Introduction to Mlflow

MLflow is an open source platform for managing the end-to-end machine learning lifecycle. It has the following primary components:

Tracking: Allows you to track experiments to record and compare parameters and results.

Models: Allow you to manage and deploy models from a variety of ML libraries to a variety of model serving and inference platforms.

Projects: Allow you to package ML code in a reusable, reproducible form to share with other data scientists or transfer to production.

Model Registry: Allows you to centralize a model store for managing models’ full lifecycle stage transitions: from staging to production, with capabilities for versioning and annotating. Databricks provides a managed version of the Model Registry in Unity Catalog.

Model Serving: Allows you to host MLflow models as REST endpoints. Databricks provides a unified interface to deploy, govern, and query your served AI models.

Utilizing the MLflow Model Registry for model versioning and management

The MLflow Model Registry is a powerful tool for managing the lifecycle of machine learning models, including versioning, staging, and deployment. Here’s how you can utilize it effectively:

**Key Features of MLflow Model Registry**

1. **Model Versioning**:
   * **Registering Models**: Use mlflow.<model\_flavor>.log\_model() to log and register models. [Each registered model is assigned a unique version number1](https://mlflow.org/docs/latest/model-registry.html).
   * [**Version Management**: Track different iterations of a model, facilitating comparison and rollback if necessary1](https://mlflow.org/docs/latest/model-registry.html).
2. **Model Staging**:
   * **Stages**: Models can be assigned to different stages such as “Staging”, “Production”, and “Archived”. [This helps manage the transition of models from development to production](https://mlflow.org/docs/latest/model-registry.html)[2](https://www.databricks.com/blog/2020/10/13/using-mlops-with-mlflow-and-azure.html).
   * **Stage Transitions**: Move models between stages to reflect their lifecycle status. [For example, promote a model from “Staging” to “Production” once it passes validation](https://mlflow.org/docs/latest/model-registry.html)[2](https://www.databricks.com/blog/2020/10/13/using-mlops-with-mlflow-and-azure.html).
3. **Model Aliasing and Tagging**:
   * **Aliases**: Assign mutable, named references to specific model versions, simplifying deployment and updates. [For example, use an alias like “champion” to refer to the current best model](https://mlflow.org/docs/latest/model-registry.html)[1](https://mlflow.org/docs/latest/model-registry.html).
   * [**Tags**: Label models with custom key-value pairs to categorize and document them effectively](https://mlflow.org/docs/latest/model-registry.html)[1](https://mlflow.org/docs/latest/model-registry.html).
4. **Model Lineage and Annotations**:
   * [**Lineage**: Track which experiment and run produced the model, providing transparency and reproducibility](https://mlflow.org/docs/latest/model-registry.html)[2](https://www.databricks.com/blog/2020/10/13/using-mlops-with-mlflow-and-azure.html).
   * [**Annotations**: Add descriptive notes to models for better collaboration and understanding](https://mlflow.org/docs/latest/model-registry.html)[1](https://mlflow.org/docs/latest/model-registry.html).

**Workflow Example**

1. **Registering a Model**:

**Python**

import mlflow

import mlflow.sklearn

# Train a model

model = ... # Your trained model

# Log and register the model

mlflow.sklearn.log\_model(model, "model")

mlflow.register\_model("runs:/<run\_id>/model", "MyModel")

1. **Transitioning Model Stages**:

**Python**

from mlflow.tracking import MlflowClient

client = MlflowClient()

client.transition\_model\_version\_stage(

name="MyModel",

version=1,

stage="Production"

)

1. **Assigning Aliases and Tags**:

**Python**

client.set\_model\_version\_tag(

name="MyModel",

version=1,

key="task",

value="classification"

)

client.set\_registered\_model\_alias(

name="MyModel",

alias="champion",

version=1

)

**Benefits**

* **Centralized Management**: A single location to manage all models, ensuring consistency and ease of access.
* **Collaboration**: Facilitates teamwork by providing a clear history and documentation of models.
* [**Scalability**: Supports large-scale model management and deployment](https://mlflow.org/docs/latest/model-registry.html)[1](https://mlflow.org/docs/latest/model-registry.html)[2](https://www.databricks.com/blog/2020/10/13/using-mlops-with-mlflow-and-azure.html).

Top of Form

Bottom of Form

Logging parameters, metrics, and artifacts during model training

Logging parameters, metrics, and artifacts during model training is essential for tracking the performance and reproducibility of your machine learning experiments. Here’s how you can do it effectively using MLflow:

**Using MLflow for Logging**

**Basic Setup**

1. **Install MLflow**:
2. pip install mlflow
3. **Initialize MLflow**:

**Python**

import mlflow

mlflow.set\_experiment("my\_experiment")

AI-generated code. Review and use carefully. [More info on FAQ](https://www.bing.com/new#faq).

**Logging Parameters, Metrics, and Artifacts**

1. **Logging Parameters**: Parameters are the hyperparameters or configuration settings used in your model training.

**Python**

mlflow.log\_param("learning\_rate", 0.01)

mlflow.log\_param("batch\_size", 32)

AI-generated code. Review and use carefully. [More info on FAQ](https://www.bing.com/new#faq).

1. **Logging Metrics**: Metrics are the performance measures of your model, such as accuracy or loss.

**Python**

mlflow.log\_metric("accuracy", 0.95)

mlflow.log\_metric("loss", 0.05)

1. **Logging Artifacts**: Artifacts are files generated during the training process, such as model weights, plots, or logs.

**Python**

mlflow.log\_artifact("path/to/model\_weights.h5")

mlflow.log\_artifact("path/to/training\_log.txt")

**Automatic Logging with**mlflow.autolog()

MLflow provides an automatic logging feature that simplifies the process. [By calling mlflow.autolog(), MLflow will automatically log parameters, metrics, and artifacts for supported libraries like TensorFlow, PyTorch, and Scikit-learn](https://mlflow.org/blog/mlflow-autolog)[1](https://mlflow.org/blog/mlflow-autolog)[2](https://www.restack.io/docs/mlflow-knowledge-mlflow-pytorch-integration).

**Example with PyTorch**

**Python**

import mlflow.pytorch

import torch

import torch.nn as nn

import torch.optim as optim

# Enable automatic logging

mlflow.pytorch.autolog()

# Define a simple model

class SimpleModel(nn.Module):

def \_\_init\_\_(self):

super(SimpleModel, self).\_\_init\_\_()

self.fc = nn.Linear(10, 1)

def forward(self, x):

return self.fc(x)

model = SimpleModel()

criterion = nn.MSELoss()

optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training loop

with mlflow.start\_run():

for epoch in range(10):

# Dummy input and target

inputs = torch.randn(32, 10)

targets = torch.randn(32, 1)

# Forward pass

outputs = model(inputs)

loss = criterion(outputs, targets)

# Backward pass and optimization

optimizer.zero\_grad()

loss.backward()

optimizer.step()

# Log metrics

mlflow.log\_metric("loss", loss.item(), step=epoch)

**Viewing Logged Data**

You can view the logged parameters, metrics, and artifacts in the MLflow UI:

1. **Start the MLflow UI**:
2. mlflow ui
3. **Access the UI**: Open your browser and go to http://localhost:5000.

**Benefits**

* **Reproducibility**: Ensures that experiments can be reproduced with the same parameters and configurations.
* **Transparency**: Provides a clear record of what was done during model training.
* [**Collaboration**: Facilitates sharing and collaboration among team members](https://mlflow.org/blog/mlflow-autolog).

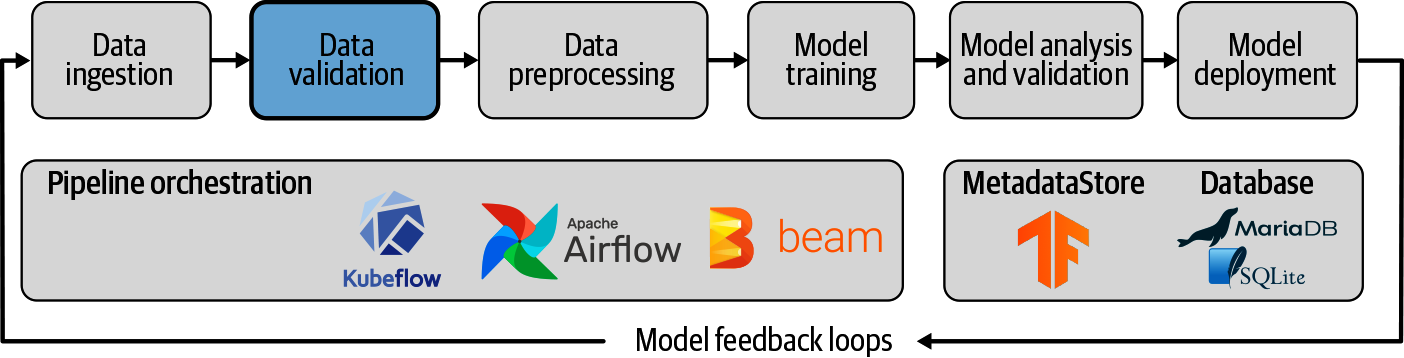
Machine Learning Pipelines

**Pipeline**

A machine learning pipeline is a way to control and automate the workflow it takes to produce a machine learning model. Machine learning pipelines consist of multiple sequential steps that do everything from data extraction and preprocessing to model training and deployment.

Machine learning pipelines are iterative as every step is repeated to continuously improve the accuracy of the model and achieve the end goal.

The term **Pipeline** is used generally to describe the independent sequence of steps that are arranged together to achieve a task.



**Key Stages of a Machine Learning Pipeline**

1. **Data Collection**:
   * **Gathering Data**: Collect raw data from various sources such as databases, APIs, or files.
   * [**Data Storage**: Store the collected data in a structured format for easy access and processing](https://www.ibm.com/topics/machine-learning-pipeline)[1](https://www.ibm.com/topics/machine-learning-pipeline).
2. **Data Preprocessing**:
   * **Cleaning**: Handle missing values, remove duplicates, and correct errors in the data.
   * [**Transformation**: Normalize, scale, or encode data to prepare it for modeling2](https://dzone.com/articles/building-a-powerful-ai-and-machine-learning-pipeli).
3. **Feature Engineering**:
   * **Feature Selection**: Identify the most relevant features for the model.
   * [**Feature Creation**: Generate new features from existing data to improve model performance](https://dzone.com/articles/building-a-powerful-ai-and-machine-learning-pipeli)
4. **Model Training**:
   * **Algorithm Selection**: Choose the appropriate machine learning algorithm based on the problem.
   * [**Training**: Train the model using the preprocessed data and selected features3](https://valohai.com/machine-learning-pipeline/).
5. **Model Evaluation**:
   * **Validation**: Evaluate the model’s performance using validation techniques like cross-validation.
   * [**Metrics**: Use metrics such as accuracy, precision, recall, and F1-score to assess the model3](https://valohai.com/machine-learning-pipeline/).
6. **Model Deployment**:
   * **Integration**: Deploy the trained model into a production environment where it can make predictions on new data.
7. [**Monitoring**: Continuously monitor the model’s performance and update it as needed1](https://www.ibm.com/topics/machine-learning-pipeline).
8. **Model Management**:
   * **Versioning**: Keep track of different versions of the model to manage updates and rollbacks.
   * [**Documentation**: Document the entire pipeline, including data sources, preprocessing steps, and model parameters2](https://dzone.com/articles/building-a-powerful-ai-and-machine-learning-pipeli).

Workflow Orchestration and Automation

Workflow orchestration and automation are key concepts in modern business operations, especially for enhancing efficiency and productivity.

**Workflow Automation**

Workflow automation focuses on automating individual tasks within a process. This involves using software to perform repetitive tasks without human intervention, such as sending emails, updating databases, or generating reports.

**Workflow Orchestration**

Workflow orchestration, on the other hand, is about managing and coordinating these automated tasks to create a seamless end-to-end process. [It ensures that all tasks are executed in the correct sequence and that data flows smoothly between different systems and applications](https://camunda.com/blog/2024/02/what-is-workflow-orchestration-guide-use-cases/)

Performance Optimization and Scalability

Optimizing performance and ensuring scalability in a machine learning (ML) pipeline are crucial for handling large datasets and complex computations efficiently. Here are some key strategies:

**Performance Optimization**

1. [**Hyperparameter Tuning**: Techniques like grid search and Bayesian optimization can help find the best model parameters, improving accuracy and efficiency1](https://www.trigyn.com/insights/advanced-machine-learning-big-data-scalability-and-performance).
2. [**Caching**: Storing intermediate results can reduce redundant computations and speed up processing](https://www.trigyn.com/insights/advanced-machine-learning-big-data-scalability-and-performance)[1](https://www.trigyn.com/insights/advanced-machine-learning-big-data-scalability-and-performance).
3. [**Efficient Data Handling**: Using optimized data structures and efficient data loading mechanisms can significantly reduce processing time](https://www.trigyn.com/insights/advanced-machine-learning-big-data-scalability-and-performance)[2](https://www.codementor.io/blog/scalable-ml-models-6rvtbf8dsd).
4. [**Parallel Processing**: Leveraging multi-threading and distributed computing frameworks like Apache Spark can accelerate data processing and model training2](https://www.codementor.io/blog/scalable-ml-models-6rvtbf8dsd).
5. [**Algorithm Optimization**: Choosing the right algorithms and optimizing their implementation can lead to faster and more efficient model training](https://www.trigyn.com/insights/advanced-machine-learning-big-data-scalability-and-performance)[1](https://www.trigyn.com/insights/advanced-machine-learning-big-data-scalability-and-performance).

**Scalability**

1. [**Distributed Computing**: Using frameworks like TensorFlow, PyTorch, or Apache Spark to distribute computations across multiple nodes can handle larger datasets and more complex models](https://www.trigyn.com/insights/advanced-machine-learning-big-data-scalability-and-performance)[2](https://www.codementor.io/blog/scalable-ml-models-6rvtbf8dsd).
2. [**Dynamic Resource Allocation**: Automatically adjusting resources based on the workload can ensure efficient use of computational power](https://www.trigyn.com/insights/advanced-machine-learning-big-data-scalability-and-performance)[3](https://fastercapital.com/content/Pipeline-optimization--How-to-improve-the-efficiency-and-performance-of-your-pipeline-using-data-analytics-and-machine-learning.html).
3. [**Load Balancing**: Distributing the workload evenly across multiple servers can prevent bottlenecks and ensure smooth operation](https://www.trigyn.com/insights/advanced-machine-learning-big-data-scalability-and-performance)[3](https://fastercapital.com/content/Pipeline-optimization--How-to-improve-the-efficiency-and-performance-of-your-pipeline-using-data-analytics-and-machine-learning.html).
4. [**Modular Pipeline Design**: Designing the pipeline in modular stages (e.g., data preprocessing, feature engineering, model training, evaluation, and deployment) allows for easier scaling and maintenance4](https://www.educba.com/machine-learning-pipeline/).

**Combining Both**

* [**Resource Utilization and Monitoring**: Continuously monitoring resource usage and performance metrics can help identify bottlenecks and optimize resource allocation2](https://www.codementor.io/blog/scalable-ml-models-6rvtbf8dsd).
* [**Model Deployment**: Using containerization (e.g., Docker) and orchestration tools (e.g., Kubernetes) can help in scaling the deployment of ML models](https://www.trigyn.com/insights/advanced-machine-learning-big-data-scalability-and-performance)[2](https://www.codementor.io/blog/scalable-ml-models-6rvtbf8dsd).

Exporting trained models for deployment

Exporting trained models for deployment involves several key steps to ensure that your model can be efficiently and reliably used in a production environment. Here’s a general guide to help you through the process:

**1. Finalize Your Model**

Before exporting, ensure your model is fully trained and validated. Confirm that it meets performance criteria on your test data and is free from issues.

**2. Choose the Right Format**

Select a format suitable for your deployment environment. Common formats include:

* **ONNX (Open Neural Network Exchange):** Supports interoperability between different frameworks.
* **TensorFlow SavedModel:** For TensorFlow models, supports various deployment scenarios.
* **TorchScript:** For PyTorch models, allows you to serialize and optimize models for production.
* **PMML (Predictive Model Markup Language):** A standard for representing predictive models.
* **3. Export the Model**

Depending on the framework you used, exporting the model will differ:

**TensorFlow**

python

Copy code

import tensorflow as tf

# Assuming `model` is your trained model

model.save('path/to/saved\_model')

**4. Optimize the Model**

Consider optimizing the model for inference. This can involve:

* **Quantization:** Reducing the model’s precision to lower its computational requirements.
* **Pruning:** Removing parts of the model that contribute less to performance.
* **Model Distillation:** Training a smaller model to mimic a larger, more complex model.

**5. Test the Exported Model**

Verify that the exported model performs correctly in a test environment. Ensure it produces the expected results and integrates well with your deployment infrastructure.

**6. Set Up the Deployment Environment**

Prepare the infrastructure where the model will be deployed. This may involve:

* **Choosing a Platform:** Cloud services (AWS, Azure, GCP) or on-premises servers.
* **Configuring APIs:** For serving the model via a REST API or gRPC.
* **Monitoring and Logging:** Implementing systems to monitor model performance and log errors.

**7. Deploy the Model**

Deploy the model to the production environment. Depending on the setup, this might involve:

* **Containerization:** Using Docker or other container solutions.
* **Serverless Deployment:** Using cloud-based functions.
* **Batch Processing:** Running the model on scheduled intervals for batch predictions.

**8. Monitor and Maintain**

After deployment, continuously monitor the model’s performance. Track metrics, gather feedback, and update the model as needed based on real-world performance.

**9. Handle Versioning**

If you update the model or deploy new versions, manage versioning to ensure smooth transitions and maintain backward compatibility if necessary.

TFX Pipeline

TensorFlow Extended (TFX) is an end-to-end platform for deploying production machine learning (ML) pipelines. It provides a comprehensive suite of components and tools to manage the entire ML lifecycle, from data ingestion to model deployment.

**Key Components of a TFX Pipeline**

1. **ExampleGen**: Ingests and optionally splits the input dataset.
2. **StatisticsGen**: Generates statistics over the dataset.
3. **SchemaGen**: Creates a schema by inferring types, categories, and ranges from the data.
4. **ExampleValidator**: Identifies anomalies in the dataset.
5. **Transform**: Performs feature engineering on the dataset.
6. **Trainer**: Trains a TensorFlow model.
7. **Tuner**: Tunes the hyperparameters of the model.
8. **Evaluator**: Analyzes the training results and validates the model.
9. **InfraValidator**: Ensures the model is servable from the infrastructure.
10. [**Pusher**: Deploys the model to a serving infrastructure](https://www.tensorflow.org/tfx)

Data Ingestion and Preprocessing using data from databaes, files, or streaming platforms

Data ingestion frameworks, platforms, and systems provide a complete end-to-end solution for data ingestion. It involves ingesting data in various formats such as structured data from databases, unstructured data from documents and files, or streaming data from sensors and other real-time sources.

Data ingestion and preprocessing are crucial steps in building robust data pipelines. Here’s an overview of how to handle data from various sources:

**Data Ingestion**

1. **From Databases**:
   * **Batch Ingestion**: Use tools like Apache Sqoop to import data from relational databases into Hadoop.
   * [**Real-Time Ingestion**: Use Change Data Capture (CDC) tools like Debezium to capture and stream changes from databases1](https://www.informatica.com/ca/resources/articles/what-is-data-ingestion.html).
2. **From Files**:
   * **Batch Ingestion**: Use tools like Apache Nifi or AWS Glue to ingest data from CSV, JSON, or XML files.
   * [**Real-Time Ingestion**: Use file monitoring tools that trigger ingestion processes when new files are added1](https://www.informatica.com/ca/resources/articles/what-is-data-ingestion.html).
3. **From Streaming Platforms**:
   * **Kafka**: Use Apache Kafka to ingest streaming data from various sources like IoT devices or log files.
   * [**Kinesis**: Use AWS Kinesis for real-time data streaming and processing2](https://www.confluent.io/learn/data-ingestion/).

**Data Preprocessing**

1. **Data Cleaning**:
   * **Handling Missing Values**: Impute missing values or remove incomplete records.
   * **Data Normalization**: Scale data to a standard range, especially for machine learning models.
2. **Data Transformation**:
   * **Feature Engineering**: Create new features from existing data to improve model performance.
   * **Data Encoding**: Convert categorical data into numerical formats using techniques like one-hot encoding.
3. **Data Validation**:
   * **Schema Validation**: Ensure data conforms to a predefined schema to catch anomalies early.
   * [**Consistency Checks**: Verify data consistency across different sources and formats](https://www.informatica.com/ca/resources/articles/what-is-data-ingestion.html)[3](https://www.ibm.com/think/topics/data-ingestion).

**Tools and Platforms**

* **Apache NiFi**: For automating data flow between systems.
* **AWS Glue**: For ETL processes and data cataloging.
* **Apache Kafka**: For real-time data streaming and ingestion.
* [**Apache Spark**: For large-scale data processing and transformation2](https://www.confluent.io/learn/data-ingestion/).

Model Training and Evaluation

In a TFX (TensorFlow Extended) pipeline, model training and evaluation are critical steps to ensure your machine learning model performs well and is ready for deployment. Here’s a detailed look at these components:

**Model Training**

1. **Trainer Component**:
   * **Purpose**: Trains your machine learning model using the preprocessed data.
   * **Implementation**: You define a training module that includes your model architecture, training loop, and evaluation metrics.
   * **Example**:

**Python**

from tfx.components import Trainer

from tfx.proto import trainer\_pb2

trainer = Trainer(

module\_file='path/to/your/trainer\_module.py',

examples=example\_gen.outputs['examples'],

transform\_graph=transform.outputs['transform\_graph'],

train\_args=trainer\_pb2.TrainArgs(num\_steps=10000),

eval\_args=trainer\_pb2.EvalArgs(num\_steps=5000)

)

**Model Evaluation**

1. **Evaluator Component**:
   * **Purpose**: Analyzes the performance of the trained model and validates it against a baseline model.
   * **Implementation**: Uses TensorFlow Model Analysis (TFMA) to evaluate the model on various metrics and data slices.
   * **Example**:

**Python**

from tfx.components import Evaluator

from tfx.proto import evaluator\_pb2

eval\_config = evaluator\_pb2.EvalConfig(

model\_specs=[evaluator\_pb2.ModelSpec(label\_key='label')],

metrics\_specs=[evaluator\_pb2.MetricsSpec(metrics=[

evaluator\_pb2.MetricConfig(class\_name='BinaryAccuracy'),

evaluator\_pb2.MetricConfig(class\_name='AUC')

])],

slicing\_specs=[evaluator\_pb2.SlicingSpec()]

)

evaluator = Evaluator(

examples=example\_gen.outputs['examples'],

model=trainer.outputs['model'],

eval\_config=eval\_config

)

**Workflow**

1. **Data Ingestion**: Data is ingested using the ExampleGen component.
2. **Data Validation and Transformation**: Data is validated and transformed using SchemaGen, ExampleValidator, and Transform components.
3. **Model Training**: The Trainer component trains the model using the transformed data.
4. [**Model Evaluation**: The Evaluator component assesses the model’s performance and validates it against a baseline](https://www.tensorflow.org/tfx/guide/evaluator)[1](https://www.tensorflow.org/tfx/guide/evaluator)[2](https://www.tensorflow.org/tfx/guide/understanding_tfx_pipelines).

**Benefits**

* **Automated Workflow**: TFX automates the entire ML pipeline, making it easier to manage and scale.
* **Consistent Evaluation**: Ensures models meet performance criteria before deployment.
* [**Scalability**: Supports distributed training and evaluation, handling large datasets efficiently](https://www.tensorflow.org/tfx/guide/evaluator).

Top of Form

Bottom of Form

Model validation with holdout dataset or cross-validation and exporting and serving model

**Model Validation**

**Holdout Dataset**

The holdout method involves splitting your dataset into two parts: a training set and a validation (or test) set. Common split ratios are 70:30 or 80:20. The training set is used to train the model, while the validation set is used to evaluate its performance. [This method is straightforward but can be less reliable for small datasets as it might not capture the variability in the data1](https://www.pluralsight.com/resources/blog/guides/validating-machine-learning-models-scikit-learn).

**Cross-Validation**

Cross-validation is a more robust method, especially for small datasets. It involves splitting the data into multiple folds (e.g., 5 or 10). The model is trained on all but one fold and validated on the remaining fold. This process is repeated until each fold has been used as the validation set. [The results are then averaged to provide a more reliable estimate of model performance](https://machinelearningmastery.com/training-validation-test-split-and-cross-validation-done-right/)

**Exporting and Serving Models**

**Exporting Models**

1. **TensorFlow**:
   * Save the model using model.save('path/to/model').
   * Export to TensorFlow SavedModel format for deployment.
2. **PyTorch**:
   * Save the model state using torch.save(model.state\_dict(), 'path/to/model.pth').
   * Export to ONNX format if needed for interoperability.

**Serving Models**

1. **TensorFlow Serving**:
   * A flexible, high-performance serving system for machine learning models designed for production environments.
   * Deploy models using Docker containers for scalability.
2. **TorchServe**:
   * A model serving framework for PyTorch models.
   * Supports multi-model serving and dynamic batching for efficient inference.

**Example TFX Pipeline for Model Validation and Serving**

1. **Model Validation**:
   * Use the Evaluator component in TFX to validate the model using cross-validation or a holdout dataset.
   * Example

from tfx.components import Evaluator

from tfx.proto import evaluator\_pb2

eval\_config = evaluator\_pb2.EvalConfig(

model\_specs=[evaluator\_pb2.ModelSpec(label\_key='label')],

metrics\_specs=[evaluator\_pb2.MetricsSpec(metrics=[

evaluator\_pb2.MetricConfig(class\_name='BinaryAccuracy'),

evaluator\_pb2.MetricConfig(class\_name='AUC')

])],

slicing\_specs=[evaluator\_pb2.SlicingSpec()]

)

evaluator = Evaluator(

examples=example\_gen.outputs['examples'],

model=trainer.outputs['model'],

eval\_config=eval\_config

)

**2 . Model Exporting and Serving**:

* Use the Pusher component in TFX to export and serve the model.
* Example:

from tfx.components import Pusher

from tfx.proto import pusher\_pb2

pusher = Pusher(

model=trainer.outputs['model'],

model\_blessing=evaluator.outputs['blessing'],

push\_destination=pusher\_pb2.PushDestination(

filesystem=pusher\_pb2.PushDestination.Filesystem(

base\_directory='path/to/serving/model'

)

))

Managing the end-to end ML pipeline worflow and coordinating the execution of different pipeline components