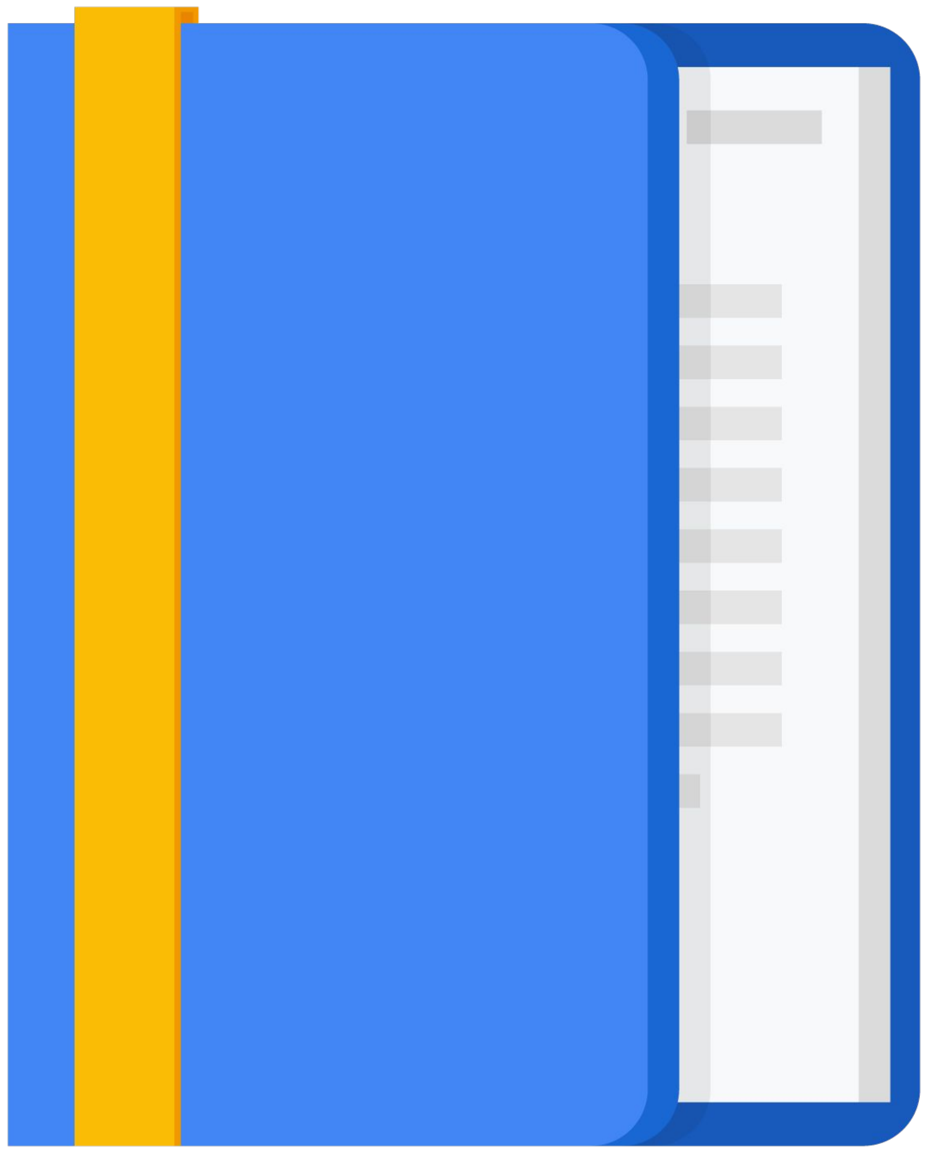
Custom components and CI/CD for TFX pipelines

Doug Kelly
ML Solutions Engineer, Google Cloud

Agenda

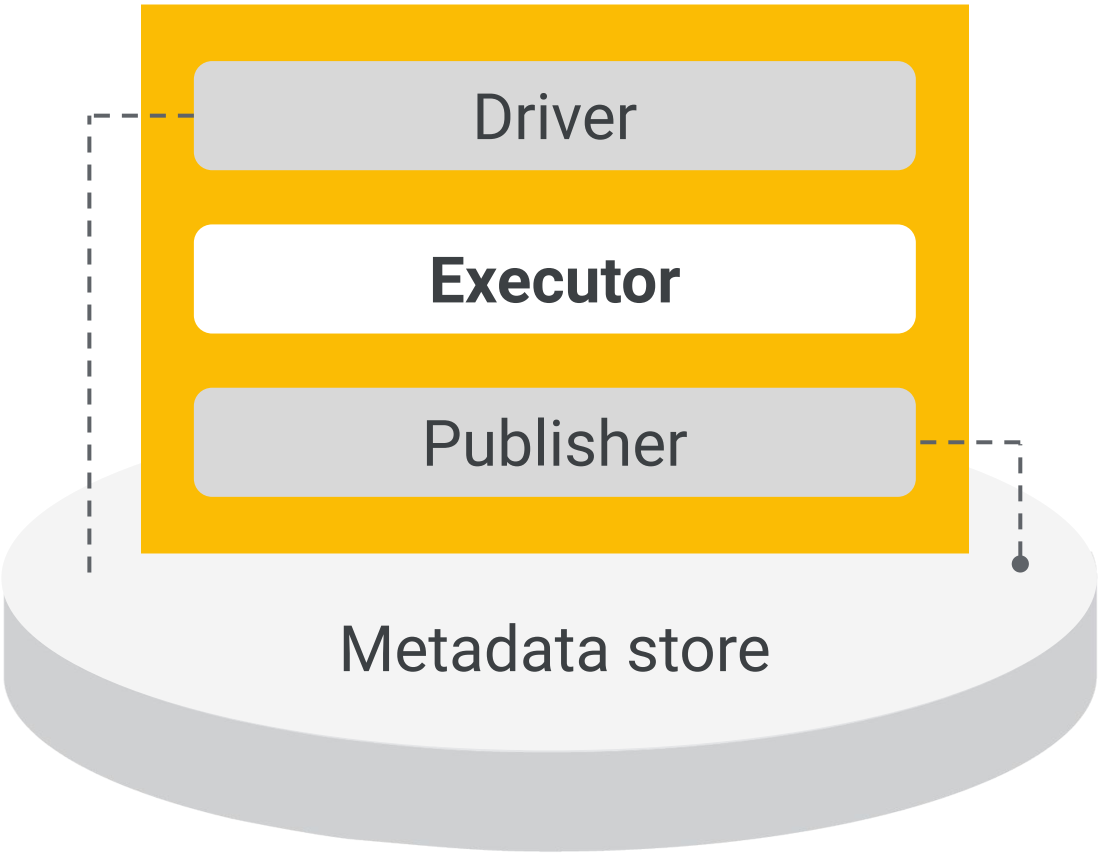
TFX custom components

CI/CD for TFX pipeline workflows



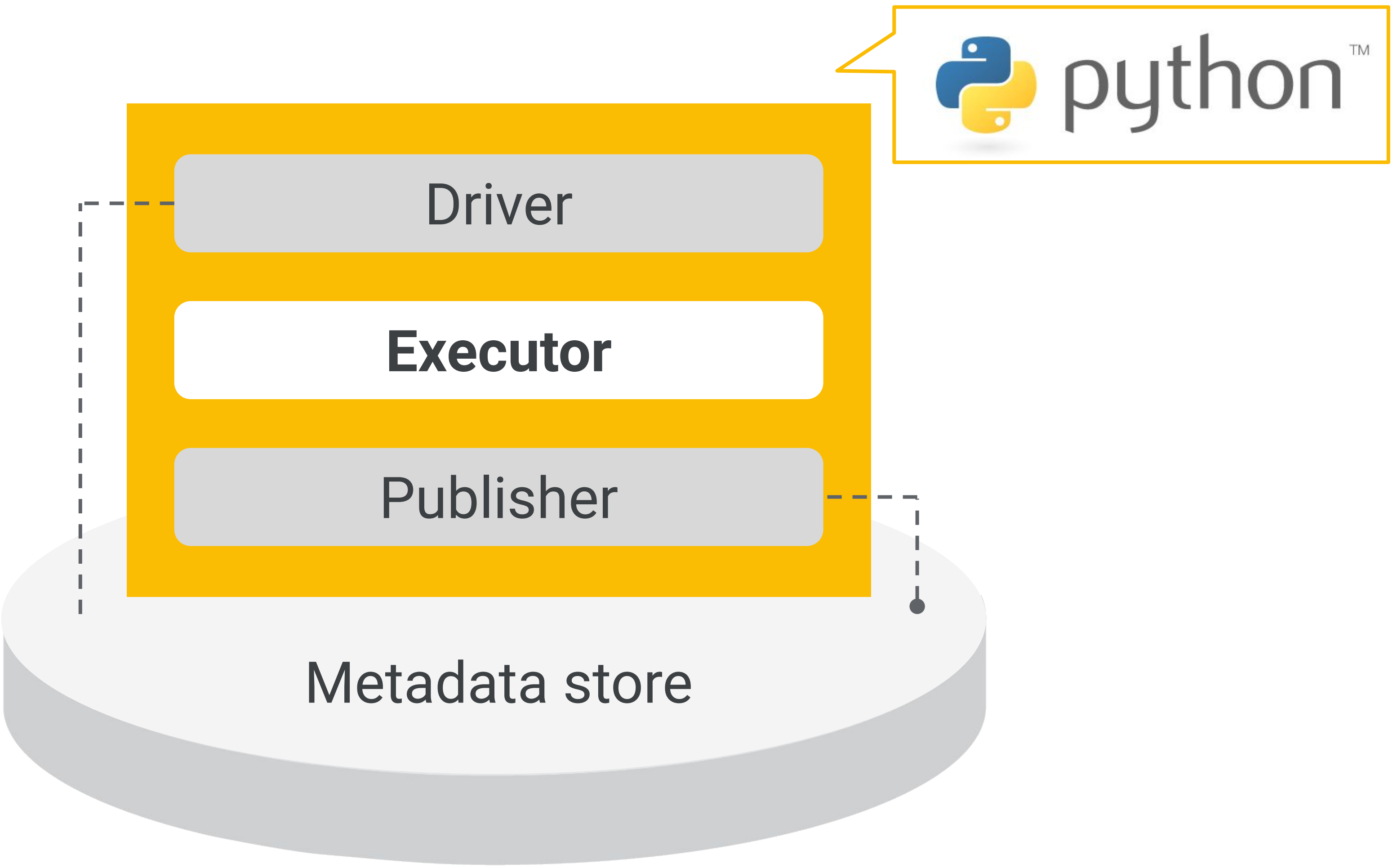
Create new custom components

1. Python functions
2. Containers
3. Extending existing component classes



Option 1: Python function

Create custom components faster from Python functions, decorators, and type annotations.



Option 1: Python function

```
from tfx.dsl.component.experimental.annotations import OutputDict
from tfx.dsl.component.experimental.annotations import InputArtifact
from tfx.dsl.component.experimental.annotations import OutputArtifact
from tfx.dsl.component.experimental.annotations import Parameter
from tfx.dsl.component.experimental.decorators import component
from tfx.types.standard_artifacts import Examples
from tfx.types.standard_artifacts import Model

@component
def MyTrainerComponent(
    training_data: InputArtifact[Examples],
    model: OutputArtifact[Model],
    dropout_hyperparameter: float,
    num_iterations: Parameter[int] = 10
    ) -> OutputDict(loss=float, accuracy=float):
  '''My simple trainer component.'''

  records = read_examples(training_data.uri)
  model_obj = train_model(records, num_iterations, dropout_hyperparameter)
  model_obj.write_to(model.uri)

  return {
    'loss': model_obj.loss,
    'accuracy': model_obj.accuracy}
```

```
# ...
trainer = MyTrainerComponent(
    examples=example_gen.outputs['examples'],
    dropout_hyperparameter=other_component.outputs['dropout'],
    num_iterations=1000)
pusher = Pusher(model=trainer.outputs['model'])
# ...
```

Step 1: apply @component decorator to your Python function.

Option 1: Python function

```
from tfx.dsl.component.experimental.annotations import OutputDict
from tfx.dsl.component.experimental.annotations import InputArtifact
from tfx.dsl.component.experimental.annotations import OutputArtifact
from tfx.dsl.component.experimental.annotations import Parameter
from tfx.dsl.component.experimental.decorators import component
from tfx.types.standard_artifacts import Examples
from tfx.types.standard_artifacts import Model

@component
def MyTrainerComponent(
    training_data: InputArtifact[Examples],
    model: OutputArtifact[Model],
    dropout_hyperparameter: float,
    num_iterations: Parameter[int] = 10
    ) -> OutputDict(loss=float, accuracy=float):
  '''My simple trainer component.'''

  records = read_examples(training_data.uri)
  model_obj = train_model(records, num_iterations, dropout_hyperparameter)
  model_obj.write_to(model.uri)

  return {
    'loss': model_obj.loss,
    'accuracy': model_obj.accuracy}
```

```
# ...
trainer = MyTrainerComponent(
    examples=example_gen.outputs['examples'],
    dropout_hyperparameter=other_component.outputs['dropout'],
    num_iterations=1000)
pusher = Pusher(model=trainer.outputs['model'])
# ...
```

Step 1: apply @component decorator to your Python function.

Step 2: annotate input artifact types.

Google Cloud

Option 1: Python function

```python
from tfx.dsl.component.experimental.annotations import OutputDict
from tfx.dsl.component.experimental.annotations import InputArtifact
from tfx.dsl.component.experimental.annotations import OutputArtifact
from tfx.dsl.component.experimental.annotations import Parameter
from tfx.dsl.component.experimental.decorators import component
from tfx.types.standard_artifacts import Examples
from tfx.types.standard_artifacts import Model

@component
def MyTrainerComponent(
    training_data: InputArtifact[Examples],
    model: OutputArtifact[Model],
    dropout_hyperparameter: float,
    num_iterations: Parameter[int] = 10
    ) -> OutputDict(loss=float, accuracy=float):
  '''My simple trainer component.'''

  records = read_examples(training_data.uri)
  model_obj = train_model(records, num_iterations, dropout_hyperparameter)
  model_obj.write_to(model.uri)

  return {
    'loss': model_obj.loss,
    'accuracy': model_obj.accuracy}
```

```python
# ...
trainer = MyTrainerComponent(
    examples=example_gen.outputs['examples'],
    dropout_hyperparameter=other_component.outputs['dropout'],
    num_iterations=1000)
pusher = Pusher(model=trainer.outputs['model'])
# ...
```

Step 1: apply @component decorator to your Python function.

Step 2: annotate input artifact types.

Step 3: annotate input parameter types.

Google Cloud

Option 1: Python function

```
from tfx.dsl.component.experimental.annotations import OutputDict
from tfx.dsl.component.experimental.annotations import InputArtifact
from tfx.dsl.component.experimental.annotations import OutputArtifact
from tfx.dsl.component.experimental.annotations import Parameter
from tfx.dsl.component.experimental.decorators import component
from tfx.types.standard_artifacts import Examples
from tfx.types.standard_artifacts import Model

@component
def MyTrainerComponent(
    training_data: InputArtifact[Examples],
    model: OutputArtifact[Model],
    dropout_hyperparameter: float,
    num_iterations: Parameter[int] = 10
    ) -> OutputDict(loss=float, accuracy=float):
  '''My simple trainer component.'''

  records = read_examples(training_data.uri)
  model_obj = train_model(records, num_iterations, dropout_hyperparameter)
  model_obj.write_to(model.uri)

  return {
    'loss': model_obj.loss,
    'accuracy': model_obj.accuracy}
```

```
# ...
trainer = MyTrainerComponent(
    examples=example_gen.outputs['examples'],
    dropout_hyperparameter=other_component.outputs['dropout'],
    num_iterations=1000)
pusher = Pusher(model=trainer.outputs['model'])
# ...
```

Step 1: apply @component decorator to your Python function.

Step 2: annotate input artifact types.

Step 3: annotate input parameter types.

Step 4: annotate output artifact types.

Option 1: Python function

```
from tfx.dsl.component.experimental.annotations import OutputDict
from tfx.dsl.component.experimental.annotations import InputArtifact
from tfx.dsl.component.experimental.annotations import OutputArtifact
from tfx.dsl.component.experimental.annotations import Parameter
from tfx.dsl.component.experimental.decorators import component
from tfx.types.standard_artifacts import Examples
from tfx.types.standard_artifacts import Model

@component
def MyTrainerComponent(
    training_data: InputArtifact[Examples],
    model: OutputArtifact[Model],
    dropout_hyperparameter: float,
    num_iterations: Parameter[int] = 10
    ) -> OutputDict(loss=float, accuracy=float):
  '''My simple trainer component.'''

  records = read_examples(training_data.uri)
  model_obj = train_model(records, num_iterations, dropout_hyperparameter)
  model_obj.write_to(model.uri)

  return {
    'loss': model_obj.loss,
    'accuracy': model_obj.accuracy}
```

```
# ...
trainer = MyTrainerComponent(
    examples=example_gen.outputs['examples'],
    dropout_hyperparameter=other_component.outputs['dropout'],
    num_iterations=1000)
pusher = Pusher(model=trainer.outputs['model'])
# ...
```

Step 1: apply @component decorator to your Python function.

Step 2: annotate input artifact types.

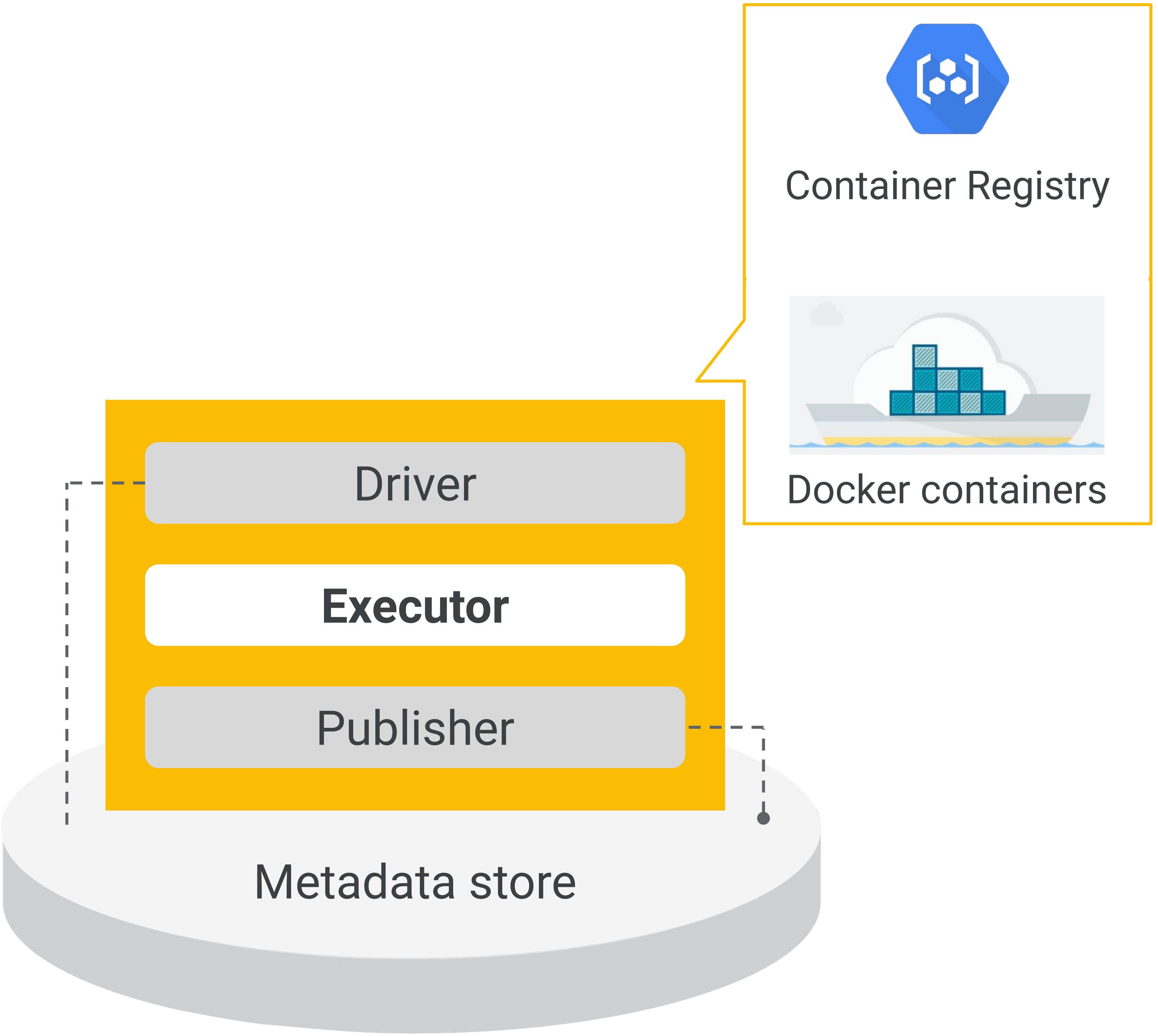
Step 3: annotate input parameter types.

Step 4: annotate output artifact types.

Step 5: initialize custom component.

Option 2: Containers

Create custom components using pre-built Docker containers for flexibility to incorporate code in any language into your TFX pipeline.



Option 2: container-based components

```
from tfx.dsl.component.experimental import container_component
from tfx.dsl.component.experimental import placeholders
from tfx.types import standard_artifacts
grep_component = container_component.create_container_component(
    name='FilterWithGrep',
    inputs={ 'text': standard_artifacts.ExternalArtifact,},
    outputs={ 'filtered_text': standard_artifacts.ExternalArtifact,},
    parameters={'pattern': str, },
    # The component code uses gsutil to upload the data to Google Cloud Storage
    image='google/cloud-sdk:278.0.0',
    command=[
        'sh', '-exc',
        '''
          pattern="$1"
          text_uri="$3"
          text_path=$(mktemp)
          filtered_text_uri="$5"
          filtered_text_path=$(mktemp)

          # Step 1
          gsutil cp "$text_uri" "$text_path"

          # Step 2
          grep "$pattern" "$text_path" >"$filtered_text_path"

          # Step 3
          gsutil cp "$filtered_text_path" "$filtered_text_uri"
        ''',
        , placeholders.InputValuePlaceholder('pattern'),
        '--text', placeholders.InputUriPlaceholder('text'),
        '--filtered-text', placeholders.OutputUriPlaceholder('filtered_text'),
    ],
)'--pattern'
```

```
Create a component from an existing
container image by wrapping it with
create_container_component(). Its
arguments:

Name: string component name.
Inputs: dictionary that maps input names to
artifact types.
Outputs: dictionary that maps output names
to artifact types.
Parameters: dictionary that maps names to
parameter types.
Image: Container image name with optional
tag.
Command: Container entrypoint command line.
```

Option 2: container-based components

```
from tfx.dsl.component.experimental import container_component
from tfx.dsl.component.experimental import placeholders
from tfx.types import standard_artifacts
grep_component = container_component.create_container_component(
    name='FilterWithGrep',
    inputs={ 'text': standard_artifacts.ExternalArtifact,},
    outputs={ 'filtered_text': standard_artifacts.ExternalArtifact,},
    parameters={'pattern': str, },
    # The component code uses gsutil to upload the data to Google Cloud Storage
    image='google/cloud-sdk:278.0.0',
    command=[
        'sh', '-exc',
        '''
          pattern="$1"
          text_uri="$3"
          text_path=$(mktemp)
          filtered_text_uri="$5"
          filtered_text_path=$(mktemp)

          # Step 1
          gsutil cp "$text_uri" "$text_path"

          # Step 2
          grep "$pattern" "$text_path" >"$filtered_text_path"

          # Step 3
          gsutil cp "$filtered_text_path" "$filtered_text_uri"
        ''',
        '--pattern', placeholders.InputValuePlaceholder('pattern'),
        '--text', placeholders.InputUriPlaceholder('text'),
        '--filtered-text', placeholders.OutputUriPlaceholder('filtered_text'),
    ],
)
```

Step 1: Use gsutil to copy ExternalArtifact text data into the container from text_uri directory.

Step 2: Filter text using grep command based on provided pattern.

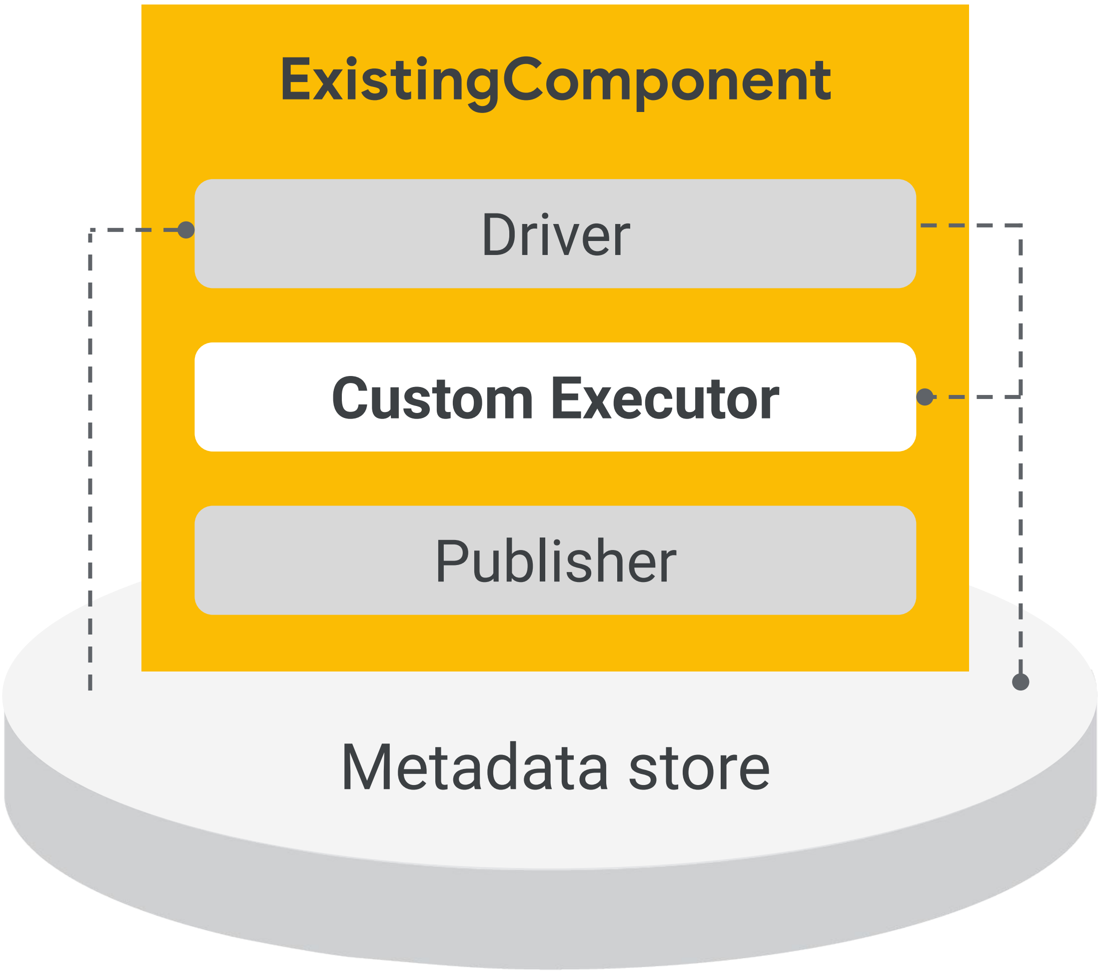
Step 3: Use gsutil to copy data out of container into text_filtered_uri directory.

Option 2: container-based components

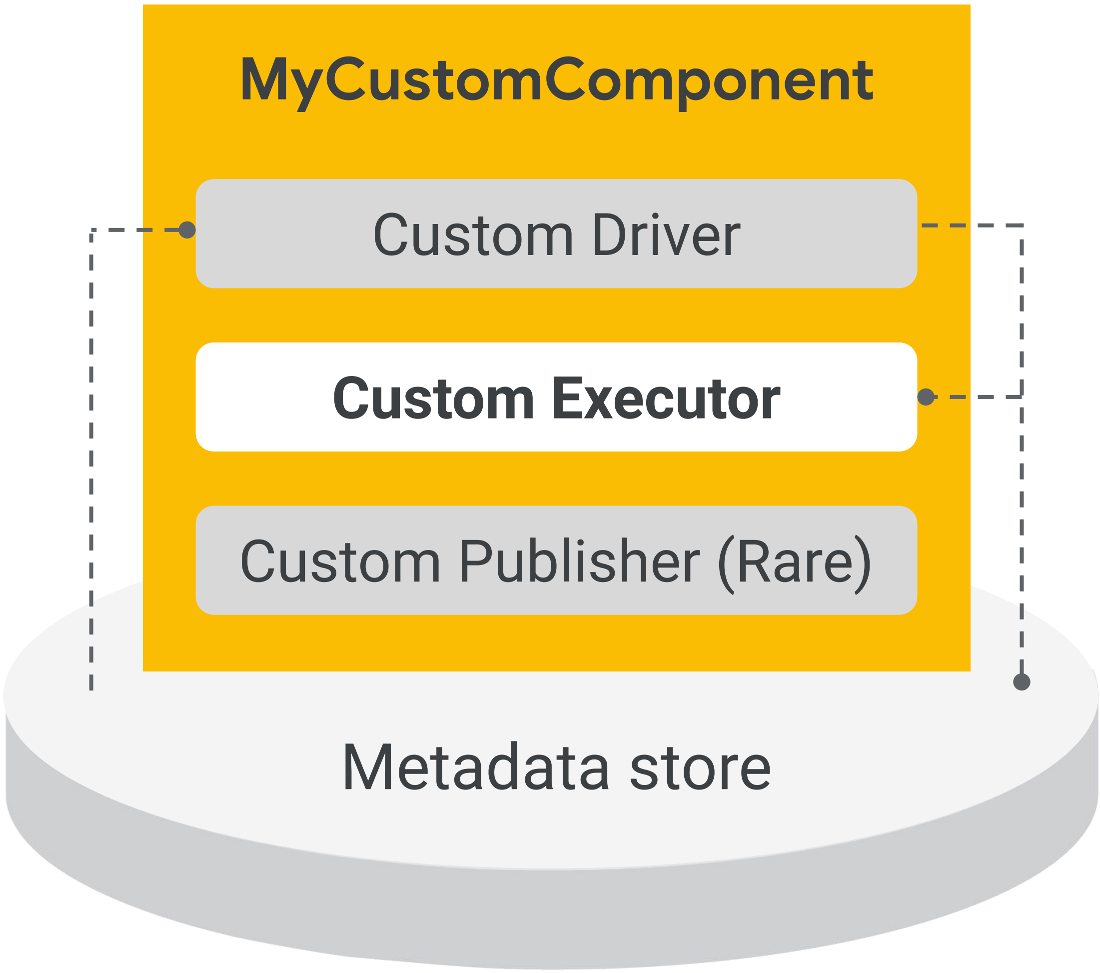
```
from tfx.dsl.component.experimental import container_component
from tfx.dsl.component.experimental import placeholders
from tfx.types import standard_artifacts
grep_component = container_component.create_container_component(
    name='FilterWithGrep',
    inputs={ 'text': standard_artifacts.ExternalArtifact,},
    outputs={ 'filtered_text': standard_artifacts.ExternalArtifact,},
    parameters={'pattern': str, },
    # The component code uses gsutil to upload the data to Google Cloud Storage
    image='google/cloud-sdk:278.0.0',
    command=[
        'sh', '-exc',
        '''
          pattern="$1"
          text_uri="$3"
          text_path=$(mktemp)
          filtered_text_uri="$5"
          filtered_text_path=$(mktemp)

          # Step 1
          gsutil cp "$text_uri" "$text_path"

          # Step 2
          grep "$pattern" "$text_path" >"$filtered_text_path"

          # Step 3
          gsutil cp "$filtered_text_path" "$filtered_text_uri"
        ''',
        '--pattern', placeholders.InputValuePlaceholder('pattern'),
        '--text', placeholders.InputUriPlaceholder('text'),
        '--filtered-text', placeholders.OutputUriPlaceholder('filtered_text'),
    ],
)
```

Placeholders are command-line arguments use in container component definitions that are replaced at runtime with artifact values or URIs.

Option 3: Extending existing components



Design Pattern # 1 (most common):
Extend existing component executor



Design Pattern # 2:
Full customization

Google Cloud

Subclass ComponentSpec to customize component artifact linkage

```
class MyComponentSpec(types.ComponentSpec):

  PARAMETERS = {
      'name': ExecutionParameter(type=Text),
  }
  INPUTS = {
      'input_data': ChannelParameter(type=standard_artifacts.Examples),
  }
  OUTPUTS = {
      'output_data': ChannelParameter(type=standard_artifacts.Examples),
  }
```

Subclass BaseExecutor for custom computation

```
class MyExecutor(base_executor.BaseExecutor):

  def Do(self, input_dict,
         output_dict,
         exec_properties):
```

Customize BaseDriver to change interactions with ML Metadata

```
class MyDriver(base_driver.BaseDriver):
  """Custom driver for MyComponent."""

  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_artifacts()."""
    del driver_args 1
    del pipeline_info

    input_dict = channel_utils.unwrap_channel_dict(input_channels) 2
    for input_list in input_dict.values():
        for single_input in input_list:
            self._metadata_handler.publish_artifacts([single_input]) 3
            absl.logging.debug("Registered input: {}".format(single_input))
            absl.logging.debug("single_input.mlmd_artifact "
                               "{}".format(single_input.mlmd_artifact)) 4
    return input_dict
```

Customize component interface to use fully custom component in pipeline

```
from tfx.components.base import base_component
from tfx import types
from tfx.types import channel_utils

class MyComponent(base_component.BaseComponent):
    """Custom MyComponent."""
    SPEC_CLASS = MyComponentSpec
    EXECUTOR_SPEC = executor_spec.ExecutorClassSpec(MyExecutor)
    DRIVER_CLASS = MyDriver

    def __init__(self, input, output_data=None, name=None):
        if not output_data:
            examples_artifact = standard_artifacts.Examples()
            examples_artifact.split_names = \
                artifact_utils.encode_split_names(['train', 'eval'])

            output_data = channel_utils.as_channel([examples_artifact])

        spec = MySpec(input=input,
examples=output_data,
name=name)
        super(MyComponent, self).__init__(spec=spec)
```

Exercise: Custom Component Brainstorm

Reflect on what you learned about building custom TFX components.

Option 1: Can you think of a part of your ML workflow that you would build a custom component for?

Option 2: How would you build a custom ExampleGen to convert image directories to TF Records?

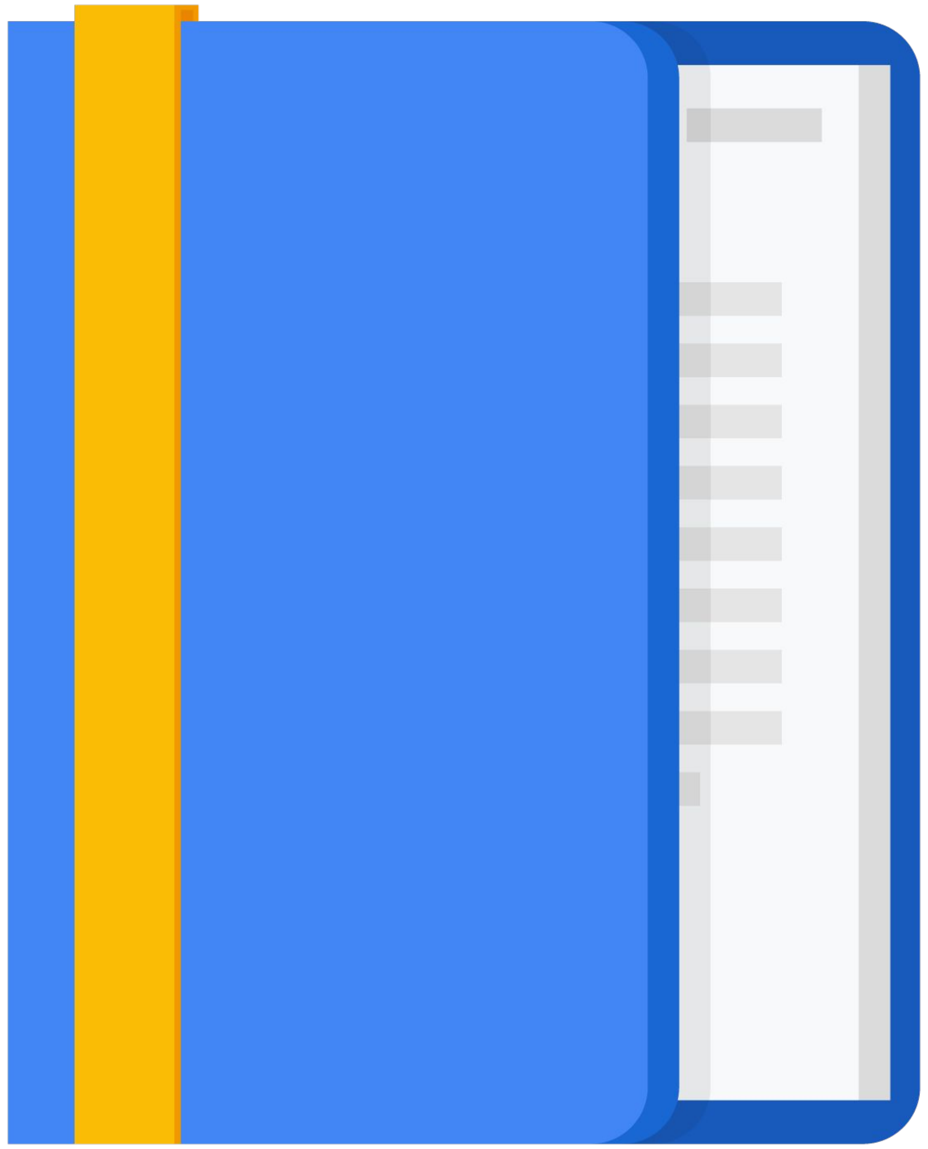
Describe the following pieces of your custom component:

- ComponentSpec()
- Executor()
- Component() interface

Agenda

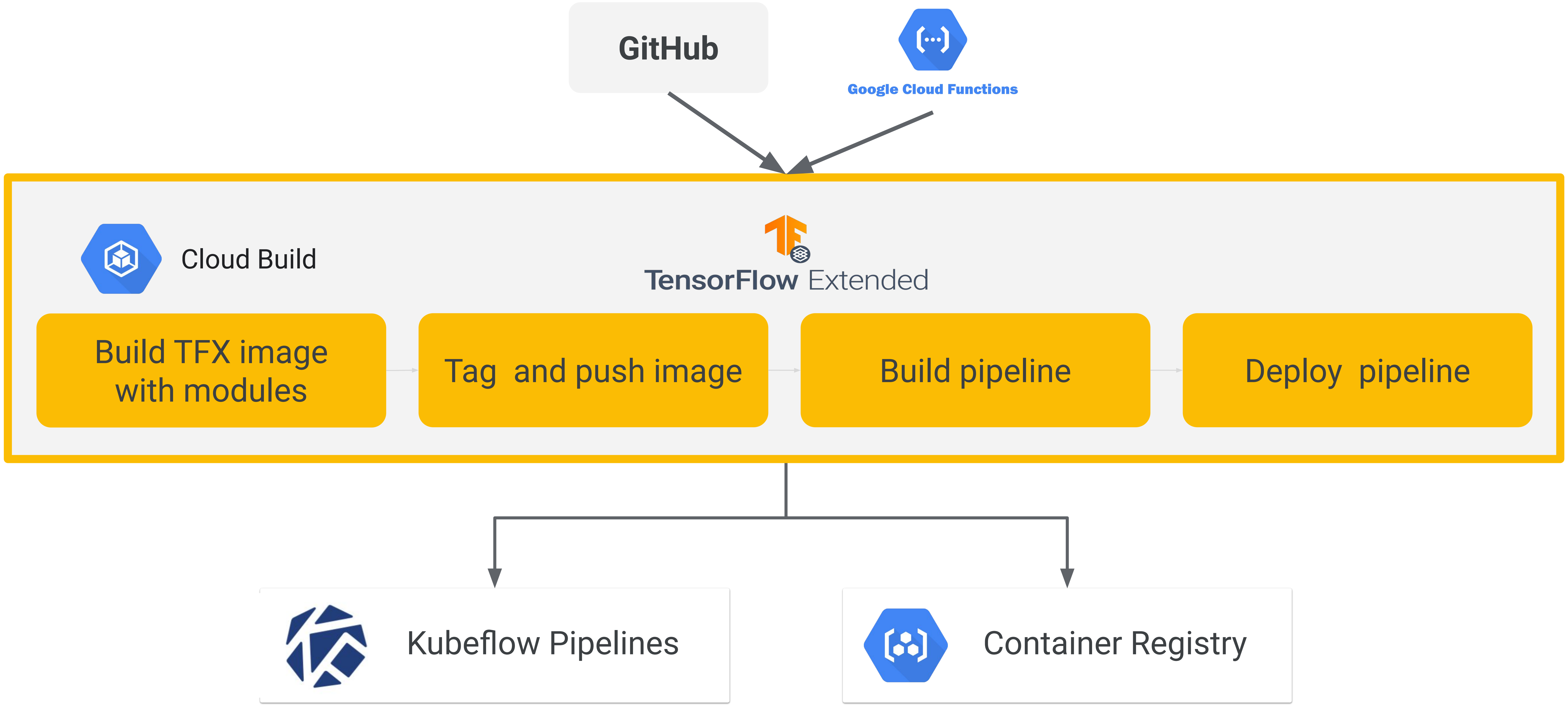
TFX custom components

CI/CD for TFX pipeline workflows

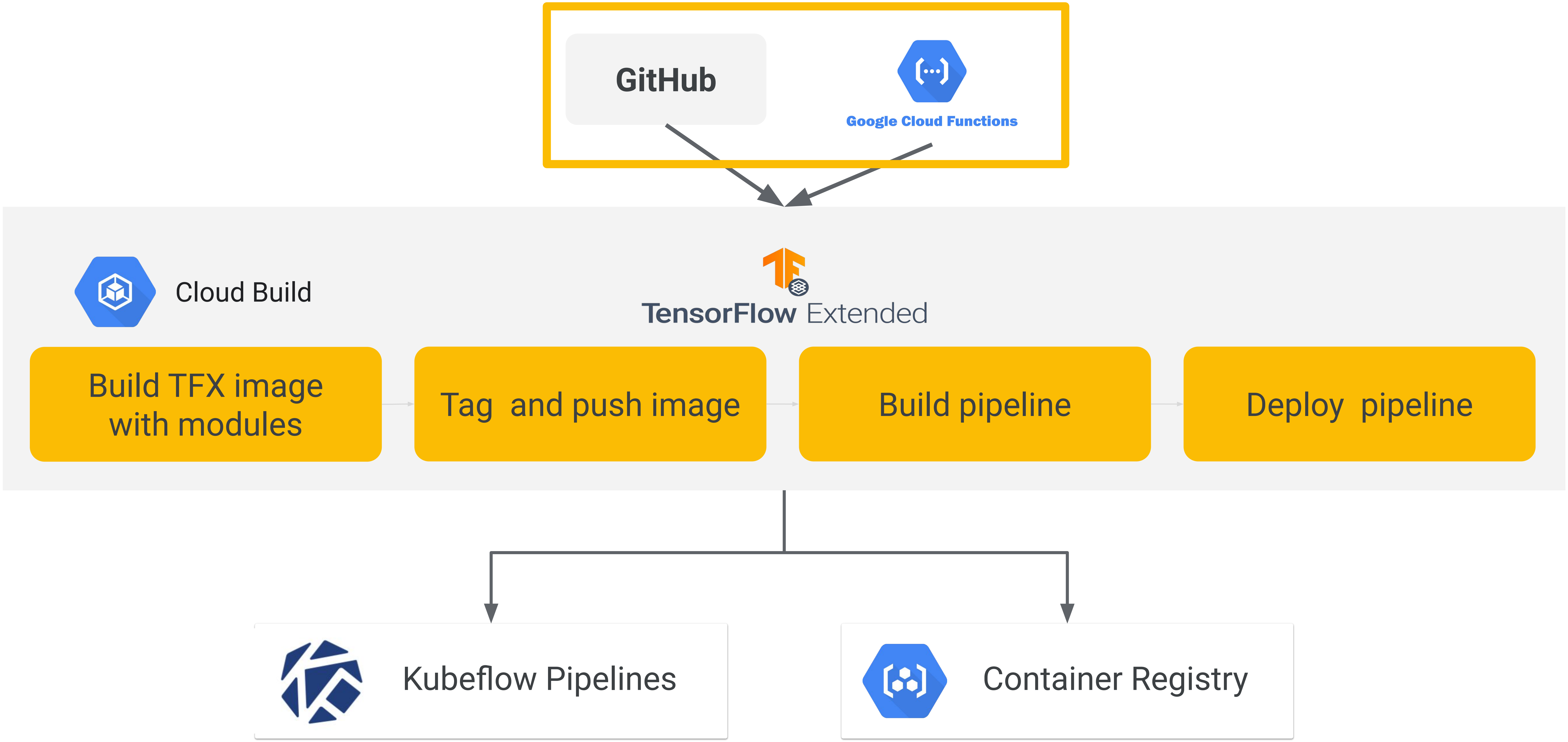


Review: TFX brings DevOps best practices to ML workflows

CI/CD for training TFX pipelines on Google Cloud



CI/CD for training TFX pipelines on Google Cloud



TFX CI/CD Workflow: Cloud Build steps

Build steps expand all

Clone Repo 2 sec
gcr.io/cloud-builders/git — clonehttps://githib.com/mlops-on-gcp/tfx-examples.git tfx-cicd-pipeline --depth 1 --verbose

Run Unit Tests 11 sec
python:3.6-slim-jessie — components/tests.sh

Build my_add Image 2 sec
gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_add:latest .

Build my_divide Image 2 sec
gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_divide:latest .

Update Component Spec Images 34 sec
gcr.io/ml-cicd-template/kfp-util:latest — pipeline/helper.py update-specs --repo_url gcr.io/ml-cicd-template --image_tage {_TAG}

Compile Pipeline 2 sec
gcr.io/ml-cicd-template/kfp-util:latest — --py pipeline/workflow.py --output pipeline/pipeline.tar.gz --disable-type-check

Upload Pipeline to GCS 5 sec
gcr.io/cloud-builders/gsutil — cp pipeline.tar.gz settings.yaml gs://ml-cicd-template/helloworld/pipelines/latest/

Deploy & Run Pipeline 8 sec
gcr.io/ml-cicd-template/kfp-util:latest — -c "/builder/kubectl.bash; python3 helper.py deploy-pipeline --package_path=pipeline.tar.gz --version=\"latest\" --experiment=\"helloworld-dev\" --run"

TFX CI/CD Workflow: Cloud Build steps

Build steps | expand all

Step	Command	Time
Clone Repo	gcr.io/cloud-builders/git — clone https://githib.com/mlops-on-gcp/tfx-examples.git tfx-cicd-pipeline --depth 1 --verbose	2 sec
Run Unit Tests	python:3.6-slim-jessie — components/tests.sh	11 sec
Build my_add Image	gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_add:latest .	2 sec
Build my_divide Image	gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_divide:latest .	2 sec
Update Component Spec Images	gcr.io/ml-cicd-template/kfp-util:latest — pipeline/helper.py update-specs --repo_url gcr.io/ml-cicd-template --image_tage {_TAG}	34 sec
Compile Pipeline	gcr.io/ml-cicd-template/kfp-util:latest — --py pipeline/workflow.py --output pipeline/pipeline.tar.gz --disable-type-check	2 sec
Upload Pipeline to GCS	gcr.io/cloud-builders/gsutil — cp pipeline.tar.gz settings.yaml gs://ml-cicd-template/helloworld/pipelines/latest/	5 sec
Deploy & Run Pipeline	gcr.io/ml-cicd-template/kfp-util:latest — -c "/builder/kubectl.bash; python3 helper.py deploy-pipeline --package_path=pipeline.tar.gz --version=\"latest\" --experiment=\"helloworld-dev\" --run"	8 sec

TFX CI/CD Workflow: Cloud Build steps

Build steps expand all

Clone Repo gcr.io/cloud-builders/git — clonehttps://githib.com/mlops-on-gcp/tfx-examples.git tfx-cicd-pipeline --depth 1 --verbose	2 sec
Run Unit Tests python:3.6-slim-jessie — components/tests.sh	11 sec
Build my_add Image gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_add:latest .	2 sec
Build my_divide Image gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_divide:latest .	2 sec
Update Component Spec Images gcr.io/ml-cicd-template/kfp-util:latest — pipeline/helper.py update-specs --repo_url gcr.io/ml-cicd-template --image_tage {_TAG}	34 sec
Compile Pipeline gcr.io/ml-cicd-template/kfp-util:latest — --py pipeline/workflow.py --output pipeline/pipeline.tar.gz --disable-type-check	2 sec
Upload Pipeline to GCS gcr.io/cloud-builders/gsutil — cp pipeline.tar.gz settings.yaml gs://ml-cicd-template/helloworld/pipelines/latest/	5 sec
Deploy & Run Pipeline gcr.io/ml-cicd-template/kfp-util:latest — -c "/builder/kubectl.bash; python3 helper.py deploy-pipeline --package_path=pipeline.tar.gz --version=\"latest\" --experiment=\"helloworld-dev\" --run"	8 sec

TFX CI/CD Workflow: Cloud Build steps

Build steps — expand all

Clone Repo	gcr.io/cloud-builders/git — clonehttps://githib.com/mlops-on-gcp/tfx-examples.git tfx-cicd-pipeline --depth 1 --verbose	2 sec
Run Unit Tests	python:3.6-slim-jessie — components/tests.sh	11 sec
Build my_add Image	gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_add:latest .	2 sec
Build my_divide Image	gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_divide:latest .	2 sec
Update Component Spec Images	gcr.io/ml-cicd-template/kfp-util:latest — pipeline/helper.py update-specs --repo_url gcr.io/ml-cicd-template --image_tage {_TAG}	34 sec
Compile Pipeline	gcr.io/ml-cicd-template/kfp-util:latest — --py pipeline/workflow.py --output pipeline/pipeline.tar.gz --disable-type-check	2 sec
Upload Pipeline to GCS	gcr.io/cloud-builders/gsutil — cp pipeline.tar.gz settings.yaml gs://ml-cicd-template/helloworld/pipelines/latest/	5 sec
Deploy & Run Pipeline	gcr.io/ml-cicd-template/kfp-util:latest — -c "/builder/kubectl.bash; python3 helper.py deploy-pipeline --package_path=pipeline.tar.gz --version=\"latest\" --experiment=\"helloworld-dev\" --run"	8 sec

TFX CI/CD Workflow: Cloud Build steps

Build steps expand all

Clone Repo 2 sec
gcr.io/cloud-builders/git — clonehttps://githib.com/mlops-on-gcp/tfx-examples.git tfx-cicd-pipeline –depth 1 –verbose

Run Unit Tests 11 sec
python:3.6-slim-jessie — components/tests.sh

Build my_add Image 2 sec
gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_add:latest .

Build my_divide Image 2 sec
gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_divide:latest .

Update Component Spec Images 34 sec
gcr.io/ml-cicd-template/kfp-util:latest — pipeline/helper.py update-specs –repo_url gcr.io/ml-cicd-template –image_tage {_TAG}

Compile Pipeline 2 sec
gcr.io/ml-cicd-template/kfp-util:latest — –py pipeline/workflow.py –output pipeline/pipeline.tar.gz –disable-type-check

Upload Pipeline to GCS 5 sec
gcr.io/cloud-builders/gsutil — cp pipeline.tar.gz settings.yaml gs://ml-cicd-template/helloworld/pipelines/latest/

Deploy & Run Pipeline 8 sec
gcr.io/ml-cicd-template/kfp-util:latest — -c "/builder/kubectl.bash; python3 helper.py deploy-pipeline –package_path=pipeline.tar.gz –version=\"latest\" –experiment=\"helloworld-dev\" –run"

TFX CI/CD Workflow: Cloud Build steps

Build steps

expand all

Clone Repo — 2 sec
gcr.io/cloud-builders/git — clone https://githib.com/mlops-on-gcp/tfx-examples.git tfx-cicd-pipeline --depth 1 --verbose

Run Unit Tests — 11 sec
python:3.6-slim-jessie — components/tests.sh

Build my_add Image — 2 sec
gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_add:latest .

Build my_divide Image — 2 sec
gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_divide:latest .

Update Component Spec Images — 34 sec
gcr.io/ml-cicd-template/kfp-util:latest — pipeline/helper.py update-specs --repo_url gcr.io/ml-cicd-template --image_tage {_TAG}

Compile Pipeline — 2 sec
gcr.io/ml-cicd-template/kfp-util:latest — --py pipeline/workflow.py --output pipeline/pipeline.tar.gz --disable-type-check

Upload Pipeline to GCS — 5 sec
gcr.io/cloud-builders/gsutil — cp pipeline.tar.gz settings.yaml gs://ml-cicd-template/helloworld/pipelines/latest/

Deploy & Run Pipeline — 8 sec
gcr.io/ml-cicd-template/kfp-util:latest — -c "/builder/kubectl.bash; python3 helper.py deploy-pipeline --package_path=pipeline.tar.gz --version=\"latest\" --experiment=\"helloworld-dev\" --run"

TFX CI/CD Workflow: Cloud Build steps

Build steps expand all

Clone Repo 2 sec
gcr.io/cloud-builders/git — clone https://githib.com/mlops-on-gcp/tfx-examples.git tfx-cicd-pipeline --depth 1 --verbose

Run Unit Tests 11 sec
python:3.6-slim-jessie — components/tests.sh

Build my_add Image 2 sec
gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_add:latest .

Build my_divide Image 2 sec
gcr.io/cloud-builders/docker — build -t gcr.io/ml-cicd-template/my_divide:latest .

Update Component Spec Images 34 sec
gcr.io/ml-cicd-template/kfp-util:latest — pipeline/helper.py update-specs --repo_url gcr.io/ml-cicd-template --image_tage {_TAG}

Compile Pipeline 2 sec
gcr.io/ml-cicd-template/kfp-util:latest — --py pipeline/workflow.py --output pipeline/pipeline.tar.gz --disable-type-check

Upload Pipeline to GCS 5 sec
gcr.io/cloud-builders/gsutil — cp pipeline.tar.gz settings.yaml gs://ml-cicd-template/helloworld/pipelines/latest/

Deploy & Run Pipeline 8 sec
gcr.io/ml-cicd-template/kfp-util:latest — -c "/builder/kubectl.bash; python3 helper.py deploy-pipeline --package_path=pipeline.tar.gz --version=\"latest\" --experiment=\"helloworld-dev\" --run"

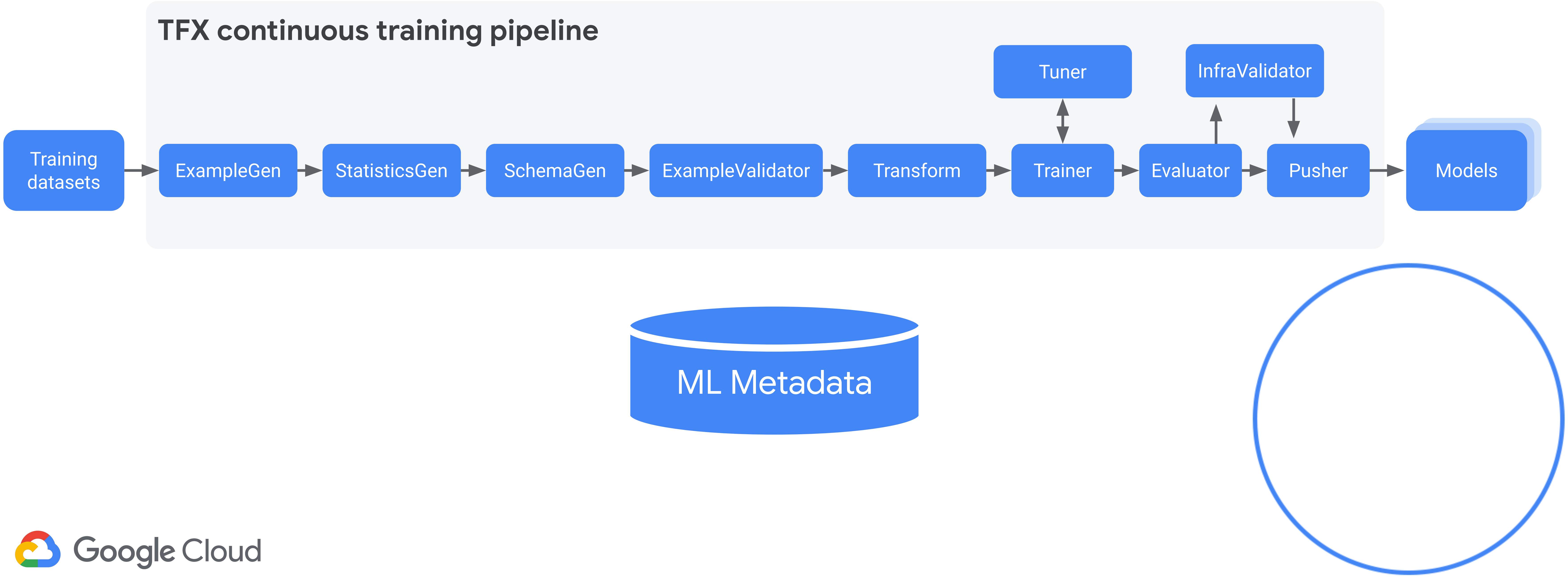
Level-0 ML process automation: TFX pipeline prototyping in notebooks

```
context = InteractiveContext()

component = MyComponent(...)
context.run(component)
context.show(component.outputs['my_output'])
```

Prototype-to-production: notebook can be converted into an orchestratable pipeline file

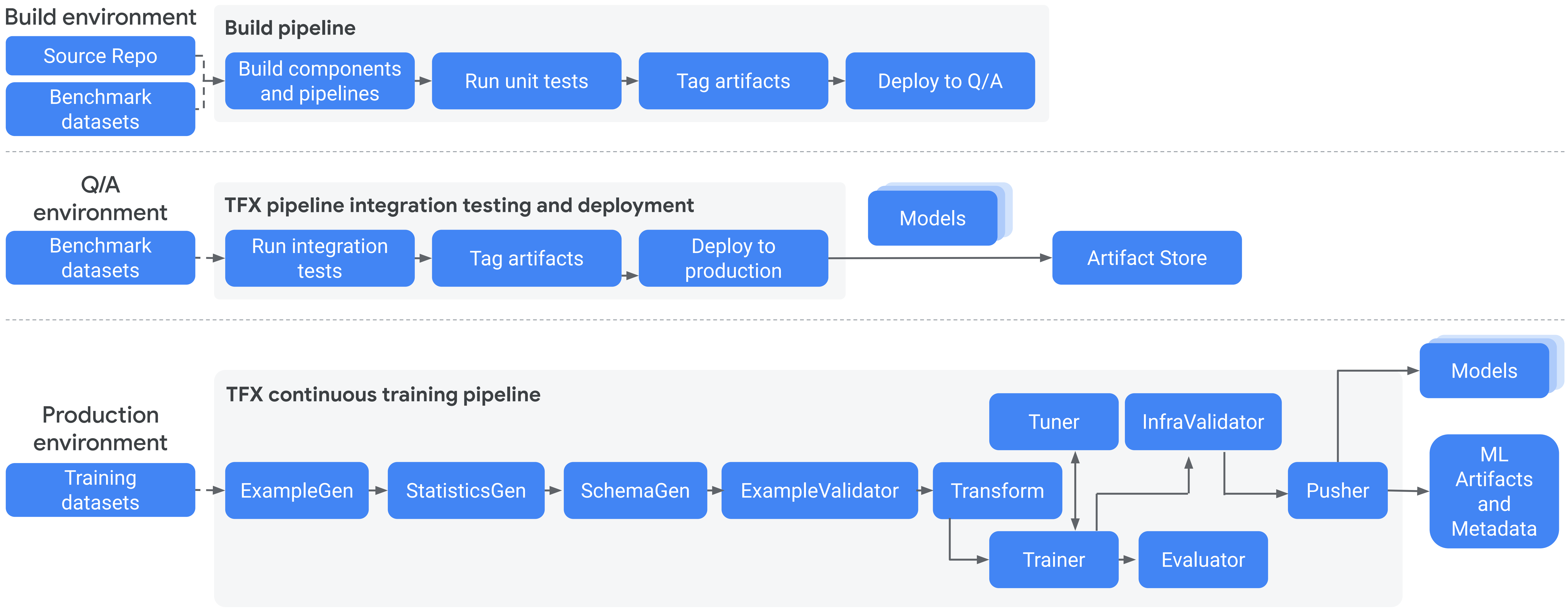
```
context.export_to_pipeline(notebook_filepath=_notebook_filepath,
                           export_filepath=_pipeline_export_filepath,
                           runner_type=_runner_type)
```

Level-1 ML development automation: TFX pipeline continuous training



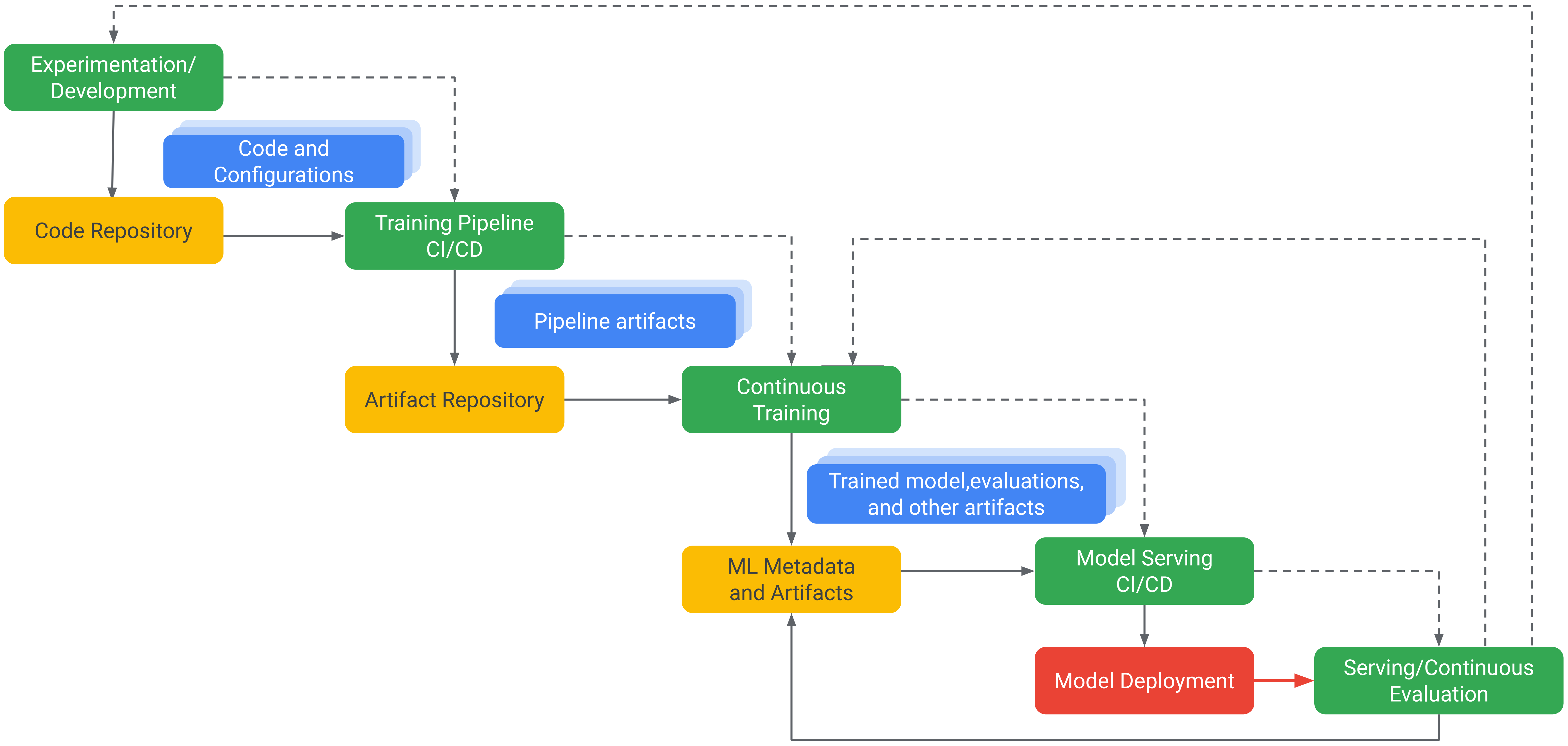
Google Cloud

Level-2 ML development automation: TFX CI/CD pipelines

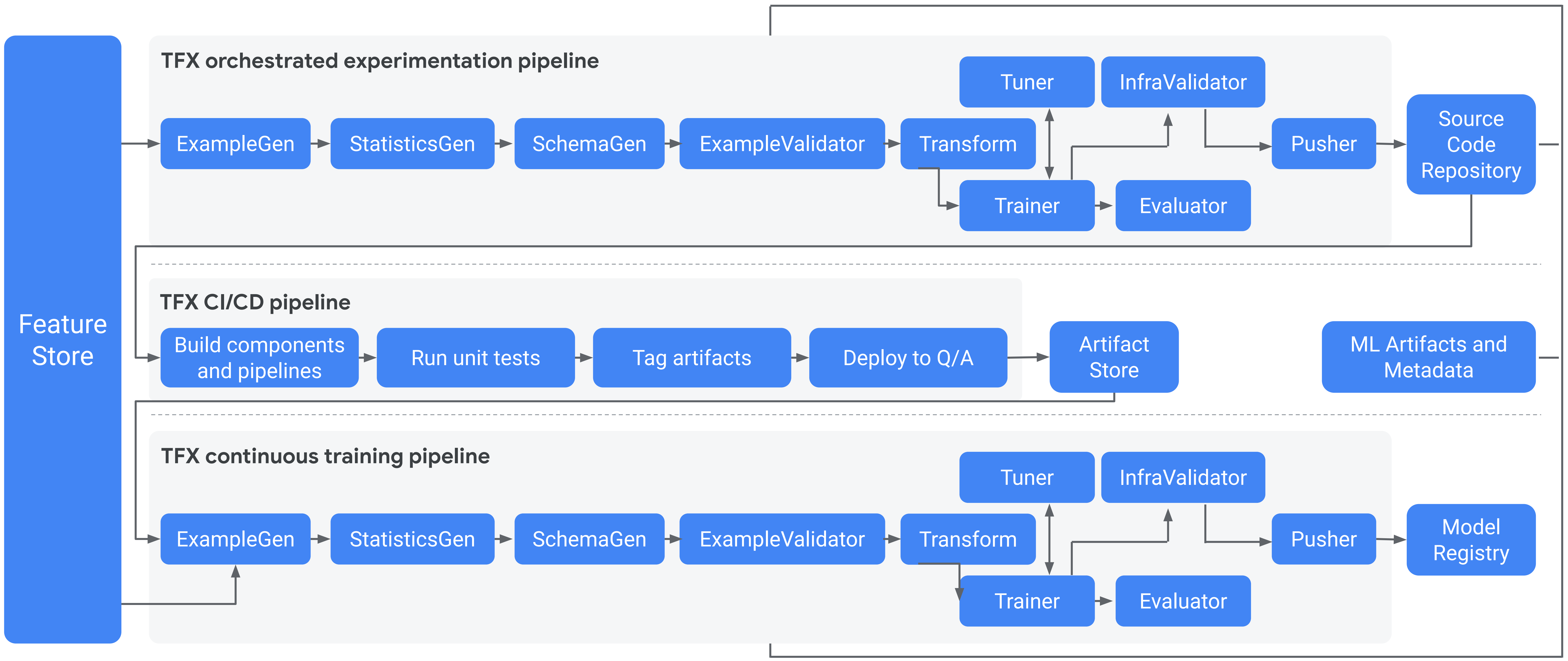


Google Cloud

End-to-end TFX MLOps workflow

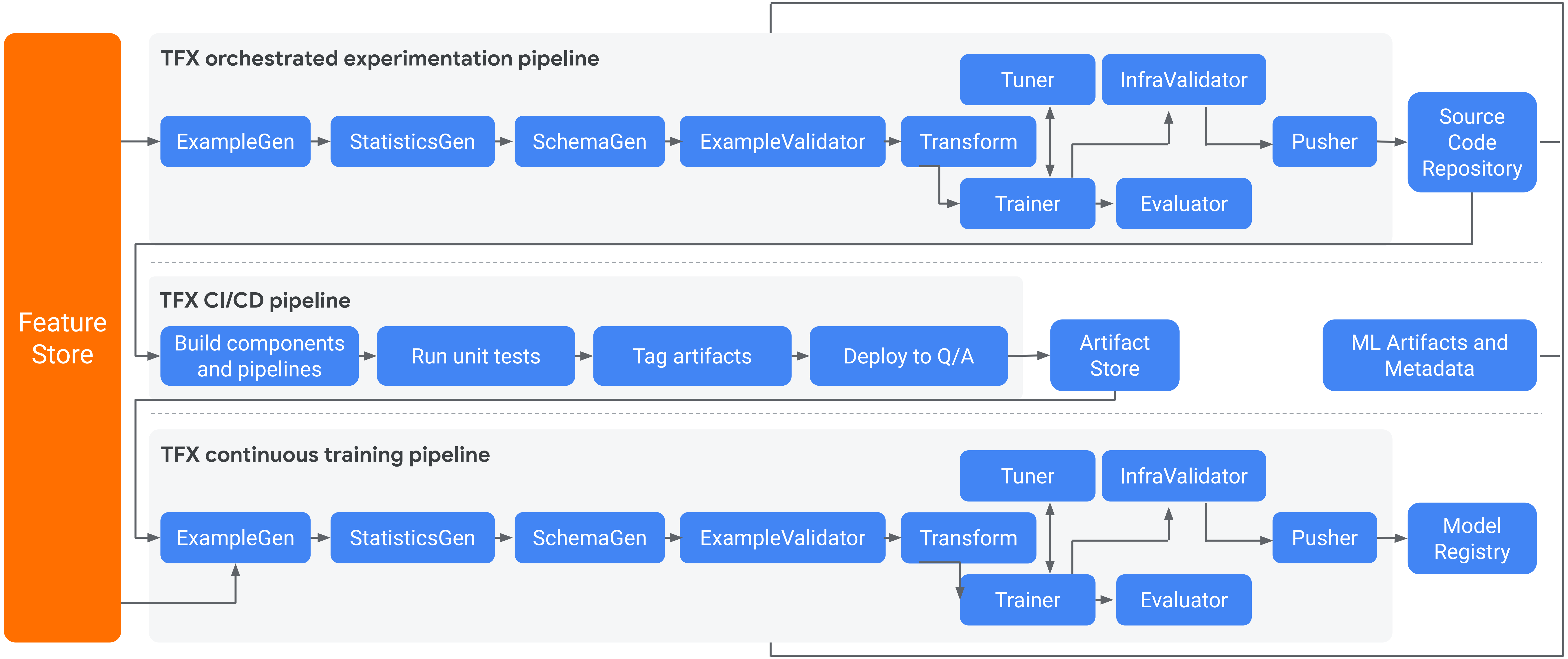


Future state of ML development:
Fully automated TFX MLOps pipeline development



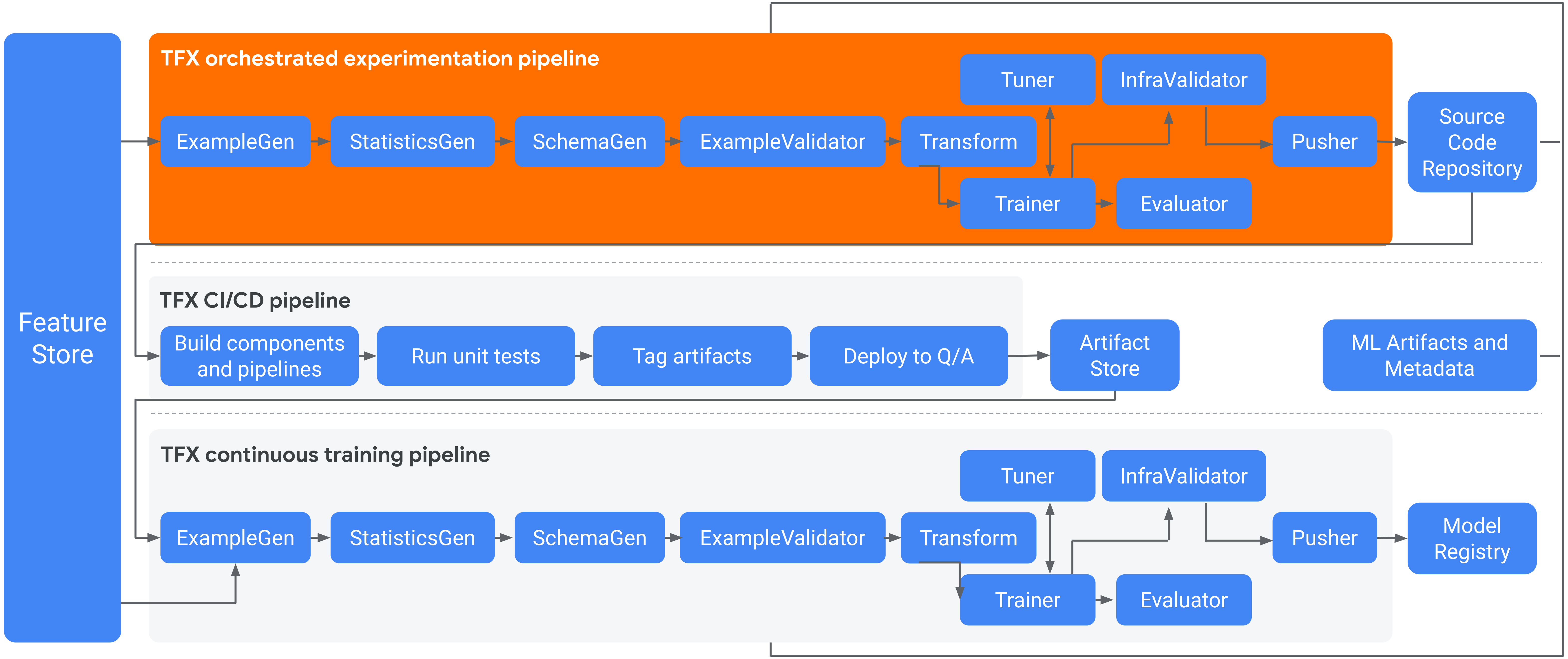
Google Cloud

Future state of ML development: Fully automated TFX MLOps pipeline development



Google Cloud

Future state of ML development: Fully automated TFX MLOps pipeline development



Lab Walkthrough

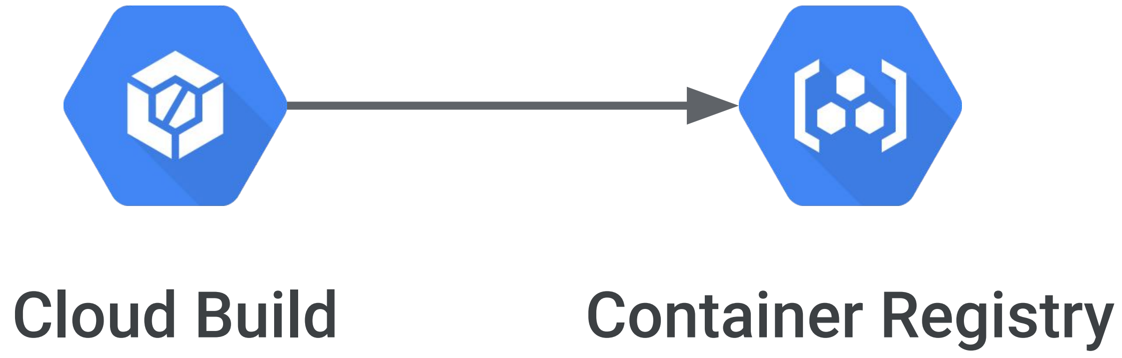
CI/CD for TFX pipelines

Google Cloud

Use Cloud Build to build and submit TFX CLI to Container Registry

/ ... / lab-03-tfx-cicd / tfx-cli /

Name	Last Modified
Dockerfile	2 months ago
requirements.txt	25 days ago



Dockerfile

```
FROM gcr.io/deeplearning-platform-release/tf2-cpu.2-3
COPY requirements.txt .
RUN python -m pip install -U -r requirements.txt

ENTRYPOINT ["tfx"]
```

requirements.txt

```
pandas>1.0.0
tfx==0.25.0
kfp==1.0.4
```

```
IMAGE_NAME='tfx-cli'
TAG='latest'
IMAGE_URI='gcr.io/{}/{}:{}'.format(PROJECT_ID,
IMAGE_NAME, TAG)
```

```
!gcloud builds submit --timeout 15m --tag {IMAGE_URI}
tfx-cli
```

Manually trigger CI/CD runs of TFX pipeline image

/ ... / tfx-caip-tf21 / lab-03-tfx-cicd /

Name	Last Modified
exercises	7 days ago
pipeline	25 days ago
tfx-cli	25 days ago
cloudbuild.yaml	2 months ago
lab-03.ipynb	7 days ago
README.md	2 months ago

Use Cloud Build to create custom TFX pipeline container and push to Container Registry.

Use Cloud Build to trigger build of TFX pipeline image on Container Registry.

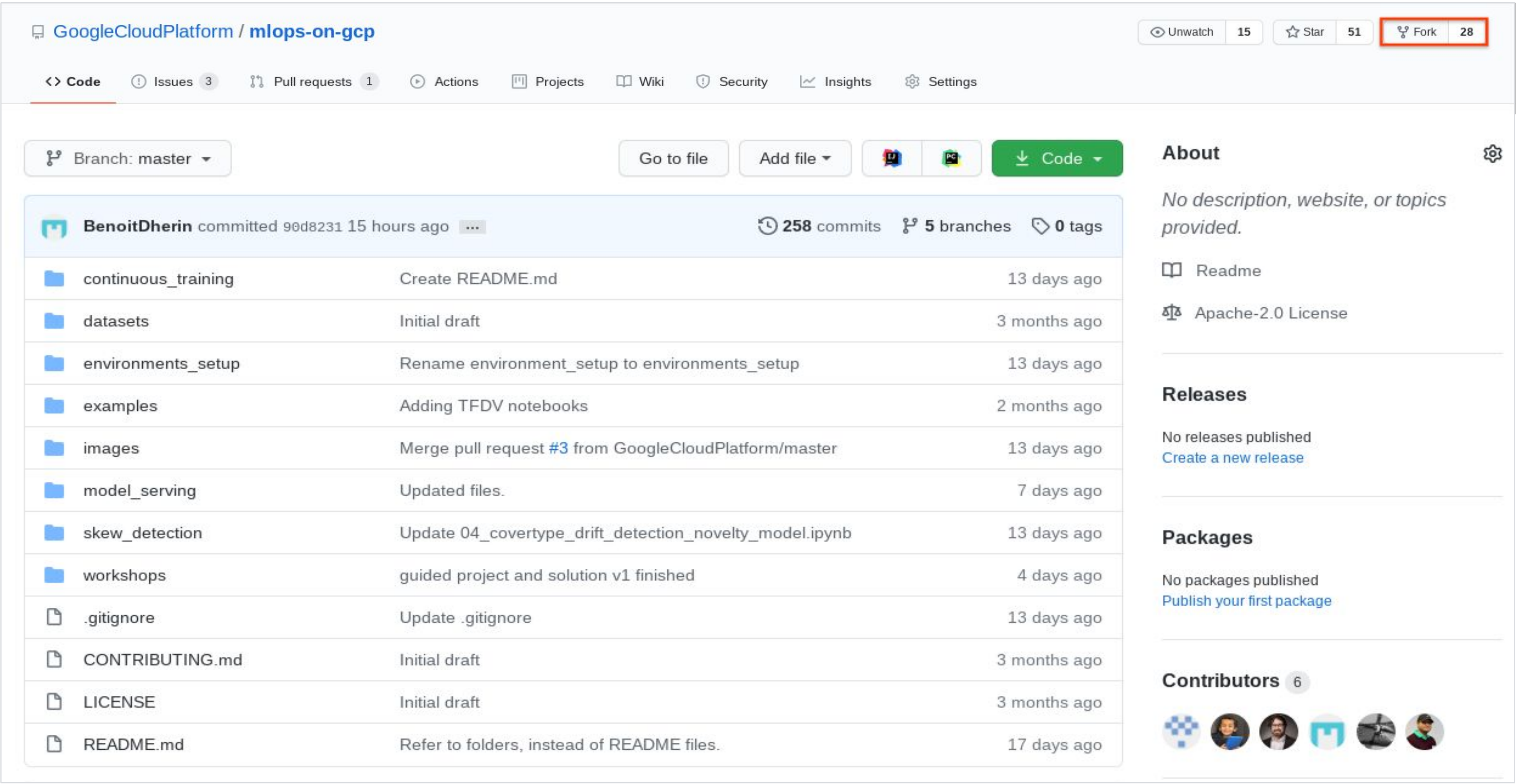
Cloud Build

```
!gcloud builds submit .
--config cloudbuild.yaml
--substitutions {SUBSTITUTIONS}
```

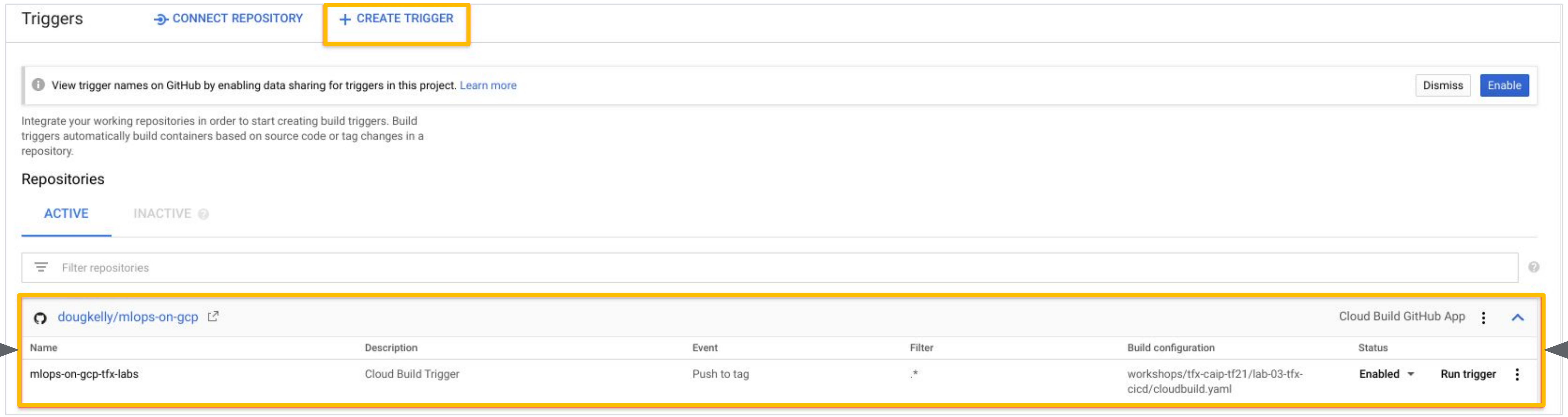
```
PIPELINE_NAME='tfx_covertype_continuous_training'
TAG_NAME='test'
TFX_IMAGE_NAME='lab-03-tfx-image'
DATA_ROOT_URI='gs://workshop-datasets/covertype/small'
MODEL_NAME='tfx_covertype_classifier'
PIPELINE_FOLDER='pipeline'
PIPELINE_DSL='runner.py'
RUNTIME_VERSION='2.3'
PYTHON_VERSION='3.7'
USE_KFP_SA='False'

SUBSTITUTIONS="""
_ENDPOINT={},\
_GCP_REGION={},\
_ARTIFACT_STORE_URI={},\
_TFX_IMAGE_NAME={},\
_DATA_ROOT_URI={},\
_MODEL_NAME={},\
TAG_NAME={},\
_PIPELINE_FOLDER={},\
_PIPELINE_DSL={},\
_PIPELINE_NAME={},\
_RUNTIME_VERSION={},\
_USE_KFP_SA={},\
_PYTHON_VERSION={}
""".format(ENDPOINT,
           GCP_REGION,
           ARTIFACT_STORE_URI,
           TFX_IMAGE_NAME,
           DATA_ROOT_URI,
           MODEL_NAME,
           TAG_NAME,
           PIPELINE_FOLDER,
           PIPELINE_DSL,
           PIPELINE_NAME,
           RUNTIME_VERSION,
           PYTHON_VERSION,
           USE_KFP_SA
           ).strip()
```

Fork mlops-on-gcp Github repository

https://github.com/GoogleCloudPlatform/mlops-on-gcp



Create a Github App trigger with Cloud Build



Create Cloud Build Github trigger

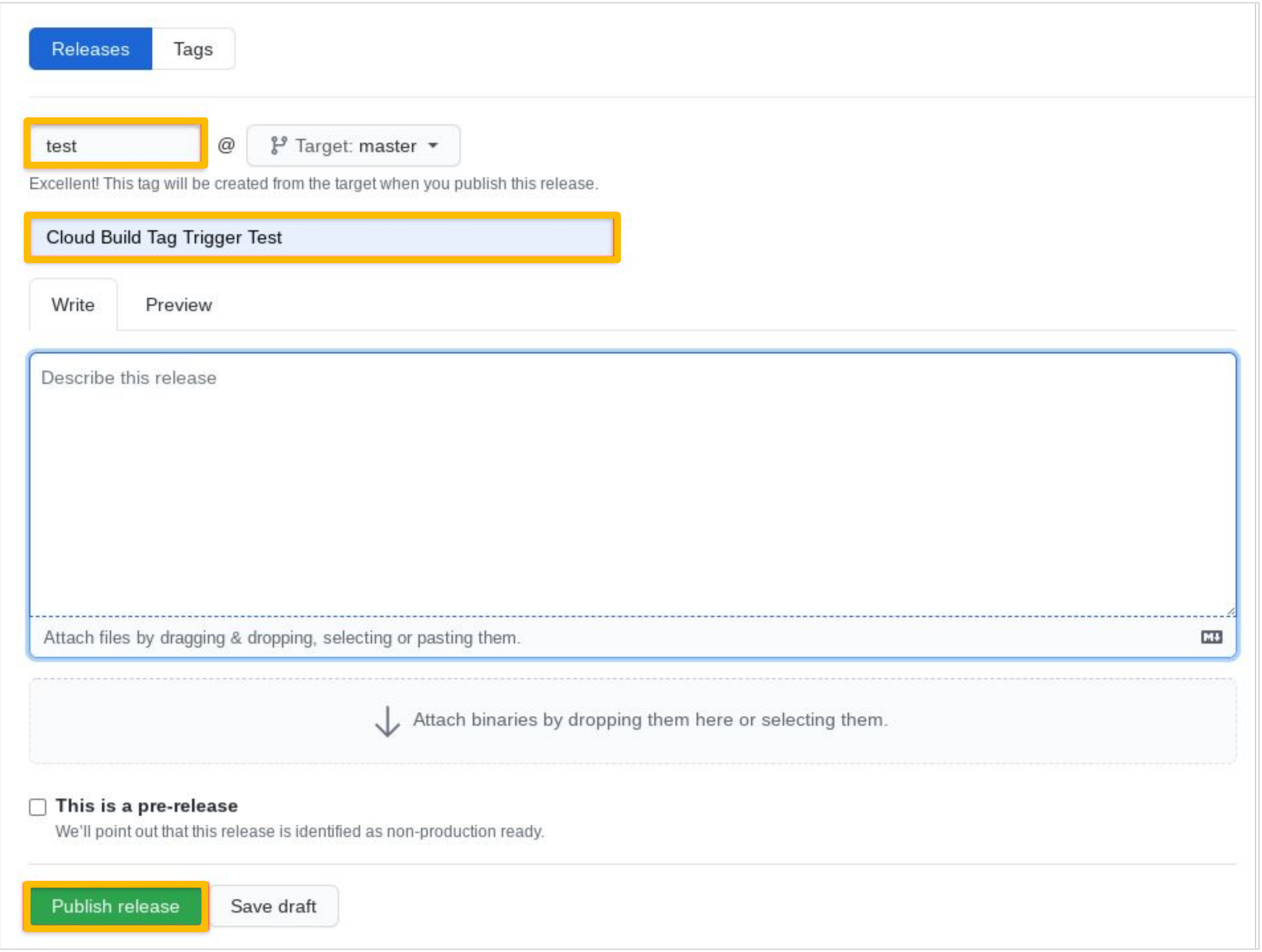
Field	Value
Name	[YOUR TRIGGER NAME]
Description	[YOUR TRIGGER DESCRIPTION]
Event	Tag
Source	[YOUR FORK]
Tag (regex)	.*
Build Configuration	Cloud Build configuration file (yaml or json)
Cloud Build configuration file location	/ workshops/tfx-caip-tf23/lab-03-tfx-cicd/cloudbuild.yaml

Enter substitution runtime variables

Variable	Value
_ENDPOINT	[Your inverting proxy host pipeline ENDPOINT]
_TFX_IMAGE_NAME	lab-03-tfx-image
_PIPELINE_NAME	tfx_covertype_continuous_training
_PIPELINE_DSL	runner.py
_DATA_ROOT_URI	gs://workshop-datasets/covertype/small
_PIPELINE_FOLDER	workshops/tfx-caip-tf23/lab-03-tfx-cicd/pipeline
_PYTHON_VERSION	3.7
_RUNTIME_VERSION	2.3
_USE_KFP_SA	False

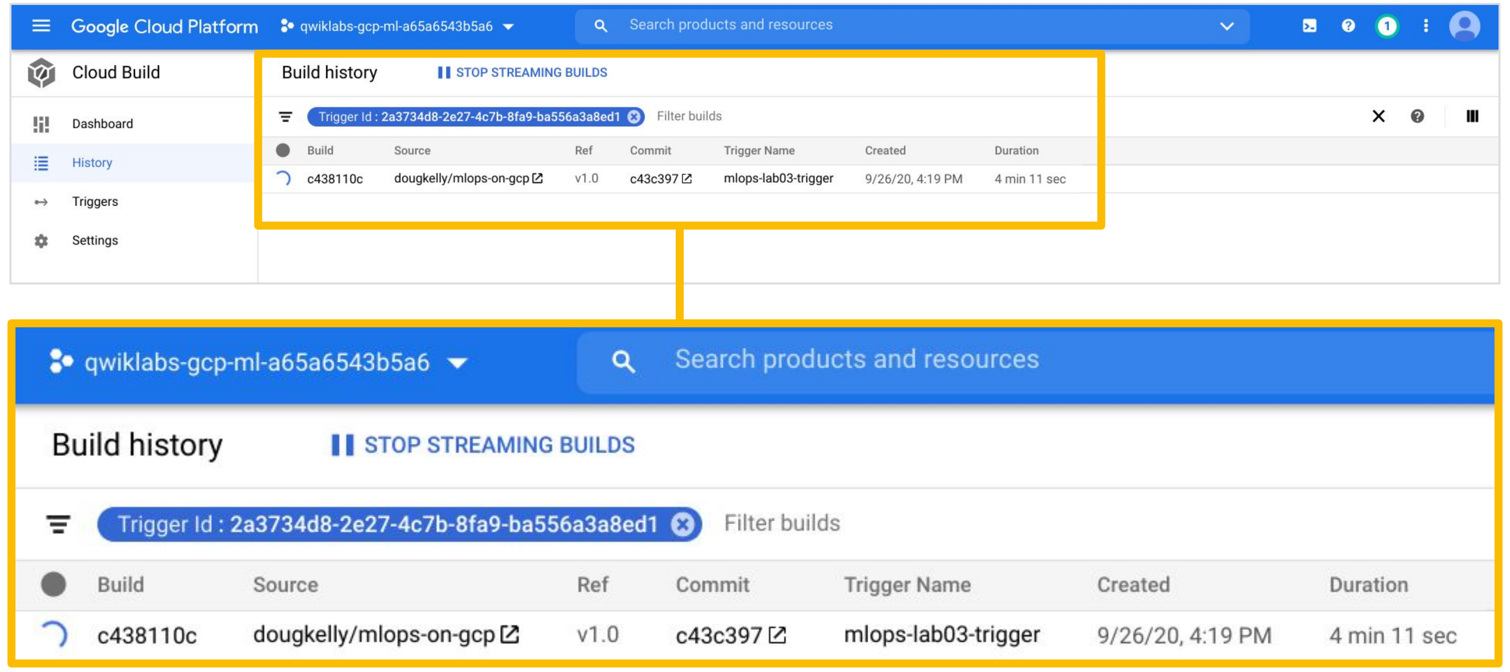
Trigger CI/CD run with Github release or Git

Create a release on Github to trigger pipeline run.



Add a release tag to trigger pipeline run via Git.

```
git tag [TAG NAME]
git push origin --tags
```



Verify pipeline run in Kubeflow Pipelines UI