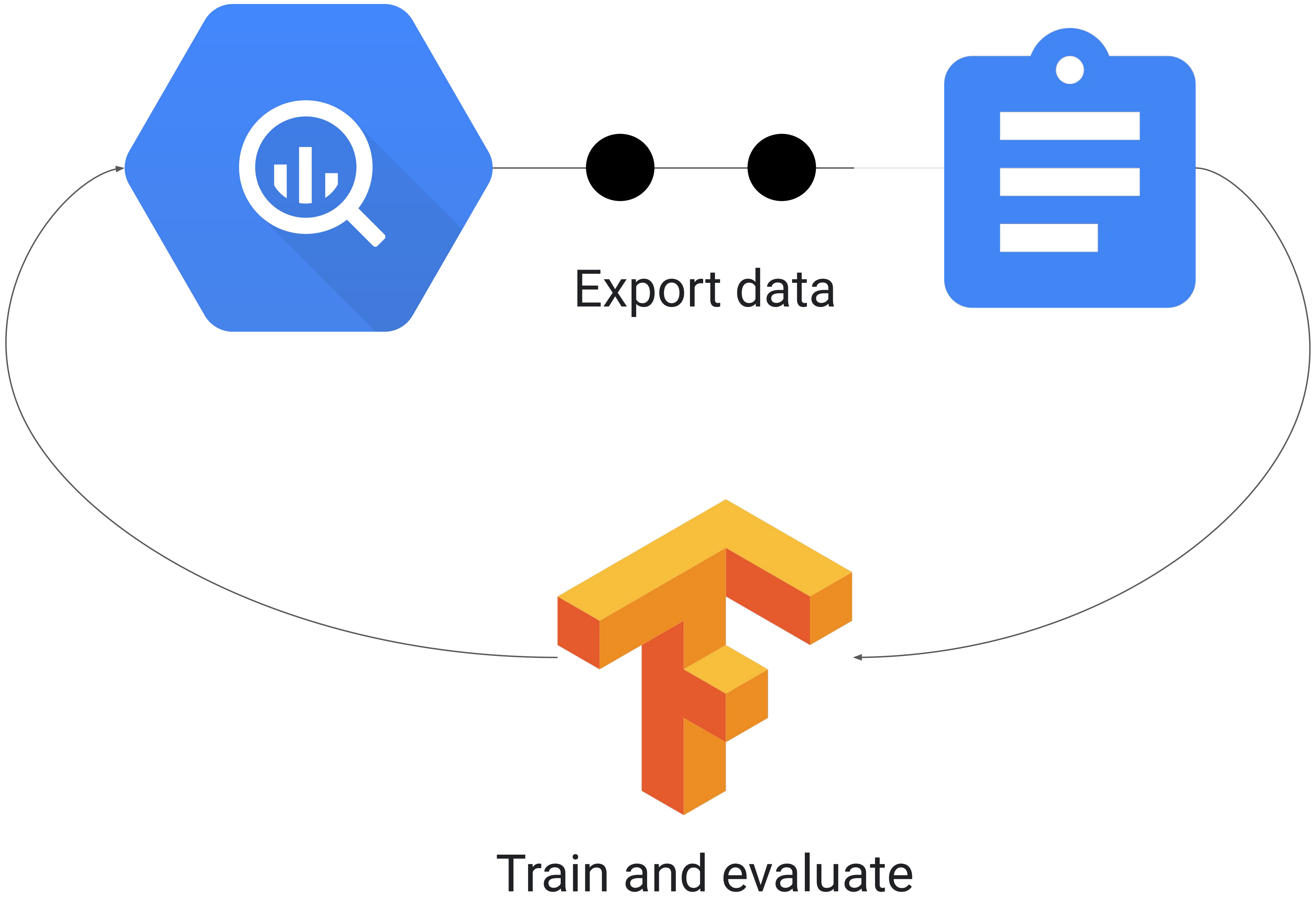
- Implement the component code.
- Package it into a Docker container.
- Write the component description.

AutoML can be launched using pre-built components

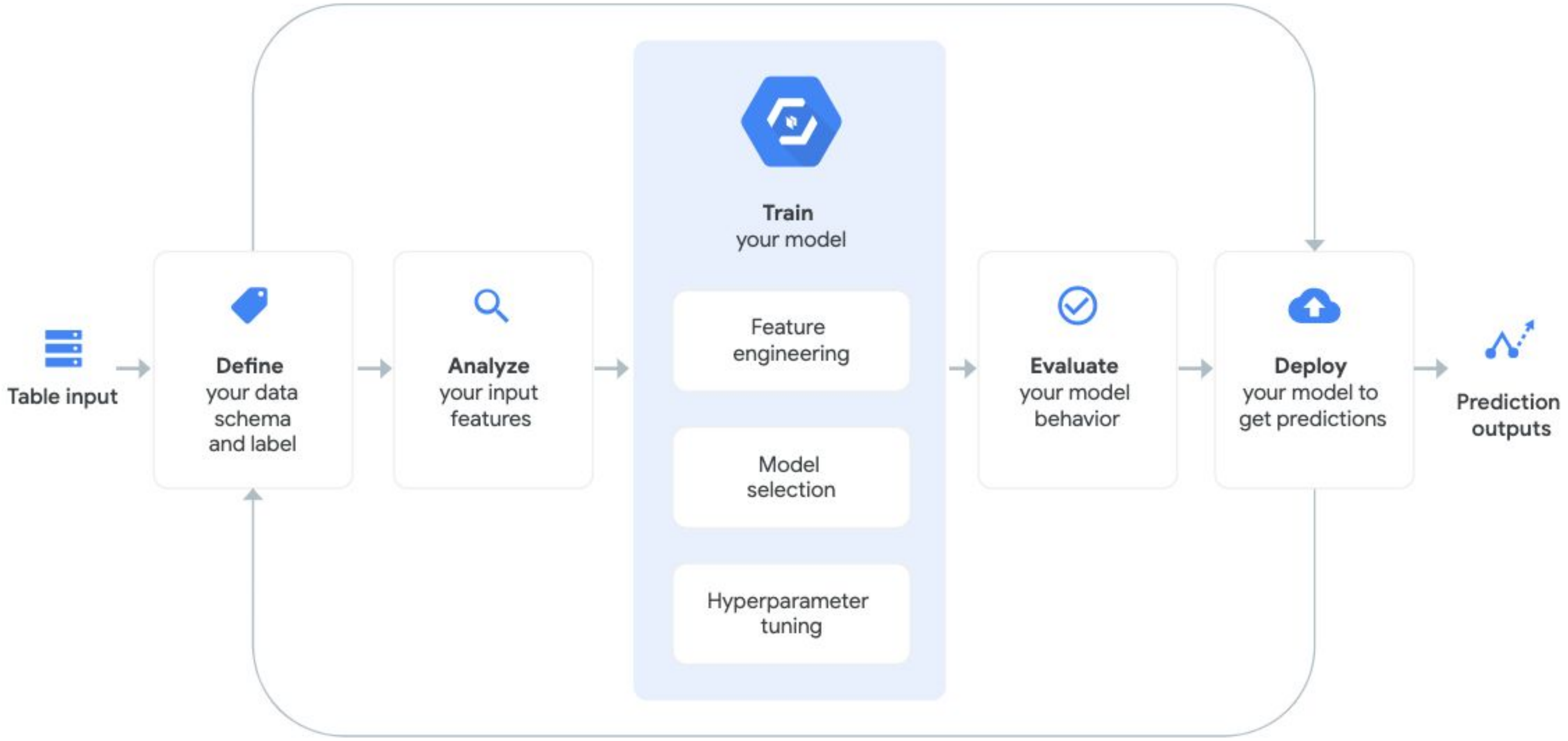
```
from google_cloud_pipeline_components.aiplatform import (
    TabularDatasetCreateOp,
    AutoMLTabularTrainingJobRunOp
    AutoMLImageTrainingJobRunOp
    AutoMLForecastingTrainingJobRunOp
    EndpointCreateOp,
    ModelDeployOp,

    # etc.
)
```

AutoML Vertex components exists for many input sources and ML problems

It can take days to months to create an ML model



Using AutoML within a Vertex Pipeline can speed up things!



Step 1: Create a Vertex Dataset

```
dataset_create_task = TabularDatasetCreateOp(
    display_name=DISPLAY_NAME,
    bq_source=DATASET_SOURCE,
    project=PROJECT,
)
```

bq://project.dataset.table"

Step 2: Launch AutoML training

```
automl_training_task = AutoMLTabularTrainingJobRunOp(
    project=PROJECT,
    display_name=DISPLAY_NAME,
    optimization_prediction_type="classification",
    dataset=dataset_create_task.outputs["dataset"],
    target_column=TARGET_COLUMN,
)
```

The output `dataset_create_task.outputs["dataset"]` is an AutoML dataset

By setting the `dataset` argument as a `dataset_create_task.outputs["dataset"]` we are implicitly ordering the pipeline.

Step 3: Deploy the trained model as before

```
endpoint_create_task = EndpointCreateOp(
    project=PROJECT,
    display_name=DISPLAY_NAME,
)

model_deploy_task = ModelDeployOp(
    model=automl_training_task.outputs["model"],
    endpoint=endpoint_create_task.outputs["endpoint"],
    deployed_model_display_name=DISPLAY_NAME,
    dedicated_resources_machine_type=SERVING_MACHINE_TYPE,
    dedicated_resources_min_replica_count=1,
    dedicated_resources_max_replica_count=1,
)
```

Lab

AutoML Pipelines on Vertex AI

In this lab, you will learn how to use Vertex AI Pipelines to build a **Vertex AutoML pipeline** to train, tune, and serve a model.

notebooks/kubeflow_pipelines/pipelines/solutions/kfp_pipeline_vertex_automl_online_predictions.ipynb

Importing Kubeflow pre-built components

Standard package

```
from google_cloud_pipeline_components.aiplatform
import (
    AutoMLTabularTrainingJobRunOp
    AutoMLImageTrainingJobRunOp
    AutoMLForecastingTrainingJobRunOp
    # etc.
    CustomContainerTrainingJobRunOp
    EndpointCreateOp,
    ModelDeployOp,
    ModelUploadOp,
)
```

Experimental Package

```
from google_cloud_pipeline_components.experimental.hyperparameter_tuning_job
import (
    HyperparameterTuningJobRunOp,
)

from google_cloud_pipeline_components.experimental.custom_job import (
    CustomTrainingJobOp,
)
```

Using pre-built components for TUNING

```
hp_tuning_task = HyperparameterTuningJobRunOp(
    display_name=f"{PIPELINE_NAME}-kfp-tuning-job",
    project=PROJECT_ID,
    location=REGION,
    worker_pool_specs=worker_pool_specs,
    study_spec_metrics=metric_spec,
    study_spec_parameters=parameter_spec,
    max_trial_count=MAX_TRIAL_COUNT,
    parallel_trial_count=PARALLEL_TRIAL_COUNT,
    base_output_directory=PIPELINE_ROOT,
)
```

```
worker_pool_specs = [
    {
        "container_spec": {
            "image_uri": TRAINING_CONTAINER_IMAGE_URI,
            "args": [
                f"--training_dataset_path={TRAINING_FILE_PATH}",
                f"--validation_dataset_path={VALIDATION_FILE_PATH}",
                "--hptune",
            ],
        },
    }
]

metric_spec = hyperparameter_tuning_job.serialize_metrics(
    {"accuracy": "maximize"}
)


parameter_spec = hyperparameter_tuning_job.serialize_parameters(
    {
        "alpha": hpt.DoubleParameterSpec(
            min=1.0e-4, max=1.0e-1, scale="linear"
        ),
        "max_iter": hpt.DiscreteParameterSpec(
            values=[1, 2], scale="linear"
        ),
    }
)
```

Using pre-built components for TRAINING

```
training_task = CustomTrainingJobOp(
    project=PROJECT_ID,
    location=REGION,
    display_name=f"{PIPELINE_NAME}-kfp-training-job",
    worker_pool_specs=worker_pool_specs_task.output,
    base_output_directory=BASE_OUTPUT_DIR,
)
```

```
worker_pool_specs_task = GetWorkerPoolSpecsOp(
    best_hyperparameters=best_hyperparameters_task.output,
    worker_pool_specs=[
        {
            "machine_spec": {"machine_type": "n1-standard-4"},
            "replica_count": 1,
            "container_spec": {
                "image_uri": TRAINING_CONTAINER_IMAGE_URI,
                "args": [
                    f"--training_dataset_path={TRAINING_FILE_PATH}",
                    f"--validation_dataset_path={VALIDATION_FILE_PATH}",
                    "--nohptune",
                ],
            },
        }
    ],
)
```

When the container is run the environment variable

AIP_MODEL_DIR will be set to BASE_OUTPUT_DIR/model

It is then used in train.py to save the model:

train.py

```
AIP_MODEL_DIR = os.environ["AIP_MODEL_DIR"]
MODEL_FILENAME = "model.pkl"
```

Using pre-built components for SERVING

```
model_upload_task = ModelUploadOp(
    project=PROJECT_ID,
    display_name=f"{PIPELINE_NAME}-kfp-model-upload-job",
    artifact_uri=f"{BASE_OUTPUT_DIR}/model",
    serving_container_image_uri=SERVING_CONTAINER_IMAGE_URI,
)


endpoint_create_task = EndpointCreateOp(
    project=PROJECT_ID,
    display_name=f"{PIPELINE_NAME}-kfp-create-endpoint-job",
)


model_deploy_op = ModelDeployOp(
    model=model_upload_task.outputs["model"],
    endpoint=endpoint_create_task.outputs["endpoint"],
    deployed_model_display_name=MODEL_DISPLAY_NAME,
    dedicated_resources_machine_type=SERVING_MACHINE_TYPE,
    dedicated_resources_min_replica_count=1,
    dedicated_resources_max_replica_count=1,
)
```

Lab

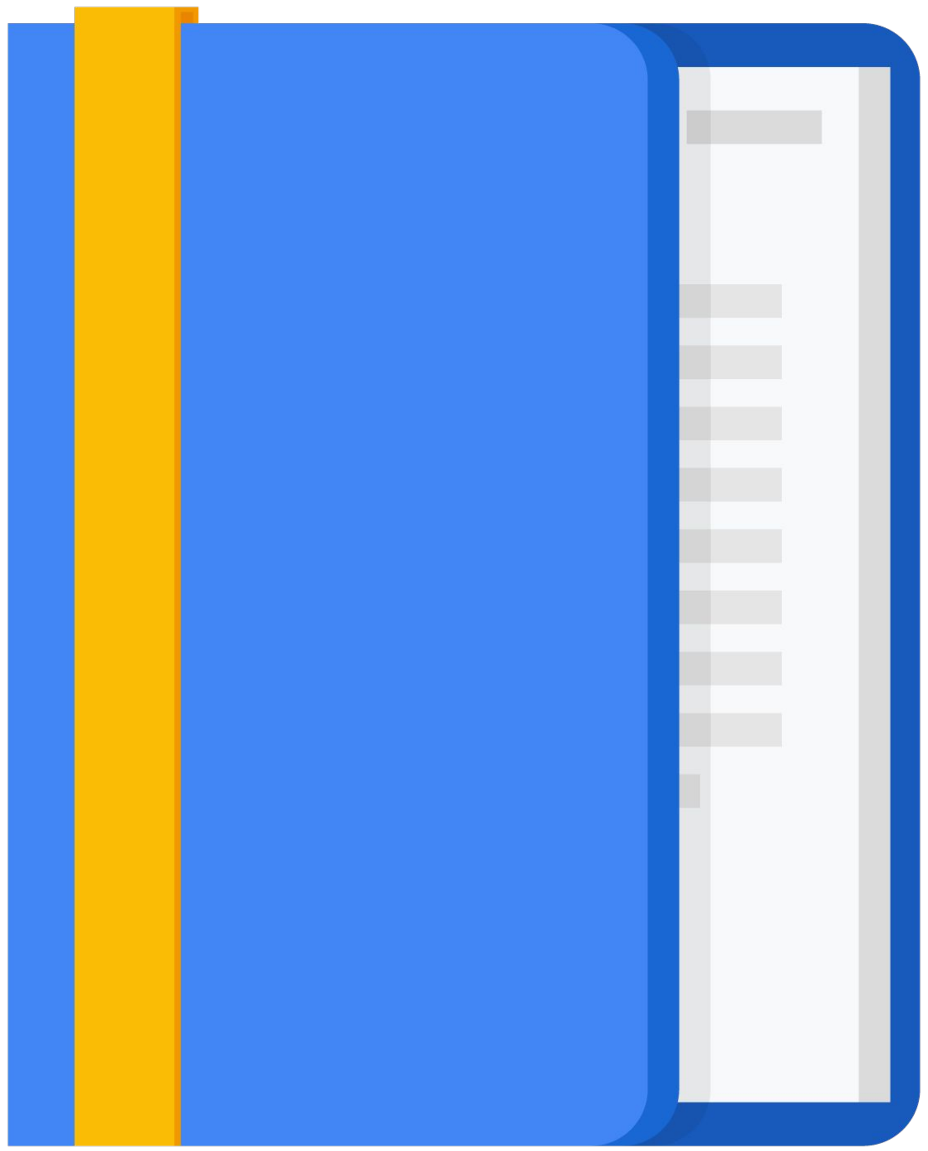
Kubeflow Pipelines on Vertex AI

In this lab, you will learn how to use Vertex AI Pipelines to build a Kubeflow pipeline to train, tune, and serve a model using **Google pre-built components.**

[kubeflow_pipelines/pipelines/labs/kfp_pipeline_vertex_prebuilt.ipynb]

Agenda

- System and Concept Overview
- Describing a Kubeflow Pipeline with KF DSL
- Compile, Upload, and Run
- Pre-built Components
- Lightweight Python Components
- AutoML Vertex Pipelines



Wrap Python functions into KF components

training_lightweight_component.py

```
@component(base_image="python:3.8",
    output_component_file="covertype_kfp_train_and_deploy.yaml",
    packages_to_install=["google-cloud-aiplatform"])
def train_and_deploy(
        project: str,
        location: str,
        container_uri: str,
        serving_container_uri: str,
        training_file_path: str,
        validation_file_path: str,
        staging_bucket: str,
        alpha: float,
        max_iter: int,
):
```

Wrap Python functions into KF components

tuning_lightweight_component.py

```python
from kfp.v2.dsl import component

@component(...)
def tune_hyperparameters(
        container_uri: str,
        # etc.
) -> NamedTuple("Outputs", [
    ("best_accuracy", float),
    ("best_alpha", float),
    ("best_max_iter", int)
]):

# etc.

return best_accuracy, best_alpha, best_max_iter
```

Use and compose the lightweight components as usual

```
tuning_op = tune_hyperparameters(
    project=PROJECT_ID,
    location=REGION,
    container_uri=training_container_uri,
    training_file_path=training_file_path,
    validation_file_path=validation_file_path,
    staging_bucket=staging_bucket,
    max_trial_count=max_trial_count,
    parallel_trial_count=parallel_trial_count,
)
```

Lab

Kubeflow Pipelines on Vertex AI

In this lab, you will learn how to use Vertex AI Pipelines to build a Kubeflow pipeline to train, tune, and serve a model using your implementing **Python lightweight components.**

[kubeflow_pipelines/pipelines/labs/kfp_pipeline_vertex_lightweight.ipynb]