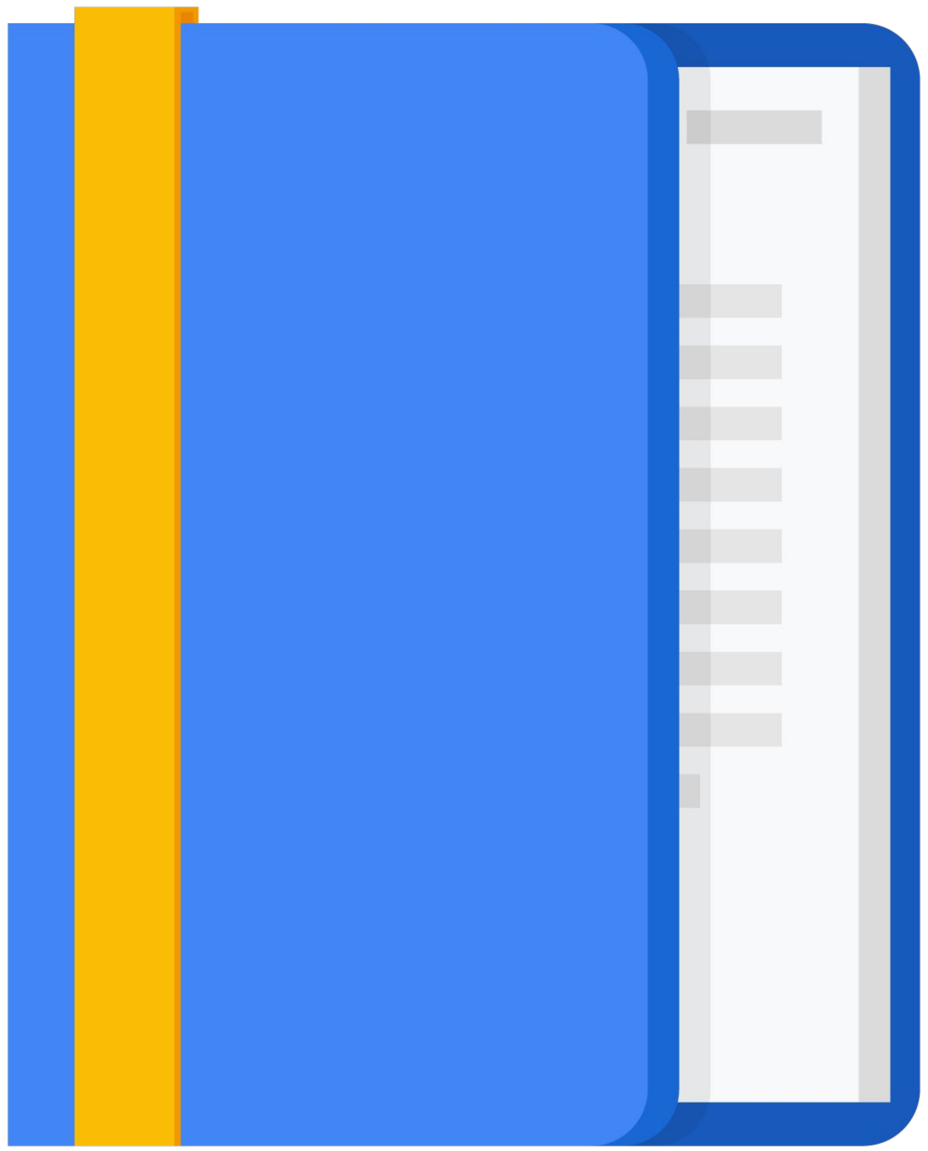
# Kubeflow Pipelines on AI Platform

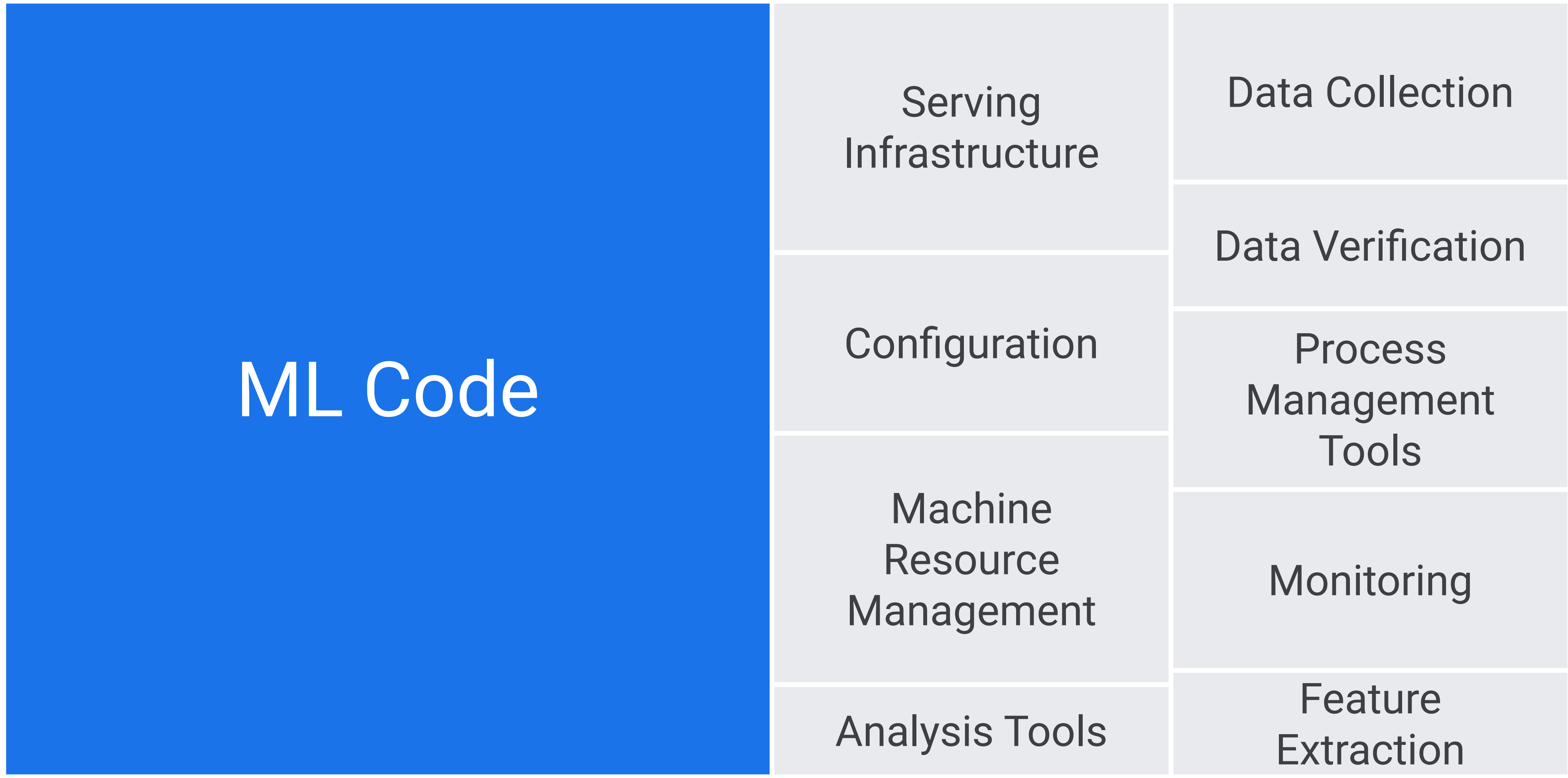
Benoit Dherin
ML Engineer, Google Advanced Solutions Lab

# Agenda

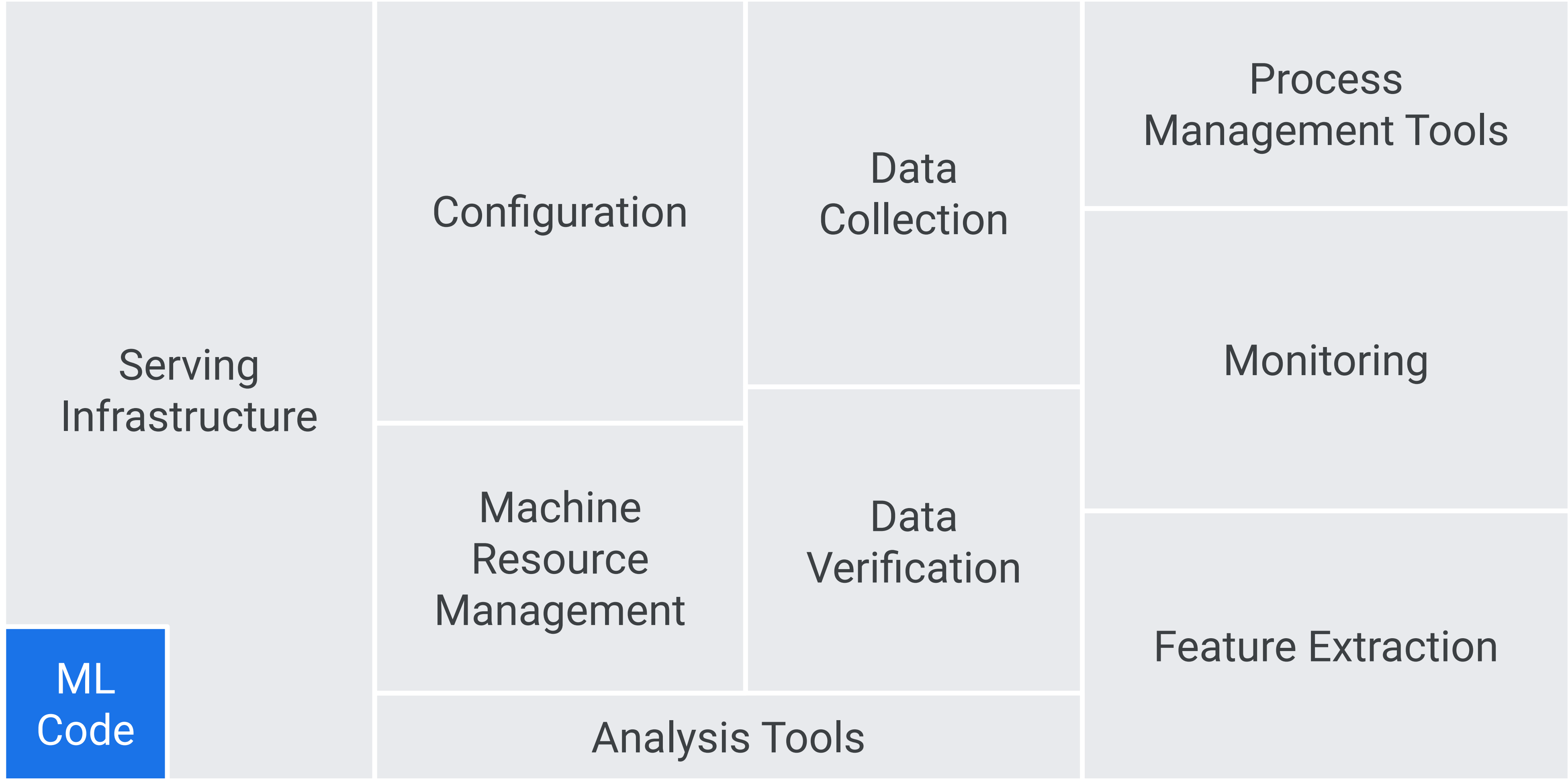
- System and Concept Overview
- Describing a Kubeflow Pipeline with KF DSL
- Pre-built Components
- Lightweight Python Components
- Custom Components
- Compile, Upload, and Run



# Perception: ML products are mostly about ML

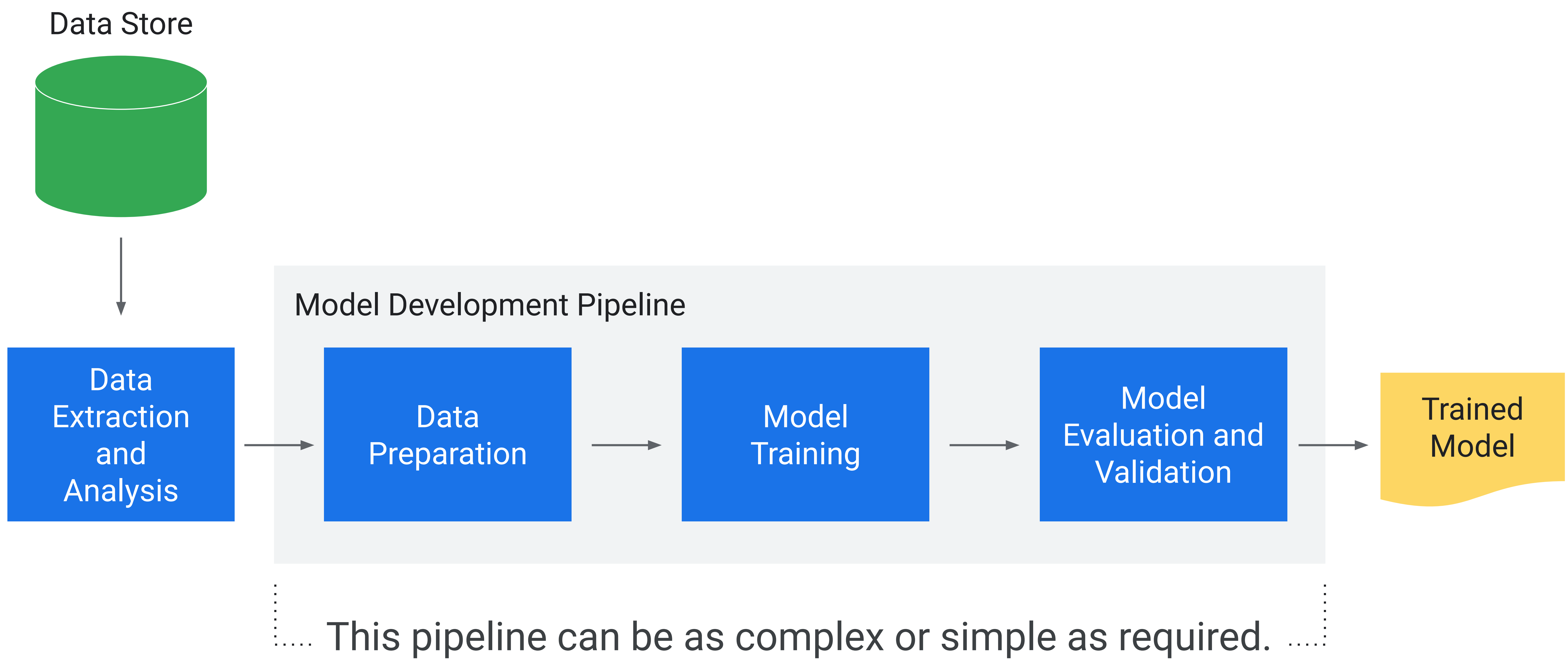


# Reality: ML Requires lots of DevOps

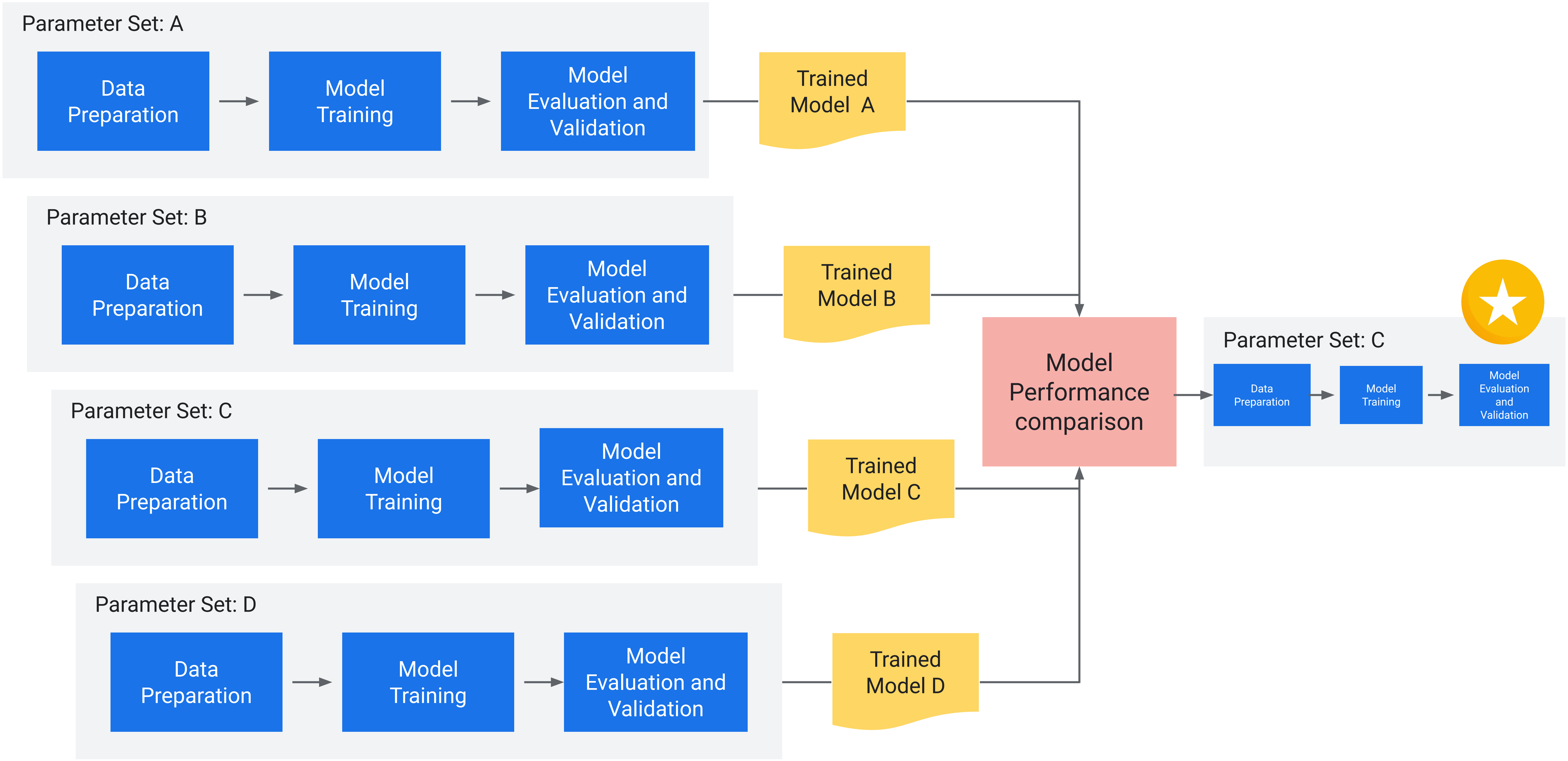



Source: Sculley et al.: Hidden Technical Debt in Machine Learning Systems

# 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**

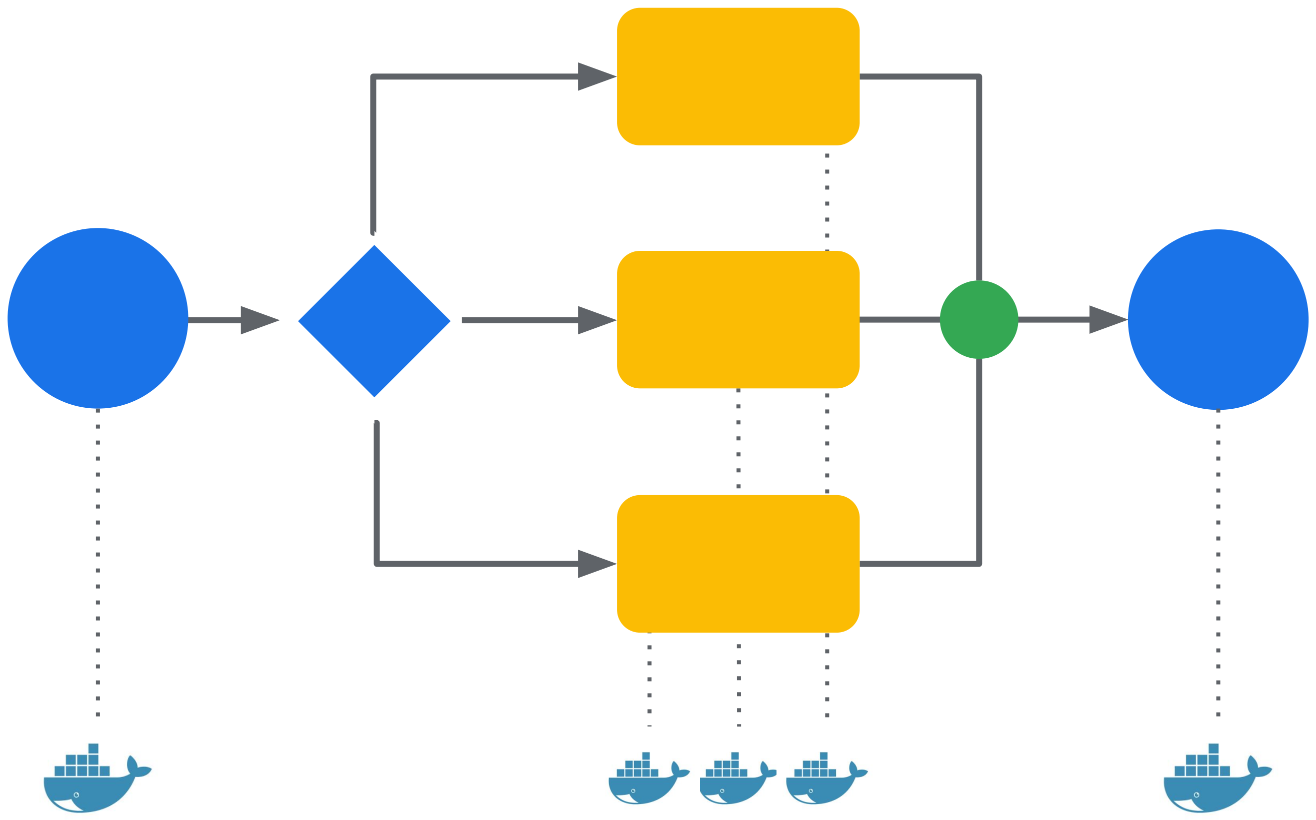
- Example of ML tasks: Data import, training, serving, model evaluation
- Containers provide portability, repeatability, and encapsulation.
- A containerized task can invoke other services, such as AI Platform, Dataflow, or Dataproc.

**Specification of the sequence of steps**

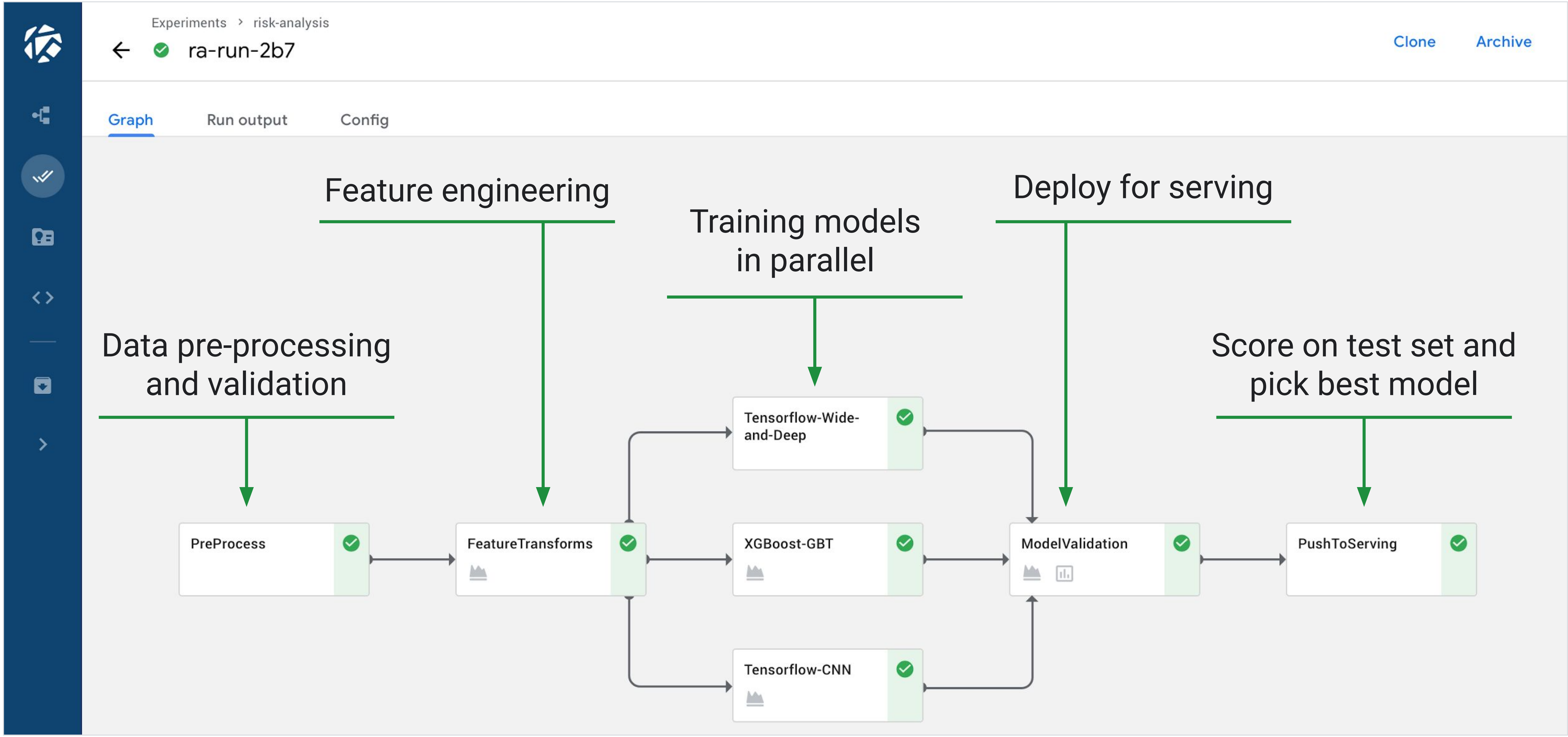
- Specified via Python SDK

**Input parameters**

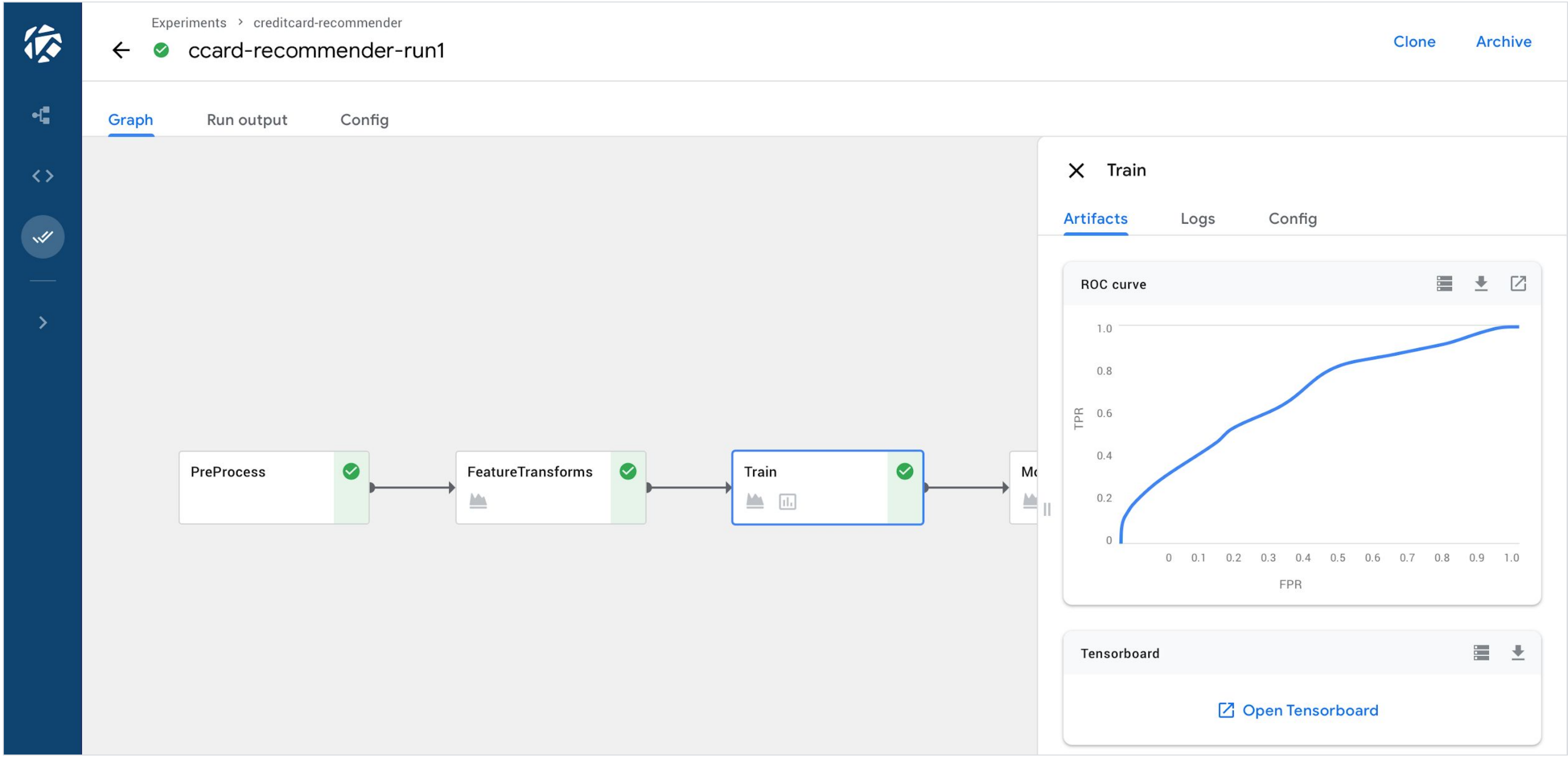
- A “Job” is a pipeline invoked w/specific parameters



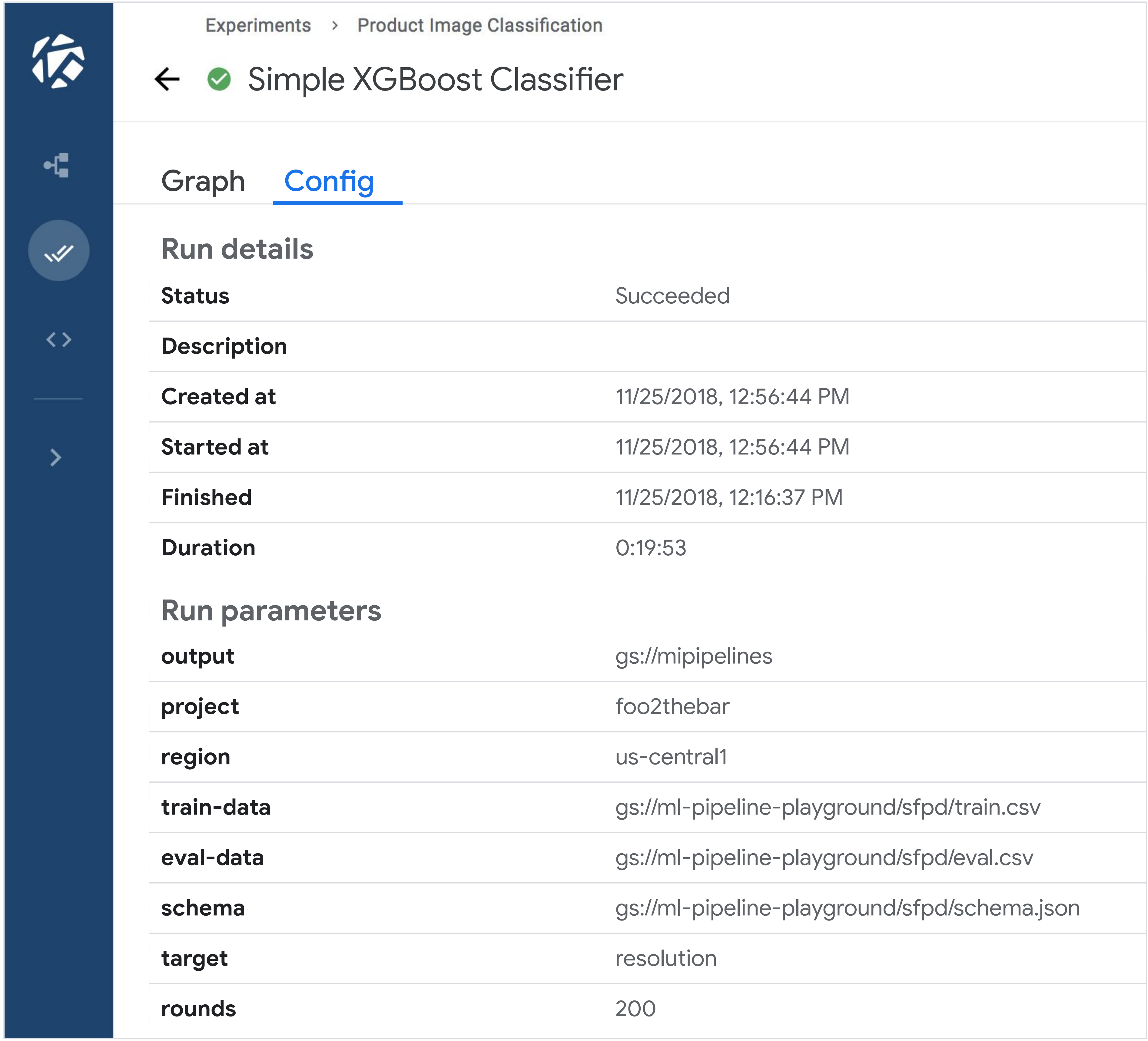
# Visual depiction of pipeline topology



# 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

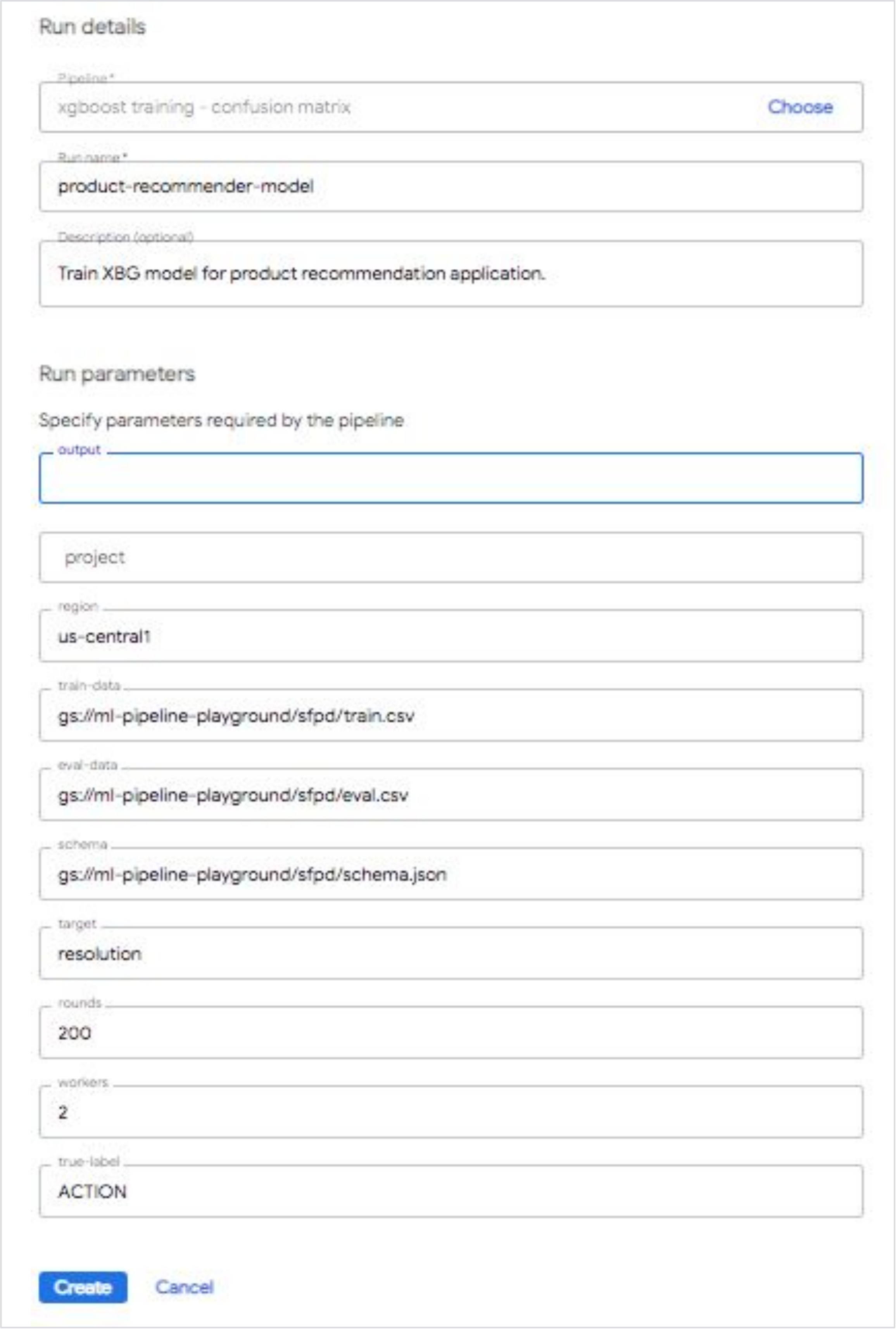
In [16]:

```
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 as zip files

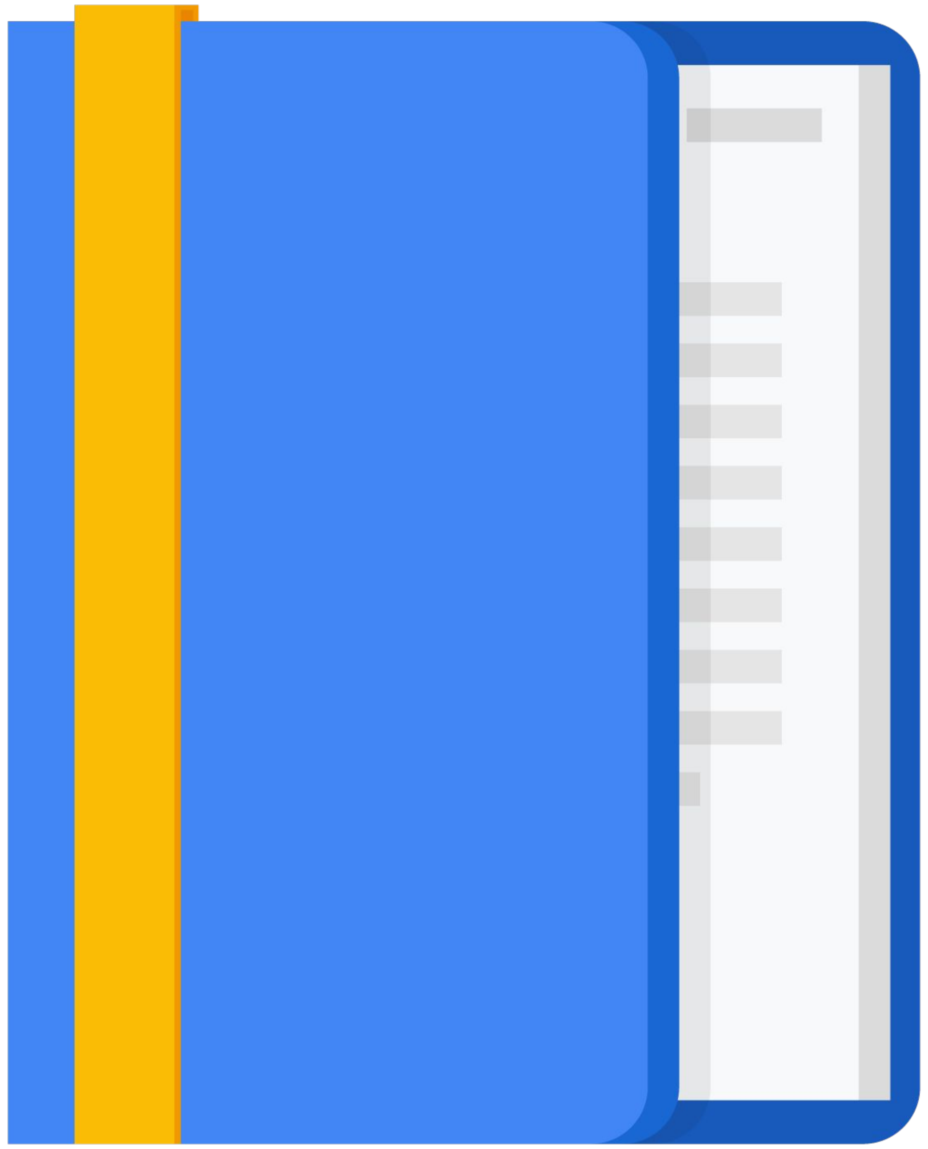
- Upload and execute pipelines via UI (in addition to API/SDK).

- Pipeline steps can be authored as reusable components.



# Agenda

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



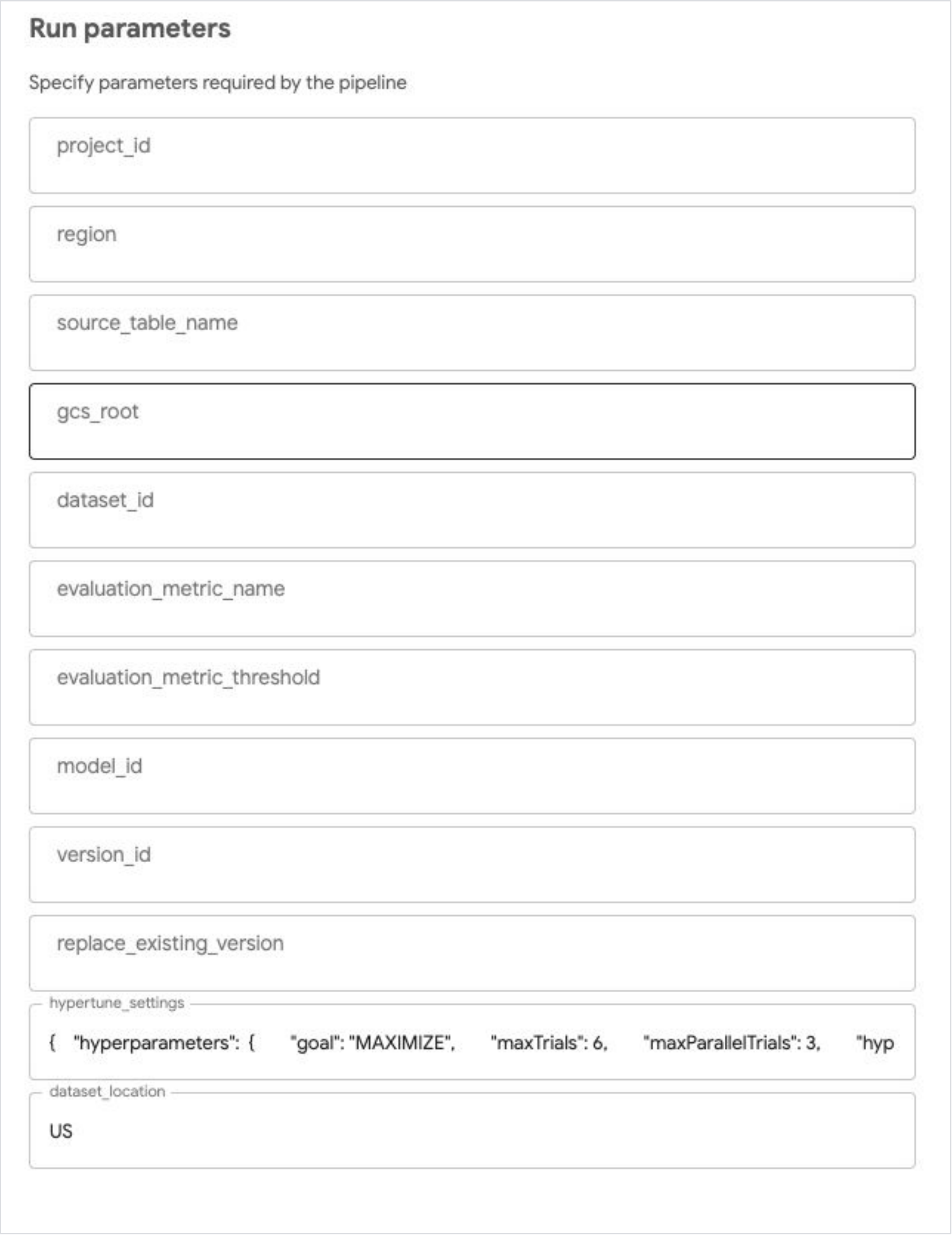
Kubeflow offers a Domain Specific Language (DSL) in Python that allows you to use Python code to describe Kubeflow tasks as they organize themselves in a Directed Acyclic Graph (DAG).

We describe this DSL next...

```
import kfp

@kfp.dsl.pipeline(
    name='Covertype Classifier Training',
    description='Covertype training and deployment pipeline’,
)
def covertype_train(project_id,
                    region,
                    source_table_name,
                    gcs_root,
                    dataset_id,
                    evaluation_metric_name,
                    evaluation_metric_threshold,
                    model_id,
                    version_id,
                    replace_existing_version,
                    hypertune_settings=HYPERTUNE_SETTINGS,
                    dataset_location='US'):
```

Pipeline Decorator

Pipeline Run Parameters




```
def covertype_train(project_id,
                    region,
                    source_table_name,
                    gcs_root,
                    dataset_id,
                    evaluation_metric_name,

evaluation_metric_threshold,
                    model_id,
                    version_id,
                    replace_existing_version,

hypertune_settings=HYPERTUNE_SETTINGS,
                    dataset_location='US'):
```

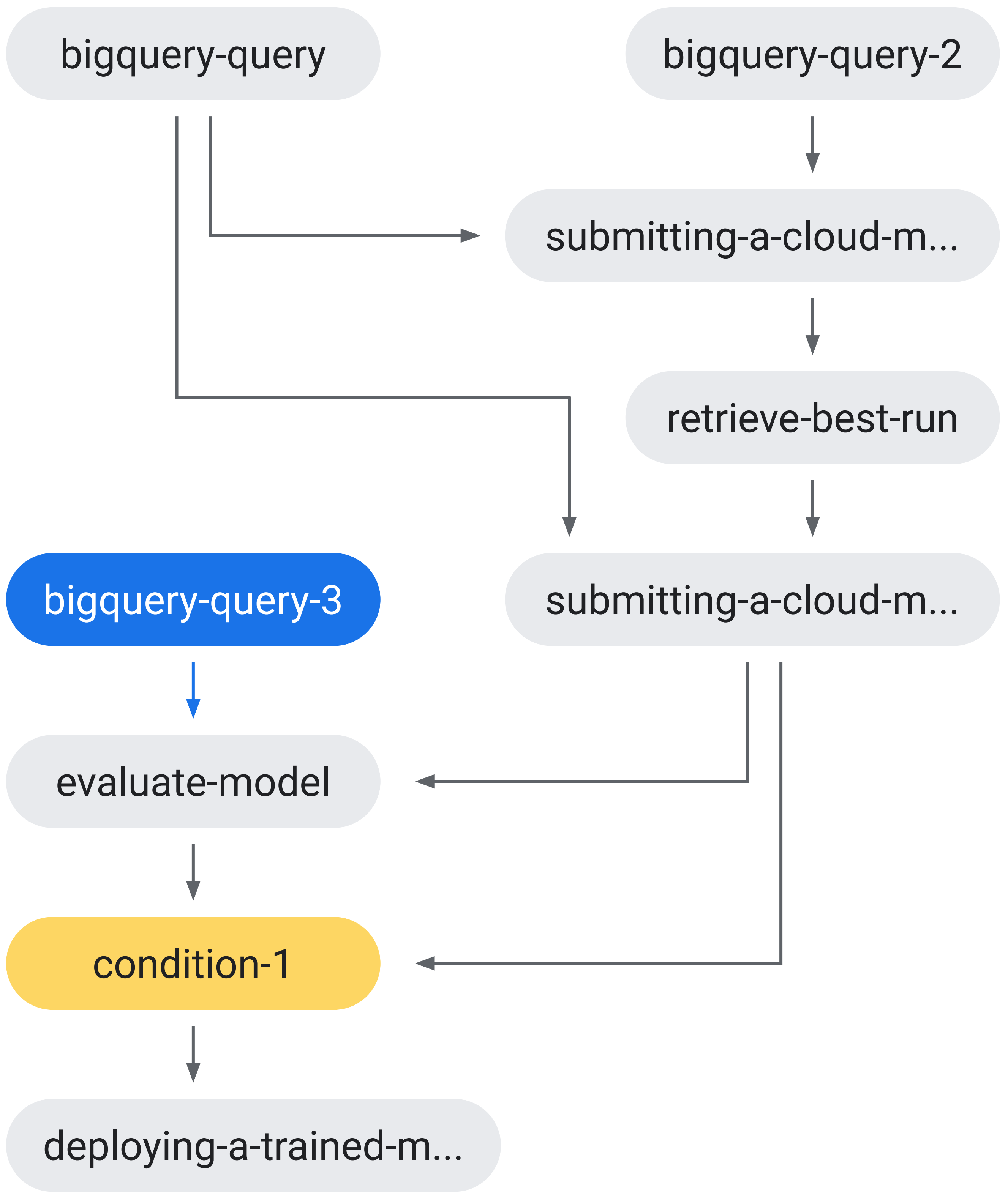
The Run Parameters are supplied at run time.

# 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

```
train_model = mlengine_train_op(
        project_id=project_id,
        region=region,
        master_image_uri=TRAINER_IMAGE,
        job_dir=job_dir,
        args=train_args)

eval_model = evaluate_model_op(
        dataset_path=str(create_testing_split.outputs['output_gcs_path']),
        model_path=str(train_model.outputs['job_dir']),
        metric_name=evaluation_metric_name)
```

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

    with kfp.dsl.Condition(eval_model.outputs['metric_value'] >
evaluation_metric_threshold):
        deploy_model = mlengine_deploy_op(
        model_uri=train_model.outputs['job_dir'],
        project_id=project_id,
        model_id=model_id,
        version_id=version_id,
        runtime_version=RUNTIME_VERSION,
        python_version=PYTHON_VERSION,
        replace_existing_version=replace_existing_version)
```

# 3 main types of Kubeflow components we will look at

01 Pre-built components

- Just load the component from its description and compose.

02 Lightweight Python components

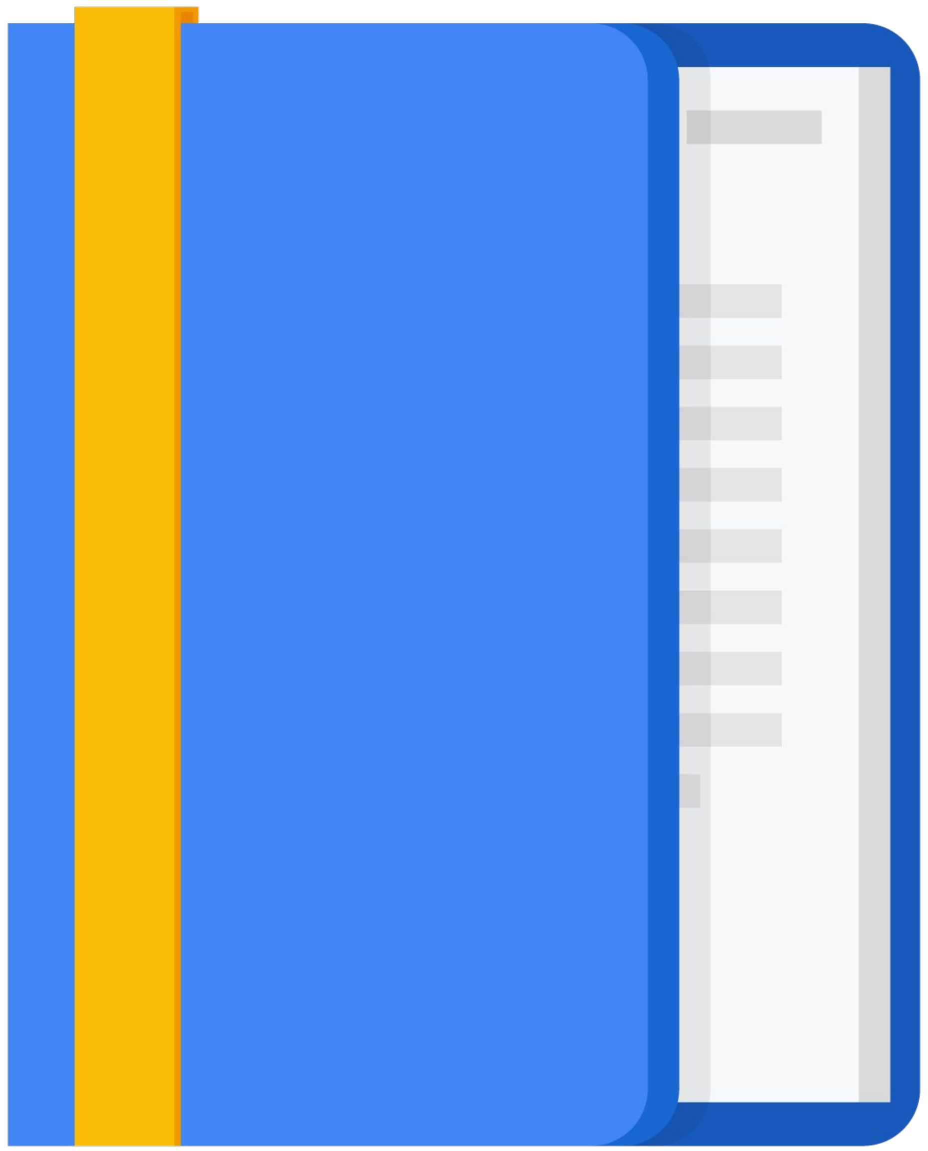
- Implement the component code.

03 Custom components

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

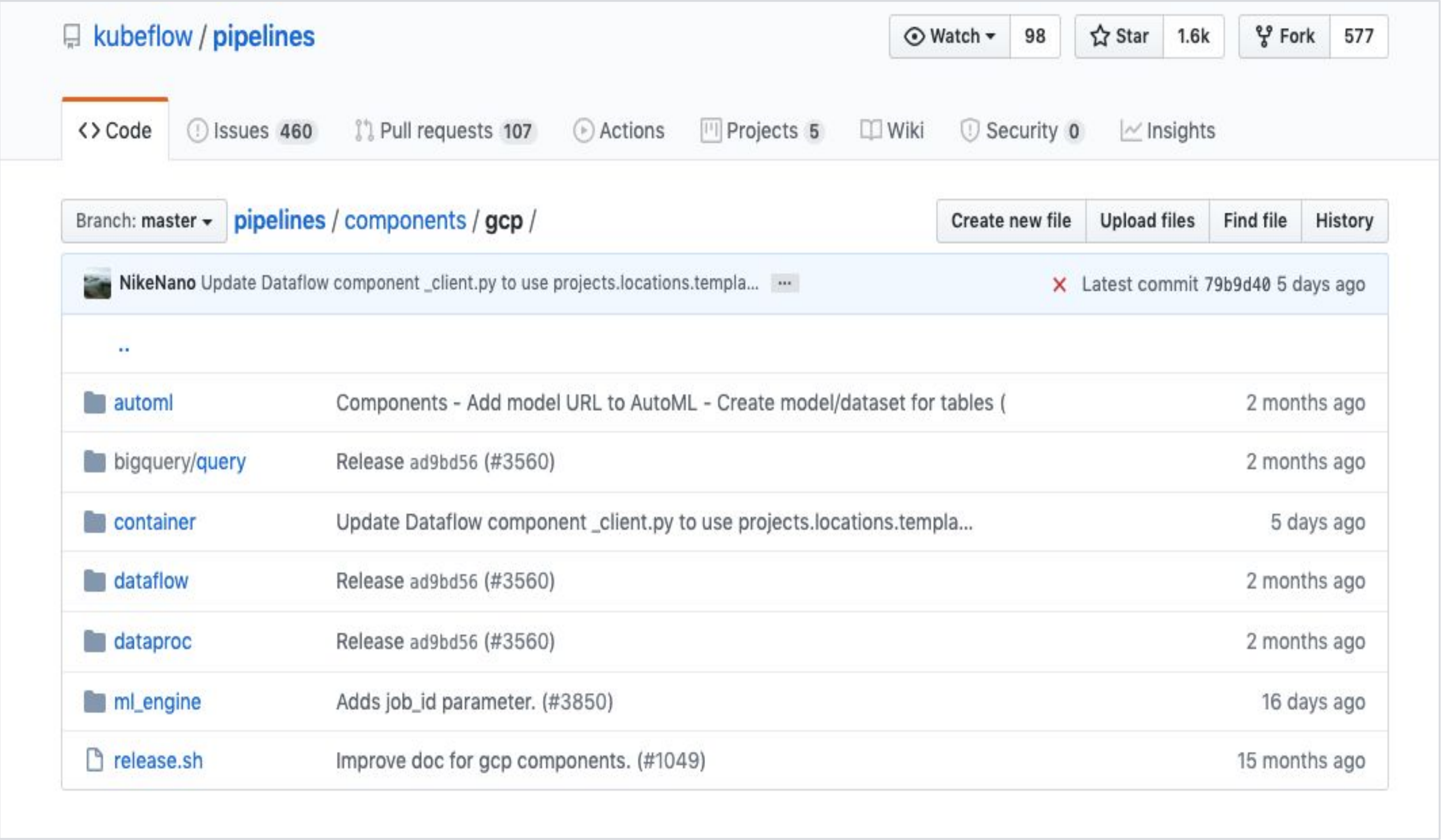
# Agenda

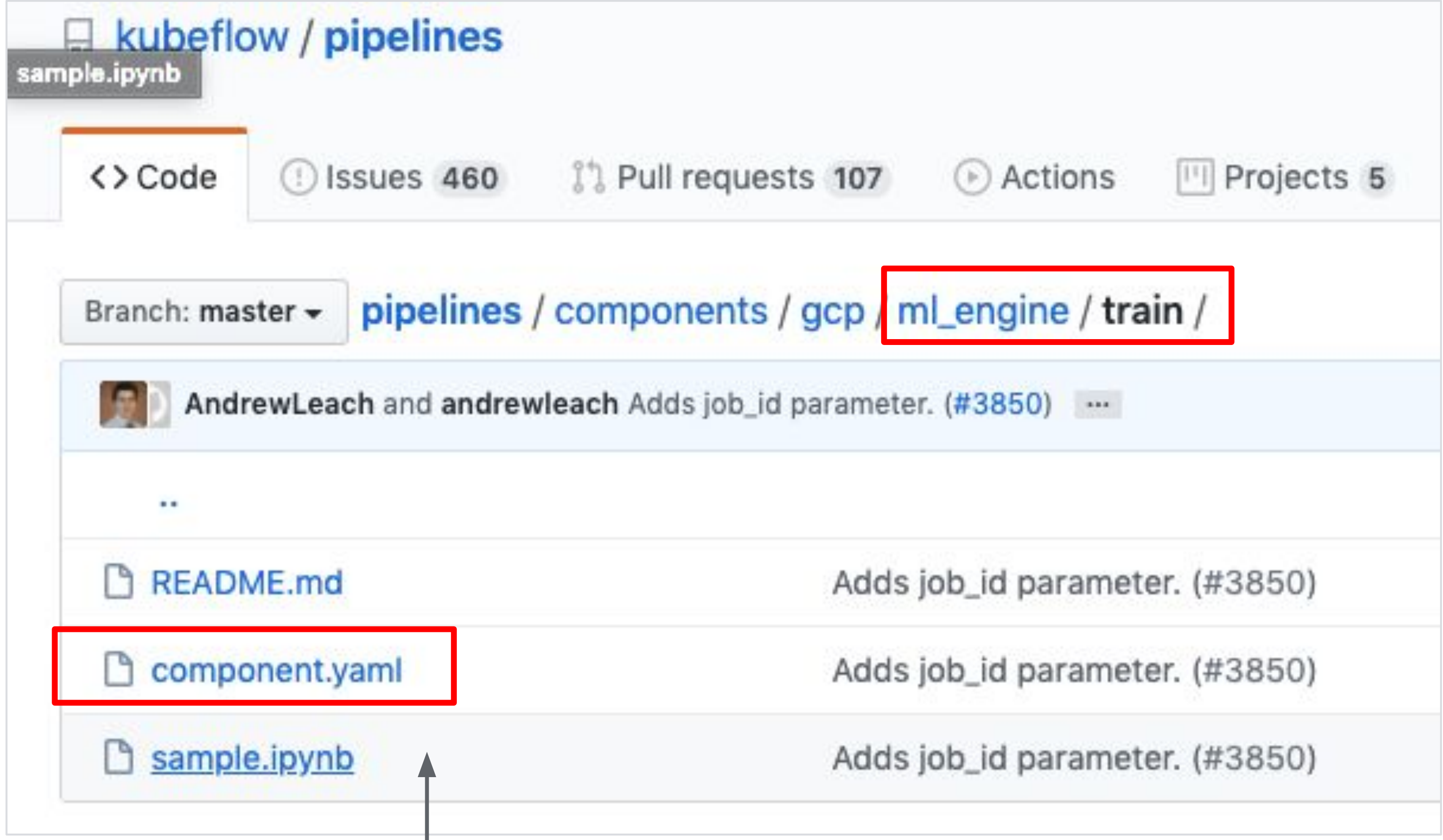
- System and Concept Overview
- Describing a Kubeflow Pipeline with KF DSL
- Pre-built Components
- Lightweight Python Components
- Custom Components
- Compile, Upload, and Run



# Github repo for pre-built Kubeflow components

https://github.com/kubeflow/pipelines/blob/master/components




Component description

# component.yaml

```
implementation:
  container:
    image: gcr.io/ml-pipeline/ml-pipeline-gcp:ad9bd5648dd0453005225779f25d8cebebc7ca00
    args: [
      --ui_metadata_path, {outputPath: MLPipeline UI metadata},
      kfp_component.google.ml_engine, train,
      --project_id, {inputValue: project_id},
      --python_module, {inputValue: python_module},
      --package_uris, {inputValue: package_uris},
      --region, {inputValue: region},
      --args, {inputValue: args},
      --job_dir, {inputValue: job_dir},
      --python_version, {inputValue: python_version},
      --runtime_version, {inputValue: runtime_version},
      --master_image_uri, {inputValue: master_image_uri},
      --worker_image_uri, {inputValue: worker_image_uri},
      --training_input, {inputValue: training_input},
      --job_id_prefix, {inputValue: job_id_prefix},
      --job_id, {inputValue: job_id},
      --wait_interval, {inputValue: wait_interval},
    ]
    env:
      KFP_POD_NAME: "{{pod.name}}"
    fileOutputs:
```

Container image URI

Run parameters

# Loading a pre-built component

```
import kfp

URI = 'https://raw.githubusercontent.com/kubeflow/pipelines/0.2.5/components/gcp/'

component_store = kfp.components.ComponentStore(
    local_search_paths=None, url_search_prefixes=[URI])

bigquery_query_op = component_store.load_component('bigquery/query')
mlengine_train_op = component_store.load_component('ml_engine/train')
mlengine_deploy_op = component_store.load_component('ml_engine/deploy')
```

# Using pre-built bigquery/query

```
create_training_split = bigquery_query_op(
    query=query,
    project_id=project_id,
    dataset_id=dataset_id,
    table_id='',
    output_gcs_path=training_file_path,
    dataset_location=dataset_location)
```

Runtime arguments:

| Argument | Description | Optional | Data type | Accepted values | Default |
|---|---|---|---|---|---|
| query | The query used by BigQuery to fetch the results. | No | String | | |
| project_id | The project ID of the Google Cloud Platform (GCP) project to use to execute the query. | No | GCPProjectID | | |
| dataset_id | The ID of the persistent BigQuery dataset to store the results of the query. If the dataset does not exist, the operation will create a new one. | Yes | String | | None |

Output:

| Name | Description | Type |
|---|---|---|
| output_gcs_path | The path to the Cloud Storage bucket containing the query output in CSV format. | GCSPath |

https://github.com/kubeflow/pipelines/tree/master/components/gcp/bigquery/query

# Using pre-built ml_engine/train

```
train_model = mlengine_train_op(
    project_id=project_id,
    region=region,
    master_image_uri=TRAINER_IMAGE,
    job_dir=job_dir,
    args=train_args)
```

Runtime arguments:

| Argument | Description | Optional | Data type | Accepted values | Default |
|---|---|---|---|---|---|
| project_id | The Google Cloud Platform (GCP) project ID of the job. | No | GCPProjectID | - | - |
| python_module | The name of the Python module to run after installing the training program. | Yes | String | - | None |
| package_uris | The Cloud Storage location of the packages that contain the training program and any additional dependencies. The maximum number of package URIs is 100. | Yes | List | - | None |

Output:

| Name | Description | Type |
|---|---|---|
| job_id | The ID of the created job. | String |
| job_dir | The Cloud Storage path that contains the output files with the trained model. | GCSPath |

https://github.com/kubeflow/pipelines/tree/master/components/gcp/ml_engine/train

# Using pre-built ml_engine/deploy

```
deploy_model = mlengine_deploy_op(

model_uri=train_model.outputs['job_dir'],
    project_id=project_id,
    model_id=model_id,
    version_id=version_id,
    runtime_version=RUNTIME_VERSION,
    python_version=PYTHON_VERSION,

replace_existing_version=replace_existing_ve
rsion)
```

https://github.com/kubeflow/pipelines/tree/master/components/gcp/ml_engine/deploy

Runtime arguments:

| Argument | Description | Optional | Data type | Accepted values | Default |
|---|---|---|---|---|---|
| model_uri | The URI of a Cloud Storage directory that contains a trained model file.<br>Or<br>An Estimator export base directory that contains a list of subdirectories named by timestamp. The directory with the latest timestamp is used to load the trained model file. | No | GCSPath | | |
| project_id | The ID of the Google Cloud Platform (GCP) project of the serving model. | No | GCPProjectID | | |
| model_id | The name of the trained model. | Yes | String | | None |
| version_id | The name of the version of the model. If it is not provided, the operation uses a random name. | Yes | String | | None |

Output:

| Name | Description | Type |
|---|---|---|
| model_uri | The Cloud Storage URI of the trained model. | GCSPath |
| model_name | The name of the deployed model. | String |
| version_name | The name of the deployed version. | String |

# Composing pre-built components: Hyper tuning

```
tune_args = [
    '--training_dataset_path',
create_training_split.outputs['output_gcs_path'],
    '--validation_dataset_path',
create_validation_split.outputs['output_gcs_path'],
    '--hptune', 'True'
]

hypertune = mlengine_train_op(
    project_id=project_id,
    region=region,
    master_image_uri=TRAINER_IMAGE,
    job_dir=job_dir,
    args=tune_args,
    training_input=HYPERTUNE_SETTINGS)
```

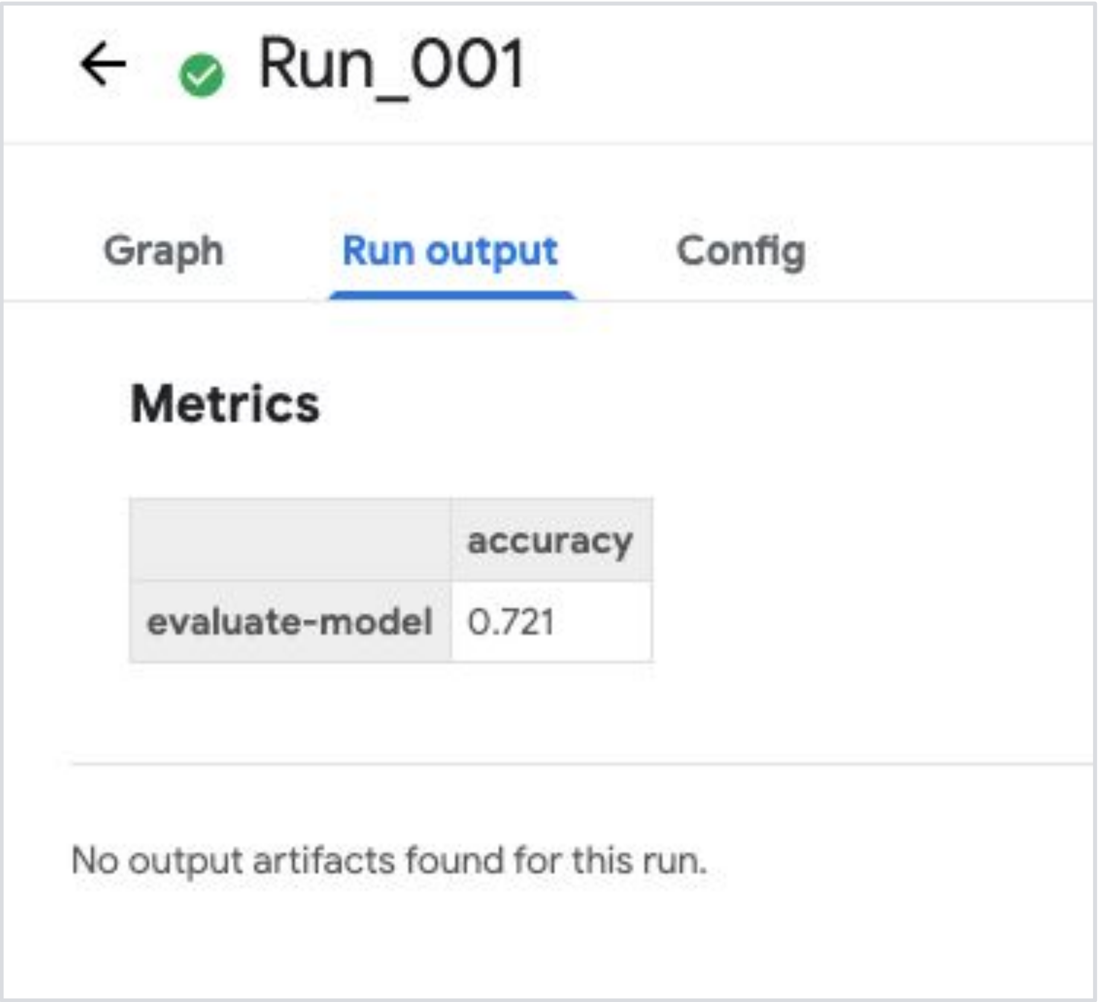
# Composing pre-built components: Hypertuning

```
HYPERTUNE_SETTINGS = """
{
    "hyperparameters":  {
        "goal": "MAXIMIZE",
        "maxTrials": 6,
        "maxParallelTrials": 3,
        "hyperparameterMetricTag": "accuracy",
        "enableTrialEarlyStopping": True,
        "params": [
            {
                "parameterName": "max_iter",
                "type": "DISCRETE",
                "discreteValues": [500, 1000]
            },
         etc.
        ]
    }
}
"""
```

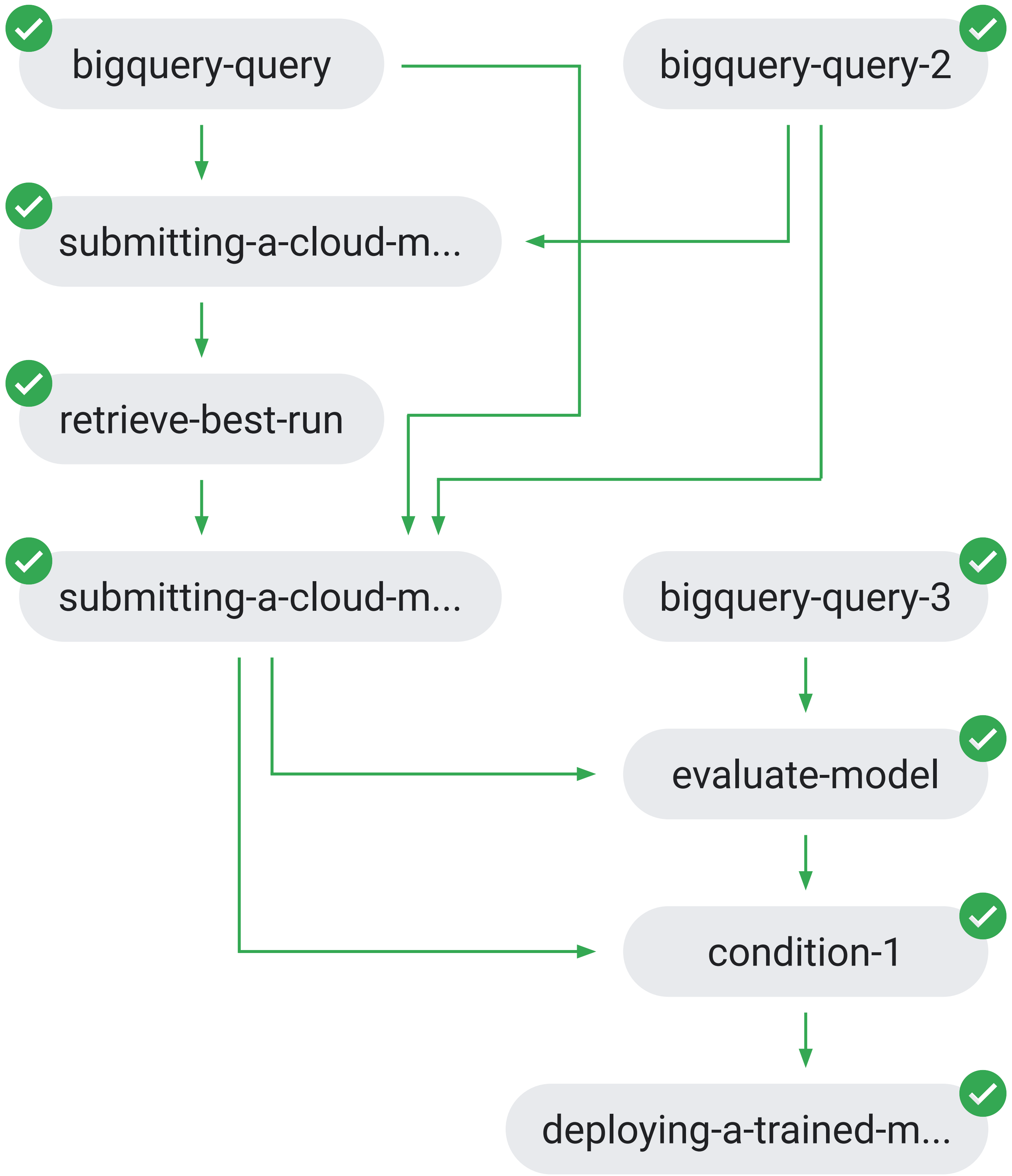
# Composing pre-built components: Training best run

```
train_args = [
    '--training_dataset_path', create_training_split.outputs['output_gcs_path'],
    '--validation_dataset_path', create_validation_split.outputs['output_gcs_path'],
    '--alpha', get_best_trial.outputs['alpha'],
    '--max_iter', get_best_trial.outputs['max_iter'],
    '--hptune', 'False'
]

train_model = mlengine_train_op(
    project_id=project_id,
    region=region,
    master_image_uri=TRAINER_IMAGE,
    job_dir=job_dir,
    args=train_args,
)
```

Components composed through their input/output



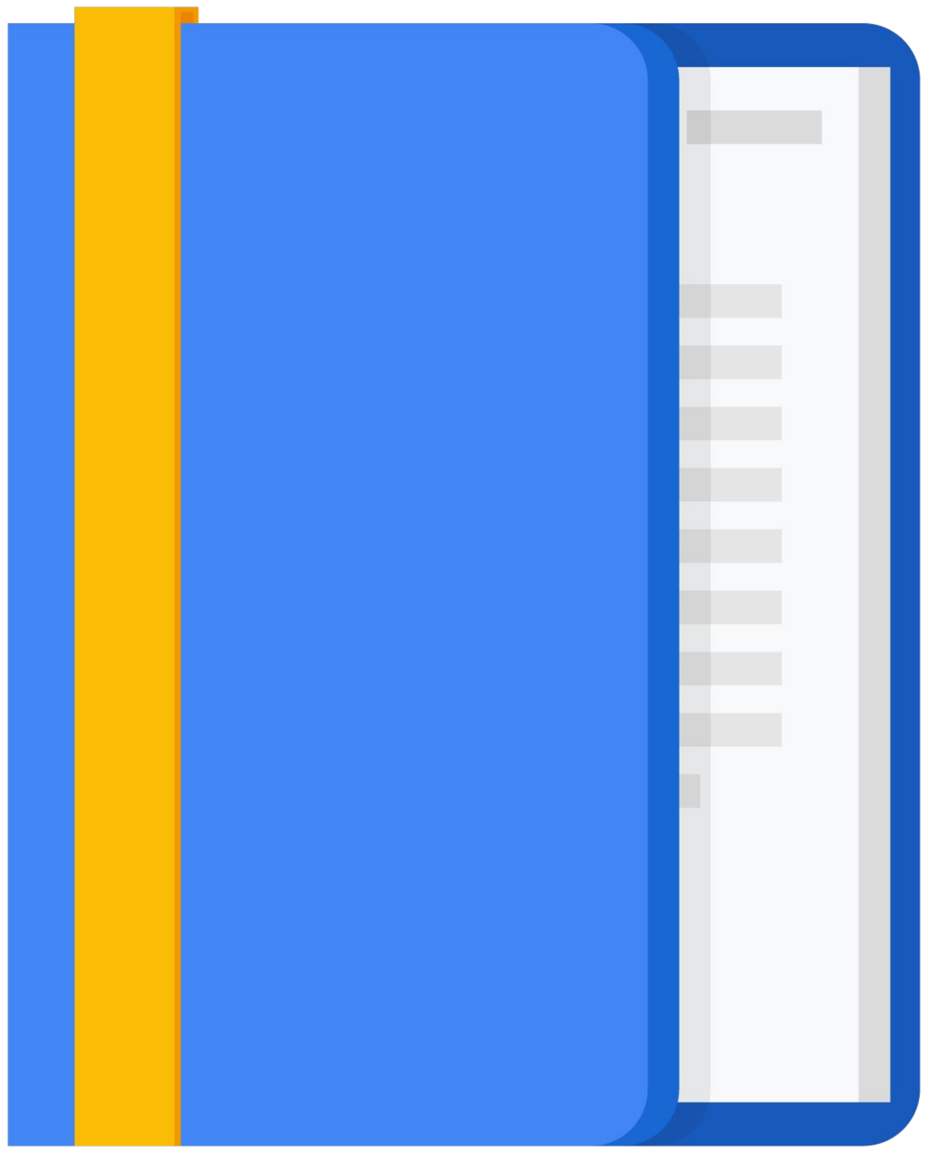
Run_001
Graph
Run output
Config
Metrics
accuracy
evaluate-model
0.721
No output artifacts found for this run.

bigquery-query
bigquery-query-2
submitting-a-cloud-m...
retrieve-best-run
submitting-a-cloud-m...
bigquery-query-3
evaluate-model
condition-1
deploying-a-trained-m...

# Agenda

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



# Wrap Python functions into KF components

helper_components.py

```
def retrieve_best_run(project_id, job_id):
    """Retrieves the parameters of the best Hypertune run."""
    # [...]
    return (metric_value, alpha, max_iter)


def evaluate_model(dataset_path, model_path, metric_name):
    """Evaluates a trained sklearn model."""
    # [...]
    return (metric_name, metric_value, json.dumps(metrics))
```

# func_to_container_op

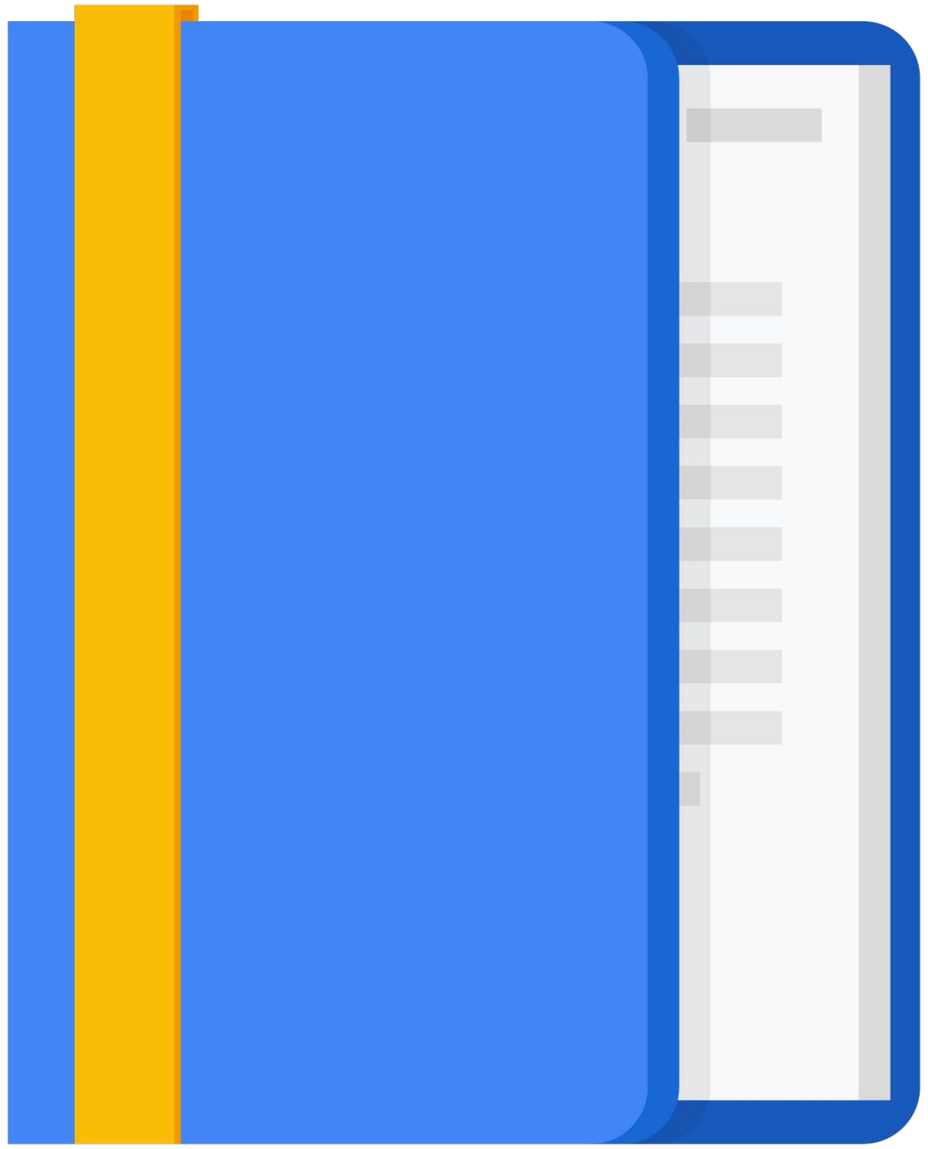
```
from helper_components import evaluate_model
from helper_components import retrieve_best_run

from kfp.components import func_to_container_op


retrieve_best_run_op = func_to_container_op(
    retrieve_best_run, base_image=BASE_IMAGE)

evaluate_model_op = func_to_container_op(
    evaluate_model, base_image=BASE_IMAGE)
```

# Use and compose the lightweight components as usual

```
get_best_trial = retrieve_best_run_op(
    project_id, hypertune.outputs['job_id'])


eval_model = evaluate_model_op(
    dataset_path=str(create_testing_split.outputs['output_gcs_path']),
    model_path=str(train_model.outputs['job_dir']),
    metric_name=evaluation_metric_name)
```

# Agenda

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



# Custom components

**Steps**

1. Write your own code (in any language).
2. Write a Dockerfile to package it into a Docker container.
3. Build and push your Docker container image to `gcr/io`.
4. Describe your component in a yaml description file.
5. Use the description file to load the component into the pipeline.

# Step 1: Write your custom code (example)

main.py

```
from google.cloud import bigquery
from google.cloud.bigquery.job import ExtractJobConfig

if __name__ == "__main__":

    bq = bigquery.Client()

    dataset_ref = bigquery.Dataset(bq.dataset("taxifare"))
    bq.create_dataset(dataset_ref)

    bq.query(TRAIN_SQL).result()
    bq.query(VALID_SQL).result()

    export_table_to_gcs(dataset_ref, TRAIN_TABLE, train_export_path)
```

# Step 2: Package it into a Docker container

Dockerfile

```
FROM google/cloud-sdk:latest

RUN apt-get update && \
    apt-get install --yes python3-pip

COPY . /code
WORKDIR /code

RUN pip3 install google-cloud-bigquery



ENTRYPOINT ["python3", "./main.py"]
```

# Step 3: Write the component description

bq2gcs.yaml

```
name: bq2gcs

description: |
    This component creates the training and
    validation datasets as BiqQuery tables and exports
    them into Cloud Storage at gs://<BUCKET>/taxifare/data.

inputs:
    - {name: Input Bucket , type: String, description: 'GCS directory path.'}

implementation:
    container:
        image: gcr.io/<YOUR PROJECT>/taxifare-bq2gcs
        args: ["--bucket", {inputValue: Input Bucket}]
```

# Step 4: Load the component into the pipeline

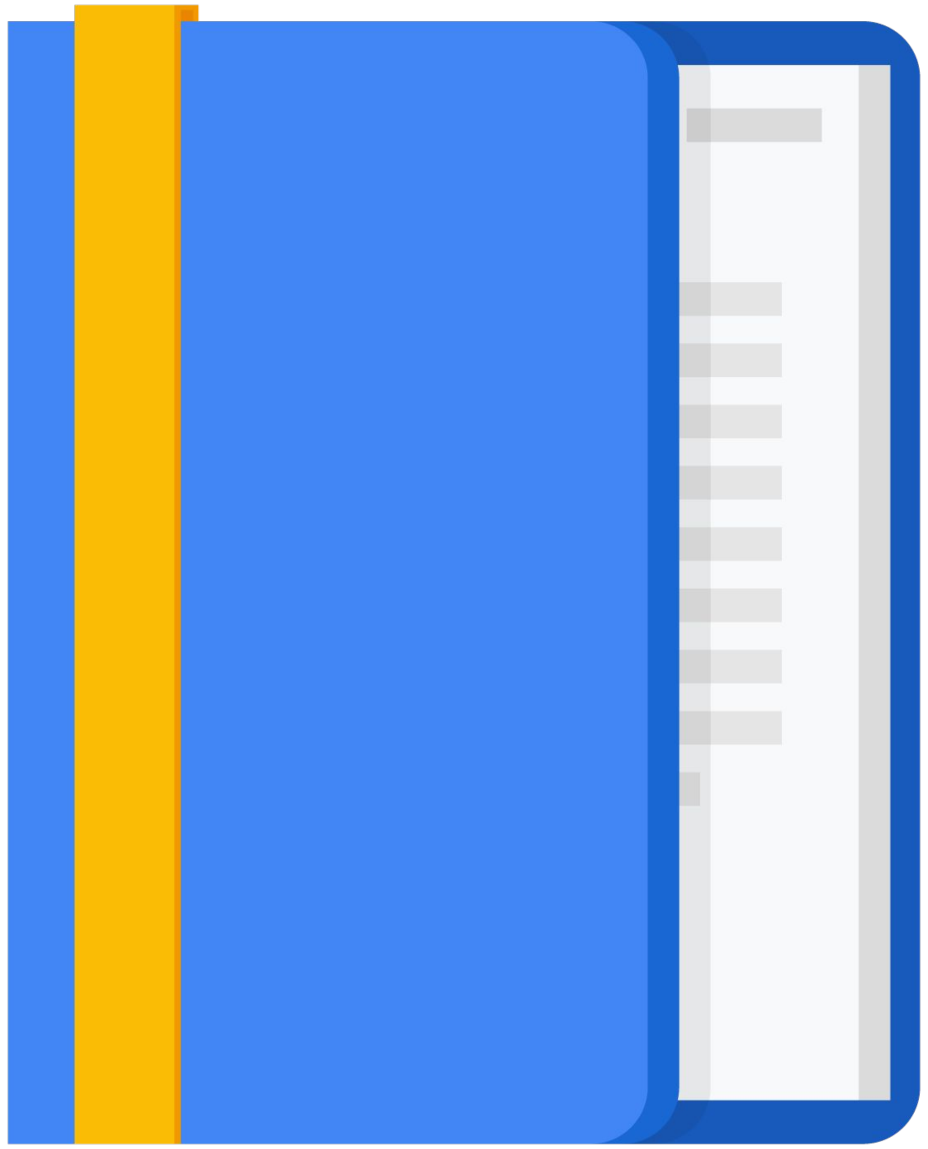
```
@dsl.pipeline(
    name='Taxifare',
    description='Train an Ml model to predict the taxi fare in NY')
def pipeline(gcs_bucket_name='<bucket where data and model will be exported>'):

    bq2gcs_op = comp.load_component_from_file(BQ2GCS_YAML)
    bq2gcs = bq2gcs_op(
        input_bucket=gcs_bucket_name,
    )

   [...]
```

# Run the pipeline with custom components!

```
import kfp

client = kfp.Client(host=HOST)

client.list_experiments()

exp = client.create_experiment(name='taxifare')

kfp.compiler.Compiler().compile(pipeline, PIPELINE_TAR)

run = client.run_pipeline(
    experiment_id=exp.id,
    job_name='taxifare',
    pipeline_package_path='taxifare.tar.gz',
    params={
        'gcs_bucket_name': BUCKET,
    },
)
```

# Agenda

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



# Step 1: Build and push the trainer container

trainer_image/Dockerfile

Required by ml_engine/train

```
FROM gcr.io/deeplearning-platform-release/base-cpu
RUN pip install -U fire cloudml-hypertune scikit-learn==0.20.4 pandas==0.24.2

WORKDIR /app

COPY train.py .

ENTRYPOINT ["python", "train.py"]
```

```
TRAINER_IMAGE='gcr.io/PROJECT_ID/TRAINER_IMAGE_NAME:TAG'

gcloud builds submit --timeout 15m --tag $TRAINER_IMAGE trainer_image
```

# Step 2: Build and push the base container

base_image/Dockerfile

Required by the Python lightweight ops

```
FROM gcr.io/deeplearning-platform-release/base-cpu

RUN pip install -U fire scikit-learn==0.20.4 pandas==0.24.2 kfp==0.2.5
```

```
BASE_IMAGE='gcr.io/PROJECT_ID/BASE_IMAGE_NAME:TAG'

gcloud builds submit --timeout 15m --tag $BASE_IMAGE base_image
```

# Step 3: Compile the Kubeflow pipeline

dsl-compile --py pipeline/covertype_training_pipeline.py --outputcovertype_training_pipeline.yaml

```
[16]: !head covertype_training_pipeline.yaml
```


```
"apiVersion": |-
  argoproj.io/v1alpha1
"kind": |-
  Workflow
"metadata":
  "annotations":
    "pipelines.kubeflow.org/pipeline_spec": |-
      {"description": "The pipeline training and deploying the Covertype classifierpipeline_yaml", "input
s": [{"name": "project_id"}, {"name": "region"}, {"name": "source_table_name"}, {"name": "gcs_root"}, {"na
me": "dataset_id"}, {"name": "evaluation_metric_name"}, {"name": "evaluation_metric_threshold"}, {"name":
"model_id"}, {"name": "version_id"}, {"name": "replace_existing_version"}, {"default": "\n{\n    \"hyperpa
rameters\":  {\n        \"goal\": \"MAXIMIZE\",\n        \"maxTrials\": 6,\n        \"maxParallelTrials\":
3,\n        \"hyperparameterMetricTag\": \"accuracy\",\n        \"enableTrialEarlyStopping\": True,\n
\"params\": [\n            {\n                \"parameterName\": \"max_iter\",\n                \"type\":
\"DISCRETE\",\n                \"discreteValues\": [500, 1000]\n            },\n            {\n
\"parameterName\": \"alpha\",\n                \"type\": \"DOUBLE\",\n                \"minValue\": 0.000
1,\n                \"maxValue\": 0.001,\n                \"scaleType\": \"UNIT_LINEAR_SCALE\"\n
}\n        ]\n    }\n}\n", "name": "hypertune_settings", "optional": true}, {"default": "US", "name": "dat
aset_location", "optional": true}], "name": "Covertype Classifier Training"}
  "generateName": |-
    covertype-classifier-training-
```

# Step 4: Upload the pipeline to the KF cluster

```
kfp --endpoint $ENDPOINT pipeline upload -p $PIPELINE_NAME \
covertype_training_pipeline.yaml
```

```
kfp --endpoint $ENDPOINT pipeline list
```

# Step 5: Run the pipeline

```
kfp --endpoint $ENDPOINT run submit \
    -e $EXPERIMENT_NAME \
    -r $RUN_ID \
    -p $PIPELINE_ID \
    project_id=$PROJECT_ID \
    gcs_root=$GCS_STAGING_PATH \
    region=$REGION \
    source_table_name=$SOURCE_TABLE \
    dataset_id=$DATASET_ID \
    evaluation_metric_name=$EVALUATION_METRIC \
    evaluation_metric_threshold=$EVALUATION_METRIC_THRESHOLD \
    model_id=$MODEL_ID \
    version_id=$VERSION_ID \
    replace_existing_version=$REPLACE_EXISTING_VERSION
```

Run parameters

# Lab

## Kubeflow Pipelines on CAIP

In this lab, you will learn how to use AI Platform Pipelines to build a
Kubeflow pipeline to train, tune, and serve a model automatically.

https://github.com/GoogleCloudPlatform/mlops-on-gcp/blob/master/workshops/kfp-caip-sklearn/lab-02-kfp-pipeline/exercises/lab-02.ipynb cloud.google.com