Architecture for MLOps using TFX, Kubeflow Pipelines, and Cloud Build

This document describes the overall architecture of a machine learning (ML) system using TensorFlow Extended (TFX) (https://www.tensorflow.org/tfx) libraries. It also discusses how to set up a continuous integration (CI), continuous delivery (CD), and continuous training (CT) for the ML system using Cloud Build (/cloud-build/docs) and Kubeflow Pipelines (https://www.kubeflow.org/docs/pipelines/overview/pipelines-overview/).

In this document, the terms *ML system* and *ML pipeline* refer to ML model *training* pipelines, rather than model *scoring* or *prediction* pipelines.

This document is for data scientists and ML engineers who want to adapt their CI/CD practices to move ML solutions to production on Google Cloud, and who want to help ensure the quality, maintainability, and adaptability of their ML pipelines.

This document covers the following topics:

- Understanding CI/CD and automation in ML.
- Designing an integrated ML pipeline with TFX.
- Orchestrating and automating the ML pipeline using Kubeflow Pipelines.
- Setting up a CI/CD system for the ML pipeline using Cloud Build.

MLOps

To integrate an ML system in a production environment, you need to orchestrate the steps in your ML pipeline. In addition, you need to automate the execution of the pipeline for the continuous training of your models. To experiment with new ideas and features, you need to adopt CI/CD practices in the new implementations of the pipelines. The following sections give a high-level overview of CI/CD and CT in ML.

ML pipeline automation

In some use cases, the manual process of training, validating, and deploying ML models can be sufficient. This manual approach works if your team manages only a few ML models that aren't retrained or aren't changed frequently. In practice, however, models often break down when deployed in the real world because they fail to adapt to changes in the dynamics of environments, or the data that describes such dynamics.

For your ML system to adapt to such changes, you need to apply the following MLOps techniques:

- Automate the execution of the ML pipeline to retrain new models on new data to capture any emerging patterns. CT is discussed later in this document in the ML with Kubeflow Pipelines (#ml-with-kubeflow-pipelines) section.
- Set up a continuous delivery system to frequently deploy new implementations of the *entire* ML pipeline. CI/CD is discussed later in this document in the CI/CD setup for ML on Google Cloud (#setting-up-ci-cd-for-ml-on-google-cloud) section.

You can automate the ML production pipelines to retrain your models with new data. You can trigger your pipeline on demand, on a schedule, on the availability of new data, on model performance degradation, on significant changes in the statistical properties of the data, or based on other conditions.

CI/CD pipeline compared to CT pipeline

The availability of new data is one trigger to retrain the ML model. The availability of a new implementation of the ML pipeline (including new model architecture, feature engineering, and hyperparameters) is another important trigger to re-execute the ML pipeline. This new implementation of the ML pipeline serves as a new version of the model prediction service, for example, a microservice with a REST API for online serving. The difference between the two cases is as follows:

- To train a new ML model with new data, the previously deployed CT pipeline is executed. No new pipelines or components are deployed; only a new prediction service or newly trained model is served at the end of the pipeline.
- To train a new ML model with a new implementation, a new pipeline is deployed through a CI/CD pipeline.

To deploy new ML pipelines quickly, you need to set up a CI/CD pipeline. This pipeline is responsible for automatically deploying new ML pipelines and components when new implementations are available and approved for various environments (such as development, test, staging, pre-production, canary, and production).

The following diagram shows the relationship between the CI/CD pipeline and the ML CT pipeline.

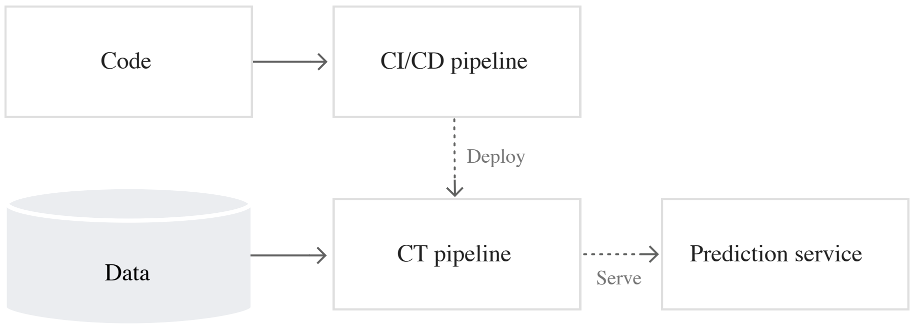


Figure 1. CI/CD and ML CT pipelines.

The output for these pipelines is as follows:

- If given new implementation, a successful CI/CD pipeline deploys a new ML CT pipeline.
- If given new data, a successful CT pipeline serves a new model prediction service.

Designing a TFX-based ML system

The following sections discuss how to design an integrated ML system using TensorFlow Extend (TFX) to set up a CI/CD pipeline for the ML system. Although there are several frameworks for building ML models, TFX is an integrated ML platform for developing and deploying production ML systems. A TFX pipeline is a sequence of components that implement an ML system. This TFX pipeline is designed for scalable, high-performance ML tasks. These tasks include modeling, training, validation, serving inference, and managing deployments. The key libraries of TFX are as follows:

- TensorFlow Data Validation (TFDV) (https://github.com/tensorflow/data-validation): Used for detecting anomalies in the data.
- TensorFlow Transform (TFT) (https://www.tensorflow.org/tfx/transform/get_started): Used for data preprocessing and feature engineering.
- TensorFlow Estimators and Keras (https://www.tensorflow.org/overview/): Used for building and training ML models.
- TensorFlow Model Analysis (TFMA) (https://www.tensorflow.org/tfx/model_analysis/get_started): Used for ML model evaluation and analysis.

- TensorFlow Serving (TFServing) (https://www.tensorflow.org/tfx/guide/serving): Used for serving ML models as REST and gRPC APIs.

TFX ML system overview

The following diagram shows how the various TFX libraries are integrated to compose an ML system.

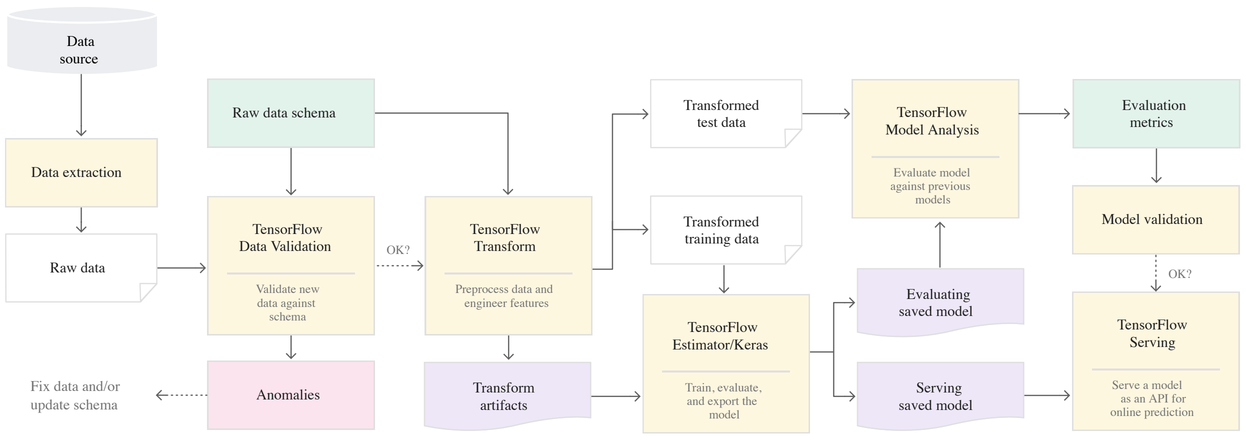

Data source
Data extraction
Raw data
Fix data and/or update schema
Raw data schema
TensorFlow Data Validation
Validate new data against schema
Anomalies
OK?
TensorFlow Transform
Preprocess data and engineer features
Transform artifacts
Transformed test data
Transformed training data
TensorFlow Estimator/Keras
Train, evaluate, and export the model
TensorFlow Model Analysis
Evaluate model against previous models
Evaluating saved model
Serving saved model
Evaluation metrics
Model validation
OK?
TensorFlow Serving
Serve a model as an API for online prediction

Figure 2. A typical TFX-based ML system.

Figure 2 shows a typical TFX-based ML system. The following steps can be completed manually or by an automated pipeline:

1. Data extraction: The first step is to extract the new training data from its data sources. The outputs of this step are data files that are used for training and evaluating the model.

2. Data validation: TFDV validates the data against the expected (raw) data schema. The data schema is created and fixed during the development phase, before system deployment. The data validation steps detect anomalies related to both data distribution and schema skews. The outputs of this step are the anomalies (if any) and a decision on whether to execute downstream steps or not.

3. Data transformation: After the data is validated, the data is split and prepared for the ML task by performing data transformations and feature engineering operations using TFT. The outputs of this step are data files to train and evaluate the model, usually transformed in `TFRecords` (https://www.tensorflow.org/tutorials/load_data/tf_records) format. In addition, the transformation artifacts that are produced help with constructing the model inputs and with exporting the saved model after training.

4. Model training and tuning: To implement and train the ML model, use the `tf.estimator` (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator) or

`tf.Keras` (https://www.tensorflow.org/guide/keras) APIs with the transformed data produced by the previous step. To select the parameter settings that lead to the best model, you can use Keras tuner (https://github.com/keras-team/keras-tuner), a hyperparameter tuning library for Keras, or you can use other services like Katib (https://github.com/kubeflow/katib). The output of this step is a saved model that is used for evaluation, and another saved model that is used for online serving of the model for prediction.

5. Model evaluation and validation: When the model is exported after the training step, it's evaluated on a test dataset to assess the model quality by using TFMA. TFMA evaluates the model quality as a whole, and identifies which part of the data model isn't performing. This evaluation helps guarantee that the model is promoted for serving only if it satisfies the quality criteria. The criteria can include fair performance on various data subsets (for example, demographics and locations), and improved performance compared to previous models or a benchmark model. The output of this step is a set of performance metrics and a decision on whether to promote the model to production
6. Model serving for prediction: After the newly trained model is validated, it's deployed as a microservice to serve online predictions using TensorFlow Serving. The output of this step is a deployed prediction service of the trained ML model. You can replace this step by storing the trained model in a model registry. Subsequently a separate model serving CI/CD process is launched.

For a tutorial that shows how to use the TFX libraries, see ML with TensorFlow Extended (TFX) (https://github.com/GoogleCloudPlatform/tf-estimator-tutorials/blob/master/00_Miscellaneous/tfx/02_tfx_end_to_end.ipynb)
.

TFX ML system on Google Cloud

In a production environment, the components of the system have to run at scale on a reliable platform. The following diagram shows how each step of the TFX ML pipeline runs using a managed service on Google Cloud, which ensures agility, reliability, and performance at a large scale.

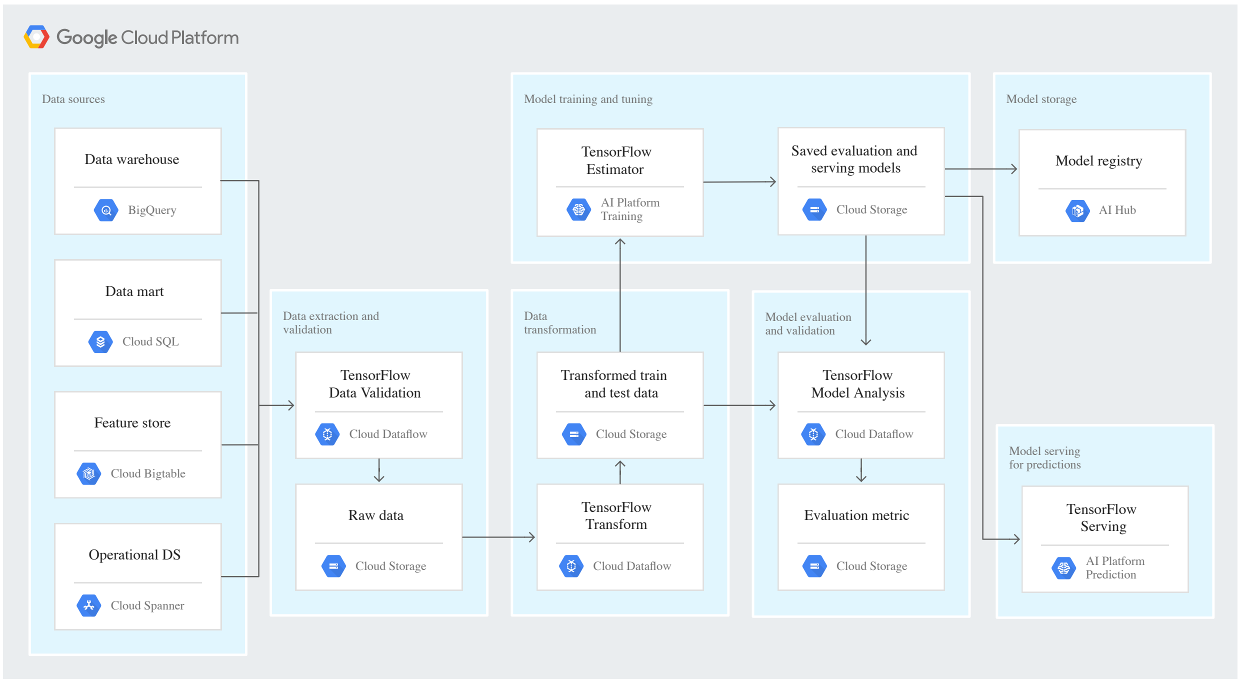


Figure 3. TFX-based ML system on Google Cloud.

The following table describes the key Google Cloud services shown in figure 3:

Step	**TFX library**	**Google Cloud service**
Data extraction and validation	TensorFlow Data Validation	Dataflow (/dataflow)
Data transformation	TensorFlow Transform	Dataflow (/dataflow)
Model training and tuning	TensorFlow	AI Platform Training (/ml-engine/docs/tensorflow/training-overview)
Model evaluation and validation	TensorFlow Model Analysis	Dataflow (/dataflow)
Model serving for prediction	TensorFlow Serving	AI Platform Prediction (/ai-platform)
Model artifacts storage	NA	AI Hub (/ai-hub)

- Dataflow (/dataflow/docs) is a fully managed, serverless, and reliable service for running Apache Beam (https://beam.apache.org/) pipelines at scale on Google Cloud. Dataflow is used to scale the following processes:
    - Computing the statistics to validate the incoming data.

    - Performing data preparation and transformation.
    - Evaluating the model on a large dataset.
    - Computing metrics on different aspects of the evaluation dataset.
- Cloud Storage (/storage) is a highly available and durable storage for binary large objects. Cloud Storage hosts artifacts produced throughout the execution of the ML pipeline, including the following:
    - Data anomalies (if any)
    - Transformed data and artifacts
    - Exported (trained) model
    - Model evaluation metrics
- AI Hub (/ai-hub/docs) is an enterprise-grade hosted repository for discovering, sharing, and reusing artificial intelligence (AI) and ML assets. To store trained and validated models, plus their relevant metadata, you can use AI Hub as a model registry.
- AI Platform (/ml-engine/docs) is a managed service to train and serve ML models at scale. AI Platform Training (/ml-engine/docs/tensorflow/training-overview) not only supports TensorFlow, Scikit-learn (/ml-engine/docs/scikit/training-scikit-learn), and XGboost (/ml-engine/docs/scikit/training-scikit-learn) models, but also supports models implemented in any framework using a user-provided custom container (/ml-engine/docs/custom-containers-training). In addition, a scalable, Bayesian (https://wikipedia.org/wiki/Bayesian_inference) optimization-based service for a hyperparameter tuning (/ml-engine/docs/tensorflow/hyperparameter-tuning-overview) is available. Trained models can be deployed to AI Platform Prediction (/ml-engine/docs/tensorflow/prediction-overview) as a microservice that has a REST API.

Orchestrating the ML system using Kubeflow Pipelines

This document has covered how to design a TFX-based ML system, and how to run each component of the system at scale on Google Cloud. However, you need an orchestrator in order to connect these different components of the system together. The orchestrator runs the pipeline in a sequence, and automatically moves from one step to another based on the defined conditions. For example, a defined condition might be executing the model serving step after the model evaluation step if the evaluation metrics meet predefined thresholds. Orchestrating the ML pipeline is useful in both the development and production phases:

- During the development phase, orchestration helps the data scientists to run the ML experiment, instead of manually executing each step.
- During the production phase, orchestration helps automate the execution of the ML pipeline based on a schedule or certain triggering conditions.

ML with Kubeflow Pipelines

Kubeflow (https://www.kubeflow.org/) is an open source Kubernetes (https://kubernetes.io/) framework for developing and running portable ML workloads. Kubeflow Pipelines (https://www.kubeflow.org/docs/pipelines/pipelines-overview/) is a Kubeflow service that lets you compose, orchestrate, and automate ML systems, where each component of the system can run on Kubeflow, Google Cloud, or other cloud platforms. The Kubeflow Pipelines platform consists of the following:

- A user interface for managing and tracking experiments, jobs, and runs.
- An engine for scheduling multistep ML workflows. Kubeflow pipelines use Argo (https://argoproj.github.io/projects/argo) to orchestrate Kubernetes resources.
- A Python SDK (https://www.kubeflow.org/docs/pipelines/sdk/) for defining and manipulating pipelines and components.
- Notebooks for interacting with the system using the Python SDK.
- An ML Metadata (https://www.kubeflow.org/docs/components/metadata/) store to save information about executions, models, datasets, and other artifacts.

The following constitutes a Kubeflow pipeline:

- A set of containerized ML tasks, or *components*. A pipeline component is self-contained code that is packaged as a Docker image (https://docs.docker.com/engine/reference/commandline/image/). A component takes input arguments, produces output files, and performs one step in the pipeline.
- A specification of the sequence of the ML tasks, defined through a Python *domain-specific language (DSL)* (https://wikipedia.org/wiki/Domain-specific_language). The topology of the workflow is implicitly defined by connecting the outputs of an upstream step to the inputs of a downstream step. A step in the pipeline definition invokes a component in the pipeline. In a complex pipeline, components can execute multiple times in loops, or they can be executed conditionally.
- A set of pipeline input parameters, whose values are passed to the components of the pipeline, including the criteria for filtering data and where to stored the artifacts that the pipeline produces.

The following diagram shows a sample graph of Kubeflow Pipelines.

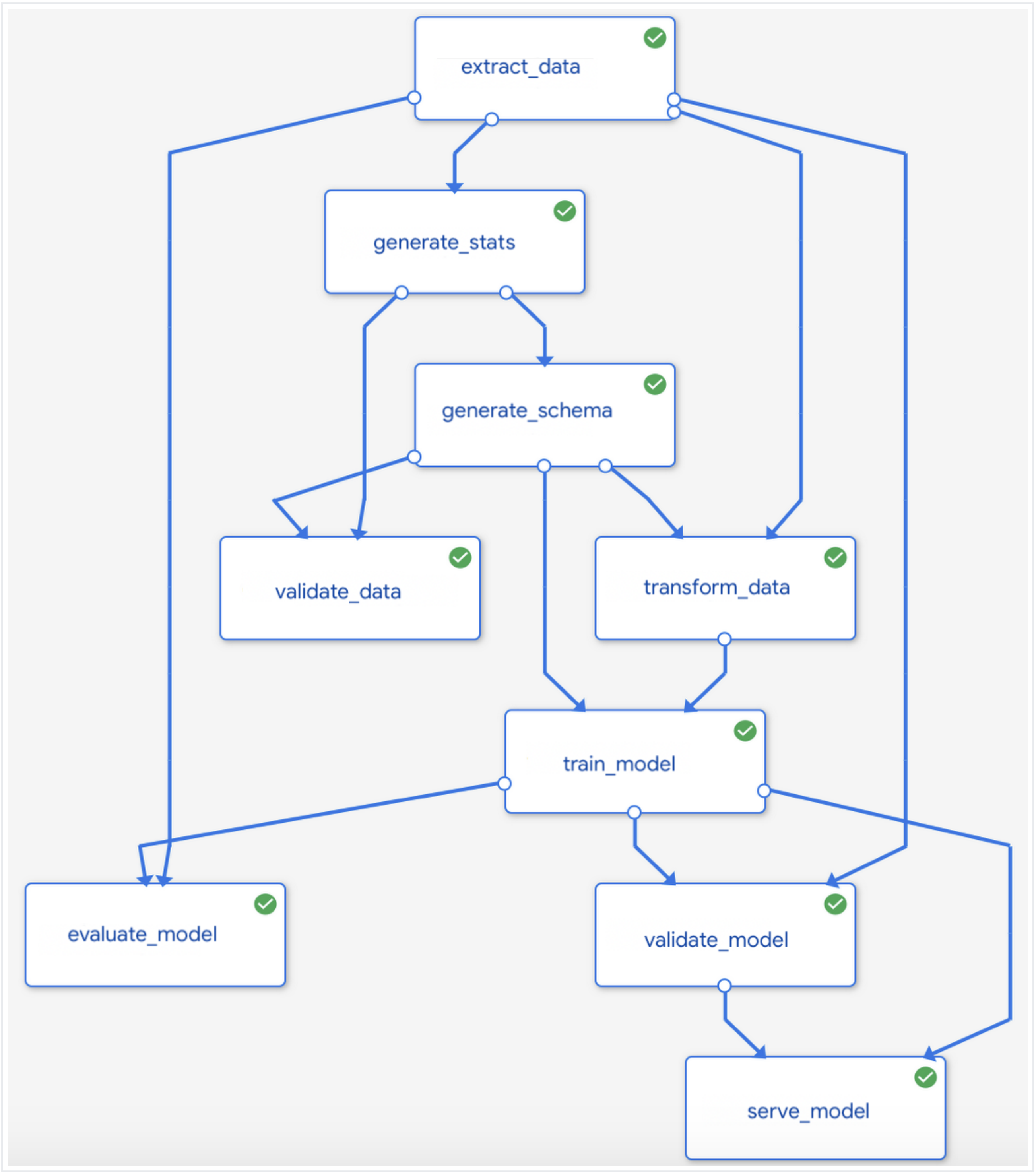


Figure 4. A sample graph of Kubeflow Pipelines.

Kubeflow Pipelines components

For a component to be invoked in the pipeline, you need to create a *component op*. You can create a component op using the following methods:

- Implementing a lightweight Python component (https://www.kubeflow.org/docs/pipelines/sdk/lightweight-python-components/): This component doesn't require that you build a new container image for every code change, and is intended for fast iteration in a notebook environment. You can create a lightweight component from your Python function using the `kfp.components.func_to_container_op` (https://kubeflow-pipelines.readthedocs.io/en/latest/source/kfp.components.html?highlight=func_to_container#kfp.components.func_to_container_op) function.

- Creating reusable component (https://www.kubeflow.org/docs/pipelines/sdk/component-development/): This functionality requires that your component incudes a component specification (https://www.kubeflow.org/docs/pipelines/reference/component-spec/) in the `component.yaml` file. The component specification describes the component to the Kubeflow Pipelines in terms of arguments, the Docker container image URL to execute, and the outputs. Component ops are automatically created from the `component.yaml` files using the `ComponentStore.load_components` (https://kubeflow-pipelines.readthedocs.io/en/latest/source/kfp.components.html#kfp.components.ComponentStore.load_component) function in the Kubeflow Pipelines SDK (https://kubeflow-pipelines.readthedocs.io/en/latest/index.html) during pipeline compilation. Reusable `component.yaml` specifications can be shared to AI Hub (/ai-hub) for composability in different Kubeflow Pipelines projects.

- Using predefined Google Cloud components (https://github.com/kubeflow/pipelines/tree/master/components/gcp): Kubeflow Pipelines provides predefined components that execute various managed services on Google Cloud by providing the required parameters. These components help you execute tasks using services such as BigQuery (https://github.com/kubeflow/pipelines/tree/master/components/gcp/bigquery), Dataflow (https://github.com/kubeflow/pipelines/tree/master/components/gcp/dataflow), Dataproc (https://github.com/kubeflow/pipelines/tree/master/components/gcp/dataproc), and AI Platform (https://github.com/kubeflow/pipelines/tree/master/components/gcp/ml_engine). These predefined Google Cloud components are also available in AI Hub. Similar to using reusable components, these component ops are automatically created from the predefined component specifications through `ComponentStore.load_components` (https://kubeflow-pipelines.readthedocs.io/en/latest/source/kfp.components.html#kfp.components.ComponentStore.load_component) . Other predefined components (https://github.com/kubeflow/pipelines/tree/master/components) are available for executing jobs in Kubeflow and other platforms.

You can also use the TFX Pipeline DSL (https://www.tensorflow.org/tfx/guide#core_concepts) and use TFX components (https://www.tensorflow.org/tfx/guide#anatomy_of_a_component). A TFX component encapsulates metadata functionality. The driver supplies metadata to the executor by querying the metadata store. The publisher accepts the results of the executor and stores them in metadata. You can also implement your custom component (https://www.tensorflow.org/tfx/guide/custom_component), which has the same integration with the metadata. TFX provides a command-line interface (CLI) that compiles the pipeline's Python code to a YAML file and describes the Argo workflow (https://argoproj.github.io/projects/argo). Then you can submit the file to Kubeflow Pipelines.

The following diagram shows how in Kubeflow Pipelines, a containerized task can invoke other services such as BigQuery jobs, AI Platform (distributed) training jobs, and Dataflow jobs.

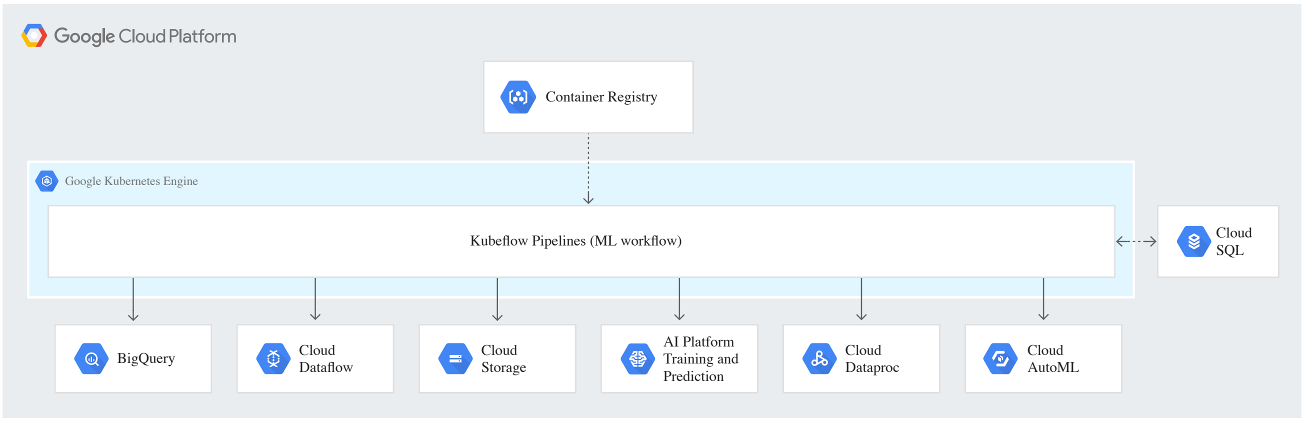


Figure 5. ML pipeline with Kubeflow Pipelines and Google Cloud managed services.

Kubeflow Pipelines lets you orchestrate and automate a production ML pipeline by executing the required Google Cloud services. In figure 5, Cloud SQL serves as the ML metadata store for Kubeflow Pipelines.

Kubeflow Pipelines components aren't limited to executing TFX-related services on Google Cloud. These components can execute any data-related and compute-related services, including Dataproc (/dataproc) for SparkML jobs, AutoML (/automl), and other compute workloads.

Containerizing tasks in Kubeflow Pipelines has the following advantages:

- Decouples the execution environment from your code runtime.

- Provides reproducibility of the code between the development and production environment, because the things you test are the same in production.
- Isolates each component in the pipeline; each can have its own version of the runtime, different languages, and different libraries.
- Helps with composition of complex pipelines.
- Integrates with ML metadata store for traceability and reproducibility.

For a comprehensive introduction to Kubeflow pipelines and an example with TFX libraries, see the Getting started with Kubeflow Pipelines (/blog/products/ai-machine-learning/getting-started-kubeflow-pipelines)blog post.

Triggering and scheduling Kubeflow Pipelines

When you deploy a Kubeflow pipeline to production, you need to automate its executions, depending on the scenarios discussed in the ML pipeline automation (#ml-pipeline-automation) section.

Kubeflow Pipelines provides a Python SDK to operate the pipeline programmatically. The `kfp.Client` (https://kubeflow-pipelines.readthedocs.io/en/latest/source/kfp.client.html) class includes APIs to create experiments, and to deploy and run pipelines. By using the Kubeflow Pipelines SDK, you can invoke Kubeflow Pipelines using the following services:

- On a schedule, using Cloud Scheduler (/scheduler).
- Responding to an event, using Pub/Sub (/pubsub) and Cloud Functions (/functions). For example, the event can be the availability of new data files in a Cloud Storage bucket.
- As part of a bigger data and process workflow, using Cloud Composer (/composer/docs)or Cloud Data Fusion (/data-fusion).

Kubeflow Pipelines also provides a built-in scheduler for recurring pipelines in Kubeflow Pipelines.

Setting up CI/CD for ML on Google Cloud

Kubeflow Pipelines enables you to orchestrate ML systems that involve multiple steps, including data preprocessing, model training and evaluation, and model deployment. In the data science exploration phase, Kubeflow Pipelines helps with rapid experimentation of the whole system. In the production phase, Kubeflow Pipelines enables you to automate the pipeline execution based on new data to train or retrain the ML model.

CI/CD architecture

The following diagram shows a high-level overview of CI/CD for ML with Kubeflow pipelines.

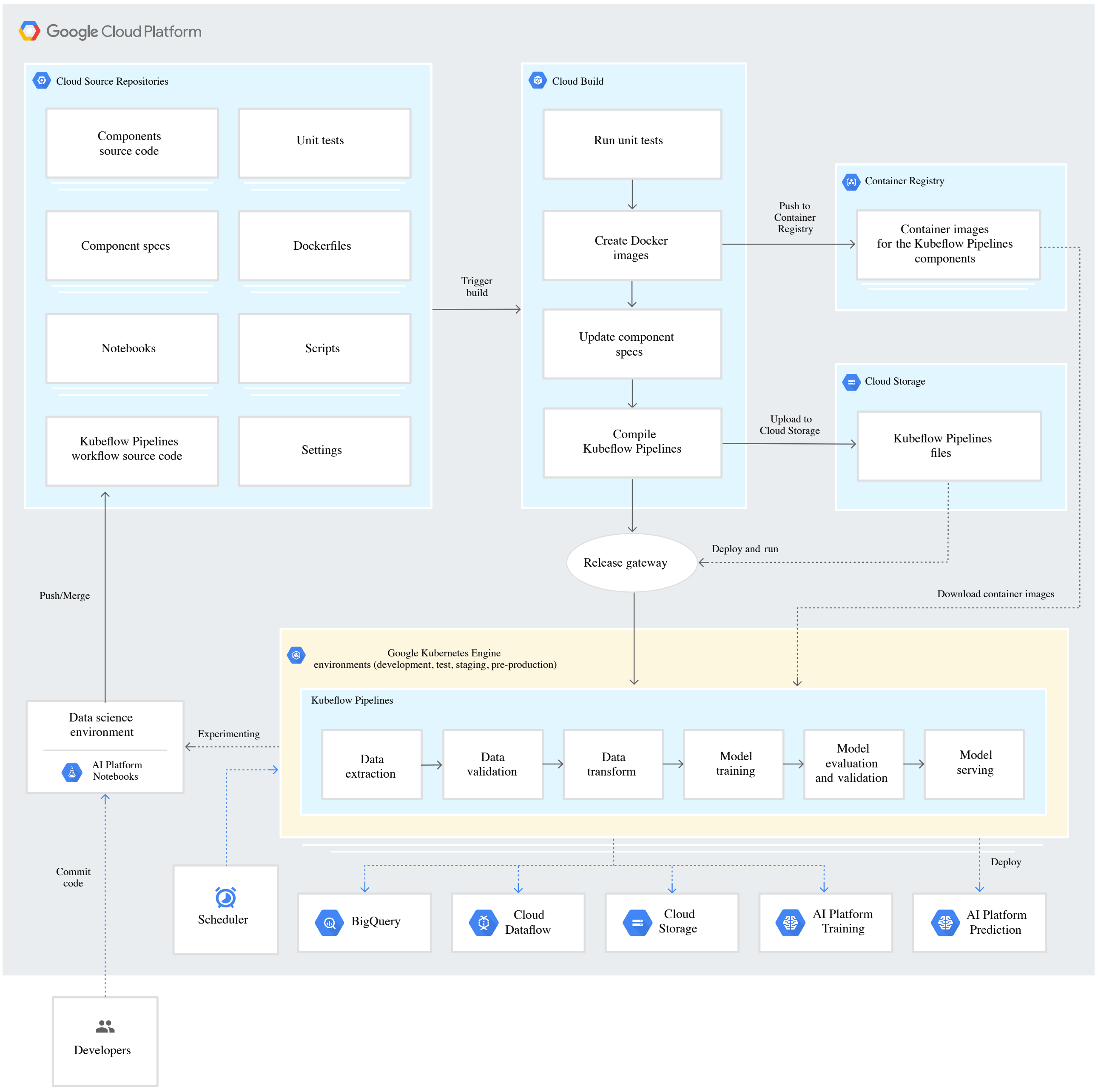


Figure 6: High-level overview of CI/CD for Kubeflow pipelines.

At the heart of this architecture is Cloud Build (/cloud-build), a managed service that executes your builds on Google Cloud infrastructure. Cloud Build can import source from Cloud Source Repositories (/source-repositories), GitHub (https://github.com/), or Bitbucket (https://bitbucket.org/), and then execute a build to your specifications, and produce artifacts such as Docker containers or Python tar files.

Cloud Build executes your build as a series of build steps, defined in a build configuration file (/cloud-build/docs/build-config) (`cloulbuild.yaml`). Each build step is run in a Docker container. You can either use the supported build steps (/cloud-build/docs/configuring-builds/build-test-deploy-artifacts) provided by Cloud Build, or write your own build steps (/cloud-build/docs/create-custom-build-steps).

The Cloud Build process, which performs the required CI/CD for your ML system, can be executed either manually (/cloud-build/docs/running-builds/start-build-manually) or through automated build triggers (/cloud-build/docs/running-builds/automate-builds). Triggers execute your configured build steps whenever changes are pushed to the build source. You can set a build trigger to execute the build routine on changes to the source repository, or to execute the build routine only when changes match certain criteria.

In addition, you can have build routines (Cloud Build configuration files) that are executed in response to different triggers. For example, you can have the following setup:

- A build routine starts when commits are made to a development branch.
- A build routine starts when commits are made to the master branch.

You can use configuration variable substitutions (/cloud-build/docs/build-config#substitutions) to define the environment variables at build time. These substitutions are captured from triggered builds. These variables include `$COMMIT_SHA`, `$REPO_NAME`, `$BRANCE_NAME`, `$TAG_NAME`, and `$REVISION_ID`. Other non-trigger-based variables are `$PROJECT_ID` and `$BUILD_ID`. Substitutions are helpful for variables whose value isn't known until build time, or to reuse an existing build request with different variable values.

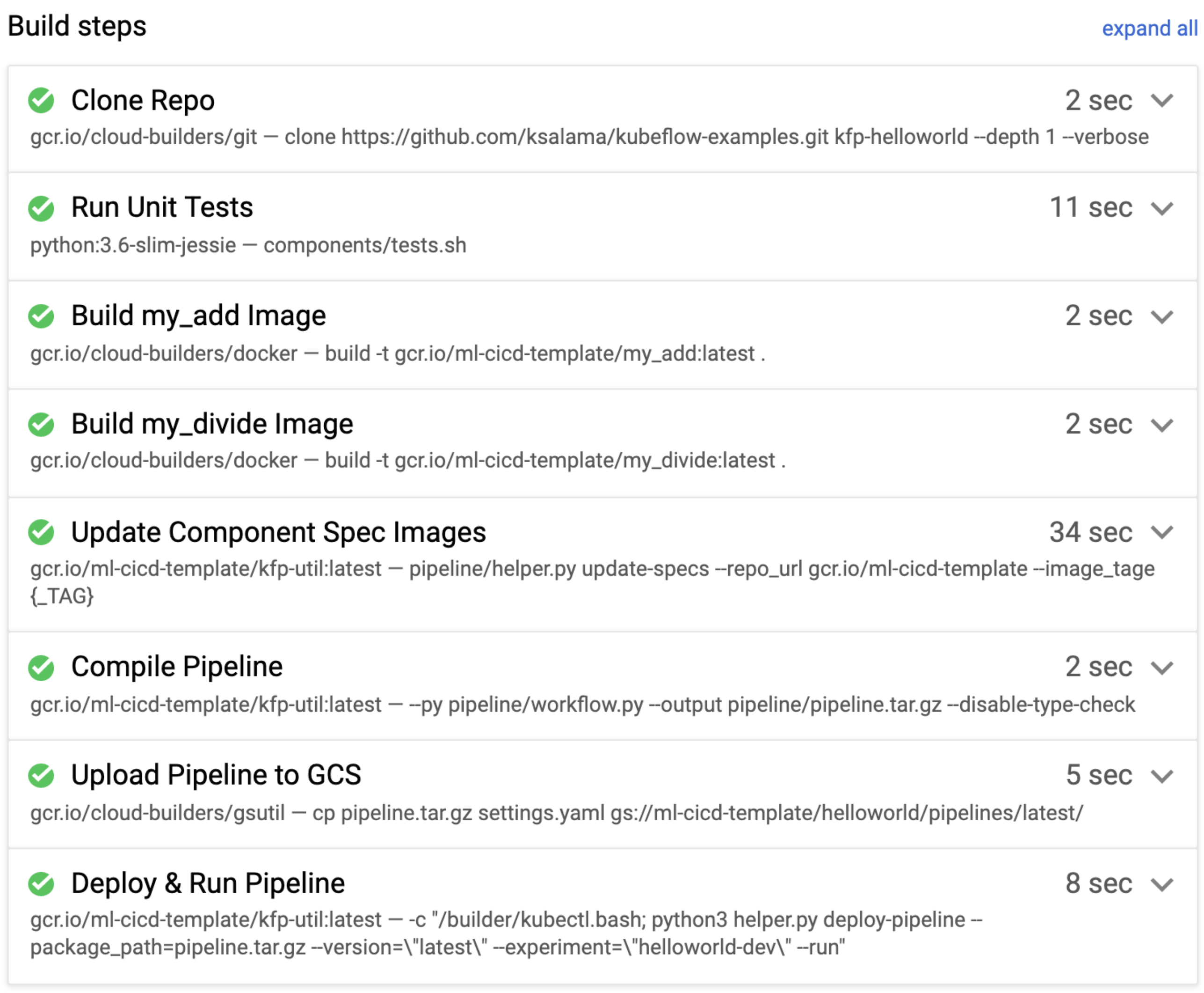
CI/CD workflow use case

A source code repository typically includes the following items:

- The Python source code for implementing the components of Kubeflow Pipelines, including data validation, data transformation, model training, model evaluation, and model serving.
- Python unit tests to test the methods implemented in the component.
- Dockerfiles that are required in order to create Docker container images, one for each Kubeflow Pipelines component.
- The `component.yaml` file that defines the pipeline component specifications (https://www.kubeflow.org/docs/pipelines/reference/component-spec/). These specifications are used to generate the component ops in the pipeline definition.

- A Python module (for example, the `pipeline.py` module) where the Kubeflow Pipelines workflow is defined.
- Other scripts and configuration files, including the `cloudbuild.yaml` files.
- Notebooks used for exploratory data analysis, model analysis, and interactive experimentation on models.
- A settings file (for example, the `settings.yaml` file), including configurations to the pipeline input parameters.

In the following example, a build routine is triggered when a developer pushes source code to the development branch from their data science environment.



Build steps expand all

✓ Clone Repo 2 sec
gcr.io/cloud-builders/git — clone https://github.com/ksalama/kubeflow-examples.git kfp-helloworld --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"

Figure 7. Example build steps performed by Cloud Build.

As shown in figure 7, Cloud Build performs the following build steps:

1. The source code repository is copied to the Cloud Build runtime environment, under the `/workspace` directory.

2. Unit tests are run.
3. (Optional) Static code analysis is run, such as Pylint (http://pylint.pycqa.org/en/latest/).
4. If the tests pass, the Docker container images are built, one for each Kubeflow Pipelines component. The images are tagged with the `$COMMIT_SHA` parameter.
5. The Docker container images are uploaded to the Container Registry.
6. The image URL is updated in each of the `component.yaml`files with the created and tagged Docker container images.
7. The Kubeflow Pipelines workflow is compiled to produce the `workflow.tar.gz` file.
8. The `workflow.tar.gz` file is uploaded to Cloud Storage.
9. The compiled pipeline is deployed to Kubeflow Pipelines, which involves the following steps:
    a. Read the pipeline parameters from the `settings.yaml` file.
    b. Create an experiment (or use an existing one).
    c. Deploy the pipeline to Kubeflow Pipelines (and tag its name with a version).
10. (Optional) Run the pipeline with the parameter values as part of an integration test or production execution. The executed pipeline eventually deploys a model as an API on AI Platform.

For a comprehensive Cloud Build example that covers most of these steps, see A Simple CI/CD Example with Kubeflow Pipelines and Cloud Build (https://github.com/ksalama/kubeflow-examples/tree/master/kfp-cloudbuild).

Additional considerations

When you set up the ML CI/CD architecture on Google Cloud, consider the following:

- For the data science environment, you can use a local machine, or an AI Platform Notebooks instance (/ai-platform-notebooks) that is based on a Deep Learning VM (/deep-learning-vm/docs) (DLVM).
- You can configure the automated Cloud Build pipeline to skip triggers (/cloud-build/docs/running-builds/automate-builds#skipping_a_build_trigger), for example, if only documentation files are edited, or if the experimentation notebooks are modified.
- You can execute the pipeline for integration and regression testing as a build test. Before the pipeline is deployed to the target environment, you can use the

`wait_for_pipeline_completion` (https://kubeflow-pipelines.readthedocs.io/en/latest/source/kfp.client.html#kfp.Client.wait_for_run_completion) method to execute the pipeline on a sample dataset to test the pipeline.

- As an alternative to using Cloud Build, you can use other build systems such as Jenkins (https://jenkins.io/). A ready-to-go deployment of Jenkins is available on Google Cloud Marketplace. (https://console.cloud.google.com/marketplace/details/google/jenkins)
- You can configure the pipeline to deploy automatically to different environments, including development, test, and staging, based on different triggers. In addition, you can deploy to particular environments manually, such as pre-production or production, typically after getting a release approval. You can have multiple build routines for different triggers or for different target environments.
- You can use Apache Airflow (https://airflow.apache.org/), a popular orchestration and scheduling framework, for general-purpose workflows, which you can run using the fully managed Cloud Composer (/composer/docs) service. For more information on how to orchestrate data pipelines with Cloud Composer and Cloud Build, see Setting up a CI/CD pipeline for your data-processing workflow (/solutions/cicd-pipeline-for-data-processing).
- When you deploy a new version of the model to production, deploy it as a canary release to get an idea of how it will perform (CPU, memory, and disk usage). Before you configure the new model to serve all live traffic, you can also perform A/B testing. Configure the new model to serve 10% to 20% of the live traffic. If the new model performs better than current one, you can configure the new model to serve all traffic. Otherwise, the serving system rolls back to the current model.

What's next

- Learn more about GitOps-style continuous delivery with Cloud Build (/kubernetes-engine/docs/tutorials/gitops-cloud-build).
- Learn more about Setting up a CI/CD pipeline for your data-processing workflow (/solutions/cicd-pipeline-for-data-processing).
- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).

Last updated 2020-11-16 UTC.