Google Cloud

Partner Certification Academy

Professional Machine Learning Engineer

The information in this presentation is classified:

Google confidential & proprietary

⚠️ This presentation is shared with you under NDA.

- Do **not** record or take screenshots of this presentation.
- Do **not** share or otherwise distribute the information in this presentation with anyone **inside** or **outside** of your organization.

Thank you!

Google Cloud

Source Materials

Some of this program's content has been sourced from the following resources:

- Google Cloud certification site
- Google Cloud documentation
- Google Cloud console
- Google Cloud courses and workshops
- Google Cloud white papers
- Google Cloud Blog
- Google Cloud YouTube channel
- Google Cloud samples
- Google codelabs
- Google Cloud partner-exclusive resources

Google Cloud

Google Cloud Skills Boost for Partners

- Professional Machine Learning Engineer Certification
- Cloud Skills Boost for Partners Professional Machine Learning Engineer Learning Path
- Partner Learning Services Instructor-Led PMLE Curriculum

Google Cloud Partner Advantage

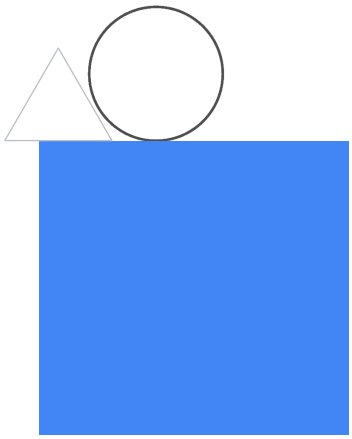
- Best practices for implementing machine learning on Google Cloud
- Artificial Intelligence
- End-to-End MLOps Go-to-Market Kit

Session Logistics

- When you have a question, please:
  - Click the Raise hand button in Google Meet.
  - Or add your question to the Q&A section of Google Meet.
  - Please note that answers may be deferred until the end of the session.
- These slides are available in the Student Lecture section of your Qwiklabs classroom.
- The session is **not recorded.**
- Google Meet does not have persistent chat.
  - If you get disconnected, you will lose the chat history.
  - Please copy any important URLs to a local text file as they appear in the chat.

Google Cloud Partner Learning Programs

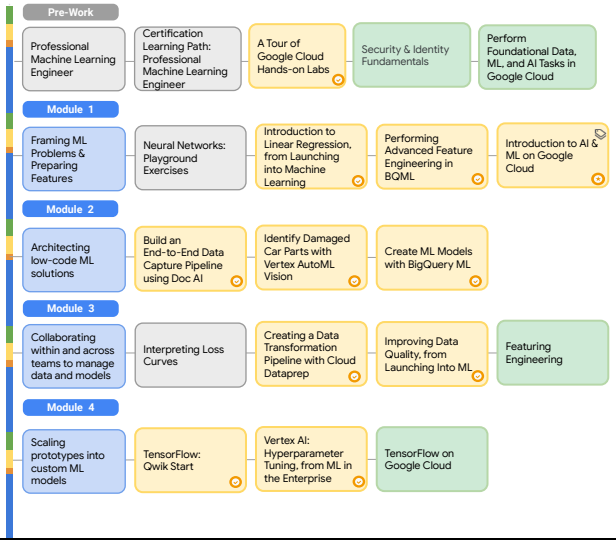
- Partner Certification Academy
- Partner Delivery Readiness Index (DRI)
- Cloud Skills Boost for Partners
- Partner Advantage



PARTNER CERTIFICATION ACADEMY

Professional Machine Learning Engineer

On-demand Course
Hands-on Skill Badge
Hands-on Lab
Resource
Live Virtual Workshop
required to earn exam voucher



A Professional Machine Learning Engineer builds, evaluates, productionizes, and optimizes ML models by using Google Cloud technologies and knowledge of proven models and techniques. The ML Engineer:

- handles large, complex datasets and creates repeatable, reusable code.
- considers responsible AI and fairness throughout the ML model development process, and collaborates closely with other job roles to ensure long-term success of ML-based applications.
- has strong programming skills and experience with data platforms and distributed data processing tools.
- is proficient in the areas of model architecture, data and ML pipeline creation, and metrics interpretation.
- is familiar with foundational concepts of MLOps, application development, infrastructure management, data engineering, and data governance.
- makes ML accessible and enables teams across the organization.

By training, retraining, deploying, scheduling, monitoring, and improving models, the ML Engineer designs and creates scalable, performant solutions.

Recommended candidate:

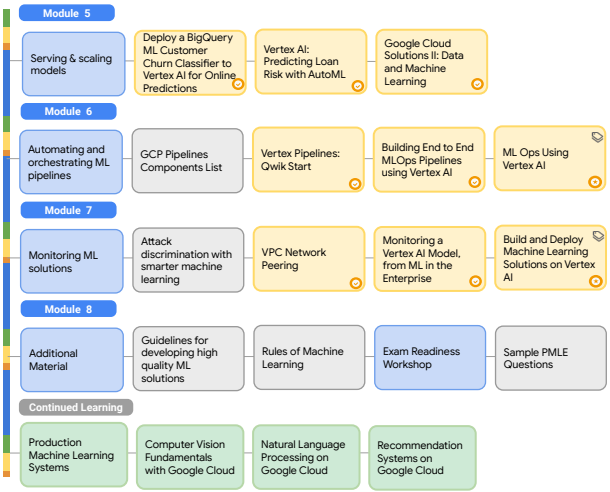
- Has in-depth experience setting up cloud environments for an organization
- Has experience deploying services and solutions based on business requirements

Google Cloud

PARTNER CERTIFICATION ACADEMY

Professional Machine Learning Engineer

On-demand Course
Hands-on Skill Badge
Hands-on Lab
Resource
Live Virtual Workshop
required to earn exam voucher



A Professional Machine Learning Engineer builds, evaluates, productionizes, and optimizes ML models by using Google Cloud technologies and knowledge of proven models and techniques. The ML Engineer:

- handles large, complex datasets and creates repeatable, reusable code.
- considers responsible AI and fairness throughout the ML model development process, and collaborates closely with other job roles to ensure long-term success of ML-based applications.
- has strong programming skills and experience with data platforms and distributed data processing tools.
- is proficient in the areas of model architecture, data and ML pipeline creation, and metrics interpretation.
- is familiar with foundational concepts of MLOps, application development, infrastructure management, data engineering, and data governance.
- makes ML accessible and enables teams across the organization.

By training, retraining, deploying, scheduling, monitoring, and improving models, the ML Engineer designs and creates scalable, performant solutions.

Recommended candidate:

- Has in-depth experience setting up cloud environments for an organization
- Has experience deploying services and solutions based on business requirements

Google Cloud

Learner Commitment

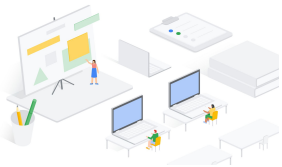
Each week, learners are to complete the learning path's course content, Cloud Skills Boost for Partner Quests/Challenge Labs and material that the mentor has recommended that will support learning.

- **Workshop Day:** Meet for the cohort's weekly 'general session'. **(≅ 2 hours)**
- **During the week:** Complete the week's course, perform hands-on labs, review any additional material suggested material for the week. **(≅ 8 - 16 hours)**

➔ **Important:** Learners must allocate time between each weekly session to study and familiarize themselves with any prerequisite knowledge they may lack. It is also recommended that learners complete the next week's course prior to the scheduled workshop.

Path to Service Excellence

Certification



Advanced Solutions Training

Delivery Readiness Index

Certification is just one step on your professional journey. Google Cloud also offers our partners access to advanced solutions training, and a new quality-focused program called Delivery Readiness Index (DRI) to help you achieve service excellence with your customers.

Benchmark your skills with DRI

DRI helps to benchmark partner proficiency and capability at any point during the customer journey however should be used primarily as a lead measure to predict and prepare for partner delivery success.

DRI assesses and analyzes Partner Consultant GCP proficiency by creating a DRI Profile inclusive of their GCP knowledge, skills, and experience.

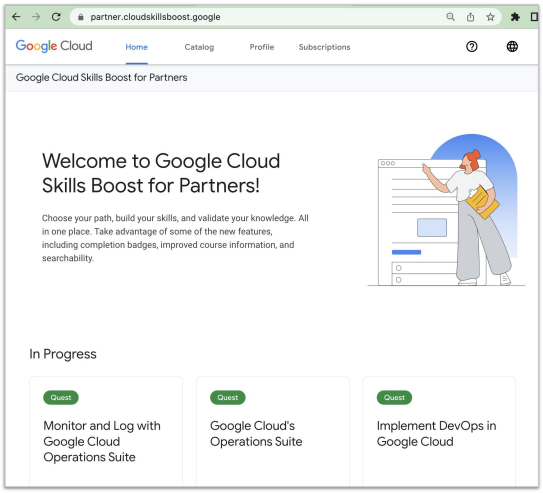
With the DRI insights, we can prescriptively advise the partner project team on the ground and bridge niche capability gaps.

DRI also takes action. For partner consultants, DRI generates a tailored L&D plan that prescribes personalized learning, training, and skill development to build GCP proficiency.

Google Cloud Skills Boost for Partners

https://partner.cloudskillsboost.google/

- On-demand course content
- Hands-on labs
- Skill Badges
- **FREE** to Google Cloud Partners!

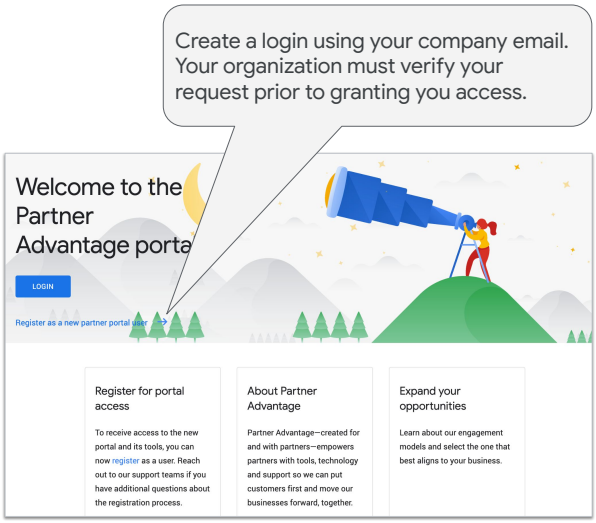


Proprietary + Confidential

Google Cloud

Partner Advantage

- Resources for Google Cloud partner organizations:
  - Recent announcements
  - Solutions/role-based training
  - Live/pre-recorded webinars on various topics
    - Partner Advantage Live Webinars
- Complements the certification self-study material presented on Google Cloud Skills Boost for Partners
- Helpful Links:
  - Getting started on Partner Advantage
  - Join Partner Advantage
  - Get help accessing Partner Advantage



https://www.partneradvantage.goog

Google Cloud

The getting started link:
https://support.google.com/googlecloud/topic/9198654#zippy=

Note the top section, **"Getting Started & User Guides"** and two key documents → Direct Partners to this if they need to enroll into Partner Advantage

1. Logging in to the Partner Advantage Portal - Quick Reference Guide
2. Enrolling in the Partner Advantage Program - Quick Reference Guide

Focus from this point on:

Some context on enrolling in PA:
Access to Partner Portal is given in 2 ways

- Partner Admin Led: Partner Administrator at Partner Company can set up users
- User Led: User can go through Self Registration
  - https://www.partneradvantage.goog/GCPPRM/s/partneradvantageportallogin?language=en_US
  - Or directly to the User Registration Form, https://www.partneradvantage.goog/GCPPRM/s/partnerselfregistration?language=en_US

Please Note

- After a user self-registers, they receive an email that essentially states:

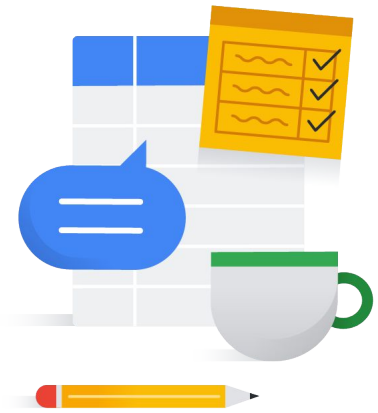
    - “Hi {Partner Name}, you are one step away from joining the Google Cloud Partner Advantage Community. Please click to continue with the user registration process. See you in the cloud, The Partner Advantage Team
- Once registered, they can access limited content until their **Partner Administrator approves the user**
- Their Partner Administrator also receive an email notifying them that a member of their organization has registered themselves on their organization’s Google Cloud Partner Advantage account.
    - It also states that this user has limited access to the portal
    - They are provided instructions on how to review and provision the appropriate access for the user that has registered
- Once their admin approves the user, they receive an email that states:
    - Hi {User Name}, Your Partner Administrator has updated your access to the Google Cloud Partner Advantage portal. You have been granted edit access to additional account information on the portal on behalf of your organization to help build your business. For additional access needs, please work with your Partner Administrator. See you in the cloud, The Partner Advantage Team

The net takeaway is, on the Support Page (the first link on this slide) Google Cloud Partner Advantage Support, there’s a section **“Issue accessing Partner Advantage Portal? Click here for troubleshooting steps”**

- The source of their issue can be related to the different items shown
- Additionally, there’s a Partner Administrator / Partner Adminstrator Team at their partner organization that has to approve their access.. Until that step is completed, they will have access issues/limitation. They will need to identify who this person or team is at their organization

Program issues or concerns?

- Problems with **accessing** Cloud Skills Boost for Partners
  - cloud-partner-training@google.com
- Problems with **a lab** (locked out, etc.)
  - support@qwiklabs.com
- Problems with accessing Partner Advantage
  - https://support.google.com/googlecloud/topic/9198654



- Problems with accessing **Cloud Skills Boost for Partners**
  - cloud-partner-training@google.com
- Problems with a **lab** (locked out, etc.)
  - support@qwiklab.com
- Problems with accessing **Partner Advantage**
  - https://support.google.com/googlecloud/topic/9198654

Module 6

Automating and orchestrating ML pipelines

Module Agenda

01 Developing end-to-end ML pipelines

02 Introduction to MLOps

03 Tracking and auditing experiments & artifact metadata

Developing end-to-end ML pipelines

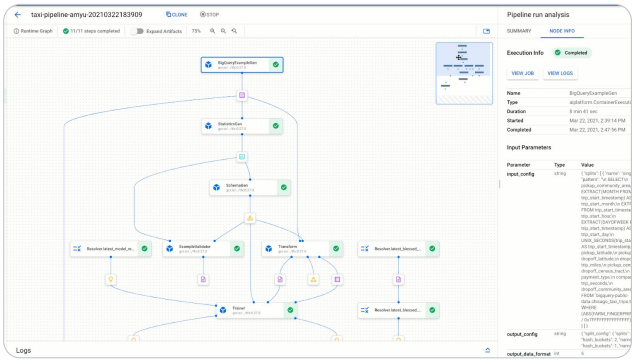
Vertex Pipelines

Streamlines building and running ML pipelines. Simplifies MLOps.

Easy to use Python SDKs
Build pipelines using data scientist–friendly SDKs like TensorFlow Extended and Kubeflow Pipelines.

Automatically tracks metadata for all artifacts produced
Enabling users to build tools for comparing, promoting, and re-training models.

New features
Security features (VPC-SC, CMEK, AxT, regionalization) and advanced flow control (for Loops; Conditional).



Vertex Pipelines supports two pipeline frameworks:

- Kubeflow
- Tensorflow Extended (TFX)

Source: MLOps Architectures with CI/CD (slides) | Y22
https://docs.google.com/presentation/d/1OiGZOviGJf6XX7E91-41PGBy3aoRPZcibmF9dtH1Cms/edit?resourcekey=0-0LF8xS6di73GKb_vAVDZjw#slide=id.g1043fd8a725_0_2031

Vertex AI Pipelines helps you to automate, monitor, and govern your ML systems by orchestrating your ML workflow in a serverless manner, and storing your workflow's artifacts using Vertex ML Metadata. By storing the artifacts of your ML workflow in Vertex ML Metadata, you can analyze the lineage of your workflow's artifacts — for example, an ML model's lineage may include the training data, hyperparameters, and code that were used to create the model.

A COMPARISON OF KUBEFLOW & TFX

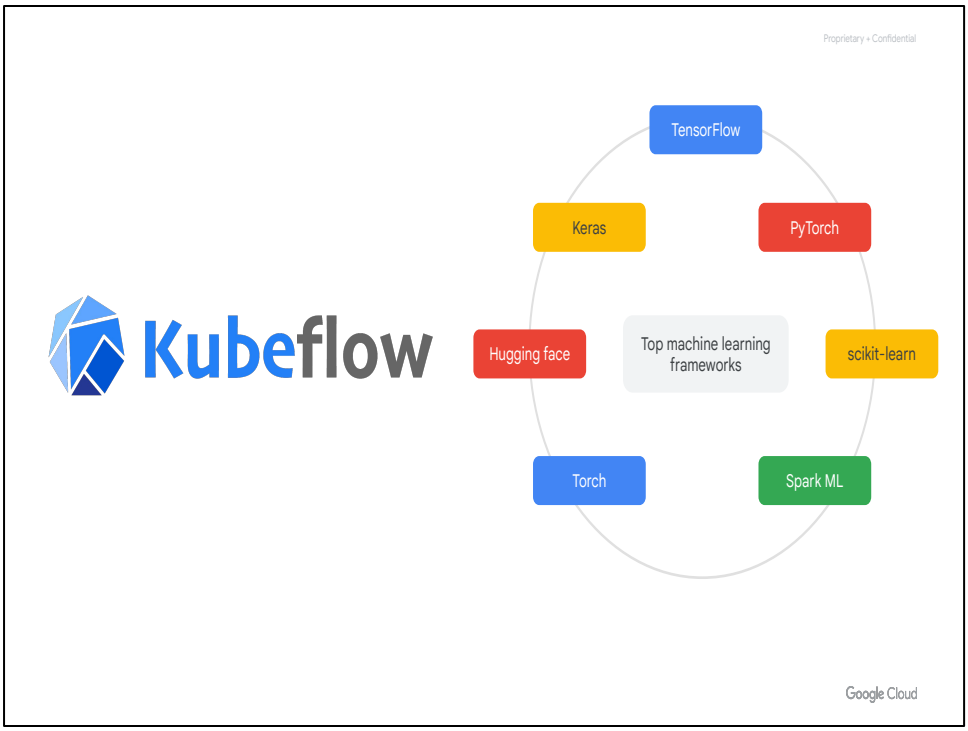
- Multi-framework support: If you need to use multiple machine learning frameworks (e.g., TensorFlow, PyTorch, Scikit-learn) in your workflow, Kubeflow Pipelines can be a good fit.
- Already running kubeflow: Kubernetes-native: If your organization is already using Kubernetes extensively, adopting Kubeflow Pipelines can lead to easier integration and a more consistent experience across your infrastructure.
- Open Sourced: Extensible to other platforms that can run Kubernetes

- TensorFlow-centric: If your project relies primarily on TensorFlow for model training and serving, TFX Pipelines can provide a more cohesive and streamlined experience.
- Pre-built components: TFX offers a set of standard components (e.g., ExampleGen, Transform, Trainer, Evaluator) that follow best practices and can save time in developing your ML workflows.
- Advanced features: TFX Pipelines provide advanced features such as model analysis, validation, and drift detection, which can help ensure your models are robust and maintain high performance over time.

In summary:

KubeFlow:
choose Kubeflow Pipelines in Vertex AI if you need multi-framework support and greater customization

TensorFlow:
Opt for TFX Pipelines if your project is primarily TensorFlow-based, you want to leverage pre-built components, and need advanced features for model management.



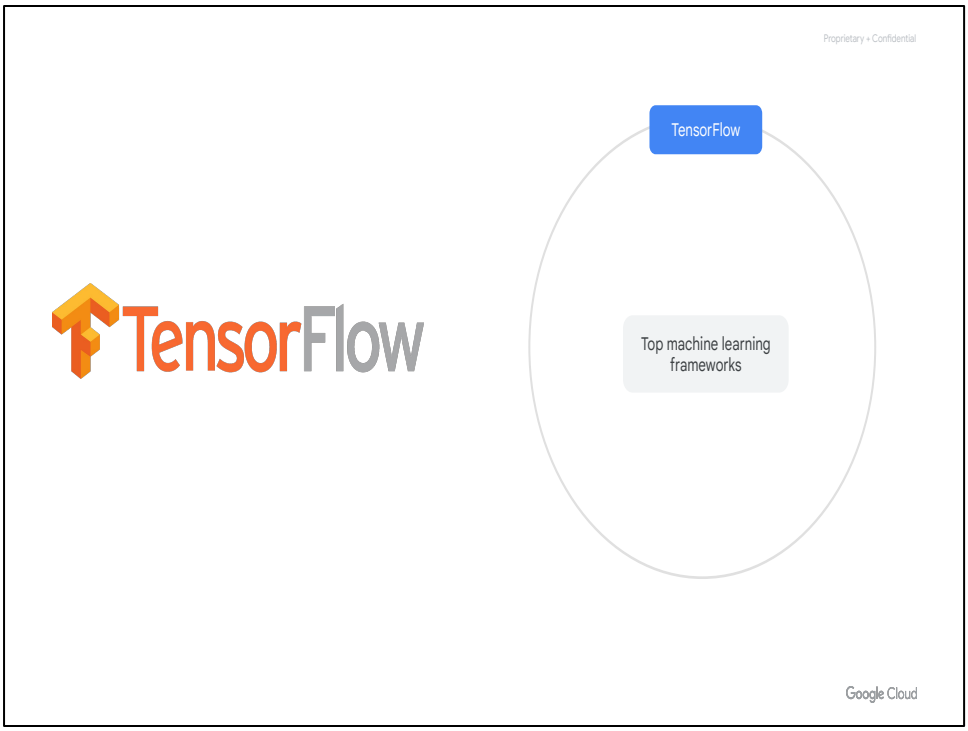
Source: Machine Learning on Google Cloud v3.0

There are many different machine learning frameworks to help solve business requirements and machine learning problems. They could use scikit Learn, Pytorch, or TensorFlow with the Keras API.

The data scientist in our use case has used scikit Learn, a Python library for machine learning.

Wait—but you also may have heard of “deep learning.” So what is the difference between machine learning and deep learning?

First, let’s look at the difference between machine learning and statistics.



There are many different machine learning frameworks to help solve business requirements and machine learning problems. They could use scikit Learn, Pytorch, or TensorFlow with the Keras API.

The data scientist in our use case has used scikit Learn, a Python library for machine learning.

Wait—but you also may have heard of “deep learning.” So what is the difference between machine learning and deep learning?

First, let’s look at the difference between machine learning and statistics.

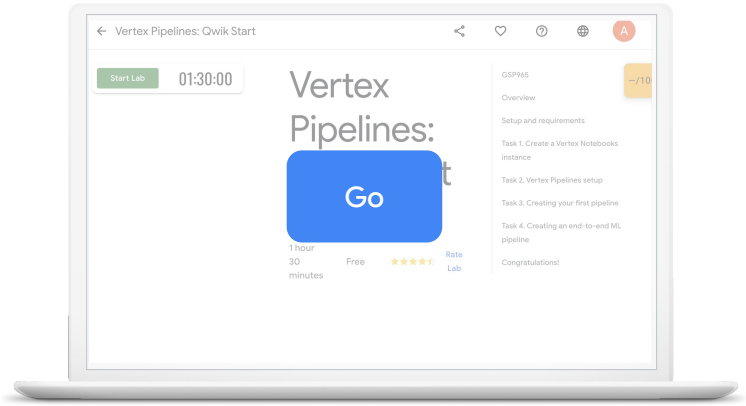
Pipelines and more Pipelines

If your company already relies on third-party pipeline tools like MLFlow, you can install them on GCP using Cloud SQL and Cloud Run.

Link:
https://dlabs.ai/blog/a-step-by-step-guide-to-setting-up-mlflow-on-the-google-cloud-platform/

Recommended Lab

Vertex Pipelines: Qwik Start



From the Course
Build and Deploy Machine Learning Solutions on Vertex AI

Google Cloud

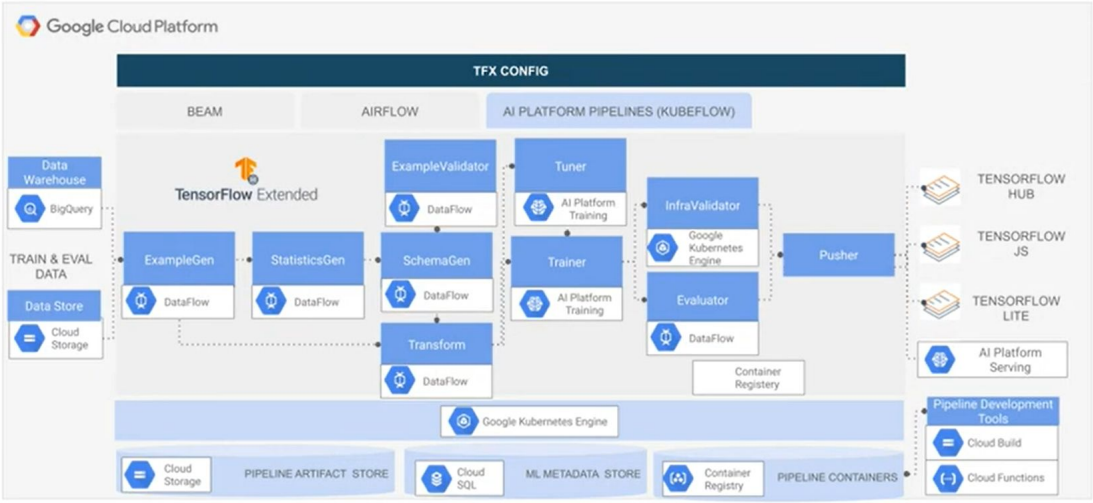
Vertex Pipelines: Qwik Start: https://partner.cloudskillsboost.google/catalog_lab/4078

Other Recommendations:

Course:
Machine Learning Operations (MLOPS) Getting Started. **(enroll on this course first!)**
https://partner.cloudskillsboost.google/course_templates/158

TensorFlow Extended (TFX)

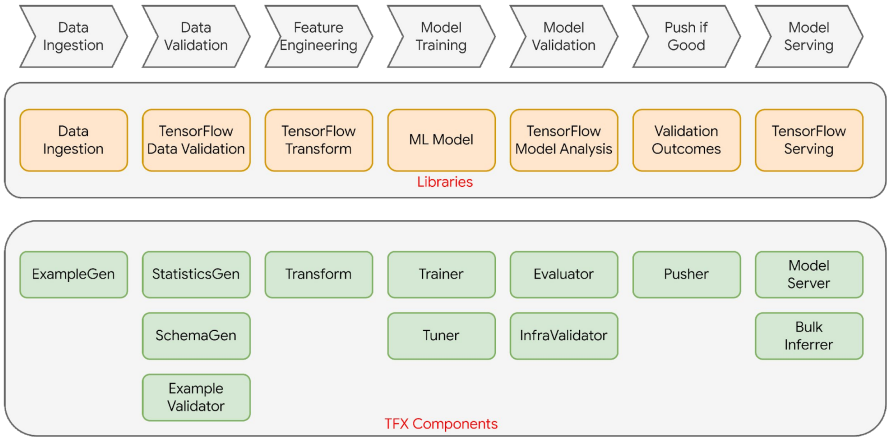
TFX: Google's production machine learning platform



[LAB] partner.cloudskillsboost.google
TFX Standard Components Walkthrough

https://partner.cloudskillsboost.google/course_sessions/2567810/labs/103958

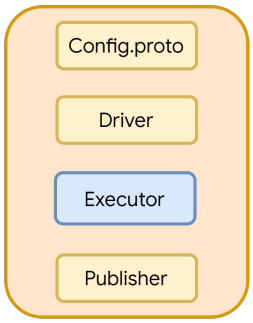
TFX Components



[LAB] partner.cloudskillsboost.google
TFX Standard Components Walkthrough

https://partner.cloudskillsboost.google/course_sessions/2567810/labs/103958

Anatomy of a TFX Component



TFX Component

- Config: Specifies how components communicates between each other via input and output artifact channels. It also includes component parameters such as hyperparameters.
- Driver: gets the artifact locations from the ML metadata store and retrieves artifact from pipeline storage.
- Executor: "the workhorse": implements the actual code of the component such as ingestion/transform/train
- Publisher: Updates the ML Metadata store

TFX components are the building blocks of a TensorFlow Extended (TFX) pipeline, each responsible for a specific task in the machine learning workflow. A TFX component typically consists of four main parts: Config, Driver, Executor, and Publisher.

Config:
The configuration is where you define the settings and parameters for a TFX component. This includes specifying input and output data, defining options for processing or training, and setting up any other component-specific configurations. In simple terms, the Config is like a recipe that tells the component what to do and how to do it.

Driver:
The Driver is responsible for preparing the component for execution. This involves reading the component's configuration, gathering input data, and performing any necessary preprocessing or setup tasks. Think of the Driver as the "helper" that gets everything ready before the main task is executed.

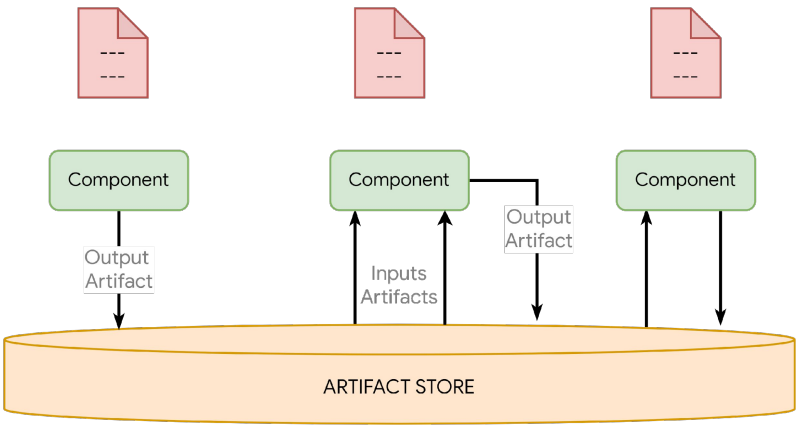
Executor:
The Executor is the "workhorse" of the component, responsible for actually performing the task defined by the component, such as training a model, transforming data, or evaluating model performance. It takes the prepared inputs from the Driver, executes the task according to the Config, and generates the output artifacts.

Publisher:
The Publisher takes the output artifacts generated by the Executor and saves them to a designated location (usually an artifact store). It also updates the ML Metadata service with information about the outputs, making them available for downstream components or future reference. The Publisher is like the "record keeper" that documents the results of the component's work.

In summary, TFX components are the building blocks of a TFX pipeline, each responsible for a specific task in the machine learning workflow. The Config sets up the component's instructions, the Driver prepares the inputs, the Executor performs the main task, and the Publisher saves the outputs and updates the metadata.

- **Config.proto (Recipe):** Imagine a detailed recipe for your favorite dish. This recipe specifies the ingredients (data sources), their preparation steps (data transformations), cooking instructions (training jobs), and even presentation suggestions (evaluation). It's the blueprint for creating your final dish (machine learning model).
- **Driver (Head Chef):** The head chef is the mastermind behind the kitchen. They review the recipe (Config.proto) and delegate tasks (data preparation, training, evaluation) to their team (Executor and Publisher). They ensure everything is done according to the recipe and handle any unexpected situations.
- **Executor (Sous Chefs & Line Cooks):** These are the skilled cooks who execute specific tasks based on the head chef's (Driver) instructions and the recipe (Config.proto). Sous chefs might handle complex tasks like data preparation (chopping, prepping ingredients), while line cooks might focus on specific steps (training - following the cooking instructions).
- **Publisher (Waiter):** Once a dish (trained model) is ready, the waiter takes it from the kitchen (Executor) and delivers it to the table (model registry or deployment location). They ensure the final product is presented well and delivered to the right place.

TFX Component Communication



TFX component communication:
refers to how the different components in a TensorFlow Extended (TFX) pipeline interact with one another and share information as they perform their tasks in the machine learning workflow.

Passing data between components: Components in a TFX pipeline often need to pass data or artifacts (such as datasets, models, or metrics) to one another. For example, a component responsible for data preprocessing might generate a preprocessed dataset that needs to be used by the next component for model training. To accomplish this, TFX components use input and output channels to share artifacts between them.

Dependencies and execution order: The order in which components are executed is important, as some components depend on the outputs of others to function correctly. TFX pipelines define these dependencies and ensure that components are executed in the correct order. For example, a model training component needs to wait until the data preprocessing component has finished its work before it can start training the model.

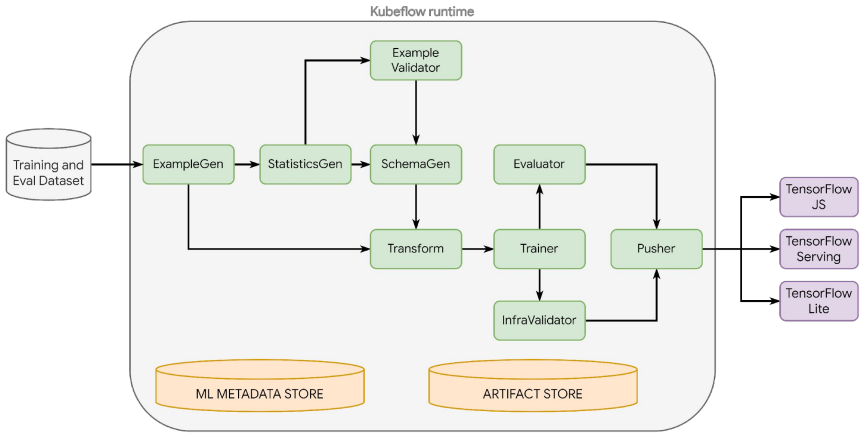
Metadata sharing: Components in a TFX pipeline generate metadata as they perform their tasks, such as information about the artifacts they produce, the configuration settings used, and the execution details. This metadata is stored in the ML Metadata service, which serves as a centralized repository for all metadata in the pipeline. Components can then access this metadata as needed, allowing them to

retrieve information about previous executions, track lineage, and make decisions based on past results.

Component configuration: To ensure that components in a TFX pipeline can work together effectively, they need to be configured with the right settings and parameters. TFX components use a standardized configuration format, making it easy to set up and modify the pipeline as needed. This consistent configuration helps ensure that components can communicate with one another and share data in a compatible manner.

In summary, TFX component communication refers to how the different components in a TFX pipeline interact and share information as they work together in the machine learning workflow. This involves passing data between components, defining dependencies and execution order, sharing metadata, and using standardized configurations to ensure compatibility and effective communication.

TFX Directed Acyclic Graph (DAG)



A TFX Directed Acyclic Graph (DAG)
represents the structure and dependencies of a TensorFlow Extended (TFX) pipeline. It is a visual representation of the pipeline components and the order in which they are executed. The DAG highlights the dependencies between components and ensures that they run in the correct sequence for a successful machine learning workflow.

In a DAG:

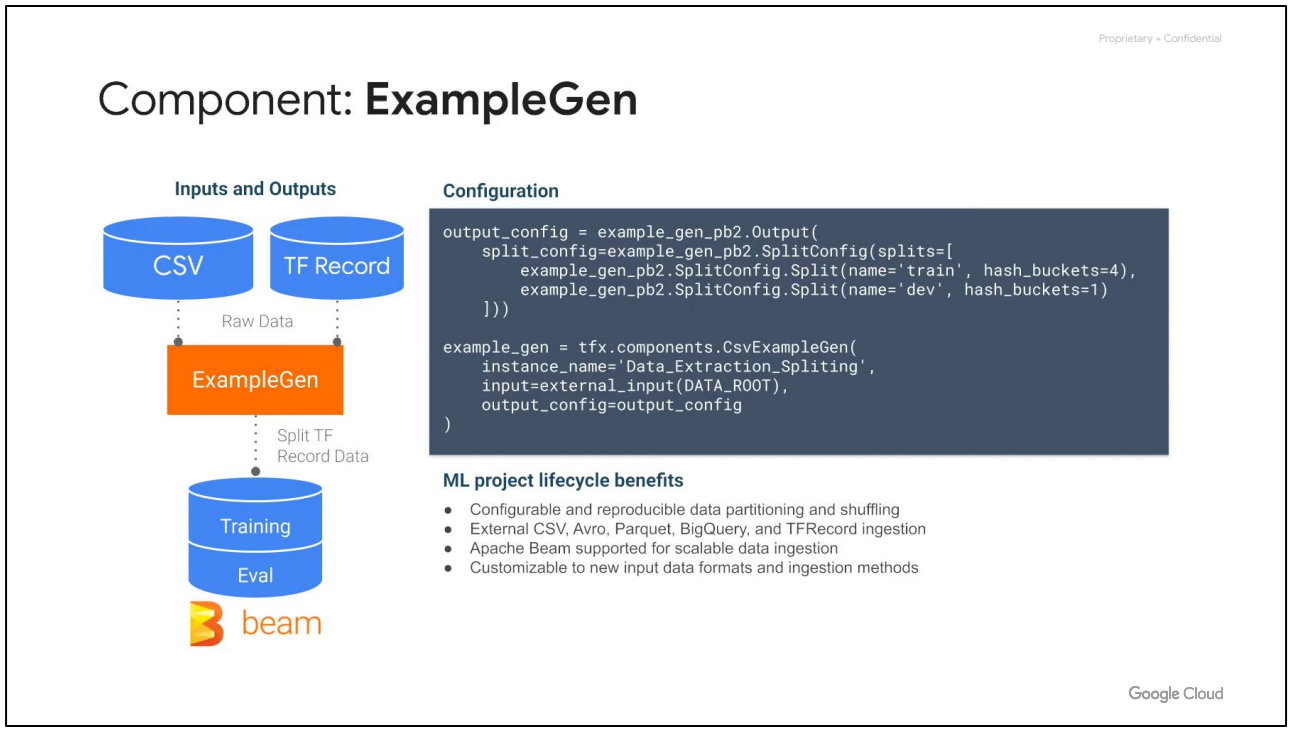
Nodes represent TFX components: Each node in the DAG corresponds to a TFX component, such as data ingestion, preprocessing, model training, evaluation, or serving. These components are the building blocks of the pipeline, responsible for specific tasks in the machine learning process.

Edges represent dependencies: Edges (or arrows) between nodes indicate the dependencies between components. An edge from one component to another shows that the output of the first component is used as input by the second component. These dependencies ensure that components are executed in the right order, and that the necessary data is available when needed.

Directed: The term "directed" refers to the fact that the edges in the graph have a direction, showing the flow of data and execution order between components. This directionality is crucial for understanding the sequence of operations in the pipeline.

Acyclic: The term "acyclic" means that the graph does not contain any cycles, ensuring that the pipeline has a clear start and end without any loops. This is important for preventing infinite loops or circular dependencies in the pipeline, which could lead to incorrect or inconsistent results.

The TFX DAG helps to visualize and understand the structure of a TFX pipeline, making it easier to manage and modify the pipeline as needed. By illustrating the dependencies and execution order of the components, the DAG provides a clear overview of the machine learning workflow, ensuring that all tasks are e



ExampleGen is a TensorFlow Extended (TFX) component responsible for ingesting and converting data from various sources into a standardized format called TFRecords. This format is optimized for use with TensorFlow and enables efficient storage, access, and processing of large-scale datasets in machine learning workflows.

The main tasks performed by the ExampleGen component include:

Data Ingestion: ExampleGen reads data from different sources such as CSV files, databases, BigQuery, or other custom data sources, allowing you to work with various types of data in your pipeline.

Data Splitting: ExampleGen can automatically split the ingested data into different subsets, such as training, validation, and testing datasets. This is important for training and evaluating machine learning models, as it allows you to assess their performance on unseen data and avoid overfitting.

Data Conversion: ExampleGen converts the ingested data into a standardized format called TFRecords, which consist of serialized TensorFlow Example or SequenceExample objects. These objects store data as feature lists, making it easy to work with structured data in TensorFlow.

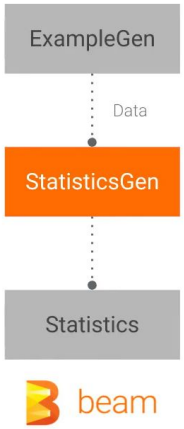
Output: The generated TFRecords are then stored as output artifacts, which can be used by downstream components in the pipeline for tasks like data preprocessing,

feature engineering, or model training.

ExampleGen plays a crucial role in the TFX pipeline by providing a consistent and efficient way to ingest and convert data for use with TensorFlow. By handling data ingestion and conversion tasks, ExampleGen simplifies the process of working with diverse data sources and enables you to focus on building and refining your machine learning model

Component: **StatisticsGen**

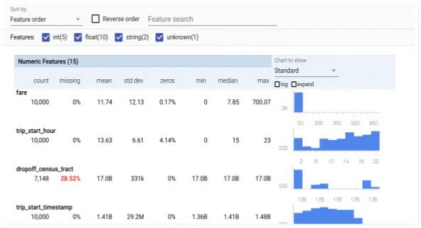
Inputs and Outputs



Configuration

```
statistics_gen = tfx.components.StatisticsGen(
    instance_name='Statistics_Generation',
    examples=example_gen.outputs['examples'])
```

TFDV Feature Statistics Visualization

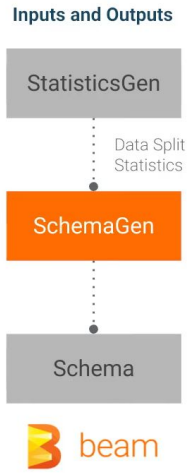


ML project lifecycle benefits

- TensorFlow Data Validation (TFDV) library for calculating feature statistics
- Scalable full-pass dataset feature statistics processing with Apache Beam

StatisticsGen:
in summary, the StatisticsGen component in TFX computes descriptive statistics on your dataset, helping you understand its properties, inform preprocessing decisions, and maintain data quality. By providing insights into the characteristics of your data, StatisticsGen plays a crucial role in building robust and reliable machine learning models.

Component: **SchemaGen**



Configuration

```
schema_gen = SchemaGen(
    statistics=statistics_gen.outputs['statistics'],
    infer_feature_shape=False)
```

Feature name	Type	Presence	Valency	Domain
'fare'	FLOAT	required	single	-
'trip_start_hour'	INT	required	single	-
'pickup_census_tract'	BYTES	optional		-
'dropoff_census_tract'	FLOAT	optional	single	-
'company'	STRING	optional	single	'company'

ML project lifecycle benefits

- TensorFlow Data Validation (TFDV) creates common data description for pipeline components
- A Schema enables continuous data monitoring and validation during pipeline training

summary:
the SchemaGen component in TFX generates a schema for your dataset based on the computed descriptive statistics, providing a blueprint for data validation and transformation in the machine learning workflow. By defining the expected properties and constraints of your features, SchemaGen helps ensure data quality and consistency throughout the pipeline.

Detail:

The main tasks performed by the SchemaGen component include:

Input: SchemaGen takes the output of the StatisticsGen component as input. This input consists of the computed descriptive statistics for each feature in your dataset across different data splits (e.g., training, validation, and testing).

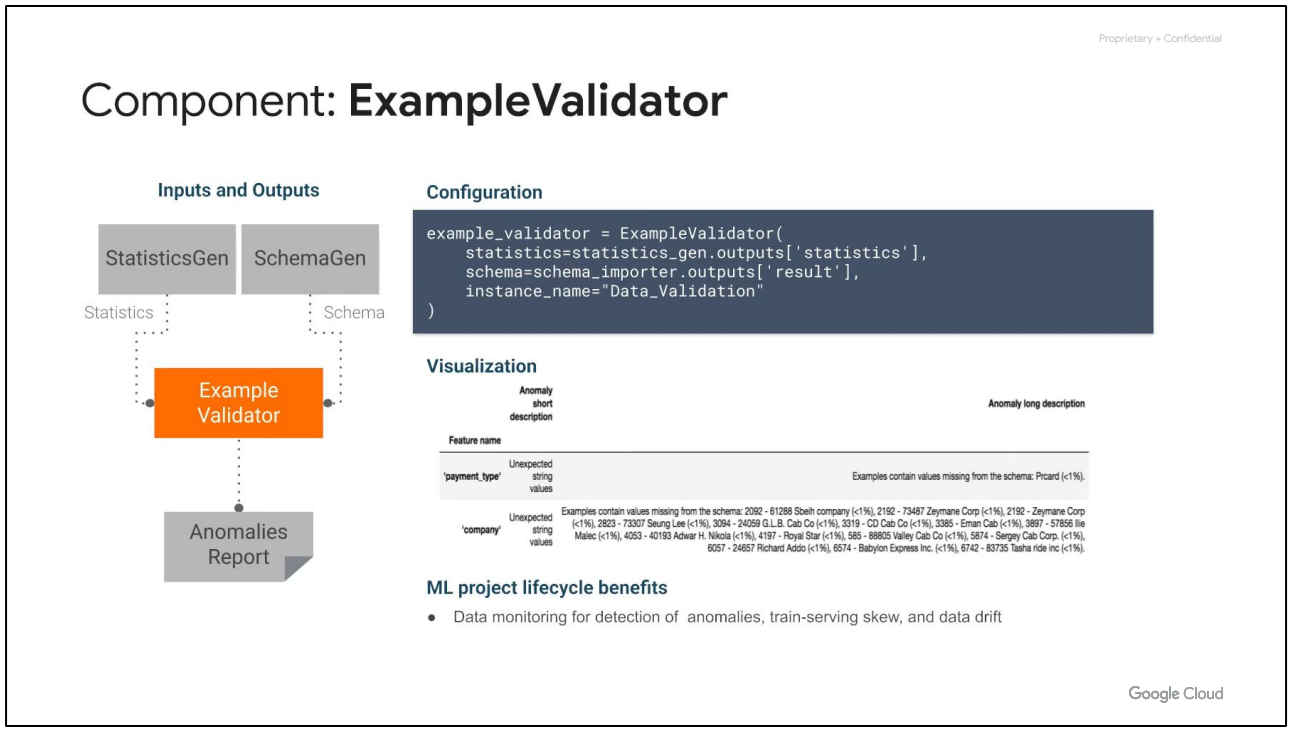
Generate Schema: The component analyzes the descriptive statistics to infer the properties of each feature, such as its data type (e.g., numeric, categorical, or text), domain (e.g., range of values or allowed categories), and presence (e.g., whether a feature is required or optional). It also detects any constraints, such as the minimum and maximum allowed lengths for text features.

Schema Customization: While SchemaGen can automatically generate a schema based on the input statistics, you can also provide your own custom schema or modify the generated one to better suit your specific requirements. This allows you to

enforce domain knowledge, add new constraints, or fine-tune the schema as needed.

Output: The generated schema is stored as an output artifact in a standardized format (protobuf), which can be consumed by downstream components in the pipeline, such as ExampleValidator, Transform, and Trainer.

Data Validation and Transformation: The schema serves as a blueprint for validating and transforming your data in the machine learning workflow. It is used by components like ExampleValidator to identify anomalies or inconsistencies in the dataset and by the Transform component to preprocess and engineer features based on the schema's specifications.



ExampleValidator:

In summary, the ExampleValidator component in TFX identifies anomalies and inconsistencies in your dataset by comparing it against the schema generated by the SchemaGen component. By ensuring data quality and consistency, ExampleValidator plays a crucial role in building robust and reliable machine learning models.

The main tasks performed by the ExampleValidator component include:

Input: ExampleValidator takes two inputs: the output of the StatisticsGen component, which consists of the computed descriptive statistics for your dataset, and the schema generated by the SchemaGen component, which defines the expected properties, types, and domains of the features in your dataset.

Data Validation: The component compares the computed statistics of the dataset against the schema to identify any anomalies or inconsistencies, such as missing or unexpected features, incorrect data types, or values outside the allowed domains. It also checks for any discrepancies between the training, validation, and testing data splits to ensure that the data is consistent across different subsets.

Anomaly Detection: ExampleValidator uses configurable detection rules to identify potential issues in your data. These rules can be customized to suit your specific

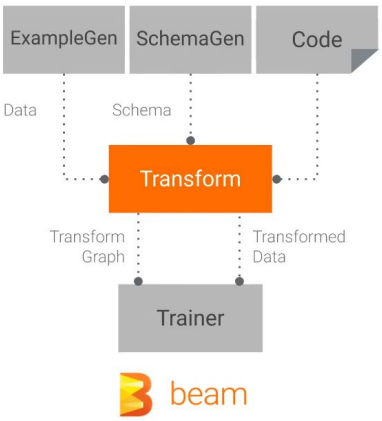
requirements, allowing you to fine-tune the anomaly detection process and focus on the most relevant issues for your use case.

Output: The identified anomalies and inconsistencies are stored as output artifacts in a standardized format (protobuf), which can be consumed by other components in the pipeline or visualized using tools like TensorFlow Data Validation (TFDV).

Data Quality Monitoring: By detecting and reporting anomalies in your dataset, ExampleValidator helps ensure data quality and maintain consistency throughout the machine learning workflow. This is particularly important when working with evolving data sources, as it allows you to identify and address potential issues before they impact your models' performance.

Component: **Transform**

Inputs and Outputs



Configuration

```
transform = Transform(
    examples=example_gen.outputs['examples'],
    schema=schema_importer.outputs['result'],
    module_file=TRANSFORM_MODULE)
```

Tensorflow Transform (TFT) library code in pipeline/preprocessing.py

```
def preprocessing_fn(inputs):

  outputs = {}

  # Scale numerical features
  for key in features.NUMERIC_FEATURE_KEYS:
    outputs[features.transformed_name(key)] = tft.scale_to_z_score(
        _fill_in_missing(inputs[key]))

  # Generate vocabularies and maps categorical features
  for key in features.CATEGORICAL_FEATURE_KEYS:
    outputs[features.transformed_name(key)] = tft.compute_and_apply_vocabulary(
        x=_fill_in_missing(inputs[key]), num_oov_buckets=1, vocab_filename=key)

  outputs[features.transformed_name(features.LABEL_KEY)] = _fill_in_missing(
      inputs[features.LABEL_KEY])

  return outputs
```


Transform:
the Transform component in TFX performs data preprocessing and feature engineering tasks on your dataset, preparing it for model training and serving. By consistently applying the specified transformations in both stages of the machine learning workflow, Transform plays a crucial role in building robust and reliable machine learning models.

Detail:
Transform is a TensorFlow Extended (TFX) component responsible for performing data preprocessing and feature engineering tasks on your dataset. It applies a series of transformations to the data to prepare it for model training and serving, ensuring that the same transformations are consistently applied in both stages of the machine learning workflow.

The main tasks performed by the Transform component include:

Input: Transform takes three inputs: the output of the ExampleGen component (the dataset in the form of TFRecords), the schema generated by the SchemaGen component (defining the properties and domains of the features), and a user-defined preprocessing function that specifies the transformations to apply to the data.

Preprocessing Function: You need to define a preprocessing function using the TensorFlow Transform library, which is a high-level API for performing data transformations. This function specifies the operations to be applied to the data, such as scaling, normalization, categorical encoding, or custom transformations based on your specific requirements.

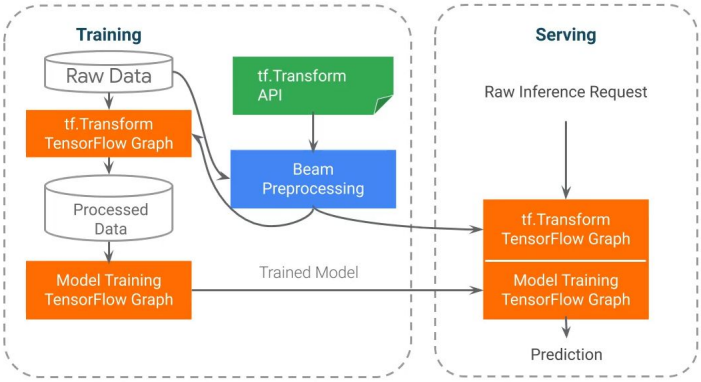
Data Transformation: The Transform component applies the preprocessing function to the input data, performing the specified operations on each feature in the dataset. These transformations ensure that the data is in the appropriate format and representation for model training and serving.

Output: The transformed data is stored as output artifacts in the form of TFRecords, which can be consumed by downstream components like the Trainer. In addition, the Transform component also generates a TensorFlow graph that encodes the preprocessing operations, ensuring that the same transformations can be consistently applied during model serving.

Consistency in Training and Serving: By generating a TensorFlow graph that encodes the preprocessing operations, the Transform component ensures that the same transformations are applied consistently during both model training and serving. This is critical for maintaining model performance and avoiding discrepancies between the training and serving stages.

I don't have access to this. Deprecated?
~~LAB: TFX components walkthrough~~
~~https://partner.cloudskillsboost.google/course_sessions/2567810/labs/103958~~

Component: **Transform**

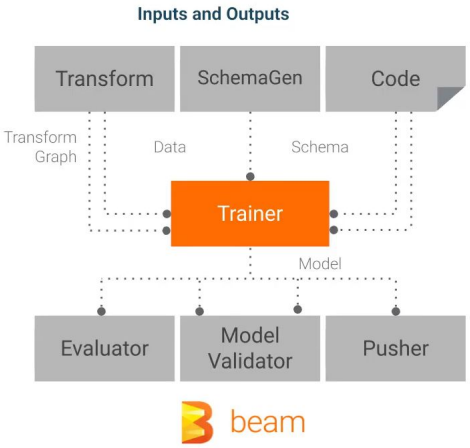


ML project lifecycle benefits

- Generates SavedModel for consistent training and serving feature engineering
- Backed by Apache Beam for scalable distributed data processing to large datasets

Data Transformation: The Transform component applies the preprocessing function to the input data, performing the specified operations on each feature in the dataset. These transformations ensure that the data is in the appropriate format and representation for model training and serving.

Component: **Trainer**



Configuration

```
trainer = Trainer(
custom_executor_spec=executor_spec.ExecutorClassSpec(trainer_executor.Gen
ericExecutor),
    module_file=model.py,
    transformed_examples=transform.outputs["transformed_examples"],
    schema=schema_importer.outputs["result"],
    transform_graph=transform.outputs["transform_graph"],
    train_args=trainer_pb2.TrainArgs(num_steps=5000),
    eval_args=trainer_pb2.EvalArgs(num_steps=1000))
```

Code -> your TensorFlow code in a pipeline/models/model.py

TFX Component Trainer:
In summary, the Trainer component in TFX is responsible for training machine learning models on the preprocessed data generated by the Transform component. It orchestrates the model training process, manages hyperparameters, and saves the trained models for evaluation and deployment. By handling the training process, Trainer plays a crucial role in building robust and reliable machine learning models.

Detail:
The main tasks performed by the Trainer component include:

Input: Trainer takes several inputs, including the transformed data from the Transform component (TFRecords), the schema generated by the SchemaGen component (defining the properties and domains of the features), and a user-defined model training function that specifies the architecture and training process for the machine learning model.

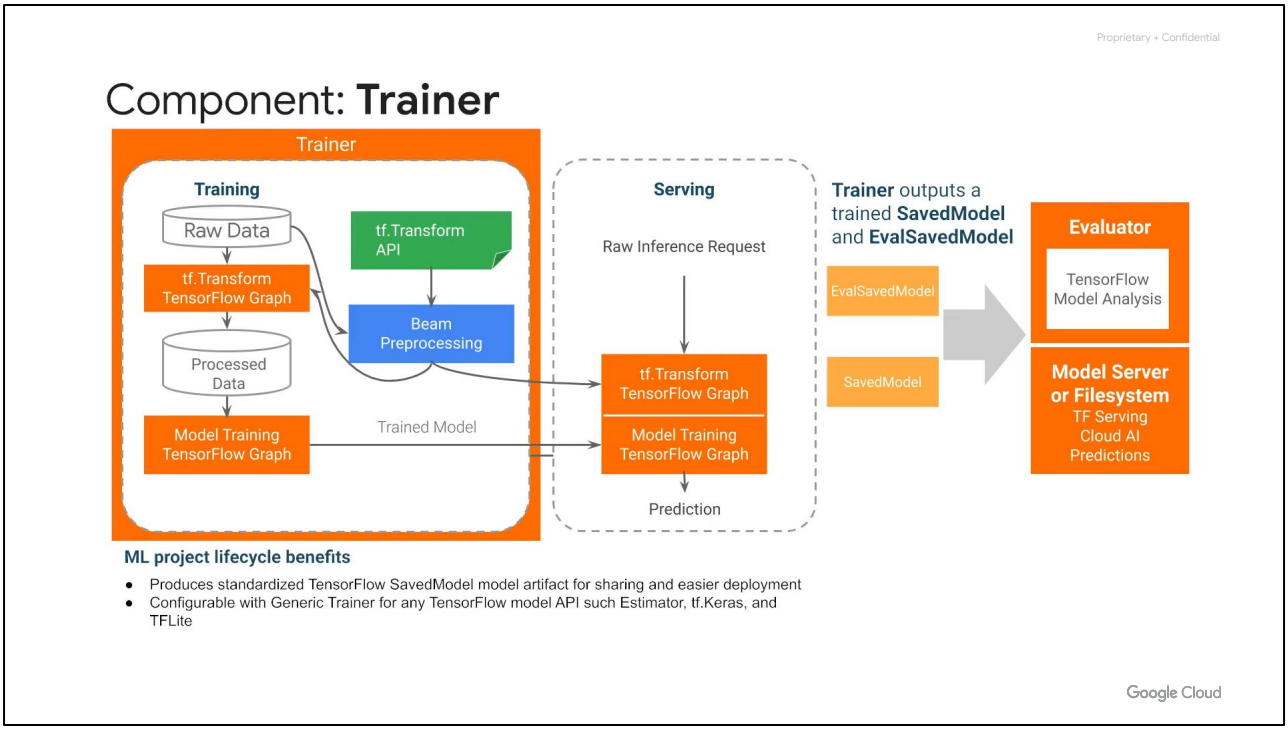
Model Training Function: You need to define a model training function using TensorFlow or other supported machine learning frameworks like Keras. This function specifies the model architecture, loss function, optimizer, and any other configurations required for training the model on the preprocessed data.

Model Training: The Trainer component orchestrates the training process by applying the model training function to the transformed data. It manages the training loop,

handling tasks like gradient updates, checkpointing, and early stopping based on your specified configurations.

Hyperparameter Tuning (Optional): If desired, Trainer can also integrate with tools like Keras Tuner or other hyperparameter optimization libraries to perform hyperparameter tuning during the training process. This helps optimize the model's performance by searching for the best combination of hyperparameters.

Output: Trainer saves the trained model as an output artifact, which can be consumed by downstream components like Evaluator and Pusher for model evaluation and deployment. The output includes the model's weights, architecture, and any other metadata required for serving the model.



TFX Component Trainer:
In summary, the Trainer component in TFX is responsible for training machine learning models on the preprocessed data generated by the Transform component. It orchestrates the model training process, manages hyperparameters, and saves the trained models for evaluation and deployment. By handling the training process, Trainer plays a crucial role in building robust and reliable machine learning models.

Detail:
The main tasks performed by the Trainer component include:

Input: Trainer takes several inputs, including the transformed data from the Transform component (TFRecords), the schema generated by the SchemaGen component (defining the properties and domains of the features), and a user-defined model training function that specifies the architecture and training process for the machine learning model.

Model Training Function: You need to define a model training function using TensorFlow or other supported machine learning frameworks like Keras. This function specifies the model architecture, loss function, optimizer, and any other configurations required for training the model on the preprocessed data.

Model Training: The Trainer component orchestrates the training process by applying the model training function to the transformed data. It manages the training loop, handling tasks like gradient updates, checkpointing, and early stopping based

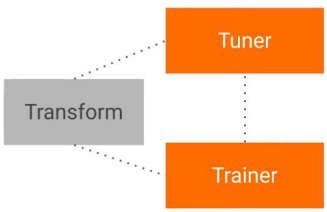
on your specified configurations.

Hyperparameter Tuning (Optional): If desired, Trainer can also integrate with tools like Keras Tuner or other hyperparameter optimization libraries to perform hyperparameter tuning during the training process. This helps optimize the model's performance by searching for the best combination of hyperparameters.

Output: Trainer saves the trained model as an output artifact, which can be consumed by downstream components like Evaluator and Pusher for model evaluation and deployment. The output includes the model's weights, architecture, and any other metadata required for serving the model.

Component: **Tuner**

Tuner works with Trainer



Configuration

```
tuner = Tuner(
    module_file=module_file,  # Contains `tuner_fn`.
    examples=transform.outputs['transformed_examples'],
    transform_graph=transform.outputs['transform_graph'],
    train_args=trainer_pb2.TrainArgs(num_steps=20),
    eval_args=trainer_pb2.EvalArgs(num_steps=5))
```

Best hyperparameters added to Trainer in pipeline

```
trainer = Trainer(
    module_file=module_file,  # Contains `run_fn`.
    custom_executor_spec=executor_spec.
        ExecutorClassSpec(GenericExecutor),
    examples=transform.outputs['transformed_examples'],
    transform_graph=transform.outputs['transform_graph'],
    schema=schema_gen.outputs['schema'],
    # This will be passed to `run_fn`.
    hyperparameters=tuner.outputs['best_hyperparameters'],
    train_args=trainer_pb2.TrainArgs(num_steps=100),
    eval_args=trainer_pb2.EvalArgs(num_steps=5))
```

ML project lifecycle benefits

- Supports KerasTuner library for hyperparameter tuning and integration with TensorFlow Trainer
- Can use Google Cloud AI Platform Optimizer for distributed tuning

Tuner component:
he Tuner component in TFX is responsible for optimizing the hyperparameters of machine learning models to improve their performance. It automates the process of searching for the best combination of hyperparameters and saves the best ones for use in the model training process. By handling hyperparameter optimization, Tuner plays a crucial role in building robust and effective machine learning models.

Detail:
The main tasks performed by the Tuner component include:

Input: Tuner takes several inputs, including the transformed data from the Transform component (TFRecords), the schema generated by the SchemaGen component (defining the properties and domains of the features), and a user-defined model training function that specifies the architecture and training process for the machine learning model. Additionally, you need to define a search space for the hyperparameters to be optimized.

Model Training Function: Similar to the Trainer component, you need to define a model training function using TensorFlow or other supported machine learning frameworks like Keras. This function specifies the model architecture, loss function, optimizer, and any other configurations required for training the model on the preprocessed data.

Search Space: You need to define a search space for the hyperparameters to be

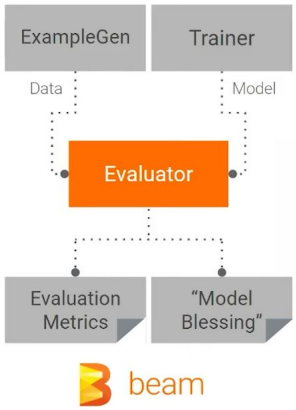
optimized. This search space includes the range of possible values for each hyperparameter, allowing the Tuner to explore different combinations and find the best ones for your specific use case.

Hyperparameter Optimization: The Tuner component performs hyperparameter optimization by training multiple instances of the model with different hyperparameter combinations and evaluating their performance on the validation data. It uses optimization algorithms like Bayesian optimization, random search, or grid search to efficiently explore the search space and identify the best combination of hyperparameters.

Output: Tuner saves the best hyperparameters as an output artifact, which can be consumed by downstream components like the Trainer. This allows you to train your final model using the optimized hyperparameters, ensuring that your model achieves the best possible performance on your specific dataset.

Component: **Evaluator**

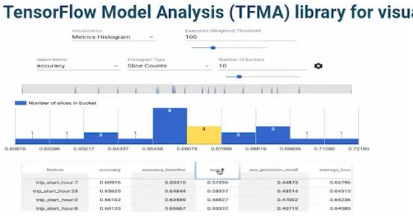
Inputs and Outputs



Configuration

```
model_analyzer = Evaluator(
    examples=example_gen.outputs.examples,
    model=trainer.outputs.model,
    baseline_model=model_resolver.outputs.model,
    eval_config=eval_config)
```

TensorFlow Model Analysis (TFMA) library for visualizing model evaluation



ML project lifecycle benefits

- Uses TensorFlow Model Analysis (TFMA) and Apache Beam for scalable model evaluation across data splits and slices.
- Validate model performance "good enough" to be pushed to production

Evaluator:
In summary, the Evaluator component in TFX assesses the performance of trained machine learning models using predefined evaluation metrics and criteria. By analyzing the model's performance and validating it against a baseline, Evaluator plays a crucial role in ensuring that only high-performing and robust models are deployed in production environments.

Detail
The main tasks performed by the Evaluator component include:

Input: Evaluator takes several inputs, including the trained model from the Trainer component, the schema generated by the SchemaGen component (defining the properties and domains of the features), and the evaluation data, typically from the validation or test split of the dataset.

Evaluation Metrics: You need to define a set of evaluation metrics to assess the model's performance. These metrics can include accuracy, precision, recall, F1 score, area under the ROC curve (AUC-ROC), and others, depending on your specific use case and requirements.

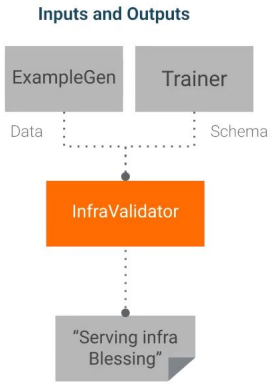
Model Evaluation: The Evaluator component applies the trained model to the evaluation data and computes the defined evaluation metrics. It assesses the model's performance against a predefined set of criteria or compares it to a baseline model to determine if the new model has improved performance.

Slicing: Optionally, you can configure Evaluator to compute evaluation metrics for different data slices, allowing you to analyze the model's performance across various segments of your dataset. This helps identify any potential biases or performance issues that might affect specific subsets of your data.

Output: Evaluator generates an output artifact that contains the computed evaluation metrics and the results of the comparison against the predefined criteria or baseline model. This output can be consumed by downstream components like the Pusher to decide whether to deploy the new model or not.

Model Validation: If the new model meets or exceeds the predefined criteria or outperforms the baseline model, it is considered validated and ready for deployment. Otherwise, the model may require further tuning or improvements before deployment.

Component: **InfraValidator**



Configuration

```
infra_validator = InfraValidator(
    model=trainer.outputs['model'],
    examples=example_gen.outputs['examples'],
    serving_spec=infra_validator_pb2.ServingSpec(...),
  validation_spec=infra_validator_pb2.ValidationSpec(...),
  request_spec=infra_validator_pb2.RequestSpec(...)
)
```

Configuration options

ServingSpec: Type of model server and infrastructure to test with e.g. TF Serving
ValidationSpec: Adjusts the infra validation criteria or workflow.
RequestSpec: Which model signature, how many examples to test.

ML project lifecycle benefits

- Validate model in model serving environment before pushing to production
- Configurable to mirror different model serving environments such as Kubernetes and TF Serving

InfraValidator
the InfraValidator component in TFX validates the trained machine learning model in a production-like environment, ensuring that it can be successfully loaded and served in the target infrastructure. By checking the model's compatibility and serving capabilities, InfraValidator plays a crucial role in reducing the risk of deployment failures and ensuring a smooth transition to production environments.

Detail:

Input: InfraValidator takes several inputs, including the trained model from the Trainer component, the schema generated by the SchemaGen component (defining the properties and domains of the features), and a serving specification that defines the target serving infrastructure and configuration.

Serving Specification: You need to define a serving specification that describes the target serving infrastructure, such as the type of serving container (e.g., TensorFlow Serving, TFLite, or a custom container), the resources required (e.g., CPU, memory, and storage), and any other relevant configuration settings.

Model Loading: The InfraValidator component starts a temporary instance of the target serving infrastructure with the specified configuration and attempts to load the trained model. This ensures that the model is compatible with the target environment and can be successfully loaded and served.

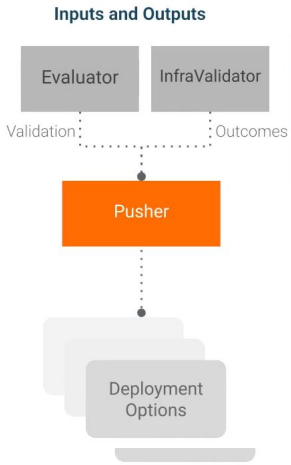
Model Validation: Once the model is loaded, InfraValidator performs a series of validation checks to ensure that the model can correctly serve predictions. This may include sending test requests to the model and verifying that the responses are as expected.

Output: InfraValidator generates an output artifact that contains the results of the validation process. This output can be consumed by downstream components like the Pusher to decide whether to deploy the new model or not.

Model Deployment Readiness: If the validation checks pass and the model can be successfully loaded and served in the target infrastructure, the model is considered ready for deployment. Otherwise, the model may require further adjustments or fixes before it can be deployed.

In summary, the InfraValidator component in TFX validates the trained machine learning model in a production-like environment, ensuring that it can be successfully loaded and served in the target infrastructure. By checking the model's compatibility and serving capabilities, InfraValidator plays a crucial role in reducing the risk of deployment failures and ensuring a smooth transition to production environments.

Component: **Pusher**



Configuration

```
pusher = Pusher(
    model=trainer.outputs['model'],
    model_blessing=model_analyzer.outputs['blessing'],
    infra_blessing=infra_validator.outputs['blessing'],
    push_destination=pusher_pb2.PushDestination(
        filesystem=pusher_pb2.PushDestination.Filesystem(
            base_directory=SERVING_MODEL_DIR)))
```

ML project lifecycle benefits

- Validation checks on model performance and serving ability before pushing to production.
- Reusable model export code that is configurable for different push destinations:
  - Filesystem (TensorFlow Lite, TensorFlow JS)
  - Model Server (TensorFlow Serving)

Pusher component:
the Pusher component in TFX is responsible for deploying trained and validated machine learning models to a target environment or serving infrastructure. It automates the deployment process, ensuring a smooth and efficient transition from the development stage to production. By handling the final step in the TFX pipeline, Pusher plays a crucial role in bringing machine learning models to production and delivering value to end-users.

Detail:

The main tasks performed by the Pusher component include:

Input: Pusher takes several inputs, including the trained model from the Trainer component, the evaluation results from the Evaluator component, and optionally, the validation results from the InfraValidator component. Additionally, you need to define a target deployment location and configuration for the serving infrastructure.

Deployment Criteria: Based on the evaluation results from the Evaluator component and optionally, the validation results from the InfraValidator component, Pusher determines whether the trained model is ready for deployment. If the model meets the predefined performance criteria and infrastructure requirements, it proceeds with the deployment process.

Target Deployment Location: You need to define a target deployment location and

configuration for the serving infrastructure, such as a TensorFlow Serving instance, a cloud-based AI Platform, or another compatible serving system. The target location specifies where the model will be deployed and served.

Model Deployment: The Pusher component deploys the trained and validated model to the target serving infrastructure. It handles tasks like copying the model files, updating the serving configuration, and triggering the necessary processes to start serving the new model.

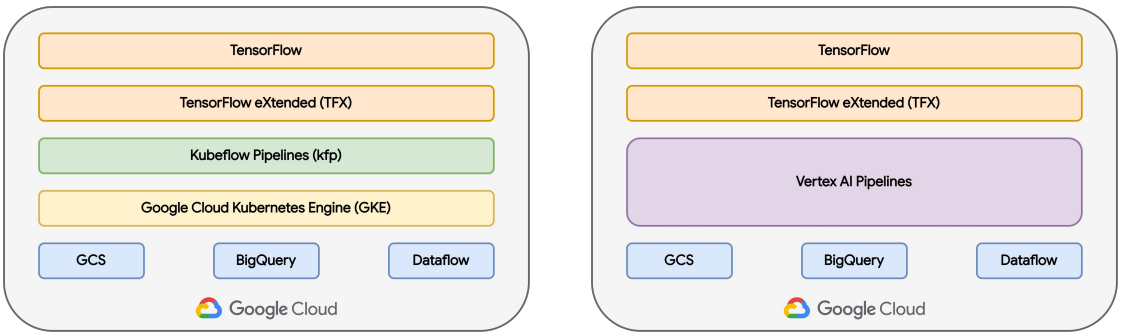
Output: Pusher generates an output artifact that contains information about the deployed model, such as its location, version, and any other relevant metadata. This output can be used for tracking and monitoring purposes or to trigger additional deployment-related tasks, such as updating a model registry or notifying stakeholders.

Component: **Build Your Own**

You can also build your own Fully Custom Components.

You can also build your own fully custom components.

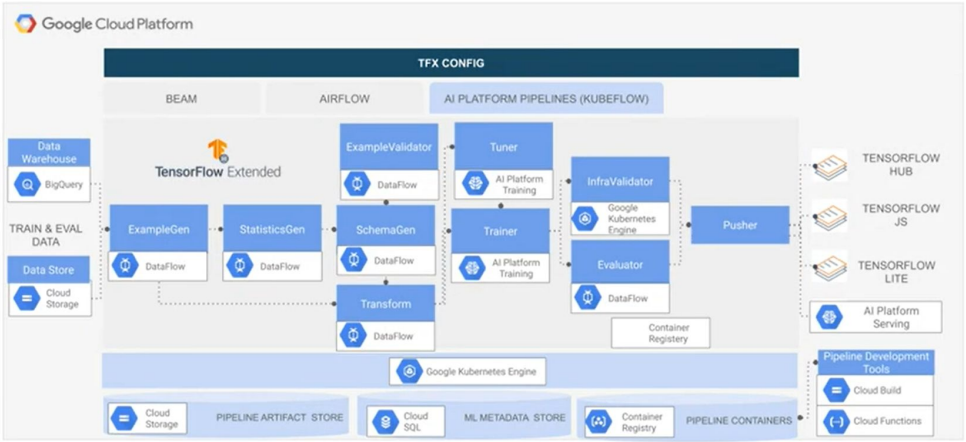
TFX can be used in Vertex AI Pipelines or run on other clusters, like your own GKE Cluster



TFX can be used within Vertex AI Pipelines or on a KubeFlow deployment in a Kubernetes cluster.

I don't have access to this. Deprecated?
~~LAB: TFX components walkthrough~~
~~https://partner.cloudskillsboost.google/course_sessions/2567810/labs/103958~~

Scaling TFX on GCP



Source: Custom

Here you can see an example of continuous training pipeline implemented with TFX on top of Google Cloud. You can notice that every single components can scale using a GCP service (Google Cloud Storage, Dataflow, BigQuery etc). In this example we are executing the pipeline on AI Platform pipelines that allows you to have a managed Kubeflow environment running on GKE. This was the only solution to implement an MLOps pipeline before Vertex AI came.

Kubeflow Pipelines

A flexible and powerful pipeline option in which you can build your own components in Python or pull from existing published components.

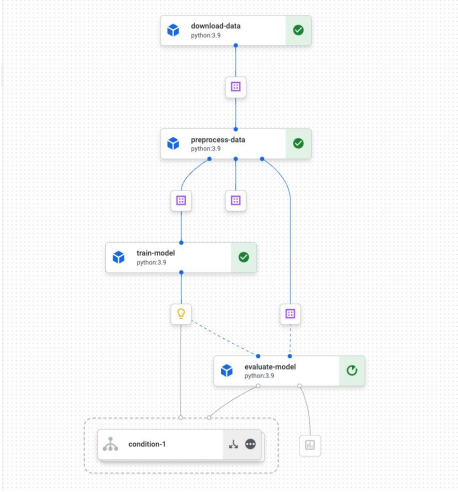
PROS

- Flexibility of building your own components
- Flexibility of using any ML Framework

CONS

- Not as opinionated with the well-structured standard pipeline of TFX, so more design is needed on your part.

Runs in Vertex AI Pipelines can include conditions, for example deciding to deploy or not deploy a new model.



Kubernetes is a great platform for ML

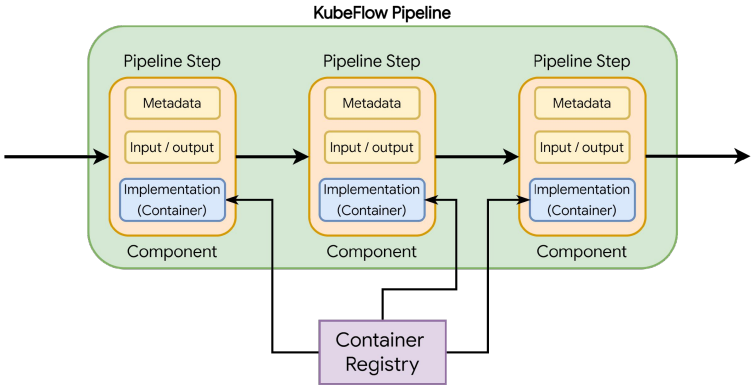
- Containers
- Scaling built-in
- Unified architecture
- Easy to integrate building blocks
  - ML APIs
  - Dataflow
- Lots of options for CI/CD
- Portability
  - Dev, On-Prem, Multi-cloud: same stack

Before we dive into the specifics of Kubeflow Pipelines, I should provide additional context.

Kubeflow Pipelines is a part of the open source project Kubeflow.
Kubeflow is a platform that provides the tools and scalable services required to develop and deploy ML workloads, all the way from distributed training, to scalable serving, to Notebooks w/ JupyterHub and workflow orchestration and much more.
Kubeflow services are built on top on Kubernetes. Kubernetes provides scalability and hybrid portability.

On GCP, You can use Kubeflow Pipelines in Vertex AI Pipelines, or deploy Kubeflow on your own Google Kubernetes Engine cluster.

Kubeflow Pipelines



Kubeflow Pipelines is a platform for building, deploying, and managing machine learning workflows on Kubernetes. It provides a set of technical components that help you create reusable, modular, and scalable pipelines. The main components of Kubeflow Pipelines are:

Pipeline:
A pipeline is a sequence of components (or steps) that define your machine learning workflow. Each component in the pipeline represents a specific task, such as data preprocessing, feature engineering, model training, or model evaluation. Pipelines are defined using Python SDK, which compiles the pipeline definition into a YAML file that can be deployed to the Kubeflow Pipelines runtime.

Components:
Components are self-contained pieces of code that perform a specific task in the pipeline. They can be created from existing container images or by defining a function in Python and converting it to a container using the Kubeflow Pipelines SDK. Components take inputs, execute a specific operation (such as data transformation or model training), and produce outputs. They can be reused across multiple pipelines or shared with other users.

Parameters and Artifacts:
Parameters and artifacts are the inputs and outputs of components. Parameters are simple values like strings, integers, or floats, while artifacts are more complex data types like datasets, models, or metrics. Artifacts are usually stored in a managed

storage solution, such as Google Cloud Storage or Amazon S3, and are automatically tracked and versioned by the Kubeflow Pipelines system.

Container:
Each component runs in a container, which provides an isolated and reproducible environment for executing the component's code. Containers are based on Docker images, which can include all the necessary dependencies and libraries required for the component to run. This allows for easy sharing and reuse of components, as well as consistent execution across different environments.

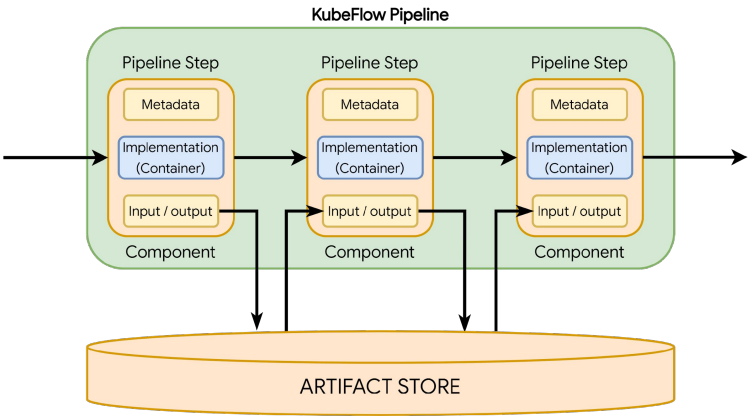
Kubernetes:
Kubeflow Pipelines runs on Kubernetes, a container orchestration platform that provides scalability, high availability, and resource management. Kubernetes handles the deployment, scaling, and management of the containers that make up your pipeline, ensuring they have the necessary resources to run and can recover from failures.

Kubeflow Pipelines UI:
The Kubeflow Pipelines UI is a web-based interface that allows you to create, manage, and monitor your pipelines. It provides an easy way to visualize your pipeline's structure, track the progress of pipeline runs, and inspect the input/output artifacts of each component. It also supports managing pipeline templates and sharing them with other users.

Metadata Store:
The metadata store is a database that keeps track of all the metadata associated with your pipelines, such as runs, artifacts, and executions. It enables advanced features like experiment tracking, versioning, and lineage tracking, allowing you to understand the relationships between different pipeline runs and components.

Together, these components enable you to build, deploy, and manage complex machine learning workflows using Kubeflow Pipelines, taking advantage of the scalability and flexibility provided by Kubernetes.

Kubeflow Pipelines Storage



An artifact store:

In summary, the Kubeflow Pipelines artifact store plays a crucial role in managing the output files generated during the execution of a pipeline, providing persistent storage, versioning, and easy access to the artifacts for analysis or future use.

By default, Kubeflow Pipelines uses the MinIO object storage system as the artifact store in standalone deployments, while managed Kubeflow Pipelines installations on cloud platforms like Google Cloud or AWS may use their respective storage services, such as Google Cloud Storage or Amazon S3.

Key features of a Kubeflow Pipelines artifact store include:

Persistence: The artifact store ensures that output files generated during the execution of a pipeline are saved and maintained even if the underlying containers or infrastructure are terminated.

Versioning: The artifact store can keep multiple versions of an artifact, allowing you to track the changes and improvements in your models or data over time.

Access control: The artifact store can be configured to provide secure access to stored artifacts, ensuring that only authorized users or systems can read or write data.

Scalability: Artifact stores like MinIO, Google Cloud Storage, or Amazon S3 are

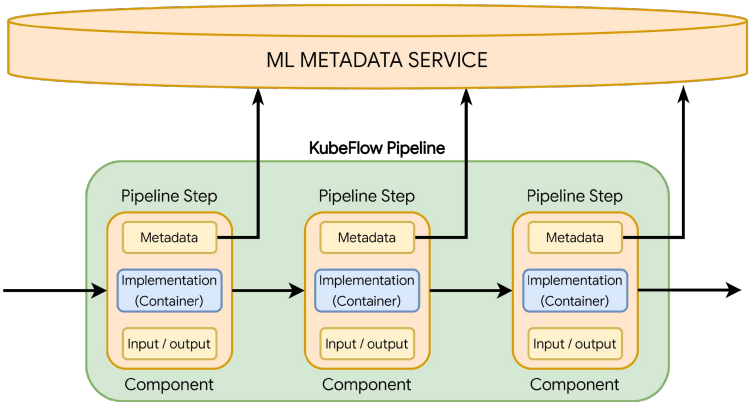
designed to handle large amounts of data and can scale as needed to accommodate growing storage requirements.

Integration: The artifact store is integrated with the Kubeflow Pipelines Metadata Store, enabling advanced features like lineage tracking, experiment tracking, and artifact comparison.

Portability: By using standard storage solutions, the artifact store can be easily migrated or accessed from different environments or platforms, ensuring the long-term availability of your pipeline artifacts.

Summary:
In summary, the Kubeflow Pipelines artifact store plays a crucial role in managing the output files generated during the execution of a pipeline, providing persistent storage, versioning, and easy access to the artifacts for analysis or future use.

Kubeflow Pipelines ML Metadata



The ML Metadata
(MLMD) service in Kubeflow Pipelines is a system that manages and tracks metadata generated throughout the machine learning workflows.

Metadata = includes information about the pipeline components, their inputs and outputs (artifacts), execution details, and more. The MLMD service helps you understand, analyze, and manage your ML workflows by providing insights into the lineage, reproducibility, and performance of your models and experiments.

Key features of the ML Metadata service in Kubeflow Pipelines include:

Metadata Storage: MLMD stores metadata in a structured and queryable format, usually in a database such as MySQL, SQLite, or PostgreSQL. This storage enables efficient querying, filtering, and retrieval of metadata for various purposes like experiment tracking or model comparison.

Lineage Tracking: MLMD allows you to track the lineage of artifacts and executions throughout the pipeline. This helps you understand how a particular artifact was created, the dependencies between artifacts, and the impact of changes in your pipeline. Lineage tracking enhances the reproducibility, auditability, and trustworthiness of your ML workflows.

Experiment Tracking: MLMD helps you organize and track multiple experiments, enabling you to compare different configurations, hyperparameters, or versions of

your models. This assists in identifying the best-performing models and understanding the factors that contribute to their success.

Artifact Versioning: MLMD supports versioning of artifacts, allowing you to track and compare different versions of models, datasets, or other outputs over time. This aids in understanding the evolution of your models and the effects of changes in your data or algorithms.

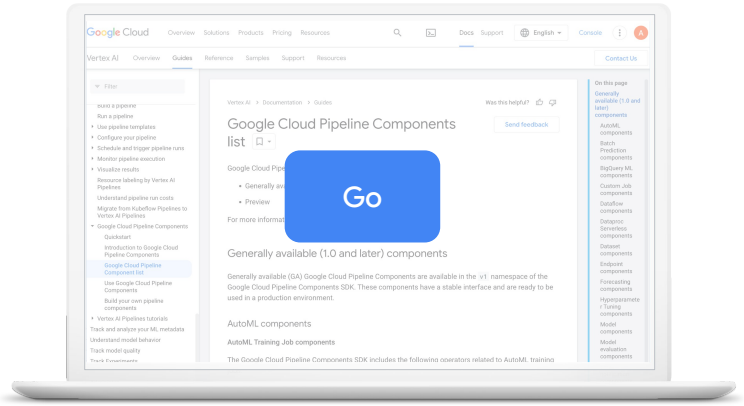
Integration: MLMD is tightly integrated with Kubeflow Pipelines, which automatically collects and stores metadata during the execution of a pipeline. The Kubeflow Pipelines UI provides an interface for visualizing and exploring the stored metadata, enabling easy access to lineage information, experiment tracking, and more.

Extensibility: MLMD is designed to be extensible, allowing you to define custom metadata types and properties to suit the specific requirements of your ML workflows.

In summary, the ML Metadata service in Kubeflow Pipelines plays a vital role in managing and tracking metadata generated during the execution of ML workflows. It provides valuable insights into lineage, reproducibility, and performance, enabling you to better understand, analyze, and manage your machine learning processes.

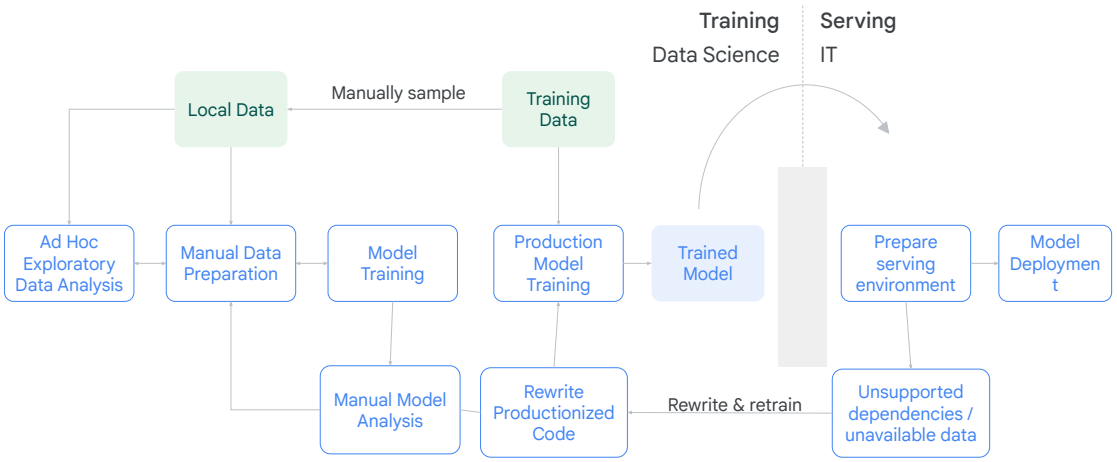
Reference

GCP Pipeline Components List



Introduction to MLOps

The problem with divided data science & engineering responsibilities.



For a period of time, teams often saw Data scientists as a separate team from Operations.

Data scientists would prepare notebooks, then someone from Ops would need to convert that notebook (possibly with code they didn't fully understand and couldn't really optimize) into a container or service.

This doesn't scale!

If you have large number of models that require frequent update with new data and new implementation... we need automation!

And if you want speedy deployments, the same team needs to be able to develop models and deploy them.

The previous workflow is good if you have a limited number of model to develop and put to production. If you have a large number of models to build/deploy/maintain this is not going to work. We need to introduce automation. MLOps has been conceived for this reason.

MLOps is a set of standardized processes and capabilities for building, deploying, and operationalizing ML systems rapidly and reliably.

This is a formal definition of MLOps. They key takeaways are standardized processes for building/deploying/maintaining ML systems.

# # #

Today, ML models are not just the domain of research teams. They are expected to be able to be built, tested, and deployed to **production environments**.

Because there are so many steps to building an ML model, we want to build towards standardizing and automating as many steps as we can.

The responsibilities of data gathering, model training, prediction pipelines, model retraining, testing, and more **are often owned by machine learning engineers** (with some overlap and collaboration with other specialists like data scientists and platform ops teams).

DevOps

DevOps is the popular practice in developing and operating large scale software systems, giving benefits such as shorter development cycles, increased deployments velocity, and more dependable releases, in close alignment with business objectives. In order to achieve this, two concepts are introduced in the software system development:

Publicly visible source: MLOps from Cloud Architecture Center

MLOps embeds DevOps culture in the ML. But there are some differences between DevOps and MLOps.

For instance, MLOps delivers ML Pipelines that result in training a Model that will be deployed.

Continuous Integration

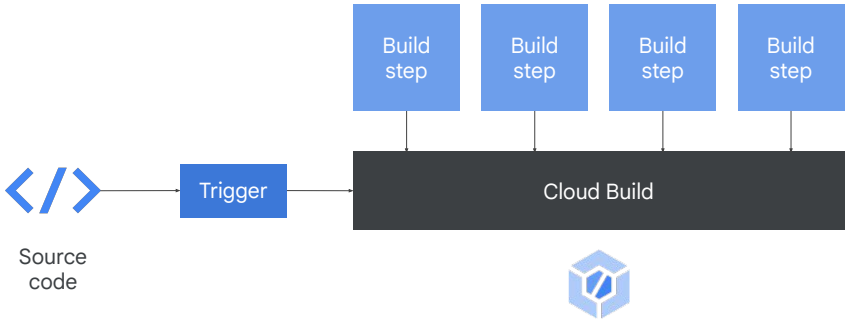
Is no longer only about testing and validating code and components, but also testing and validating data, data schemas, and models.

Continuous Delivery

Is no longer about a single software package or a service, but a system (ML training pipeline) that should automatically deploy another service.

Proprietary + Confidential

CICD in Cloud Build



Part of MLOps is also about triggering builds and steps when new code is committed.

For that we use the Continuous Integration Continuous Delivery/Deployment (CICD) service Cloud Build.

Cloud Build is a CI/CD tool that can run containerized steps when updates to a repo are made, for example to deploy updated versions of pipelines or other GCP services.

My recommendation for many GCP Terraform deployments is to deploy the Cloud Build Trigger rather than the final resource itself, so that as updates are made, new deployments are made.

Source: MLOps Architectures with CI/CD (slides) | Y22
https://docs.google.com/presentation/d/1OiGZOviGJf6XX7E91-41PGBy3aoRPZcibmF9dtH1Cms/edit?resourcekey=0-0LF8xS6di73GKb_vAVDZjw#slide=id.g1043fd8a725_0_2031

Proprietary + Confidential

cloudbuild.yaml

The **cloudbuild** file defines the work to be done by Container Builder, can be YAML or JSON

Defines:

- Build steps
- Timeout
- Logs bucket
- Build options
- Substitutions
- Tags
- Secrets
- Image name

```
steps:
- name: NAME_OF_STEP
  args:
  env:
  dir:
  id:
  waitFor:
  entrypoint:
  secretEnv:
  volumes:
- name: NAME_OF_STEP
  ...
- name: NAME_OF_STEP
...
timeout:
logsBucket:
options:
substitutions:
tags:
secrets:
images:
- [IMAGE_NAME_, IMAGE_NAME, ...]
```

Build configuration file schema | Cloud Build Documentation | Google Cloud

Google Cloud

Source: MLOps Architectures with CI/CD (slides) | Y22
https://docs.google.com/presentation/d/1OiGZOviGJf6XX7E91-41PGBy3aoRPZcibmF9dtH1Cms/edit?resourcekey=0-0LF8xS6di73GKb_vAVDZjw#slide=id.g1043fd8a725_0_2031

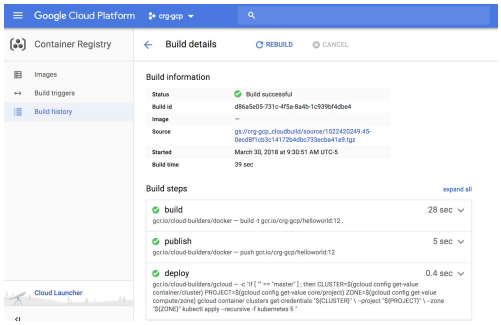
In Cloud Build a CI/CD pipeline is coded as yaml file and consists of build steps, options etc.

Proprietary + Confidential

Example YAML template

```
# cloudbuild.yaml
steps:
- name: gcr.io/cloud-builders/gcloud
  args: ['app', 'deploy']
```

```
gcloud container builds submit \
       --config cloudbuild.yaml  .
```



Source: MLOps Architectures with CI/CD (slides) | Y22
https://docs.google.com/presentation/d/1OiGZOviGJf6XX7E91-41PGBy3aoRPZcibmF9dtH1Cms/edit?resourcekey=0-0LF8xS6di73GKb_vAVDZjw#slide=id.g1043fd8a725_0_2031

You can create a CI/CD pipeline by using the command shows in the slide. In Cloud Build you can maintain the history of every build.

MLOps

From a CI/CD perspective, ML and other software systems are largely similar in continuous integration of source control, unit testing, and integration testing, and continuous delivery of software modules or packages. However, in ML, there are a few notable differences:

Publicly visible source: MLOps from Cloud Architecture Center

Source: MLOps Final Report Template | PSO | Y21 - modified from https://docs.google.com/document/d/19os-tE7Pai6QQ1_pt7DsARgcv0Gvw-ATqYfBApLtywc/edit?resourcekey=0-4Z1XA9OhvuAjt73EnhwmhQ#heading=h.l9lpvb1ep2ok

MLOps also includes an additional phase that in DevOps is not present. Indeed, Continuous Training is the part of the process where the ML Pipeline is re-triggered in order to create a new fresh model to be deployed in production.

Continuous Integration

Is no longer only about testing and validating code and components, but also testing and validating data, data schemas, and models.

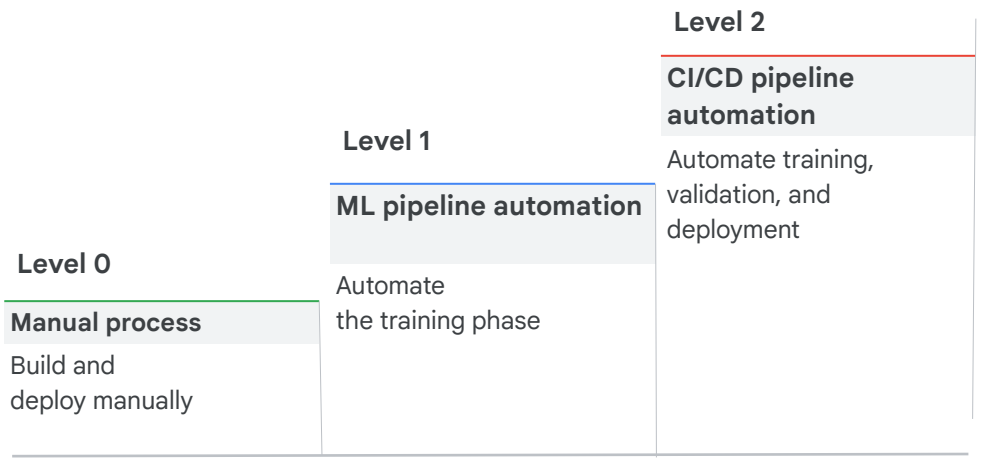
Continuous Delivery

Is no longer about a single software package or a service, but a system (ML training pipeline) that should automatically deploy another service.

Continuous Training

A new property, specific to ML systems, concerning automatically retraining and serving the models.

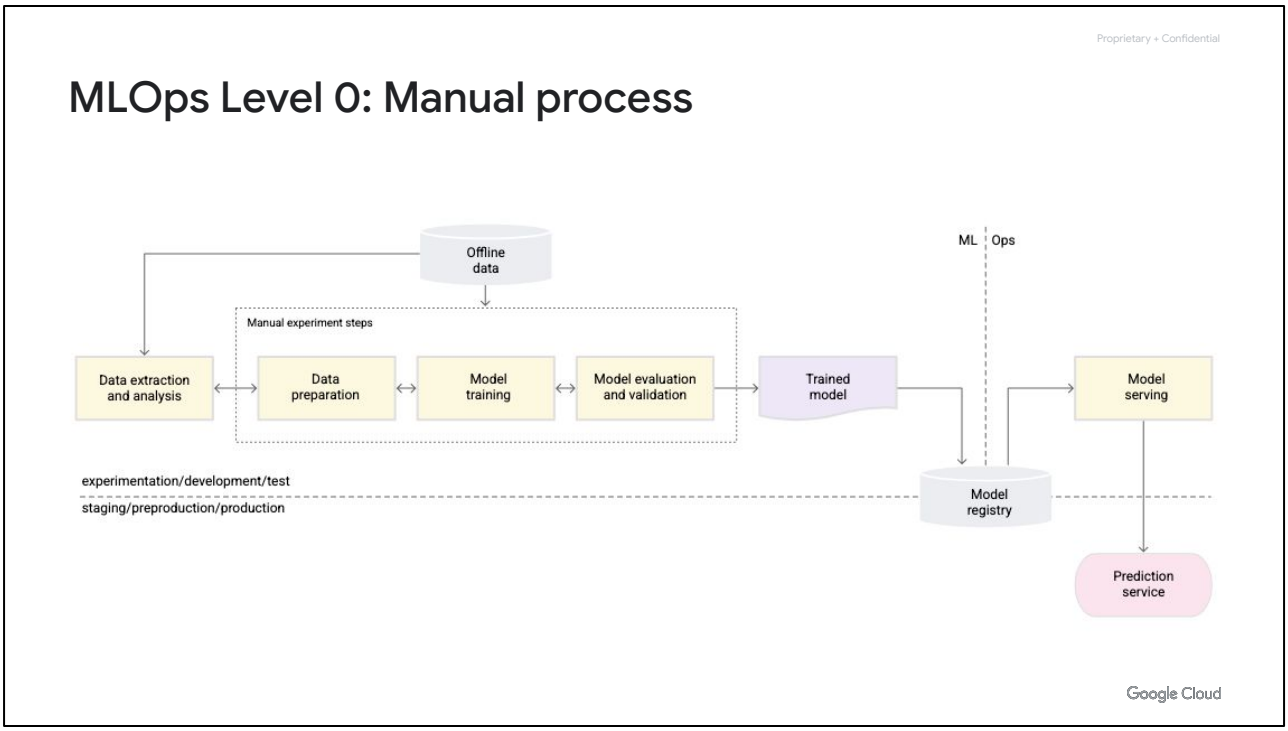
MLOps Operational Models



Publicly visible source: MLOps Level 0 from Cloud Architecture Center

Source: MLOps Architectures with CI/CD (slides) | Y22
https://docs.google.com/presentation/d/1OiGZOviGJf6XX7E91-41PGBy3aoRPZcibmF9dtH1Cms/edit?resourcekey=0-0LF8xS6di73GKb_vAVDZjw#slide=id.g1043fd8a725_0_2031

Achieving full automation in MLOps requires several steps. In the next slides we will explore how to move from manual building and deploying to CI/CD Pipeline automation.



Publicly visible source: [MLOps Level 0](#) from Cloud Architecture Center

Source: MLOps Architectures with CI/CD (slides) | Y22
https://docs.google.com/presentation/d/1OiGZOviGJf6XX7E91-41PGBy3aoRPZcibmF9dtH1Cms/edit?resourcekey=0-0LF8xS6di73GKb_vAVDZjw#slide=id.g1043fd8a725_0_2031

Let's review Level 0 (all manual) process. In the experimentation/development/test phase (there is no distinction between dev/prod environment) Data Scientist extract and analyse the data with different tools (Pandas/Seaborn etc). Then they prepared the data (cleaning and feature engineering operations) to feed the model. Then they train the model and evaluate it against some others models that they have trained. Here we go for the best one that provides the best metric. The best model is then validated against the validation dataset to see if it meets the business requirements. If so the trained model is placed into a Model Registry for serving. This process is highly manual and may involve several iterations which requires a lot of time and effort. When the model is in the registry Data scientists job is over. They are not involved in the model deployment. This effort is on the Ops team that they have to verify that the model works on the serving infrastructure. Because the model has not been tested before there is an high chance that the model will not work and thus an high chance to return to ML development.

MLOps Level 0: Manual process

Characteristics

- Manual, script-driven, and interactive process. Every step is manual.
- Disconnection between ML and operations.
- Infrequent release iterations.
- No CI & No CD
- Lack of active performance monitoring

Challenges

Successful operation would require you to:

- Actively monitor the quality of your model in production
- Frequently retrain your production models
- Continuously experiment with new implementations to produce the model

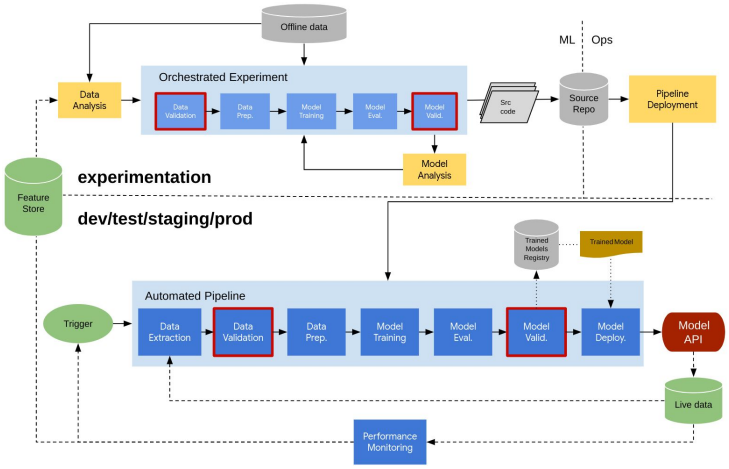
Publicly visible source: MLOps Level 0 from Cloud Architecture Center

Source: MLOps Final Report Template | PSO | Y21
https://docs.google.com/document/d/19os-tE7Pai6QQ1_pt7DsARgcv0Gvw-ATqYfBApLtywc/edit?resourcekey=0-4Z1XA9OhvuAjt73EnhwmhQ#

MLOps Architectures with CI/CD (slides) | Y22
https://docs.google.com/presentation/d/1OiGZOviGJf6XX7E91-41PGBy3aoRPZcibmF9dtH1Cms/edit?resourcekey=0-0LF8xS6di73GKb_vAVDZjw#slide=id.g1043fd8a725_0_2031

Here there is the list of Characteristics and challenges in Level 0.

MLOps Level 1: ML pipeline automation



Publicly visible source: MLOps Level 1 from Cloud Architecture Center

In the first level of MLOps, we introduce the pipeline automation.

1. We first divide experimentation to dev/test/statging/prod pipeline. They are very similar between each other but they differs in some aspects. Experimentation pipeline has the goal of let Data scientist experiment models while dev/test/staging/prod pipeline has the goal to create a model to deploy. Experimentation pipelines produces source codes not models. This source codes will be used later in the dev/test/staging/prod pipeline for model training. In fact, here the output is the source code of the pipeline and all its components.

2. The model is built on a sampled data set that can be extracted from the offline data or provided from feature store. Once the pipeline and its components have been created the Ops team deploy it into the other Automated pipeline that can be maintained separated depending on the environment we play. The automated pipeline includes similar steps that have been done in the experimentation

1. pipeline. This time the dataset is given as whole and we produce a model ready to deploy. This is stored in a model registry for versioning of course if the performance of the new model are better than what we currently have. Moreover, the model performs prediction on live data and its performance are continuously monitored to catch any skews. If there are some performance degradation, the dev/test/staging/prod pipeline is re-triggered automatically. Or if the model is old and need to adapt to a new problem we can go back to model experimentation

MLOps Level 1: ML pipeline automation

Characteristics

- Rapid experimentation
- Continuous training (CT) of the model in production including deployment
- Experimental-operational symmetry (data pipeline used for training is used for inference)
- Modularized code for components and pipelines

New Components

- Data & model validation
- Feature store (optional but helpful)
- Metadata management
- ML Pipeline triggers

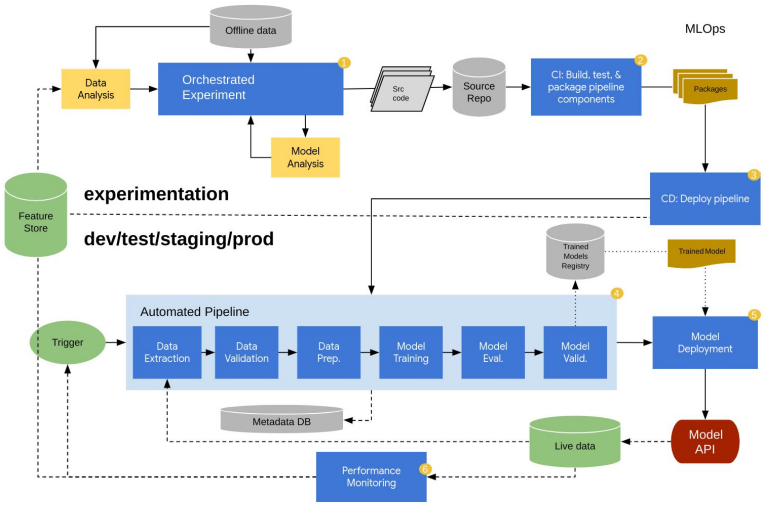
Publicly visible source: MLOps Level 1 from Cloud Architecture Center

Source: MLOps Final Report Template | PSO | Y21
https://docs.google.com/document/d/19os-tE7Pai6QQ1_pt7DsARgcv0Gvw-ATqYfBApLtywc/edit?resourcekey=0-4Z1XA9OhvuAjt73EnhwmhQ#

MLOps Architectures with CI/CD (slides) | Y22
https://docs.google.com/presentation/d/1OiGZOviGJf6XX7E91-41PGBy3aoRPZcibmF9dtH1Cms/edit?resourcekey=0-0LF8xS6di73GKb_vAVDZjw#slide=id.g1043fd8a725_0_2031

MLOps Level 1 of course is an improvement of Level0. However, there is room for improvements as you can see from the list.

MLOps Level 2: CI/CD pipeline automation



Publicly visible source: [MLOps Level 2] from Cloud Architecture Center

MLOps Level 2 introduces CI/CD in both pipeline components and model deployment. This is done fully automatically. Indeed, the source code of every single component that is coming out from the experimentation pipeline is built, tested and pushed to container registry as packages. Then a Continuous deploy pipeline verify that the modules do work before getting to the continuous training production pipeline. They might be deployed in the dev/test/staging environment before being adopted into production. Automated tests, smoke tests and other are important to avoid having a non working component in the continuous training production pipeline. Later, in the continuous training pipeline we train a new model based on the latest working components. In this processes, we record metadata for later comparisons with live production. The resulting model is compared with the current model deployed and if the performance are better than a continuous deployment process take the new model and deploy it to production. Again, here some test are conducted in order to see if the model works, make the appropriate predictions and they have the performance expected by the business. We continuously monitor the model in production to detect any training-serving skew, data skew or slowness. If some skew have been detected, performance monitor can restart a new model training process

hopefully with some new fresh data being integrated into the Feature Store.

Source: MLOps Final Report Template | PSO | Y21
https://docs.google.com/document/d/19os-tE7Pai6QQ1_pt7DsARgcv0Gvw-ATqYfBApLtywc/edit?resourcekey=0-4Z1XA9OhvuAjt73EnhwmhQ#

MLOps Architectures with CI/CD (slides) | Y22
https://docs.google.com/presentation/d/1OiGZOviGJf6XX7E91-41PGBy3aoRPZcibmF9dtH1Cms/edit?resourcekey=0-0LF8xS6di73GKb_vAVDZjw#slide=id.g1043fd8a725_0_2031

MLOps Level 2: ML pipeline automation

CI Characteristics

- Connects source control, so new code experiments are able to be tested quickly
- Unit testing of feature engineering logic
- Testing of model convergence
- Testing that each component produces the expected artifacts

CD Characteristics

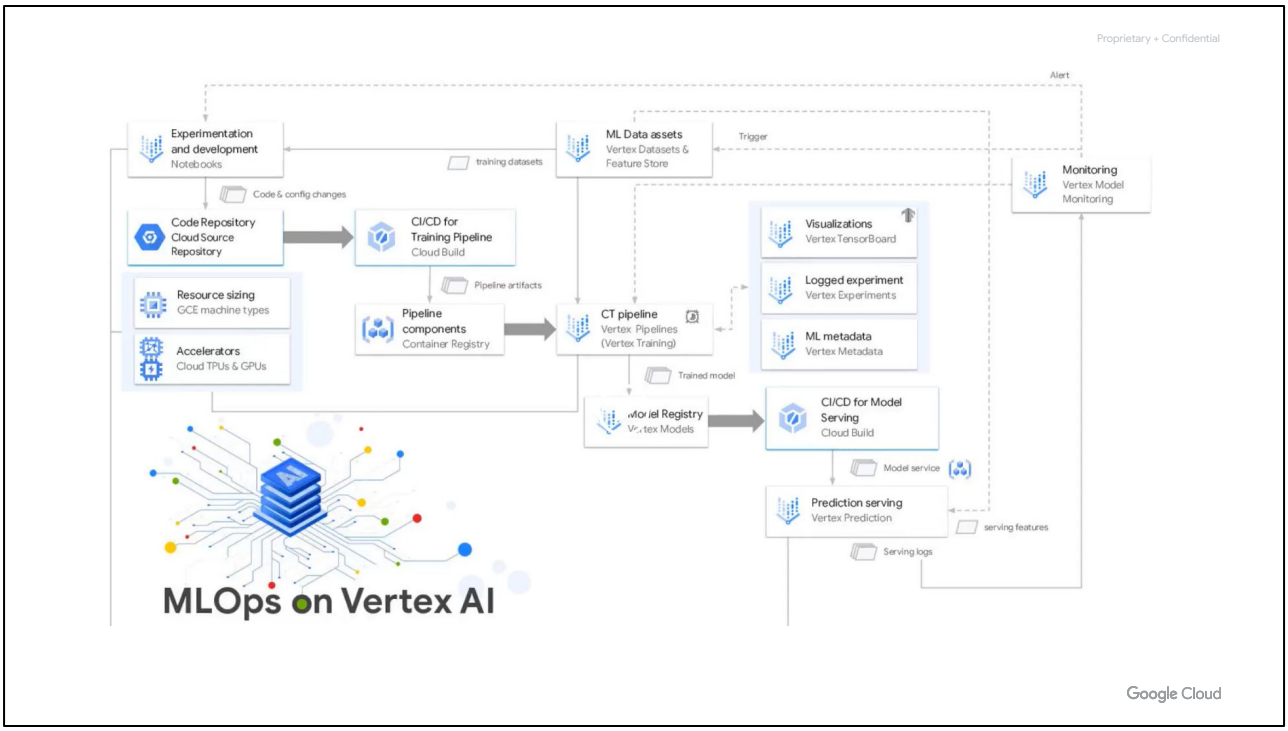
- Testing the prediction service itself (API call yields expected results)
- Testing prediction service performance
- Validating input data

Publicly visible source: MLOps Level 1 from Cloud Architecture Center

Source: MLOps Final Report Template | PSO | Y21
https://docs.google.com/document/d/19os-tE7Pai6QQ1_pt7DsARgcv0Gvw-ATqYfBApLtywc/edit?resourcekey=0-4Z1XA9OhvuAjt73EnhwmhQ#

MLOps Architectures with CI/CD (slides) | Y22
https://docs.google.com/presentation/d/1OiGZOviGJf6XX7E91-41PGBy3aoRPZcibmF9dtH1Cms/edit?resourcekey=0-0LF8xS6di73GKb_vAVDZjw#slide=id.g1043fd8a725_0_2031

MLOps Level 1 of course is an improvement of Level0. However, there is room for improvements as you can see from the list.



This pictures shows a full MLOps end-to-end solution with Vertex AI.

We start from top left using Vertex AI Workbench for Model experimentation and Development.

Data can be retrieved from Vertex AI Datasets, BigQuery, GCS.

Once a model is trained, a pipeline can be developed from a notebook. That code can be pushed to a git repository which can be implemented with Source Repositories.

From here a CI/CD pipeline is triggered to build test and deploy each component in a pre-staging continuous training pipeline.

The Pipeline components (artifacts) are store in a Container Registry for later use in the production continuous training pipeline which is hosted on Vertex Pipeline. When this pipeline is triggered (scheduled or even based) fetches data from Vertex Datasets/BigQuery/GCS or from Vertex Feature Store which stores features already pre-transformed ready to use. During pipeline execution Metadata (parameters, artifacts, metrics) are store in Vertex ML Metadata. These are useful to answer the questions like which dataset was used to train a certain model? Or Which run produced the most accurate model and what hyperparameters were used to train the model? Vertex AI Experiments lets you store the result of your experiments and visualize it with TensorBoard. This help to keep track of the performance of each model that has been trained in the pipeline. During model continuous training you can

leverage to multiple machines or accelerators to scale the training process.

The output of the continuous training pipeline is the trained model that goes to Vertex AI model registry. Here another CI/CD pipeline is trigger to that the model to the serving infrastructure after testing and a particular deployment strategy is chosen (integration testing, smoke testing, canary deployment etc). This can be done using Cloud Build, Container Registry, Git (to pull a container to wrap the model). The Vertex AI Prediction scales the serving infrastructure. Once the model is here you can enable model continuous monitoring to monitor for training-serving skew and/or data skew. If something is detected an alarm is triggered and through a cloud function model re-training happens. The alarm may be send to a data scientist to make new experiments to find an alternative model. This is often due to concept drift.

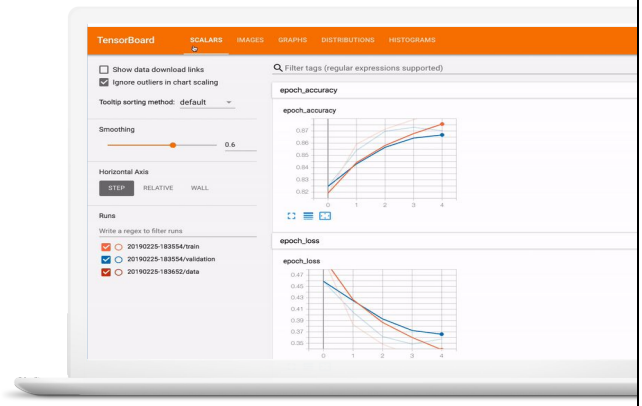
Source: MLOps Architectures with CI/CD (slides) | Y22
https://docs.google.com/presentation/d/1OiGZOviGJf6XX7E91-41PGBy3aoRPZcibmF9dtH1Cms/edit?resourcekey=0-0LF8xS6di73GKb_vAVDZjw#slide=id.g1043fd8a725_0_2031

Tracking and auditing experiments & artifact metadata

Vertex Tensorboard

Allow Data Scientists and ML Researchers to easily collaborate by making it seamless to track, compare, and share their experiments.

- Familiar experience: Experience matches the OSS TensorBoard tool users already know and love
- No Setup: Readily available TensorBoard that can be used with no installation or additional deployment
- Easy Collaboration: Share results and insights with peers to accelerate ML research and development
- Search & Compare: Quickly find specific experiments (e.g. based on hyperparameters) and compare them
- Enterprise Ready: Secure, cost-effective, scalable and integrated with the rest of GCP



Google Cloud

TensorBoard allows the team to easily collaborate by making it seamless to track compare and share their experiments using TensorBoard.

LAB: Vertex AI: Qwik Start: https://partner.cloudskillsboost.google/catalog_lab/3899

- Section: Use Vertex TensorBoard to visualize model performance.

Managed experiment tracking for faster model selection

Track, analyze, and discover ML experiments within your organization.

Customers can:

Track all runs and metrics in a central place

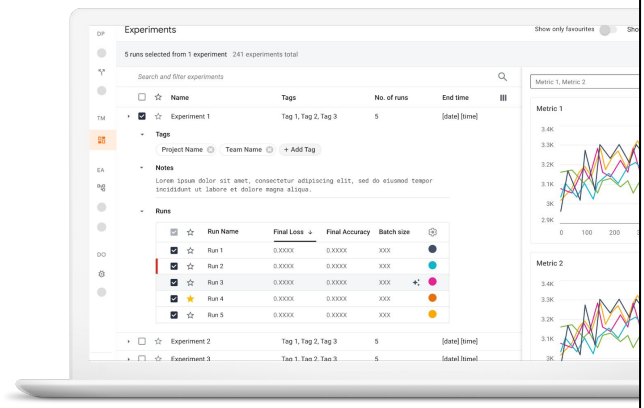
A central dashboard for training and evaluation runs across frameworks and environments in a single-pane view.

Visualize and compare multiple experiments

Analyze different training runs with rich, built-in visualizations. Quickly compare experiments and select the best model.

Share experiments and collaborate effectively

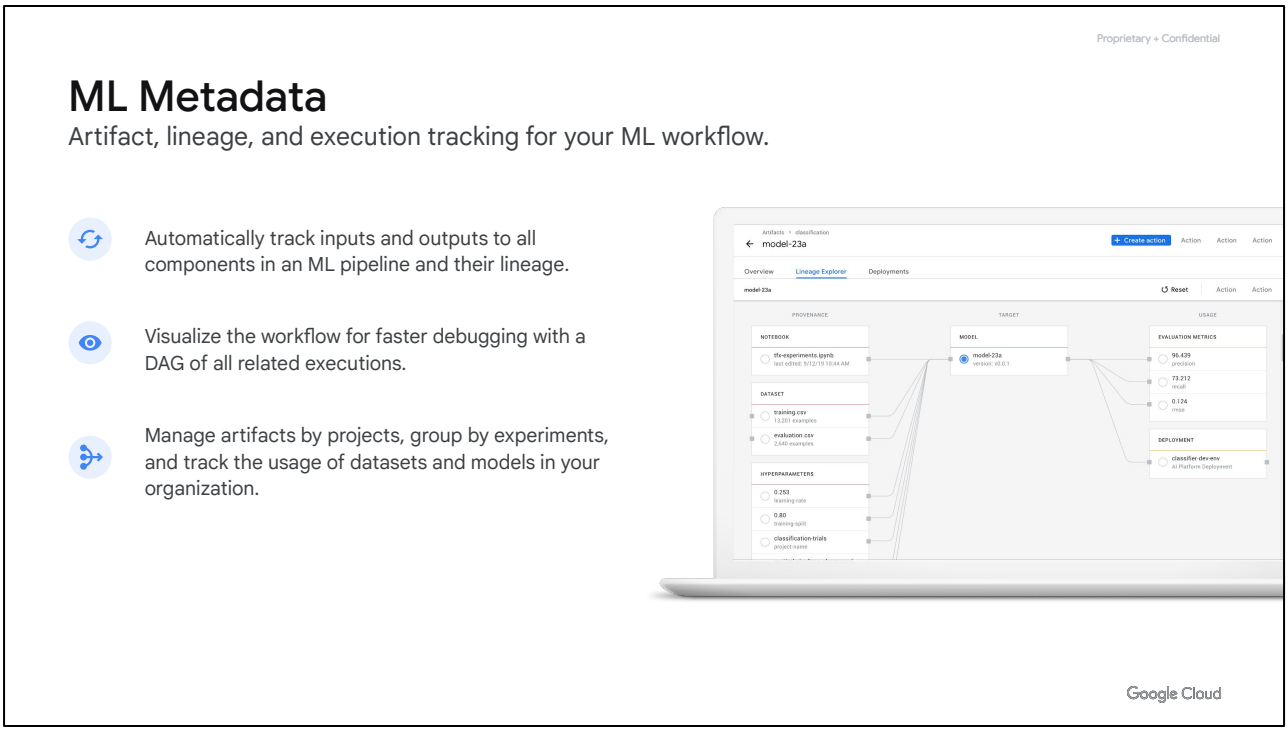
Search and discover experiments across the organization with ease; share findings easily.



Google Cloud

Source: MLOps Architectures with CI/CD (slides) | Y22
https://docs.google.com/presentation/d/1OiGZOviGJf6XX7E91-41PGBy3aoRPZcibmF9dtH1Cms/edit?resourcekey=0-0LF8xS6di73GKb_vAVDZjw#slide=id.g1043fd8a725_0_2031

This slide shows the advantages of using Vertex Experiments.



A critical part of the scientific method is recording both your observations and the parameters of an experiment. In data science, it is also critical to track the parameters, artifacts, and metrics used in a machine learning (ML) experiment. This metadata helps you:

- Analyze runs of a production ML system to understand changes in the quality of predictions.
- Analyze ML experiments to compare the effectiveness of different sets of hyperparameters.
- Track the lineage of ML artifacts, for example datasets and models, to understand just what contributed to the creation of an artifact or how that artifact was used to create descendant artifacts.
- Rerun an ML workflow with the same artifacts and parameters.
- Track the downstream usage of ML artifacts for governance purposes.

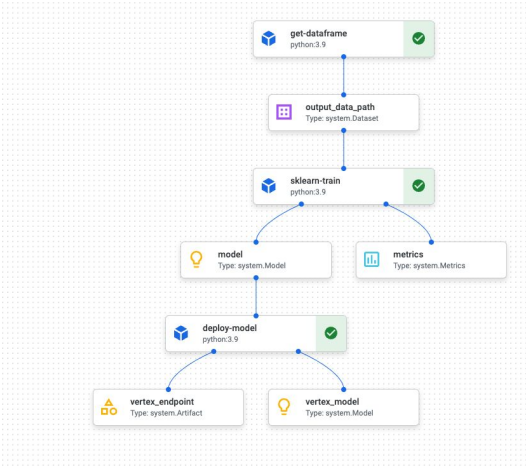
Vertex ML Metadata lets you record the metadata and artifacts produced by your ML system and query that metadata to help analyze, debug, and audit the performance of your ML system or the artifacts that it produces.

Vertex ML Metadata builds on the concepts used in the open source ML Metadata (MLMD) library that was developed by Google's TensorFlow Extended team.

Vertex ML Metadata captures your ML system's metadata as a graph.In the metadata graph, artifacts and executions are nodes, and events are edges that link artifacts as inputs or outputs of executions. Contexts represent subgraphs that are used to logically group sets of artifacts and executions. You can apply key-value pair metadata to artifacts, executions, and contexts. For example, a model could have metadata that describes the framework used to train the model and performance metrics, such as the model's accuracy, precision, and recall.

Learn more about tracking your ML system's metadata. If you're interested in analyzing metadata from Vertex AI Pipelines, check out this step-by-step tutorial.

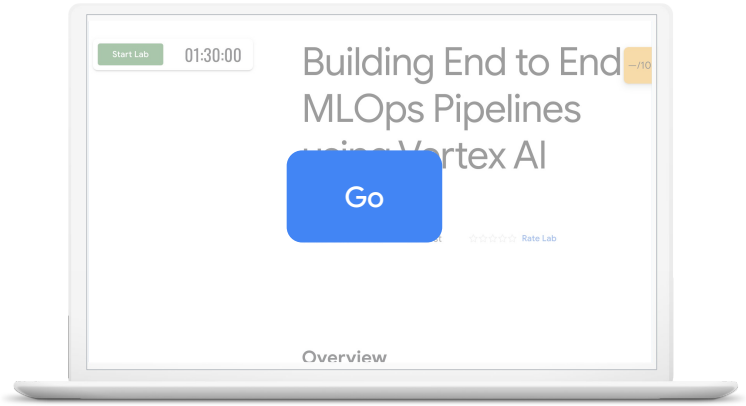
Example artifacts including Datasets, Models, Metrics, and generic Artifacts.



Recommended Lab

Building End to End MLOps Pipelines using Vertex AI

From the Course
ML Ops Using Vertex AI



Questions and answers

Thank you for attending this training!

We love your feedback! Please take a minute to complete the survey and help us improve our courses.


Google Cloud