AI Factory ML/OPS Design

# Purpose

The purpose of this document is to provide the guidelines for various activities related to ML/OPS and Model Lifecycle management for the AI Factory Implementation at SCB.

# Requirements

These requirements detail the core functionalities necessary for the solution while aiming to establish the fundamental ML/OPS capabilities required for AI Factory implementation at SCB.

## Functional Requirements

These requirements focus on the functionality required from the solution. The intention here is to capture the information around “What” are the core ML/OPS capabilities needed for the AI Factory Implementation.

| **ID** | ***Type*** | **Description** |
| --- | --- | --- |
| FR-01 | AI Lifecycle Management | The system shall provide the ability to manage AI development lifecycle, i.e. Exploratory Data Analysis, training, model selection, model testing, model validation, deployment, inference and monitoring |
| FR-02 | AI Lifecycle Management | The system shall provide the ability for users to perform various activities across AI Lifecycle using IDEs and Frameworks, i.e. Databricks Notebooks, PyTorch, Tensorflow |
| FR-03 | AI Lifecycle Management | The system shall provide the ability for Experiment Tracking and reproducibility (e.g. MLFlow, Weights and Biases) |
| FR-04 | AI Lifecycle Management | The system shall provide the ability for Model Versioning and Registry and Model Deployment |
| FR-05 | Data Management | The system shall provide the ability for creation and management of Data Ingestion Pipelines (Structured, Unstructured, Streaming) |
| FR-06 | Data Management | The system shall provide the ability for Data Transformation, feature engineering, feature selection and Feature store integration |
| FR-07 | Data Management | The system shall provide the ability for Data Versioning and Lineage - Training, Testing, Validation and Inference |
| FR-08 | Data Management | The system shall provide the ability for managing Data Quality, Integrity and Validation |
| FR-09 | Data Management | The system shall provide the ability for Data Privacy Management (PI, PCI DSS, Synthetic Data Generation) |
| FR-10 | Compute Orchestration | The system shall provide the ability for :   * Distributed Training Capabilities - Data Parallelism and Model Parallelism * Job Scheduling and Workload Management * Auto Scaling |
| FR-11 | Model Testing, Validation, Monitoring and Observability | The system shall provide the ability for :   * Performing various tests - Data Suitability, robustness, model selection, A/B, RAG Triad etc. for various types of algorithms, i.e. ML, Deep Learning, GenAI * Drift detection * Performance Monitoring (accuracy, latency) * Logging and metrics collection * Feedback Loops * Champion Challenger |
| FR-12 | Collaboration Tools | The system shall provide the ability to collaborate using:   * Shared Notebooks, Dashboards * Team based project and model management * Notifications and Alerting |

## Non - Functional Requirements

These sets of requirements focus on the non-functional aspects such as availability, scalability, reliability etc that impact the performance of the overall solution.

| **ID** | **Type** | **Description** | **Relevant Section** |
| --- | --- | --- | --- |
| NFR-01 | Scalability | * The system should be able to handle Increasing Models, Data Volumes and Users. * The system should be able to handle Horizontal Scaling of Compute and Storage. |  |
| NFR-02 | Performance | The system should be able to handle Low Latency model inference (esp. real-time use cases).  The system should be able to handle High Throughput for Batch Workloads. |  |
| NFR-03 | Availability and Reliability | The should be able to provide:   * High Uptime (95% or more) * Fault Tolerance and Disaster Recovery | NA for ML/OPS since it is a platform requirement |
| NFR-04 | Security and Compliance | The system should be able to handle:   * GDPR and other regulation compliance * Secure Data Zones and role-based access control | NA for ML/OPS since it is a platform requirement |
| NFR-05 | Interoperability | The system should be able to provide:   * Support for open standards * Plug and play for tools like MLFlow | NA for ML/OPS since it is a platform requirement |
| NFR-06 | Maintainability | The system should be able to provide:   * Easy Upgrades, patching and system health checks * Modular architecture for faster upgrades | NA for ML/OPS since it is a platform requirement |
| NFR-07 | Auditability and Traceability | The system should be able to provide:   * End to end traceability of data, features, models and predictions * Full audit trail for model decisions and updates |  |

# Architectural Decisions

This section showcases key technical decisions that impact the overall solution architecture. Each decision influences the overall design of the solution and choice of components moulded into the architecture. The intent of this section is to showcase the thought process that went into making critical decisions around architecture.

# 

| **ID** | **Type** | **Questions** | **Status** |
| --- | --- | --- | --- |
| [AD-01](https://docs.google.com/document/d/1yM23jIDHVe4L977MhXGzKaIHka9mZYn6t0kDoZa7s78/edit?tab=t.0#heading=h.lqoqnsem9n6) | Model Deployment | What is the model deployment pattern chosen to deploy models to Dev, Staging and Prod? I.e. deploy code vs deploy model | Open |
| [AD-02](#_rp28kudb67zm) | Infrastructure | How shall developers/data scientists provision required databricks [components](#_4l1j9j6tzx4i) in the Azure Databricks sandbox workspace? | Open |
| AD-03 | Monitoring | How shall Data Scientists view model training metrics and evaluation metrics from production given that they wont have access to the Databricks Production Workspace? | WIP |

| AD-01 | What is the model deployment pattern chosen to deploy models to Prod? I.e. deploy code vs deploy model |
| --- | --- |
| Category | Model Deployment |
| Description | How should we deploy models to production? Should we train all models in the sandbox environment and then deploy the model to prod or should we deploy code to each environment and retrain the model in the production environment on production data. ([reference](https://docs.databricks.com/aws/en/machine-learning/mlops/deployment-patterns)) |
| FRs/NFRs Addressed |  |
| Options | **Option - 1**: Deploy Code (recommended option)  In this pattern, the code to train models is developed in the sandbox environment and then the code is deployed to each environment. The model is then trained on each environment, i.e. dev, staging (subset of the data) and production (full production data) to create the model.  **Option - 2**: Deploy Model  In this pattern, the model artifact is generated by training code in the sandbox environment, tested in dev/staging and deployed to production. This pattern is typically used when operating in a single workspace environment or model training is very expensive and hard to reproduce. |
| Justification | **Pros/Cons of Option-1:**  ➕.Allows for training on production data, where production data is restricted in dev/sandbox environment.  ➕Safer automated model retraining, since training code is tested and validated before deployment.  ➕Feature engineering and any supporting code follows the same process and goes through integration tests in dev and staging, resulting in a stabler and reproducible system.  ➖May present a steeper learning curve in terms of Data Scientists handing off code to collaborators (i.e. deployment team/ml engineers) for deployment in production, since Data Scientists typically would have read-only access in production.  **Pros/Cons of Option-2:**  ➕Handoff by data scientists is simpler, since only a trained model needs to be deployed.  ➕Model only needs to trained once. In cases where model training is expensive, this could be an advantage.  ➖Model retraining could present challenges, since the model training code does not undergo unit testing/integration testing in dev/staging.  ➖Supporting code, i.e. feature engineering, inference and monitoring needs to be deployed separately.  ➖Might not be viable in scenarios where access to production data is restricted. |
| Databricks Recommendation | Option-1 ( Deploy Code) |
| Decision |  |

| AD-02 | How shall developers/data scientists provision required databricks [components](#_4l1j9j6tzx4i) in the Azure Databricks *sandbox* workspace? |
| --- | --- |
| Category | Infrastructure |
| Description | In the context of SCB, there is often segregation in terms of the resources being provisioned, i.e. Databricks workspace and Azure storage accounts provisioning and access is managed by the cloud platform team, whereas the creation of Unity Catalog and other components is managed by the AI Platform team. This point raises the need for clarity around how developers should provision resources in the ***sandbox*** environment and then if needed deploy the required resources in the dev, staging and prod environments. |
| FRs/NFRs Addressed |  |
| Options | **Option-1**: Create an Azure Devops Pipeline using a terraform-based deployment pattern using the databricks-terraform-provider for provisioning of Databricks [components](#_4l1j9j6tzx4i) across all environments i.e. sandbox, dev, staging and prod. This process to be managed by the AI Platform team and to be updated on a request basis by developers.  **Option-2**: Allow BU/Workspace Admins, privileges to create required databricks components in the sandbox environment only. Developers can raise a request to the BU Admins for creation of resources (either manual/automated). Follow option-1 for deploying required resources in dev, staging and production. |
| Justification | **Pros/Cons of Option-1:**  ➕Components in sandbox are created using the same IaC templates as the rest of the environments (i.e. dev, staging prod), reducing configuration drift and manual overhead.  ➕Ensures repeatability, auditability and centralized provisioning of components.  ➖Slower turnaround for experimentation or quick changes in the sandbox due to dependency on platform team.  ➖AI Platform team becomes a bottleneck if request queues grow.  **Pros/Cons of Option-2:**  ➕Teams can iterate and test quickly in sandbox environments, accelerating development cycles.  ➕BU/Workspace Admins act as gatekeepers within their domains, distributing provisioning responsibilities.  ➕Enables rapid prototyping and innovation without platform team involvement.  ➕Reduces burden on AI Platform team for non-prod environments.  ➖Manual provisioning in sandbox environments may lack traceability unless rigorously logged..  ➖BU/Workspace Admins may inadvertently misconfigure components (e.g., UC permissions, external locations) without guardrails.  ➖BU Admins need adequate training and access controls to avoid mismanagement of sensitive components.  ➖Sandbox configuration effort may not translate easily to production deployment due to lack of standardized automation. |
| Databricks Recommendation | NA (Depends on SCB Requirements) |
| Decision |  |

# Design

## Overview

This section outlines the design related to creation of feature stores (feature tables), model training, deployment and monitoring. We also outline how to implement the CI/CD process for promoting models based on required criteria.

The content is divided in the following sections:

1. **Feature Management**: Outlines the creation of feature stores, feature table creation/updation, guiding principles, limitations and feature serving endpoints.
2. **Model Training and Deployment**: Outlines the process for creation of models, registering in MLFlow, scoring and deployment of the model.
3. **Model Serving**: Outlines the process for creating Model Serving endpoints.
4. **MLOPS Process**: Outlines the CI/CD process for the end to end lifecycle.

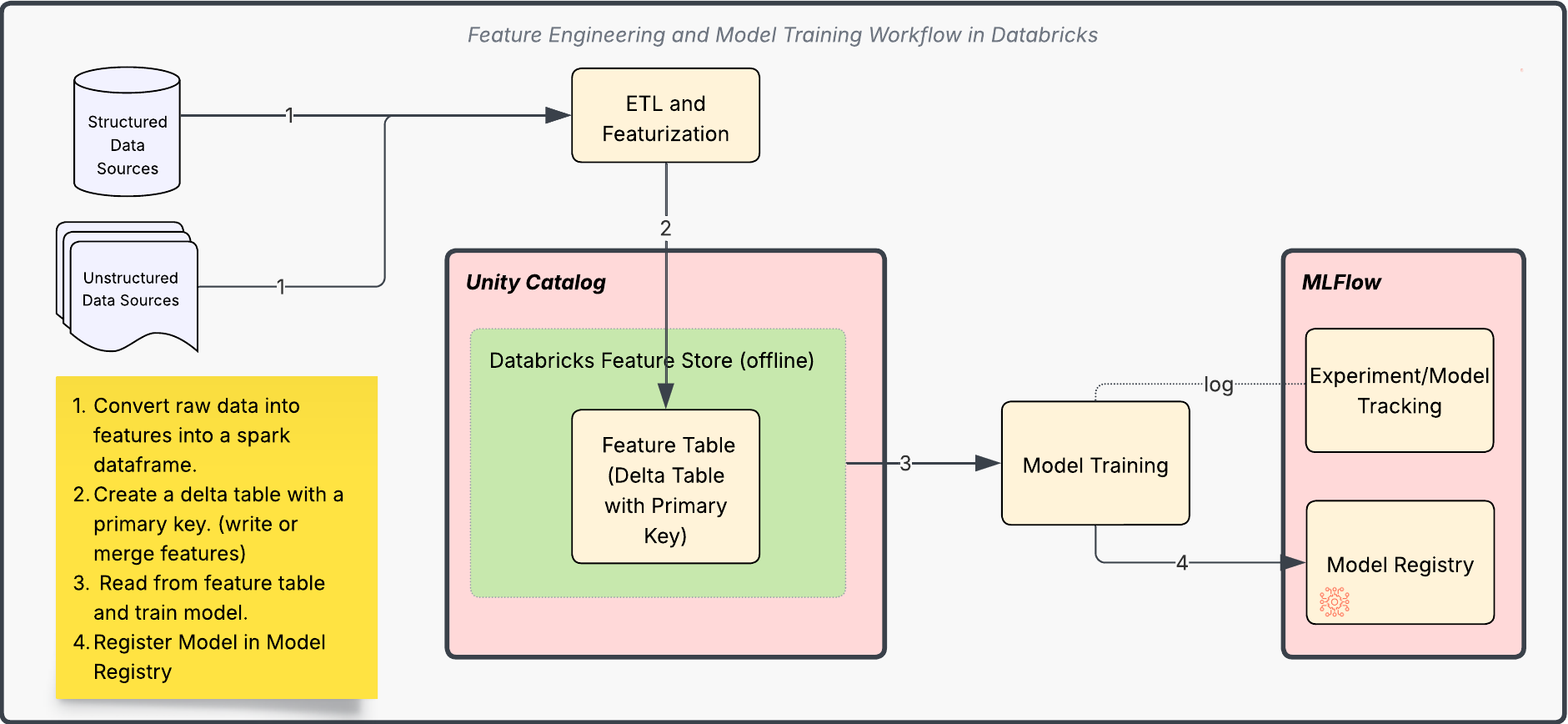
## Feature Management

This section outlines the process for creation and management of feature stores, tables etc.

### Glossary of Terms

| **Term** | **Definition** | **Links** |
| --- | --- | --- |
| [Feature Engineering](#_dex77cg6ivz4) | ML processes create models to predict a future outcome from *existing* data. Feature Engineering is the process of transforming/pre-processing the *existing raw data* into *features* before it can be used to build a model. |  |
| [Feature Store](#_wltq9wo2i3sk) | A centralized repository that enables data scientists to find and share features. Using a feature store also ensures that the code used to compute feature values is the same during model training and when the model is used for inference. |  |
| [Feature Tables](#_h2btzrvh6160) | Features are organized as feature tables. Each table must have a primary key, and is backed by a Delta table and additional metadata. Feature table metadata tracks the data sources from which a table was generated and the notebooks and jobs that created or wrote to the table. |  |
| [Feature Lookup](#_b1yarrqsfp6l) | Provides the functionality to lookup required features from a single/multiple feature tables. | [Doc reference](https://docs.databricks.com/aws/en/machine-learning/feature-store/concepts#featurelookup) |
| [Online Feature Store](#_m69u2wkes9c) | It is a high-performance, scalable solution for serving feature data to online applications and real-time machine learning models. Uses [Databricks Lakebase](#_bf9da4zgrpka) in the backend. | [Doc reference](https://docs.databricks.com/aws/en/machine-learning/feature-store/concepts#online-feature-store) |
| [Feature Serving endpoint](#_n343aa7jsqx) | These make feature data available to models or applications deployed outside of Azure Databricks, offering high availability and low latency with automatic scaling | [Doc reference](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/feature-function-serving) |
|  |  |  |

*The below diagram depicts the high level workflow of Feature Engineering and Model Training and Registration in Databricks.*



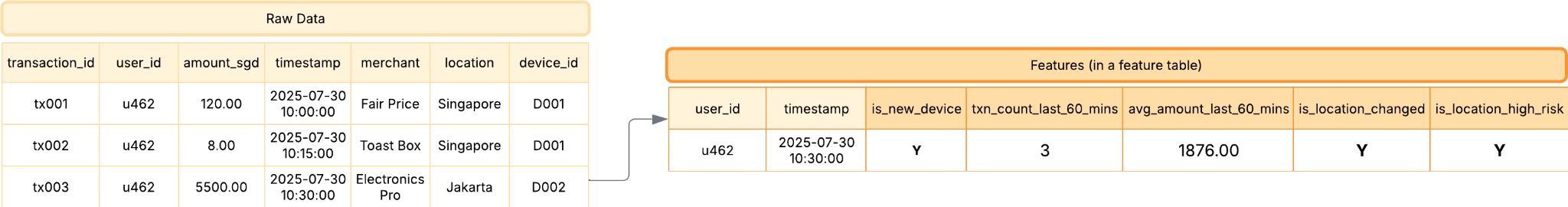
### Feature Engineering

Machine learning models don’t simply consume raw data. The *functions* of raw data are what’s meaningful for a machine learning problem. In a model that predicts fraudulent transactions, for example, one might find:

* Aggregations of raw data over time windows, like number of transactions in the past 24 hours
* Joined combinations of data sets, like user account details joined with transaction history
* Complex functions of transaction behavior, like deviation from typical spending pattern

The process of creating these values, i.e. *features* from raw data is ***feature engineering***.

*Below representation highlights the difference between raw data and features for customer transactions.*



### Feature Store

A feature store acts as a centralized repository where stakeholders and data scientists can find, share, and manage features.

The primary benefits of a feature store are outlined below:

* **Discovery**: *Features* may need to be shared and reused in different models. A feature store helps in discovery of features that have already been defined.
* **Lineage**: Feature producers often need to understand the downstream dependencies like what models and deployments depend on it, when recomputing features. Likewise feature consumers would like to know the upstream dependencies, i.e. who owns it, how it was computed, in order to reliably use it. A Feature store helps in surfacing the lineage of the features for all stakeholders.
* **Managing Feature Transformation logic**: Databricks feature store by design ensures that the code used for computing features remains consistent between model training and inference.

| **Note on managing feature transformation logic, i.e. why its important:**   * Feature engineering is essentially executing logic to transform raw data, but this happens in two possibly quite different contexts: training the model and applying it to new data (inference, or scoring). * Models may be trained and deployed in one environment, like Databricks, and called, from another type of environment entirely, like a Java web application using the model as a service. * Once the model has been built, reproducing the necessary input data and data transformations logic for inference may be diffcult because model training and production tech stacks could be entirely different and managed by different teams. * This problem of online/offline skew, or the difference between the inference and training environment, is somewhat unique to machine learning, and appears quickly once teams move to production. * Managing input data can have its own challenges. In a large organization, it can be difficult to guarantee that the source of data used for batch feature computation is the same upstream source used in inference at real time. When multiple teams manage feature computation and ML models in production, minor yet significant skew in upstream data at the input of a feature pipeline can be very hard to detect and fix. * Databricks Feature Store by design solves this data and compute skew problem. Instead of aiming to make arbitrary featurization logic portable and fast, which would be almost impossible in the general case, feature stores typically aim to make the features portable — that is, the data itself. |
| --- |

### Feature Tables

A feature table in Databricks is a Delta table used to store and manage machine learning features—i.e., the input variables used by models for training and inference. These tables are an integral part of the Databricks Feature Store, which helps with feature reuse, discovery, versioning, and lineage tracking. Some points for feature tables have been highlighted below:

| * *Feature tables* in Unity Catalog are *Delta Tables* which contain features for one or more entities. * Feature tables **must** have a PRIMARY KEY constraint. * Feature tables *may* also utilize TIMESERIES column for doing point-in-time lookups * Feature tables, like other data assets in Unity Catalog, are accessed using a three-level namespace: <catalog-name>.<schema-name>.<table-name> * Feature tables are managed and searched (lookup) using the FeatureEngineeringClient from the databricks.feature\_engineering sdk ([ref link](https://api-docs.databricks.com/python/feature-engineering/latest/feature_engineering.client.html)). Examples in the below sections highlight how this is used in an ML Workflow. * The [supported datatypes](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/#supported-data-types) for a feature table are: *IntegerType, FloatType, BooleanType, StringType, DoubleType, LongType, TimestampType, DateType, ShortType, ArrayType, BinaryType, DecimalType, MapType, and StructType* * Since ML Models often require training on historical data as it existed at specific points in time, it is beneficial to either *partition* the data by date or use [*liquid clustering*](https://learn.microsoft.com/en-gb/azure/databricks/delta/clustering) on the primary key and timestamp columns. |
| --- |

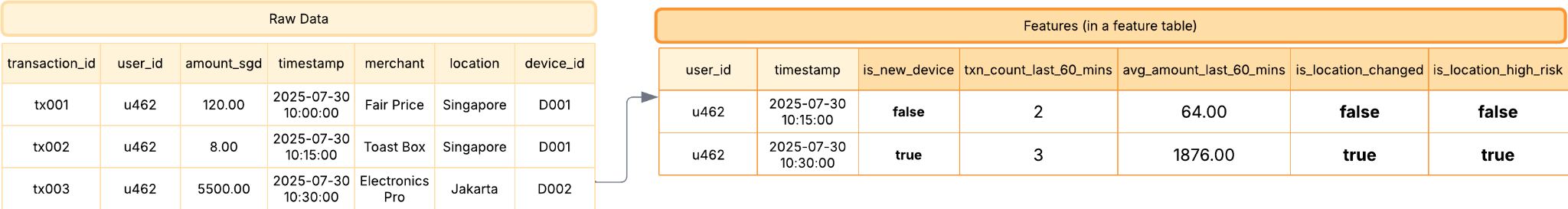
#### TimeSeries Feature Table

In a time series feature table, features are tracked and versioned over time, making it possible to recreate the exact state of data at any point in the past for training, inference, or monitoring in a temporal or sequential context.

They include a timestamp column that ensures that each row in the training dataset represents the latest known feature values as of the row's timestamp. They should be used whenever feature values change over time, for example with time series data, event-based data, or time-aggregated data.

As a consequence, they support point-in-time correct joins to avoid data leakage during training.

In the following example, we see that the customer transactions for a user in the feature table are clustered by time and they may differ as to when we query the feature table.



In the above example, to ensure the features are time-series aware, we would create a time-series feature table. More details on creating time series feature tables are in the `[Creating Feature Tables](#_xh0eqizbllir)` section.

A more detailed example may be found in the databricks documentation link [here](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/time-series).

#### Feature Table vs Normal Delta table

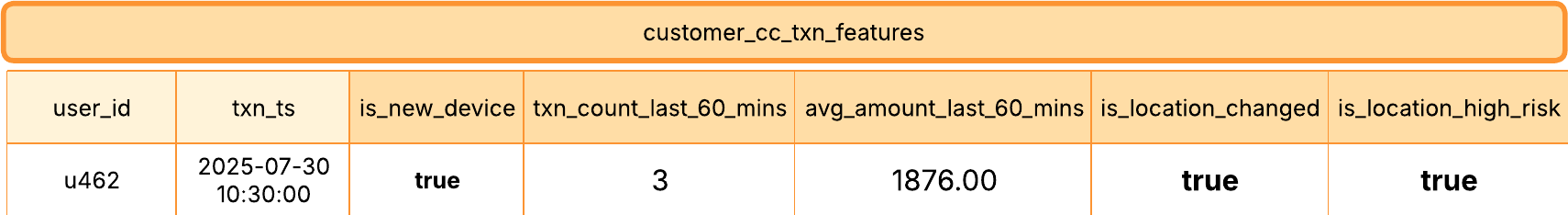
A feature table is a delta table with additional metadata w.r.t features to support model training and inference.

| ***Aspect*** | ***Feature Table*** | ***Normal Delta Table*** |
| --- | --- | --- |
| *Purpose* | Stores **ML features** with metadata to support model training and inference | Generic data storage for any structured data |
| *Metadata* | Has feature-specific metadata (e.g. description, feature lineage) | No built-in metadata for ML features |
| *Lineage & Versioning* | Tracks how features were created and used in ML workflows | No ML lineage; only supports Delta versioning |
| *Feature Lookup* | Supports feature lookup in training sets and real-time inference | Not directly usable in feature lookup APIs |
| *Integration* | Integrated with Databricks ML tools, Feature Store APIs, and Model Registry | Generic; needs custom logic for ML integration |
| *Discoverability* | Features are searchable in Feature Store UI/API | No discoverability beyond Hive metastore/catalog |

#### Creating feature tables

This section outlines the pre-requisites and creation of feature tables highlighting various methodologies and scenarios.

The sample feature table to be used in relevant examples is highlighted in the below figure.



##### Pre-requisites

* Unity Catalog Metastore: The Azure Databricks Workspace must be enabled for Unity Catalog.
* Databricks Runtime:
  + Feature Engineering in Unity Catalog requires Databricks Runtime 13.2 or above.
  + Feature Tables with the TIMESERIES keyword requires Databricks Runtime 13.3 LTS or above.
* Feature tables require a PRIMARY KEY constraint.

##### Feature Table using Databricks SQL (**Recommended Option**)

The following example highlights creating a delta table with a primary key constraint, with the purpose of it being used as a feature table.



CREATE TABLE cib\_cc\_dev.fraud\_detection.customer\_cc\_txn\_features (

user\_id string NOT NULL,

txn\_ts timestamp NOT NULL

is\_new\_device boolean,

txn\_count\_last\_60\_minutes int,

avg\_amount\_last\_60\_minutes double,

is\_location\_changed boolean,

is\_location\_high\_risk boolean

CONSTRAINT customer\_txn\_features\_pk PRIMARY KEY (user\_id, txn\_ts TIMESERIES)

)

COMMENT 'Cusotmer Credit Card Transaction Features'

-- [OPTIONAL] -- CLUSTER BY AUTO or PARTITION BY DATE if required

;



| **Note**:   * To create a *time series* feature table, add a time column as a primary key column and specify the TIMESERIES keyword. This is optional and is only required for the point-in-time lookup scenario. * To create feature tables which do not require a time-series, omit the TIMESERIES keyword. * You may also utilize *liquid clustering* to improve query times for large tables by using the CLUSTER BY keyword while creating the tables. You may specify the CLUSTER BY AUTO when predictive is enabled at the Databricks Account Level. Specific columns may also be specified, i.e. cluster by col1, col2 etc. for more details on liquid clustering, refer to the databricks documentation [here](https://learn.microsoft.com/en-gb/azure/databricks/delta/clustering). |
| --- |

###### Updating Created Feature Tables

For the feature tables created using SQL, this section highlights how features will be updated using the FeatureEngineeringClient using the databricks.feature\_engineering sdk.

from databricks.feature\_engineering import FeatureEngineeringClient

from pyspark.sql import DataFrame

#invoke the feature engineering client

fe = FeatureEngineeringClient()

df = read\_raw\_data() # define a function to read the raw data

# Compute features from the raw data and return a feature dataframe

def compute\_customer\_features(df:DataFrame)->DataFrame:

''' Feature computation code returns a DataFrame with user\_id and txn\_ts as primary key '''

# compute and return the feature df

pass

customer\_features = compute\_customer\_features(df)

# merge the features to the table

fe.write\_table(

name = 'cib\_cc\_dev.fraud\_detection.customer\_cc\_txn\_features'

df = customer\_features

mode = 'merge'

)



##### Creating Feature table using Python SDK

The following example shows how to create a feature table by using the FeatureEngineeringClient from the databricks.feature\_engineering sdk.

from databricks.feature\_engineering import FeatureEngineeringClient

from pyspark.sql import DataFrame

fe = FeatureEngineeringClient()

# Prepare feature DataFrame

def compute\_customer\_features(df:DataFrame):

''' Feature computation code returns a DataFrame with user\_id as primary key '''

pass

customer\_features\_df = compute\_customer\_features(df)

# Create feature table with `customer\_id` as the primary key.

# Take schema from DataFrame output by compute\_customer\_features

customer\_feature\_table = fe.create\_table(

name='cib\_cc\_dev.fraud\_detection.customer\_cc\_txn\_features',

primary\_keys=['user\_id', 'txn\_ts'],

timeseries\_columns = 'txn\_ts'

schema=customer\_features\_df.schema,

description='Customer features'

)

##### 

##### Use an existing Delta table in Unity Catalog as a feature table

In Databricks, *any delta table with a primary key constraint* can be used as a feature table.

In such a scenario, where the delta table is chosen to function as a feature table (i.e. it has the relevant features required for training a model) and if the existing delta table does not have a primary key constraint, it can be created as shown in the below sample code.

Please note that only the TABLE OWNER can set the primary key constraint for a table.

-- 1. Set primary key columns to NOT NULL. For each primary key column, run:

ALTER TABLE <full\_table\_name> ALTER COLUMN <pk\_col\_name> SET NOT NULL

-- 2. Alter the table to add the primary key constraint:

-- for non-time series feature table

ALTER TABLE <full\_table\_name> ADD CONSTRAINT <pk\_name> PRIMARY KEY(pk\_col1, pk\_col2, ...)

-- for time series feature table

ALTER TABLE <full\_table\_name> ADD CONSTRAINT <pk\_name> PRIMARY KEY(pk\_col1 TIMESERIES, pk\_col2, ...)

##### Use a streaming table or materialized view created by Lakeflow Declarative Pipelines as a feature table ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/uc/feature-tables-uc#use-a-streaming-table-or-materialized-view-created-by-lakeflow-declarative-pipelines-as-a-feature-table))

It is possible to utilize an existing streaming table or materialized view created a Lakeflow Declarative pipeline as a Feature Table.

In such a scenario, where a streaming table or materialized view is chosen to function as a feature table, i.e. it has the relevant features for training a model, and if it does not have a primary key defined, we need to modify the object to have a valid primary key constraint.

To set primary keys for an existing streaming table or materialized view, update the schema of the streaming table or materialized view in the notebook that manages the object. Then, [refresh the table](https://learn.microsoft.com/en-gb/azure/databricks/dlt/updates#refresh-selection) to update the Unity Catalog object.

The following is the syntax to add a primary key to a materialized view:

CREATE OR REFRESH MATERIALIZED VIEW existing\_live\_table(

id int NOT NULL PRIMARY KEY,

...

) AS SELECT ...



| **Note**: For more details on Lakeflow Declarative Pipelines (formerly DLT), please refer to the databricks documentation [here](https://learn.microsoft.com/en-gb/azure/databricks/dlt/concepts#what-is-lakeflow-declarative-pipelines). |
| --- |

##### Use an existing view in Unity Catalog as a feature table ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/uc/feature-tables-uc#use-an-existing-view-in-unity-catalog-as-a-feature-table))

It is possible to use a view in unity catalog as a feature table. In the scenario, where a view is chosen to be used as a feature table, then only a *simple* SELECT view in Unity Catalog can be used as a feature table with databricks-feature-engineering version 0.7.0 or above (built into Databricks Runtime 16.0 ML).

| **Note**:   * A simple SELECT view in this context means that it is created from a single Delta table, and its primary keys are selected without JOIN, GROUP BY, or DISTINCT clauses. Acceptable keywords include SELECT, FROM, WHERE, ORDER BY, LIMIT, and OFFSET. * Also note that Feature tables backed by views do not appear in the Features UI and cannot be published to online stores |
| --- |

An example for creating a view to be used as a feature tables is outlined below.

CREATE OR REPLACE VIEW ml.recommender\_system.content\_recommendation\_subset AS

SELECT

user\_id,

content\_id,

user\_age,

user\_gender,

content\_genre,

content\_release\_year,

user\_content\_watch\_duration,

user\_content\_like\_dislike\_ratio

FROM

ml.recommender\_system.content\_recommendations\_features

WHERE

user\_age BETWEEN 18 AND 35

AND content\_genre IN ('Drama', 'Comedy', 'Action')

AND content\_release\_year >= 2010

AND user\_content\_watch\_duration > 60;

#### Limitations

This section outlines some of the limitations while creating feature tables and working with feature tables during model training or inference.

##### Feature Table

| **General** | * **Metadata Immutability**: The primary key, partition key, name, or data type of an existing feature in a feature table **should not be updated**, as this can break downstream pipelines ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/uc/feature-tables-uc#update-a-feature-table-in-unity-catalog)) |
| --- | --- |
| **Time Series Feature Table** | * **Timestamp Key and Partitions:** A time series feature table must have **one timestamp key and cannot have any partition columns.** * **Timestamp Key Data Types:** The timestamp key column must be of *TimestampType* or *DateType*. ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/time-series#workspacefeaturestoreclientv0133andbelow)) * **Update Requirements:** When writing features to the time series feature tables, your DataFrame must supply values for all features of the feature table. This constraint reduces the sparsity of feature values across timestamps in the time series feature table. ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/time-series#update-a-time-series-feature-table)) * **Online Store Point-in-Time Lookup:** When time series features are published to an online store, the online store supports primary key lookup but does not support point-in-time lookup ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/time-series)) |
| **Feature Tables Backed by Views** | * **UI Visibility:** Feature tables backed by simple SELECT views do not appear in the Features UI * **Online Store Publishing:** Feature tables backed by views can be used for offline model training and evaluation, but cannot be published to online stores or served ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/uc/feature-tables-uc#use-an-existing-view-in-unity-catalog-as-a-feature-table)) |
| **Views as Feature Tables** | Feature tables backed by views can be used for offline model training and evaluation, but cannot be published to online stores or served ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/uc/feature-tables-uc#use-an-existing-view-in-unity-catalog-as-a-feature-table)) |

##### Model Training and Inference

* **Table and Function Count:** A model can use at most **50 tables and 100 functions for training** ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/troubleshooting-and-limitations#limitations))
* A maximum of 100 [on-demand features](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/on-demand-features) can be used in a model.
* **Lakeflow Declarative Pipelines Compute:** Databricks Runtime ML clusters **are not supported when using Lakeflow Declarative Pipelines as feature tables** for training. Instead, a standard access mode compute resource is required, and the client must be manually installed ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/troubleshooting-and-limitations))

### Features from Unstructured Data

Feature stores adopt a tabular paradigm, where features are organized into tables with typed columns and a primary key. In this context, unstructured (or simply differently structured) data is “raw” data in the feature store paradigm, like the individual transactions that might feed that customer churn model.

Models processing text or images do not typically handle them as raw bytes. A crucial step in numerous deep learning models involves learning an "embedding" of such data. An embedding constitutes a vector, or numerical list, that effectively summarizes the input. For instance, it can condense extensive, intricate text documents or video inputs into a compact vector, which is more conducive to learning tasks and prepared for model input. However, computing an embedding can be computationally expensive. Consequently, embeddings of unstructured data present strong candidates for features. For example, an organization maintaining user forum post text might embed these posts and store the embeddings as features, serving as a valuable summary. Numerous machine learning tasks requiring insights into forum posts could then repurpose this embedding.

Some sources for unstructured data are highlighted below.

* Call and chat transcripts, email text, transaction notes
* Social media posts
* Scanned documents (PDFs, images), free-text fields
* Internal memos, reports
* Web logs etc
* Other sources for images, video, audio etc

Such unstructured data follows the following process in feature engineering.

* Read unstructured data (images as binary, text as string) into a Spark Dataframe
* Process and Convert the unstructured data as embeddings.
* Store the embeddings into a feature table

These raw embeddings can then be used for training sentiment analysis models or generating sentiment using pre-built models.

An example highlighting the above outlined process is shown below.

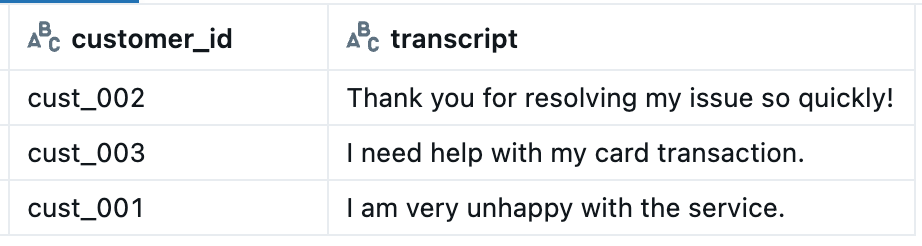
In this simplified example, we will see how to read unstructured data from *customer chat messages,* generate *sentence embeddings* and store them into the Azure Databricks feature store. These embeddings may then be looked up and used to create models for sentiment analysis and/or customer churn prediction as necessary.

This example assumes that we have raw text data ingested and stored into a bronze delta table.

This example primarily demonstrates how we can handle raw text and convert them into embeddings. This concept can further be applied to any form of data, i.e. images, audio , video etc, wherein the process to convert such data into embeddings and storing them into a feature table would be somewhat similar, only the format which is read i.e. binary type would differ.

**Data ingestion (read unstructured data)**

df = spark.read.table(f"cib-cc-dev.bronze.customer\_chat\_transcripts")



**Create feature table**

Create a Feature Table in Unity Catalog (with Primary Key Constraints).

-- Create Feature Table

CREATE TABLE IF NOT EXISTS ${schema}.customer\_chat\_embeddings(

customer\_id STRING NOT NULL,

transcript\_embedding ARRAY<DOUBLE> NOT NULL,

CONSTRAINT customer\_chat\_embeddings\_pk PRIMARY KEY (customer\_id)

)

CLUSTER BY AUTO

COMMENT "TF-IDF embeddings of transcripts for sentiment models";



**Create embedding from raw transcripts** (sample code for reference only)

from pyspark.ml.feature import (

Tokenizer, StopWordsRemover, HashingTF, IDF

)

# Tokenize

tokenizer = Tokenizer(inputCol="transcript", outputCol="words")

df\_words = tokenizer.transform(df)

# Remove Stop Words

remover = StopWordsRemover(inputCol="words", outputCol="filtered\_words")

df\_filtered = remover.transform(df\_words)

#TF : Converts words into a fixed-length vector using hashing

hashing\_tf = HashingTF(inputCol="filtered\_words", outputCol="raw\_features", numFeatures=100)

df\_tf = hashing\_tf.transform(df\_filtered)

# IDF : Inverse Document Frequency

# Learns the inverse document frequency (how rare a word is across all documents)

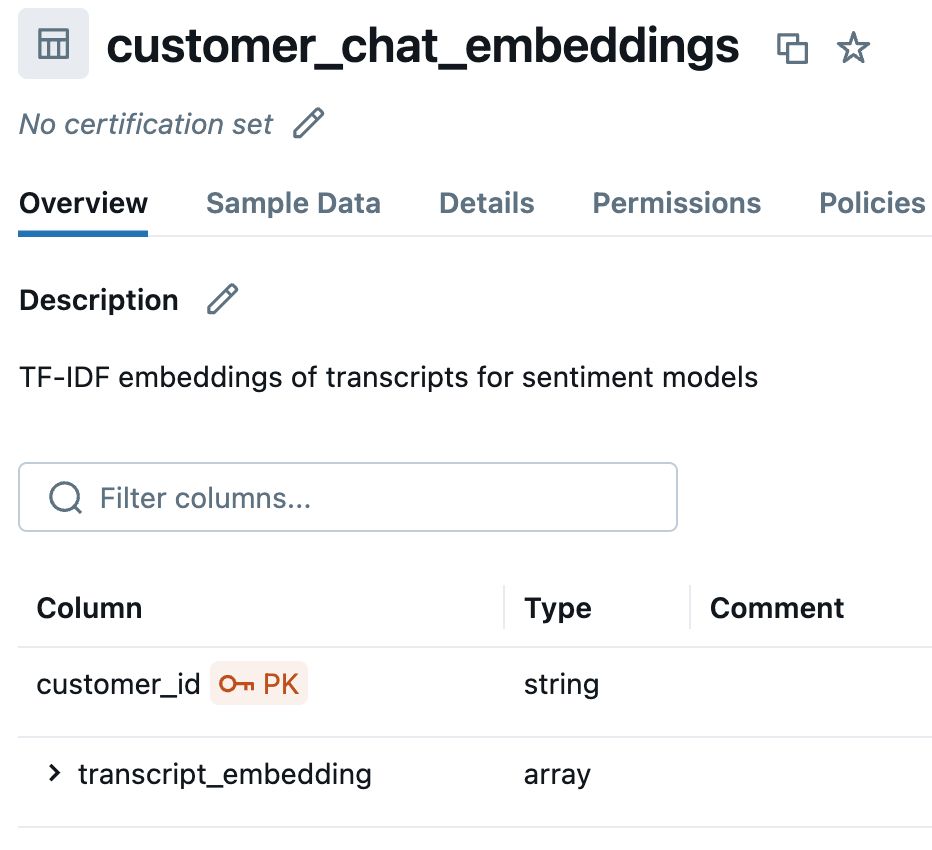
# Multiplies TF values by IDF scores to downweight common words

idf = IDF(inputCol="raw\_features", outputCol="transcript\_embedding")

idf\_model = idf.fit(df\_tf)

df\_embeddings = idf\_model.transform(df\_tf)





**Store feature to feature tables**

Write features to the feature tables in Unity Catalog via [FeatureEngineeringClient](https://api-docs.databricks.com/python/feature-engineering/latest/feature_engineering.client.html)’s write\_table API

# Assume that `df\_embeddings` is the resulting feature DataFrame generated from the feature engineering process from the example above.

display(df\_embeddings)



To store vectors, tensors, or embeddings in a feature table, use either *ArrayType* or *MapType*, depending on whether the data is dense or sparse.

For demonstration purposes and to keep the code clear, the following example shows how to convert a vector to an *ArrayType* and store it in a feature table.

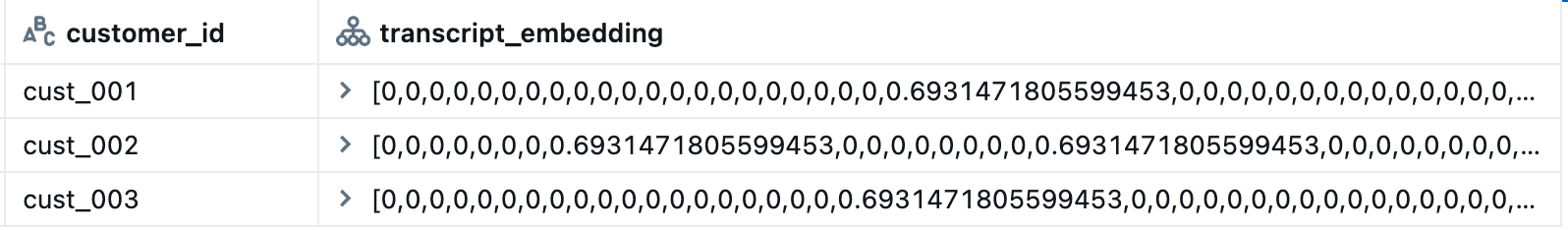
For converting to *MapType*, the code snippet is included in the note below.

Use the built-in function [vector\_to\_array](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.functions.vector_to_array.html) to convert a dense vector into an array.

from pyspark.ml.functions import vector\_to\_array

feature\_df = df\_embeddings.withColumn("transcript\_embedding", vector\_to\_array("transcript\_embedding")).select(

"customer\_id","transcript\_embedding")



Use FeatureEngineeringClient.write\_table to store features.

from databricks.feature\_engineering import FeatureEngineeringClient

# Create an instance of the Feature Engineering client.

fe = FeatureEngineeringClient()

fe.write\_table(

name=f"{catalog}.{schema}.customer\_chat\_embeddings",

df=feature\_df

)



Note.

* The default behavior of writing data to the same feature is merge, i.e. upsert.
* If you want to overwrite the feature table, run  *DELETE FROM <table name>;* to delete all rows, or drop and recreate the table before calling the *write\_table* method.
* For converting vectors to *MapType*,we can use a user-defined function (UDF) to convert a sparse vector into a key-value map. (Pandas UDF doesn’t support vector as an input)

from pyspark.ml.linalg import SparseVector

from pyspark.sql.functions import udf

from pyspark.sql.types import MapType, IntegerType, FloatType

def sparse\_to\_map(sparse\_vec: SparseVector):

return {int(i): float(v) for i, v in zip(sparse\_vec.indices, sparse\_vec.values)}

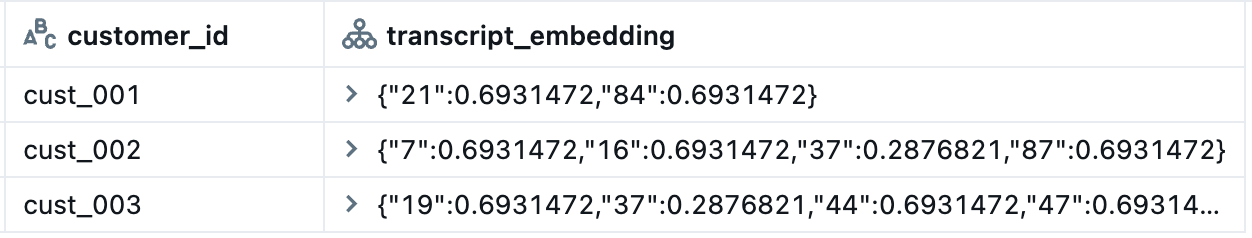
sparse\_to\_map\_udf = udf(sparse\_to\_map, MapType(IntegerType(), FloatType()))

feature\_map\_df = df\_embeddings.select(

"customer\_id",

sparse\_to\_map\_udf(col("transcript\_embedding")).alias("transcript\_embedding")

)



### 

### Online Feature Store

Databricks online feature store is used for serving feature data to online applications and real time machine learning models. Internally it uses Databricks Lakebase (OLTP) for providing low latency access to feature data at scale. *Please note that as of writing of this document, this feature is in Beta (*[*link*](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/online-feature-store)*).*

This is achieved by provisioning an online feature store and subsequently publishing the offline feature table to it as an online feature table.

In the subsequent sections, we briefly describe the following:

* What is *Databricks Lakebase*
* How to *provision an online feature store*
* How to *publish offline features to the online feature store*
* How to *use the online feature store for applications*

#### Databricks Lakebase

Databricks Lakebase is a fully managed Postgres OLTP database engine, integrated into the Databricks Platform. It allows users to efficiently handle high volumes of real-time transactional data and integrate OLTP workloads with the Databricks Lakehouse. A few important points have been outlined below:

* Allows for creating and managing OLTP databases stored in Databricks managed storage.
* Leverages existing Databricks infrastructure to deploy instances with decoupled compute and storage, managed changed data capture with Delta Lake tables and support for multi cloud deployments
* Uses Postgres v16 and includes PostGIS, pgvector extensions
* Integrated support for AI and ML use cases like feature serving, retrieval-augmented-generation (RAG) and others.
* Can be integrated with Unity Catalog for authentication and governance
* In the machine learning context, it is primarily used as part of the ***online feature store*** integration, i.e. *to serve feature data at low-latency to models*

For more details on Databricks Lakebase database instances and capacity and other features, please refer to the documentation [here](https://learn.microsoft.com/en-gb/azure/databricks/oltp/instance).

#### Provisioning an Online Feature Store

The online feature store can be provisioned using either of the two approaches:

* Using the databricks\_online\_store resource ([link](https://registry.terraform.io/providers/databricks/databricks/latest/docs/resources/online_store)) from the databricks terraform provider
* Using the FeatureEngineeringClient.create\_online\_store from the *databricks.feature\_engineering* sdk ([link](https://api-docs.databricks.com/python/feature-engineering/latest/feature_engineering.client.html))

Below we see an example of creating the online store using the *FeatureEngineeringClient.* Please note the same functionality can be achieved using the terraform resource.

from databricks.feature\_engineering import FeatureEngineeringClient

# initialize the client

fe = FeatureEngineeringClient()

# create the online feature store

fe.create\_online\_store(

name = "cib-cc-online-features" # allows alphanumeric chars and hyphens only

capacity = "CU\_1" # valid options: CU\_1, CU\_2, CU\_4, CU\_8

read\_replica\_count = None # optional parameter, max\_allowed = 2

)



| **Capacity**: While creating the online store, capacity refers to the different performance tiers of the underlying Lakebase instances. Each capacity unit allocates 16GB of RAM along with all associated CPU and local SSD resources. Scaling up increases these resources linearly, i.e. CU\_1 = 16GB memory, CU\_2 = 32 GB memory and CU\_4 = 64 GB memory  **Read Replica**: This is an option to distribute the read traffic automatically across the read replicas to reduce latency and improve performance for high-concurrency workloads. A maximum of 2 read replicas can be created for an online feature store. |
| --- |

##### Managing online feature store

As mentioned earlier the online feature store can be created, updated and destroyed using the terraform resource. We can also use the *FeatureEngineeringClient* to update or delete an online feature store, as shown below.

from databricks.feature\_engineering import FeatureEngineeringClient

# initialize the client

fe = FeatureEngineeringClient()

# get information about an existing online store

online\_store = fe.get\_online\_store(name="cib-cc-online-features")

# returns an object of type

# DatabricksOnlineStore(

# name='cib-cc-online-features',

# capacity='CU\_1',

# read\_replica\_count=0

# creator='<name of creator>'

# creation\_time='<creation timestamp in UTC format>'

# state='AVAILABLE' # or STARTING

# )

# update the online store

fe.update\_online\_store(

name='cib-cc-online-features',

capacity = 'CU\_2' # increasing the capacity

)

# destroy the online store

fe.delete\_online\_store('cib-cc-online-features')



#### Publishing to an online feature store

The previous step creates the online feature store. Once it is available, we can then publish an offline feature table to the online store using the sample shown below.

Please note that to publish to an online store, the feature table must have change data feed enabled. This will ensure that the changes are incrementally merged into the online store.

from databricks.feature\_engineering import FeatureEngineeringClient

# initialize the client

fe = FeatureEngineeringClient()

# get the online store

online\_store = fe.get\_online\_store(name="cib-cc-online-features")

# enable change data feed on the offline delta feature table

spark.sql("""

ALTER TABLE cib\_cc\_dev.fraud\_detection.cust\_cc\_txn\_features

SET TBLPROPERTIES (delta.enableChangeDataFeed = true)

""")

# publish the feature table to the online store

# source\_table\_name : denotes the offline delta feature table

# online\_table\_name : denotes the online feature table name

fe.publish\_table(

online\_store = online\_store,

source\_table\_name = 'cib\_cc\_dev.fraud\_detection.cust\_cc\_txn\_features'

online\_table\_name = 'cib\_cc\_dev.fraud\_detection.online\_cc\_txn\_features',

features = ['is\_new\_device','is\_location\_changed'], # optional

filter\_condition = "txn\_dt>'2024-12-31'" # optional

)



| **Note**: *Change Data Feed (*[*link*](https://learn.microsoft.com/en-gb/azure/databricks/delta/delta-change-data-feed#enable)*)* must be enabled on a delta feature table before it can be published to an online store. Any user with MODIFY privilege on the table can alter and set the table properties. Alternatively, if this table has been created using terraform, the TBLPROPERTIES can be set as part of the terraform sql\_tables yaml config, using the *properties* tag like so:  sql\_tables:  - name: my\_delta\_table  catalog\_name: my\_catalog  schema\_name: my\_schema  table\_type: MANAGED  comment: Example table with TBLPROPERTIES  properties:  delta.enableChangeDataFeed: "true"  columns:  - name: id  type: string  nullable: false  - name: features  type: array<double>   |
| --- |

#### Using the online feature store

Once the online table is published, it can be seen in the Unity Catalog UI under the appropriate catalog and schema. This table can also be queries using the SQL Editor to run PostgreSQL queries on the online table ([link](https://learn.microsoft.com/en-gb/azure/databricks/oltp/query/sql-editor)).

The online features can also be used in the following use-cases:

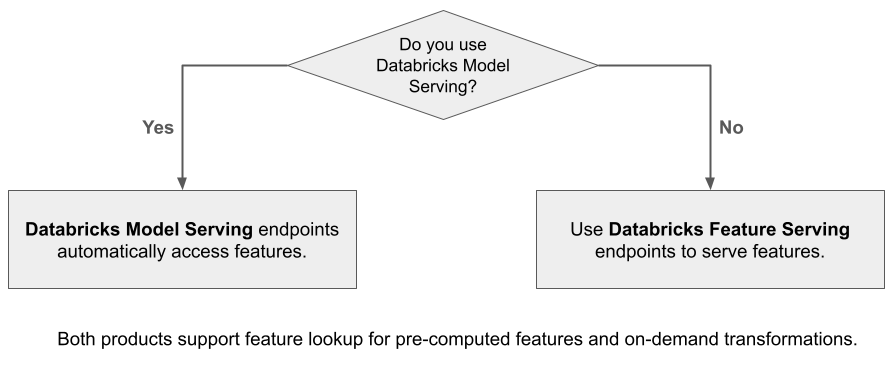
* Feature Serving Endpoints: Serve features to real-time applications and services
* Model Serving with automatic Feature lookup: Models that are trained using features from Databricks automatically track lineage to the features they were trained on. When deployed as endpoints, these models use Unity Catalog to find appropriate features in online stores.

##### Feature Serving Endpoints

Databricks feature serving endpoints are used to make data in the Databricks platform available to models or applications deployed outside of Databricks.

Feature Serving Endpoints are managed by Databricks, they automatically scale to adjust to the volume of serving requests and are deployed in a secure network boundary and use dedicated compute that terminates when the endpoint is deleted or scaled to zero.

One thing to note however is that if you plan to use Databricks Model Serving, then you don’t need to use Feature Serving Endpoints as represented in the figure below.



For the latest library and compute requirements to deploy and use Feature Serving Endpoints, please refer to the Databricks documentation [here](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/feature-function-serving#requirement).

The process of creating a feature serving endpoint is outlined below:

1. Publish feature tables to an online feature store: Covered in [Publishing to an online feature store](#_1y46ry7nl4bs) section
2. Create a FeatureSpec
3. Create a FeatureServingEndpoint

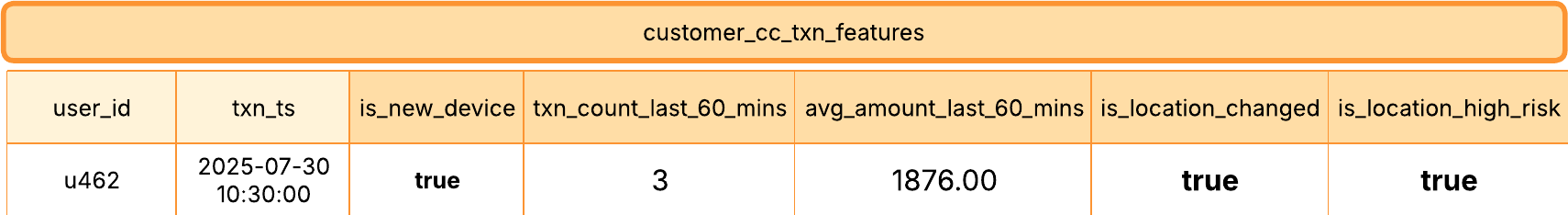
###### Create a FeatureSpec

A FeatureSpec, i.e. feature specification is a user-defined set of features and functions that define the functionality of the endpoint in terms of the features being served. They essentially define which features would be served using the endpoint.

In the following example, we take a look at how to create a FeatureSpec using the FeatureEngineeringClient.create\_feature\_spec function from the databricks.feature\_engineering sdk ([link](https://api-docs.databricks.com/python/feature-engineering/latest/feature_engineering.client.html)).

Please note that, as of writing of this document, this is the only supported way of creating a feature specification, i.e. using the databricks-feature-engineering library.

The feature table used in the following example is shown below.



from databricks.feature\_engineering import (

FeatureFunction,

FeatureLookup,

FeatureEngineeringClient,

)

# initialize the client

fe = FeatureEngineeringClient()

features = [

# Lookup column `is\_new\_device`, `is\_location\_changed` and `is\_location\_high\_risk` from a online feature table in UC by the input `user\_id`.

FeatureLookup(

table\_name="'cib\_cc\_dev.fraud\_detection.online\_cc\_txn\_features'",

lookup\_key="user\_id",

timestamp\_lookup\_key='txn\_ts'

feature\_names=["is\_new\_device","is\_location\_changed"] # optional, if None, returns all features,

# optionally specify default values, may not be valid for all scenarios

default\_values = {

"is\_new\_device" : "N",

"is\_location\_changed" : "N"

),

]

# Create a `FeatureSpec` with the features defined above.

# The `FeatureSpec` can be accessed in Unity Catalog as a function.

fe.create\_feature\_spec(

name="cib\_cc\_dev.ml\_assets.cc\_txn\_feature\_spec",

features=features,

)



###### Create a FeatureServingEndpoint

After the FeatureSpec has been created, the next step is to serve it using and endpoint.

In the following example, we see how to create a feature serving endpoint using the FeatureEngineeringClient.create\_feature\_serving\_endpoint from the databricks.feature\_engineering sdk ([link](https://api-docs.databricks.com/python/feature-engineering/latest/feature_engineering.client.html)).

Please note that the FeatureServingEndpoint once created can be viewed in the Databricks UI under AI/ML->Serving

from databricks.feature\_engineering import FeatureEngineeringClient

from databricks.feature\_engineering.entities.feature\_serving\_endpoint import (

ServedEntity,

EndpointCoreConfig,

)

# initialize the client

fe = FeatureEngineeringClient()

# refer the feature\_spec (created in the previous step)

my\_feature\_spec\_name = 'cib\_cc\_dev.ml\_assets.cc\_txn\_feature\_spec'

# create the feature serving endpoint

fe.create\_feature\_serving\_endpoint(

name = "cib-cc-txn-feature-endpoint'

config = EndpointCoreConfig(

served\_entities = ServedEntity(

feature\_spec\_name=my\_feature\_spec\_name,

workload\_size = "Small", #allowed\_vals:Small,Medium, Large

scale\_to\_zero\_enabled=True

)

)

)

# get the feature serving endpoint

endpoint = fe.get\_feature\_serving\_endpoint(name='cib-cc-txn-feature-endpoint')

# delete the feature serving endpoint

fe.delete\_feature\_serving\_endpoint(name='cib-cc-txn-feature-endpoint')

###### Querying a Feature Serving Endpoint

There are multiple ways to query a feature serving endpoint, i.e. using a Rest API, MLFlow Deployments SDK and via the Databricks->AI/ML->Serving UI. ([link](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/feature-store/feature-function-serving#query-an-endpoint))

Below we see an example of querying a feature serving endpoint using the predict API from the MLFlow Deployments SDK ([link](https://mlflow.org/docs/latest/api_reference/python_api/mlflow.deployments.html#mlflow.deployments.DatabricksDeploymentClient.predict)).

Please note, for authentication to the managed Databricks MLFlow, please specify the following parameters:

export DATABRICKS\_TOKEN=<databricks-personal-access-token>

export DATABRICKS\_HOST=https://<workspace-name>.cloud.databricks.com

export MLFLOW\_TRACKING\_URI=databricks

export MLFLOW\_REGISTRY\_URI=databricks-uc

In scenarios where Personal Access Token is not allowed, please use any of the supported authentication mechanisms outlined in the documentation reference [here](https://learn.microsoft.com/en-gb/azure/databricks/dev-tools/auth/#auth-types).

For installing MLFlow outside a databricks env, refer to the link [here](https://pypi.org/project/mlflow/).

import mlflow.deployments

client = mlflow.deployments.get\_deploy\_client("databricks")

response = client.predict(

endpoint = 'cib-cc-txn-feature-endpoint',

inputs={

"dataframe\_records": [

{"user\_id":"u2056"},

{"user\_id":"u1001"},

]

})



## Model Training and Deployment

### Overview

This section outlines the process of model training and deployment to unity catalog in Azure Databricks while using tracking, logging, monitoring and registration in Databricks managed MLFlow and deploying the model version into Unity Catalog.

For more information on MLFlow features, refer to the Azure Databricks documentation [here](https://learn.microsoft.com/en-gb/azure/databricks/mlflow/).

| **Note**: It is recommended to use the FeatureEngineeringClient python api ([link](https://docs.databricks.com/aws/en/machine-learning/feature-store/python-api)) for scenarios like *feature lookup*, *creating training dataset* and to log the models into MLFlow and Unity Catalog, to ensure the model retains references to the source features. This is beneficial, as during inference, the model can optimally retrieve the feature values automatically, given that the caller provides the primary key. |
| --- |

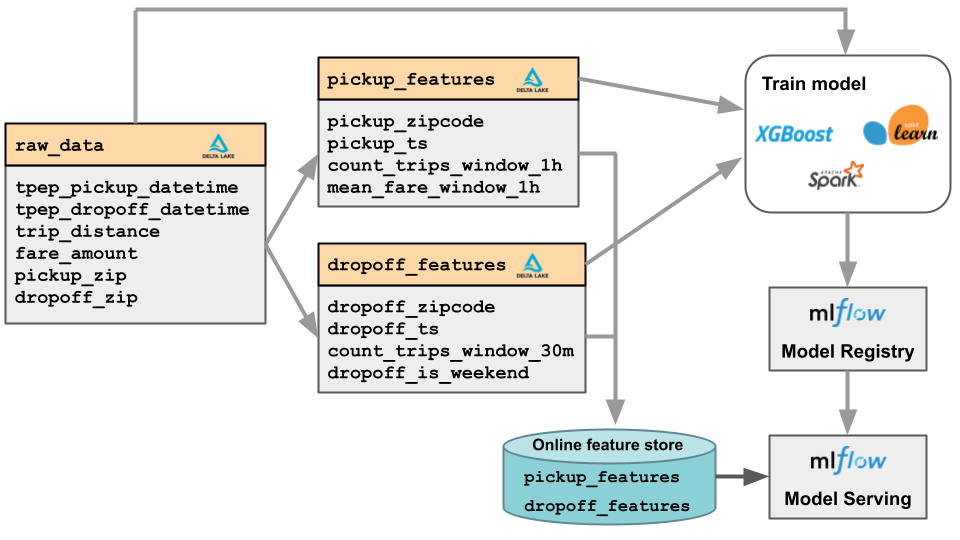
In this section, we will be using the NYC Taxi Dataset which can be found under /databricks\_datasets in a databricks workspace, as shown below.

raw\_data = spark.read.format("delta").load("/databricks-datasets/nyctaxi-with-zipcodes/subsampled")

display(raw\_data)

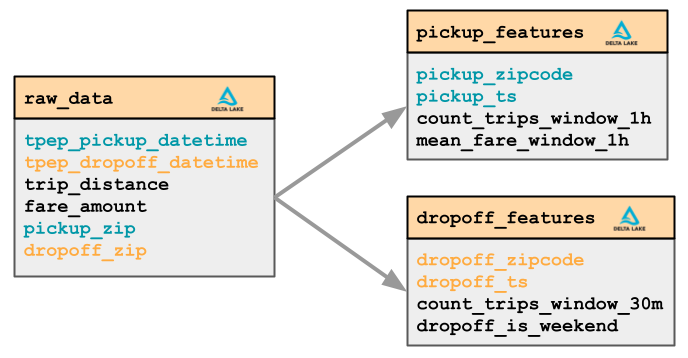


Using this sample, we will create features, create the training dataset, train the model and log the model into MLFlow and Unity Catalog.



### Create Features (As covered in Feature Management section)

As noted in the feature engineering section, we will create a delta table with a primary key to create two feature tables from the raw\_data, namely, *pickup\_features* and *dropoff\_features*.



The sample code to create the feature tables has been shown below.

CREATE TABLE IF NOT EXISTS taxi\_example.trip\_pickup\_time\_series\_features(

zip INT NOT NULL,

ts TIMESTAMP NOT NULL,

mean\_fare\_window\_1h\_pickup\_zip FLOAT,

count\_trips\_window\_1h\_pickup\_zip INT,

CONSTRAINT trip\_pickup\_time\_series\_features\_pk PRIMARY KEY (zip, ts TIMESERIES)

)

CLUSTER BY AUTO

COMMENT "Taxi Fares. Pickup Time Series Features";

CREATE TABLE IF NOT EXISTS taxi\_example.trip\_dropoff\_time\_series\_features(

zip INT NOT NULL,

ts TIMESTAMP NOT NULL,

count\_trips\_window\_30m\_dropoff\_zip INT,

dropoff\_is\_weekend INT,

CONSTRAINT trip\_dropoff\_time\_series\_features\_pk PRIMARY KEY (zip, ts TIMESERIES)

)

CLUSTER BY AUTO

COMMENT "Taxi Fares. Dropoff Time Series Features";

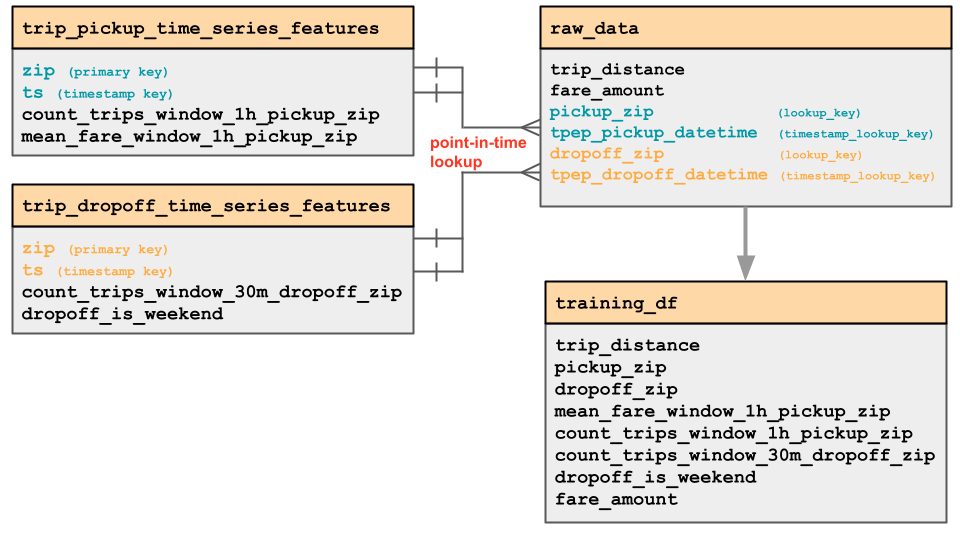
Please note that the code to write to feature tables has been omitted for brevity as it has already been covered in the Feature Management section. For the code reference to the complete example, refer to the notebook [here](https://docs.databricks.com/aws/en/notebooks/source/machine-learning/feature-store-with-uc-taxi-example.html).

### Create Training Dataset

In order to train a model, we need to create a training dataset which consists of

* Raw Input Data : We need the primary keys and timeseries columns to join with features to ensure point in time correctness ([link](https://learn.microsoft.com/en-us/azure/databricks/machine-learning/feature-store/time-series#create-a-training-set-with-a-time-series-feature-table)).
* Features from the feature tables in unity catalog: We need the calculated features which are not present in the raw data.

The following diagram depicts a visual overview of the raw input data being combined with the features in unity catalog to produce the training dataset. For more details related to concepts of creating training datasets refer to the databricks documentation [here](https://learn.microsoft.com/en-us/azure/databricks/machine-learning/feature-store/train-models-with-feature-store#create-a-training-dataset).



In the code below, we will do the following:

* Create feature lookup for each needed feature from the feature tables
* Create the training dataset by joining the raw\_data with the feature lookup

#### Create feature lookup

Next we take a look at loading the features from Unity Catalog for model training by using the FeatureLookup function from the databricks-feature-engineering sdk ([link](https://api-docs.databricks.com/python/feature-engineering/latest/ml_features.feature_lookup.html?highlight=feature_lookup#databricks.ml_features.entities.feature_lookup.FeatureLookup)). It is recommended to use the FeatureLookup function to create the lookups, so that when we create the training dataset and log the model, it retains the lineage w.r.t which features it used for training.

from databricks.feature\_engineering import FeatureLookup

import mlflow

pickup\_features\_table = "ml.taxi\_example.trip\_pickup\_time\_series\_features"

dropoff\_features\_table = "ml.taxi\_example.trip\_dropoff\_time\_series\_features"

pickup\_feature\_lookups = [

FeatureLookup(

table\_name=pickup\_features\_table,

feature\_names=[

"mean\_fare\_window\_1h\_pickup\_zip",

"count\_trips\_window\_1h\_pickup\_zip",

],

lookup\_key=["pickup\_zip"],

timestamp\_lookup\_key="tpep\_pickup\_datetime",

),

]

dropoff\_feature\_lookups = [

FeatureLookup(

table\_name=dropoff\_features\_table,

feature\_names=["count\_trips\_window\_30m\_dropoff\_zip", "dropoff\_is\_weekend"],

lookup\_key=["dropoff\_zip"],

timestamp\_lookup\_key="tpep\_dropoff\_datetime",

),

]

#### Create training dataset

When fe.create\_training\_set(..) is invoked in the code below, the following steps take place:

1. A TrainingSet object is created, which selects specific features from feature tables to use in training your model. Each feature is specified by the FeatureLookup's created previously.
2. Features are joined with the raw input data according to each FeatureLookup's lookup\_key.
3. Point-in-Time lookup is applied to avoid data leakage problems. Only the most recent feature values, based on timestamp\_lookup\_key, are joined.

The TrainingSet is then transformed into a DataFrame for training. This DataFrame includes the columns of taxi\_data, as well as the features specified in the FeatureLookups.

import mlflow

mlflow.set\_registry\_uri("databricks-uc")

# Start an mlflow run, which is needed to log the model

mlflow.start\_run()

# Since the timestamp columns would likely cause the model to overfit the data

# unless additional feature engineering was performed, exclude them to avoid training on them.

exclude\_columns = ["tpep\_pickup\_datetime", "tpep\_dropoff\_datetime"]

# Create the training set that includes the raw input data merged with corresponding features from both feature tables

training\_set = fe.create\_training\_set(

df=raw\_data,

feature\_lookups=pickup\_feature\_lookups + dropoff\_feature\_lookups,

label="fare\_amount",

exclude\_columns=exclude\_columns,

)

# Load the TrainingSet into a dataframe which can be passed into sklearn for training a model

training\_df = training\_set.load\_df()



### Model Training

Using the training set created from above, we will next train a LightGBM model and then log the model using FeatureEngineeringClient.log\_model from the databricks-feature-engineering sdk ([link](https://api-docs.databricks.com/python/feature-engineering/latest/feature_engineering.client.html?highlight=log_model#databricks.feature_engineering.client.FeatureEngineeringClient.log_model)) to ensure the model will be packaged along with the feature metadata.

| **Note**: This section outlines the process of model training in the context of using feature tables, feature lookup, creating training dataset and logging the trained model into MLFlow and Unity Catalog along with the relevant metadata.  For detailed model training examples on specific use cases, please refer to the databricks documentation here: [machine learning examples](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/train-model/training-examples), [distributed deep learning examples](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/train-model/deep-learning), [recommendation models](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/train-recommender-models), [automl](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/automl/). |
| --- |

The reference code to train the model has been outlined below

from sklearn.model\_selection import train\_test\_split

from mlflow.tracking import MlflowClient

import lightgbm as lgb

import mlflow.lightgbm

from mlflow.models.signature import infer\_signature

features\_and\_label = training\_df.columns

# Collect data into a Pandas array for training

data = training\_df.toPandas()[features\_and\_label]

train, test = train\_test\_split(data, random\_state=123)

X\_train = train.drop(["fare\_amount"], axis=1)

X\_test = test.drop(["fare\_amount"], axis=1)

y\_train = train.fare\_amount

y\_test = test.fare\_amount

mlflow.lightgbm.autolog()

train\_lgb\_dataset = lgb.Dataset(X\_train, label=y\_train.values)

test\_lgb\_dataset = lgb.Dataset(X\_test, label=y\_test.values)

param = {"num\_leaves": 32, "objective": "regression", "metric": "rmse"}

num\_rounds = 100

# Train a lightGBM model

model = lgb.train(param, train\_lgb\_dataset, num\_rounds)

### Model Deployment (MLFlow and Unity Catalog)

Using the model created above we will next log the model using the FeatureEngineeringClient.log\_model to ensure it gets packaged with feature information.

After the model is logged it should be visible in the Databricks unity catalog and should have the required feature lineage.

# Log the trained model with MLflow and package it with feature lookup information.

fe.log\_model(

model=model,

artifact\_path="model\_packaged",

flavor=mlflow.lightgbm,

training\_set=training\_set,

registered\_model\_name="ml.taxi\_example.taxi\_example\_fare\_time\_series\_packaged",

)

#### Adding pre-processing or post-processing steps to model

If we need to add pre-processing or post-processing steps to the model i.e. generate processed predictions with batch inference, we have to package the model as a custom PyFunc MLFlow model that encapsulates such methods.

In the below example, we see how we can package the model with a post\_processing step that returns a customer string output based on the prediction by the model

# custom class to package a model with preprocessing and post processing steps

class fareClassifier(mlflow.pyfunc.PythonModel):

def \_\_init\_\_(self, trained\_model):

self.model = trained\_model

def preprocess\_result(self, model\_input):

return model\_input

def postprocess\_result(self, results):

"""Return post-processed results.

Creates a set of fare ranges

and returns the predicted range."""

return [

"$0 - $9.99" if result < 10 else "$10 - $19.99" if result < 20 else " > $20"

for result in results

]

def predict(self, context, model\_input):

processed\_df = self.preprocess\_result(model\_input.copy())

results = self.model.predict(processed\_df)

return self.postprocess\_result(results)

# we package the previously created LIGHTGBM model with this wrapper

pyfunc\_model = fareClassifier(model)

# End the current MLflow run and start a new one to log the new pyfunc model

mlflow.end\_run()

# we then log the new packaged model

with mlflow.start\_run() as run:

fe.log\_model(

model=pyfunc\_model,

artifact\_path="pyfunc\_packaged\_model",

flavor=mlflow.pyfunc,

training\_set=training\_set,

registered\_model\_name="ml.taxi\_example.pyfunc\_taxi\_fare\_time\_series\_packaged",

)



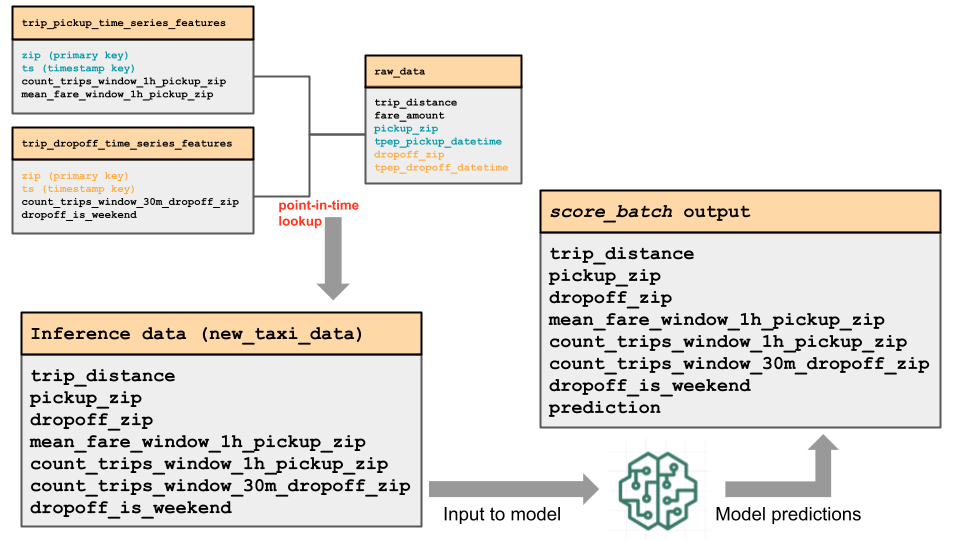
### Scoring (using batch inference)

It might be necessary to apply this model to a new batch of data or a different data slice to evaluate the model performance.

In this scenario, we will use the FeatureEngineeringClient.score\_batch function from the databricks-feature-engineering sdk ([link](https://api-docs.databricks.com/python/feature-engineering/latest/feature_engineering.client.html?highlight=score_batch#databricks.feature_engineering.client.FeatureEngineeringClient.score_batch)).

The score\_batch api, will retrieve the appropriate features using point-in-time lookups with the packaged metadata during model training. The data that we provide to the score\_batch function must contain a timestamp column with the same name and DataType as the timestamp\_lookup\_key of the FeatureLookup provided to the fe.create\_training\_set function above in the previous section.

The following diagram provides a visual representation of the process.



The sample code below outlines the process of scoring using the model from unity catalog.

# get the latest model version

latest\_pyfunc\_version = get\_latest\_model\_version(

"ml.taxi\_example.pyfunc\_taxi\_fare\_time\_series\_packaged"

)

# create the model uri

pyfunc\_model\_uri = ( f"models:/ml.taxi\_example.pyfunc\_taxi\_fare\_time\_series\_packaged/{latest\_pyfunc\_version}"

)

# get the predictions from the model

pyfunc\_predictions = fe.score\_batch(

model\_uri=pyfunc\_model\_uri, df=new\_taxi\_data, result\_type="string"

)

# view the predictions from the model

display(pyfunc\_predictions.select("fare\_amount", "prediction"))



## Model Serving

This section outlines the model serving options for various use-cases. Primarily there are two modes of model serving in the Azure Databricks Platform.

* Batch Inference (offline scoring)
* Model Serving endpoint (real-time use cases)

Note.

* In this document, we focus on traditional machine learning models. Specifically, the model type we refer to in Databricks is a custom model, which is a Python-based model packaged using the MLflow format. Databricks also supports other model types, including foundation models, which are typically used in large language model (LLM) scenarios. ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/model-serving/#models-you-can-deploy))

### Batch Inference

For use-cases which involve large scale predictions on historical data, or where some predictions that need to be made on a schedule(i.e. daily, weekly), a batch inference mode is recommended.

In this scenario, we load the data into a spark dataframe, load the required model from the databricks registry, make predictions and write the predictions into a delta table. The example has been outlined below:

# load model from model registry and create a spark user defined function

predict\_udf = mlflow.pyfunc.spark\_udf(spark, model\_uri)

# get the predicted output using the user defined function

df\_result = df\_spark.withColumn("prediction", predict\_udf(\*df\_spark.columns))

# write the prediction results to a delta table

df\_result.write.mode("overwrite").saveAsTable(output\_table)

For more information on performance tuning and other examples refer to the databricks documentation [here](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/model-inference/dl-model-inference#performance-tuning-for-model-inference).

### Model Serving Endpoint

Mosaic AI Model Serving is a unified interface for deploying, governing, and querying AI and ML models for both real-time serving and batch inference. It is designed to offer a highly available and low-latency service that automatically scales up or down to meet demand changes.

For real-time inference, Mosaic AI Model Serving makes each served model available as a REST API endpoint that can be integrated into web or client applications.

This section demonstrates how to create a model serving endpoint and how to send inference requests to the endpoint.

##### Create the Model Serving Endpoint

To create a model serving endpoint, we can use either the Databricks UI, Python SDK/REST API, or MLflow Deployments SDK. ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/model-serving/create-manage-serving-endpoints#create))

Note. As of July 31, 2025, creating a model serving endpoints via Terraform is not supported.

In the code sample below, we will see how to create a model serving endpoint using the MLFlow deployments sdk.

from mlflow.deployments import get\_deploy\_client

client = get\_deploy\_client("databricks")

endpoint = client.create\_endpoint(

name = f"{prefix}-{domain}-{sub\_domain}-{use\_case}",

config = {

"served\_entities": [

{

"name": "ads-entity"

"entity\_name": f"{schema}.{model\_name}",

"entity\_version": F"{model\_version}",

"workload\_size": "Small",

"scale\_to\_zero\_enabled": False

}

],

"traffic\_config": {

"routes": [

{

"served\_model\_name": f"{model\_name}-{version}",

"traffic\_percentage": 90

},

{

"served\_model\_name": f"{model\_name}-{previous\_version}",

"traffic\_percentage": 10

}

]

}

}

)



##### Configure Key Parameters

When creating the endpoint, you will configure several key parameters

| Parameter | Description |
| --- | --- |
| Served entities | Specify the model and its version to be served.   * It is able to serve more served entities. Note. Cannot serve different model types (e.g., custom and external) within the same endpoint * You can define a **traffic percentage** for each **served entity** (i.e., model). This allows you to control how inference requests are distributed across models.This is useful for A/B testing, canary releases, or comparing model performance in production. |
| Compute Type | Choose between CPU or GPU compute for your workloads |
| Compute scale-out | Define the size: the number of concurrent requests the served model can process at the same time.   * Small: 0-4 requests * Medium: 8-16 requests * Large: 16-64 requests * Custom: 512 is the max concurrency   Scale to zero: Specify if the endpoint should scale down to zero when inactive.   * Pros: It saves costs when not in use * Cons: Potential "cold start" latency for the first request after inactivity. Not recommended for production endpoints.   Enable tracing  Enabling tracing will record inputs, outputs, and parameters from instrumented steps during model inference. Trace data will be written to the (required) configured Inference Table. |
| Route optimization | This feature improves the network path for inference requests, lowering overhead latency and increasing Queries Per Second (QPS)   * Recommended for high throughput or low latency workloads. * Not support external model and Foundation Model APIs * Databricks in-house OAuth tokens are the only supported authentication for route optimization. Personal access tokens are not supported. |
| AI Gateway | **Enable usage tracking**  Tracking and monitoring of data usage metrics related to your model serving endpoints([Usage tracking table schema](https://learn.microsoft.com/en-us/azure/databricks/ai-gateway/configure-ai-gateway-endpoints#usage-schema))   * Pros: Useful for gaining insights into how your models are being utilized and for purposes like cost attribution   **Enable inference tables**  Automatically log incoming requests and responses from your endpoint to Unity Catalog Delta tables ([Inference table schema](https://learn.microsoft.com/en-gb/azure/databricks/ai-gateway/inference-tables#ai-gateway-enabled-inference-table-schema))   * Pros: Valuable for monitoring model quality, debugging production issues   **Rate limits**  Manage and specify the number of queries per minute (QPM) an endpoint can receive, both overall and per user/group/service principal   * Caveat: Multiple requests made over the rate limit of an endpoint might lead to failure for additional requests. |
|  |  |
|  |  |

##### Query the Model Serving Endpoint ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/model-serving/score-custom-model-endpoints#querying-methods-and-examples))

The section describes what options for sending scoring/inferencing requests to a served model.

| Method | Usage / Description | ML Scenario (When needed / Relevant) |
| --- | --- | --- |
| Serving UI | 1. Navigate to the **Serving** page in your Databricks workspace. 2. Select the target **Serving endpoint**. 3. Click the **"Use"** button located in the top right corner of the **Query endpoint** section. 4. In the **Query endpoint** page, insert the model input in **JSON format**, then click **"Send Request"** to submit it. 5. Example code snippets for making requests via **cURL**, **Python**, and **SQL** are also available on the same page for reference. | * For manual testing of the model. * Used to validate model predictions. * For ad-hoc inference with sample input. * Useful when debugging model responses. |
| SQL function | You can invoke model inference directly from SQL using the ai\_query SQL function | * For batch inference on structured data via SQL. * Useful for integrating ML into SQL-based data pipelines. |
| REST API | Call and query the model using the REST API   * The REST API can be retrieved from “Serving endpoints” page * Syntax: POST [/serving-endpoints/{name}/invocations](https://docs.databricks.com/api/azure/workspace/servingendpoints/query) * For scoring requests to endpoints serving multiple models, see the [reference](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/model-serving/serve-multiple-models-to-serving-endpoint#query) | * For integrating model inference into external applications or services. * When triggering inference from non-Databricks environments. |
| MLflow Deployments SDK | Use MLflow Deployments SDK's [predict()](https://mlflow.org/docs/latest/python_api/mlflow.deployments.html#mlflow.deployments.DatabricksDeploymentClient.predict) function to query the model. | * For programmatic inference within Python applications. * When integrating model inference into ML pipelines or notebooks. * Useful for real-time or batch inference in code-based workflows. |

#### Expectations and Limitations

The section describes known expectations and limitations for serving custom models using Model Serving

| Deployment and Update Times | * **Deploying a new model version** (includes packaging the model and its environment) can **take approximately 10 minutes** |
| --- | --- |
| Request Timeouts | * By default, it’s 120 seconds ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/model-serving/custom-models)) * The default can be increased up to 250 seconds ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/model-serving/model-serving-timeouts)) |
| Scaling behavior ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/model-serving/custom-models#endpoint-scaling-expectations)) | * Scale up almost immediately with increased traffic * Scale down every five minutes to match reduced traffic. * Caveat: Scale to zero ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/model-serving/custom-models#endpoint-scaling-expectations))   + - While endpoints can scale to zero, the first request after scaling to zero will experience a "cold start," leading to higher latency due to the need to warm up. This is generally not recommended for latency-sensitive production endpoints |
| Concurrency and Payload Limits ([ref](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/model-serving/model-serving-limits)) | * There are default limits on queries per second (**200 QPS per workspace**, increasable with route optimization) and provisioned concurrency (200 per model/workspace, increasable by support) * **Payload size** per request is **16 MB** for custom models. * Requests/responses over 1 MB will not be logged to inference tables |

## Monitoring

<ToDo>

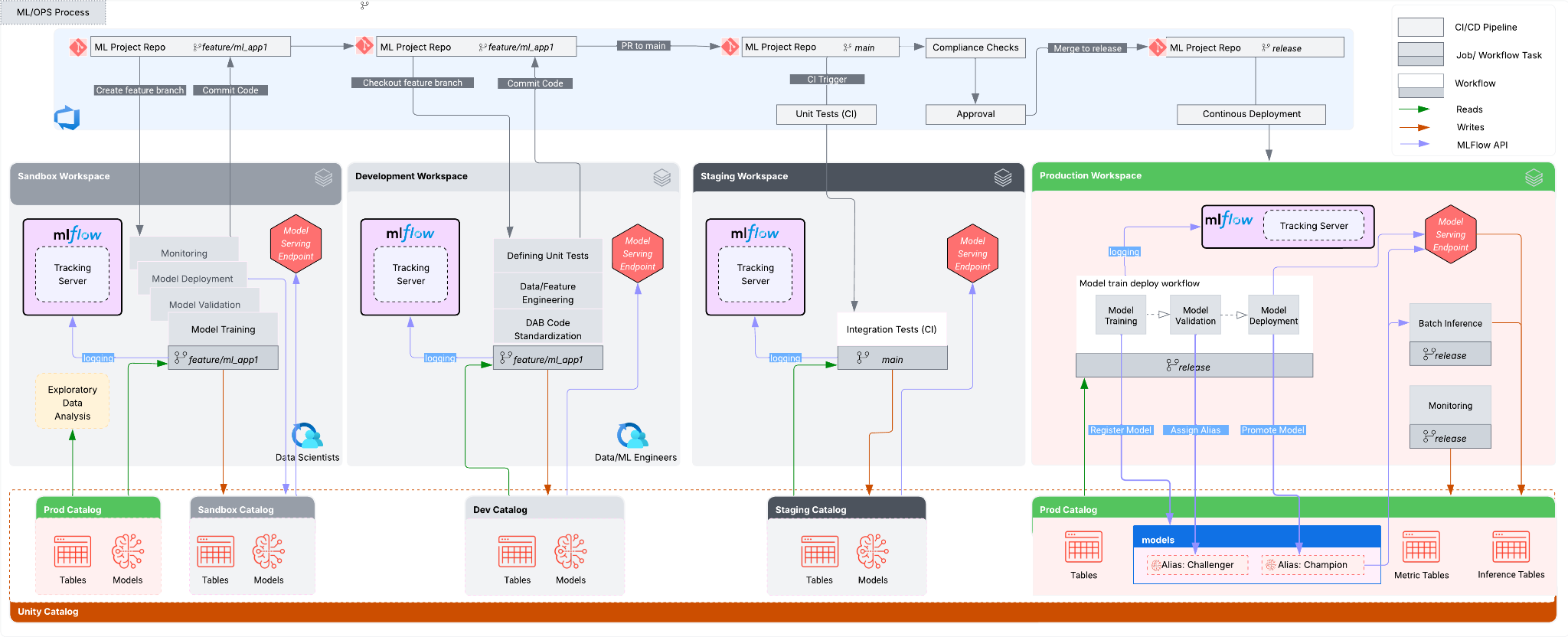
[Inference tables for monitoring and debugging models](https://learn.microsoft.com/en-gb/azure/databricks/machine-learning/model-serving/inference-tables)

[Monitor endpoint usage](https://learn.microsoft.com/en-us/azure/databricks/ai-gateway/configure-ai-gateway-endpoints#monitor-endpoint-usage)

## MLOPS Process

### Overview

This section outlines the automations and CI/CD process in the context of ML/OPS for SCB AI Factory.



The diagram above illustrates the comprehensive ML/OPS process across the planned environments at SCB. Key aspects are detailed below.

* As discussed in the *Unity Catalog* design, Azure Databricks Workspaces shall be set up for each *domain* and *environment*.
* A *domain* may be a *business unit* (BU) i.e. CIB or a *sub-domain/team* i.e. CIB\_CASH, CIB\_CC
* Environments primarily would be *dev*, *stage* and *prod*, and a *sandbox* environment shall be provisioned on request.
* An environment consists of an *Azure Databricks Workspace*, a domain specific *Catalog*, and other essential Databricks components as required for the *domain/use case*. The components that may be provisioned have been outlined in the [CI/CD process](#_o0be81gq9ka3)/ [Components](#_4l1j9j6tzx4i) section.
* The ML/OPS process has been set in the context of a single domain and depicts how the process would deploy artefacts across multiple environments using a CI/CD process.

### Components

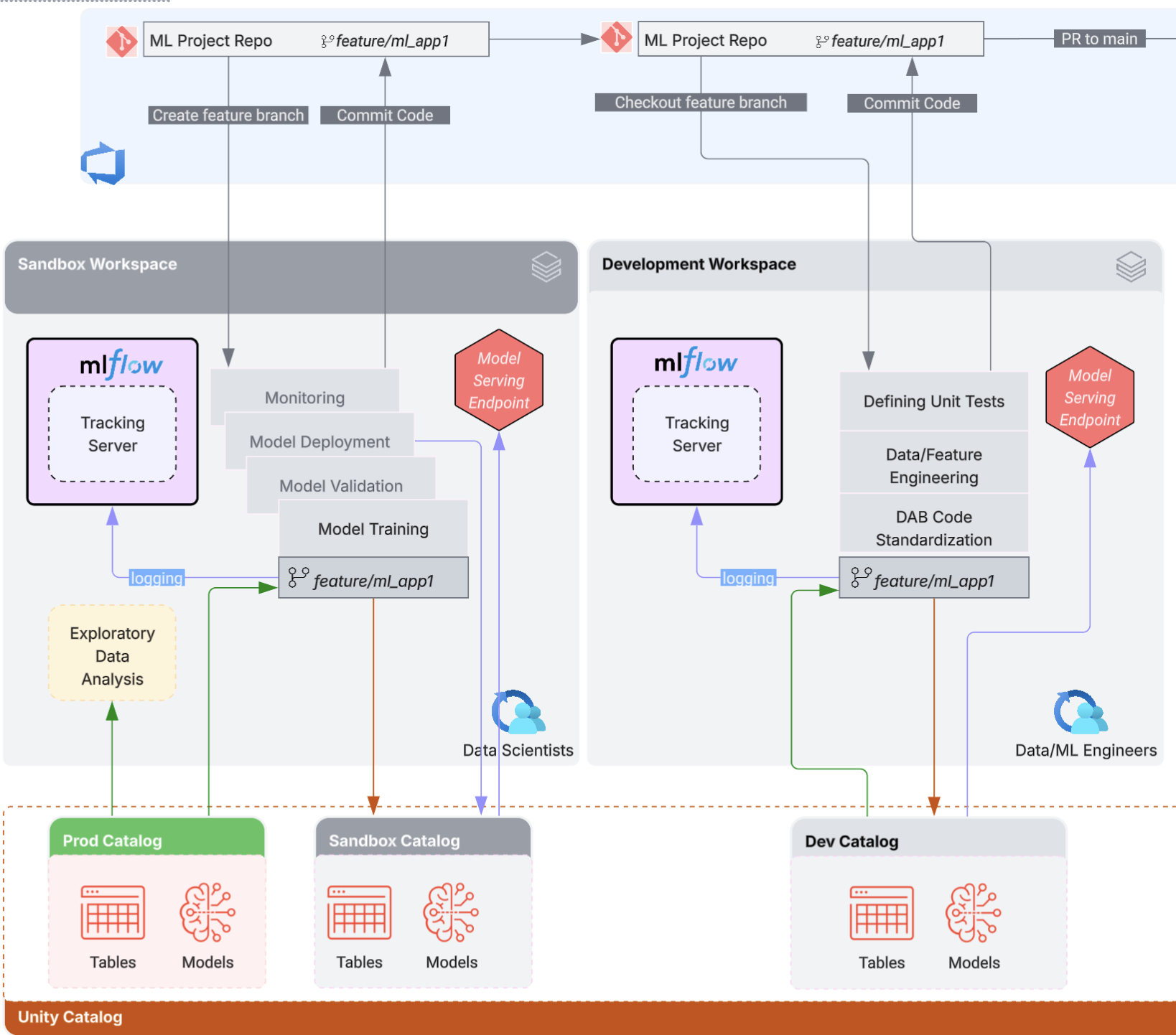
This section outlines the components identified for ML/OPS and outlines how they would be deployed across all environments, including Sandbox.

| ***Component Type*** | ***Mode Of Deployment*** | ***ML Scenario (When needed / Relevant)*** | ***Comments*** |
| --- | --- | --- | --- |
| Models | Databricks Asset Bundles (DAB) | When trained models are ready for deployment, either for batch or real-time inference. | Deployed as Code through DAB and created in each environment. |
| Experiments | Databricks Asset Bundles (DAB) | Needed to track training runs, hyperparameters, and metrics. | Deployed as Code through DAB and created in each environment in MLFlow during model training run. |
| Lakeflow Pipelines | Databricks Asset Bundles (DAB) | Required for orchestrating feature engineering, model training, and batch inference workflows. Can be used for near real-time processing or batch processing as per requirement. | Deployed as Code through DAB and created in each environment. |
| Streaming Tables/ Materialized Views | Databricks Asset Bundles (DAB) | Used in near real-time data processing pipelines, often for online feature generation or fast-refreshing features. | Created as part of Lakeflow Pipelines Deployment using DAB |
| Feature Tables (Delta tables and views only) | Terraform | Needed when defining and managing reusable, governed feature sets used by multiple models. | Deployed as unity catalog objects through terraform, and populated through pipelines which are deployed using DAB. |
| Model Serving Endpoints | Terraform | Required for real-time inference use cases like fraud detection, recommendations, or chatbots. | One Model Serving Endpoint is required per model per environment, in use-cases involving real-time model inference. (Note: A single use case may have multiple models)  This is not required for batch-inference use cases. |
| Vector Search Endpoints | Terraform | Required for GenAI / RAG (Retrieval-Augmented Generation) use cases needing semantic search or contextual memory. | Every Vector Search Endpoint has a limit of 50 vector search indexes. (The number of Vector Search Endpoints depends on the number logical collections of documents that need to be indexed for vector search) |
| Lakebase (database instances) | Terraform | Used to ground LLMs with structured enterprise data for hybrid search or context augmentation. | Required for GEN-AI use cases (1 per use case) |
| Mosaic AI Gateway | Terraform | Enables safe, governed, and cost-aware access to GenAI services and LLMs in production environments. | If rate limits or budgets need to be set for LLMs (SCB currently have their own implementation of GuardRails ) |
| Databricks Apps | Terraform | Required when exposing model outputs via custom interfaces or dashboards (e.g., explainability, manual overrides, user validation). | Required for MCP and custom UI. (Provide self-manage ability in Sandbox, Needs to be promoted to Prod) |
| SQL Warehouses | Terraform | Used for batch scoring, BI integration, or joining predictions with other business data using SQL. | Managed by SCB AI Platform team. |
|  |  |  |  |

### Development Lifecycle

This section outlines the development lifecycle for the MLOPS process.

A high level representation of the development lifecycle in the Sandbox and Development Azure Databricks Workspaces has been outlined below.



Some important points have been outlined below:

* *Data Scientists* shall have access to the Sandbox databricks workspace for doing exploratory data analysis and experimentation on prod like data and shall create feature engineering and model training workflows in the sandbox environment, i.e. for ML Projects needing access to prod like data, a sandbox environment is a pre-requisite.
* Developers (Data Scientists/Data Engineers) shall use [Databricks Asset Bundles](https://learn.microsoft.com/en-gb/azure/databricks/dev-tools/bundles/) for development and deployment.
* Developers shall request for creation of required Databricks Components (as outlined in the above [Components](#_4l1j9j6tzx4i) section) in the sandbox and development environments through an *Resource Provisioning* ADO Pipeline process managed by the SCB AI Platform Team. *<Add link to the process here: TBD>*
* It is recommended for an ADO ML Project repository to be created for each bu/subdomain/project, i.e. cib\_cash\_ml\_app1.
* The project repository shall be initialized using the Databricks Asset Bundles Init command, i.e. databricks bundle init mlops-stacks, for more details see [ML Project folder structure](#_tcxi26pwwndm)
* The developers (Data and ML Engineers) shall add required supporting code, workflows, unit tests, validation and monitoring notebooks etc to the feature branch in the ADO Repository in the Development workspace.
* The developers (Data/ML engineers) will work with the Data Scientists to standardize model training code into the required Databricks Asset Bundle structure if needed.

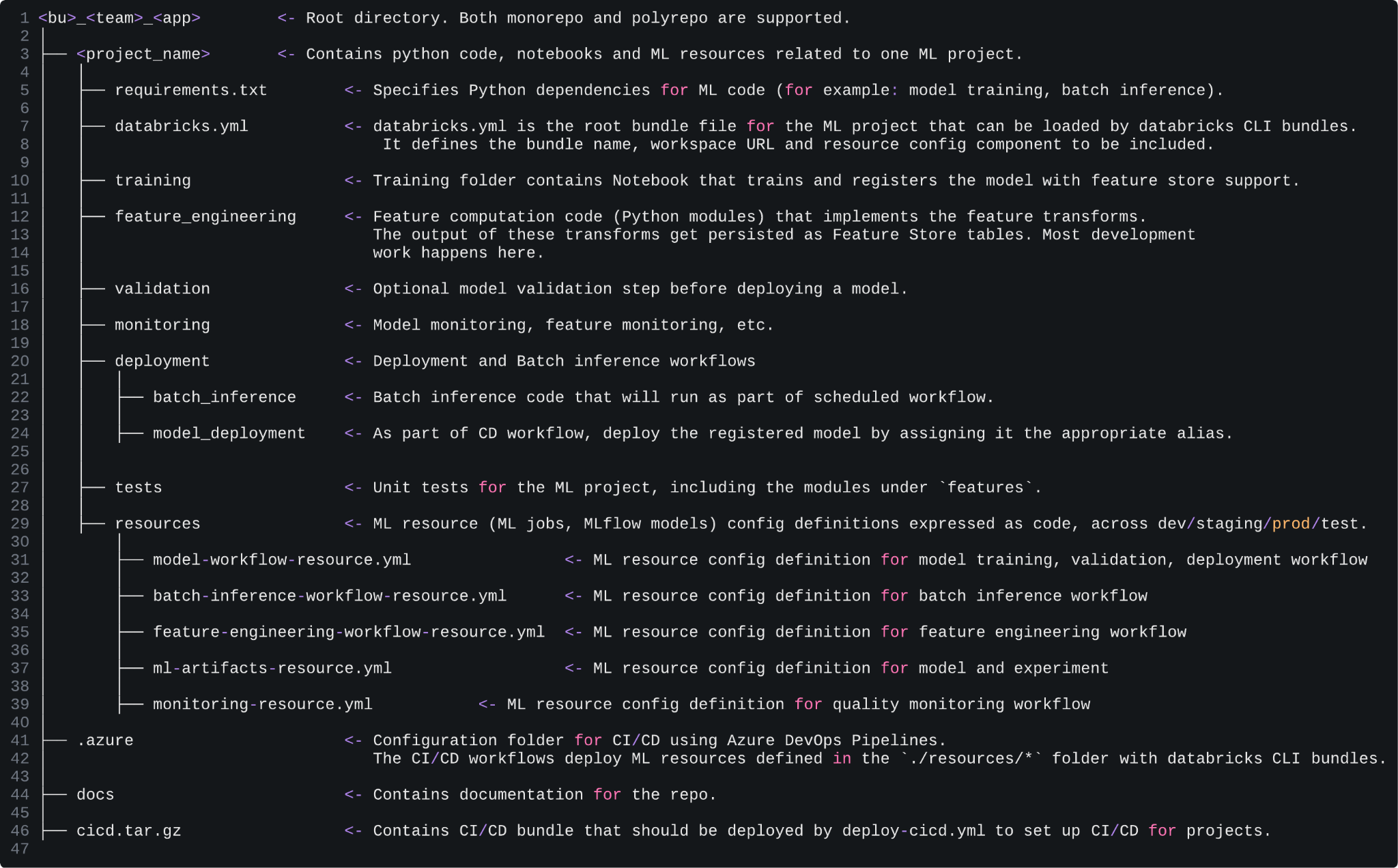
#### ML Project folder structure

It is proposed that the following folder structure be used for all ML Projects part of the SCB AI Factory. This is meant to provide a re-usable template across projects for standardization of development and deployment processes.

The following template can also be created using the databricks bundles init command with a project type of mlops-stacks ([link](https://learn.microsoft.com/en-gb/azure/databricks/dev-tools/bundles/mlops-stacks#step-2-create-the-bundle-project)).

databricks bundle init mlops-stacks





| **Note**: For more details on the mlops-stacks, refer to the databricks documentation [here](https://learn.microsoft.com/en-gb/azure/databricks/dev-tools/bundles/mlops-stacks) and the github repo [here](https://github.com/databricks/mlops-stacks). |
| --- |

<ToDo: Add steps for the development lifecycle and add provide reference>

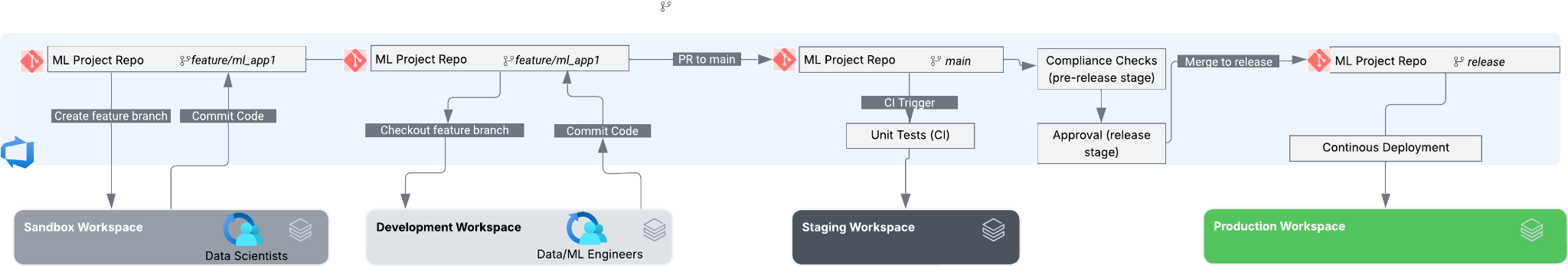
### Deployment Lifecycle: Azure Devops Pipelines

This section outlines the deployment process using Azure Devops Pipelines to be configured for the ML/OPS implementation for the SCB AI Factory.

It is proposed to configure a Code Deployment Azure Devops Pipeline, for Deployment of ML Project resources (code, workflows) from Dev to the Staging and Production environments.

#### Code Deployment Pipeline

The CI/CD Code Deployment pipeline set up in Azure Devops will be utilized to deploy ML Code and Workflows to the Development, Staging and Production environments.



* Developers shall use *Databricks Asset Bundle* structure for developing the code. This means when the ML Project Repository is first created, the repository will be initialized with the relevant structure for supporting the Databricks Asset Bundle either by using the `bundle init` command (preferred) or manually by creating the required yaml files. For more information around this, please refer to the Databricks documentation reference [here](https://learn.microsoft.com/en-gb/azure/databricks/dev-tools/bundles/templates#create-a-bundle-using-a-template).
* Developers shall create a feature branch from the ML Project repository and make their changes. Once development is complete, the code for the model training pipelines along with any supporting code (i.e. feature engineering pipelines) shall be committed to the feature branch, and a pull request to be created by the developer to merge to the main branch.
* The pull request for merging into main, will serve as the trigger for kicking off the ADO Pipeline which will trigger the CI/CD process of deploying the code to environments.
* Unit Tests(CI): The ADO Pipeline to trigger the unit tests before deploying to any environment. These shall include the following:
  + Check for hard coded secrets
  + Validate the Service Account credentials
  + Run the databricks asset bundle validation ([link](https://learn.microsoft.com/en-gb/azure/databricks/dev-tools/cli/bundle-commands#validate)), to ensure all code to be deployed has been correctly configured.
  + Request a manual approval for the validated changes prior to deployment
* Deployment to Staging: Once the unit tests are complete, code assets and jobs to be deployed to the Dev/Staging environment and Integration tests to be run in each environment. The integration tests to including the following:
  + Deploy code and workflows.
  + Run the feature engineering workflows (if any)
  + Run model training workflows (end to end) on a smaller dataset and register the experiments and logs in ml/flow.
  + Once model training is complete, save the model version in unity catalog.
  + Run any other validation job as defined by the developers based on the septic use-case
* Pre-Release (*Compliance Checks*): This stage to execute the required compliance checks required by the scb security, cloud and ai platform teams.
* Release (*Approval*): This stage to request the required approvals from the relevant stakeholders prior to deployment to the production environment.
* Deployment to Production: Once approved, the code assets and jobs shall be deployed to production. Post-deployment, the entire model training workflow shall be run which may include the following steps:
  + Run feature engineering workflows(if any).
  + Run model training workflows(end to end) on the entire production dataset and register the experiments and logs in ml/flow.
  + Once model training is complete, save the model version in unity catalog.
  + Run validation and deployment steps for promoting the model, as outlined in the [Environment Steps](#_yzpx2u4s9nun) section below under `Production`

### Environment/Steps Description

Some of the environment specific steps and descriptions are outlined below. For a detailed description please refer to the databricks documentation [here](https://docs.databricks.com/aws/en/machine-learning/mlops/mlops-workflow#general-recommendations-for-mlops).

| ***Environment*** | ***Description*** | ***Comments*** |
| --- | --- | --- |
| [Sandbox](https://docs.databricks.com/aws/en/machine-learning/mlops/mlops-workflow#development-stage) | * Sandbox environment primarily to be used for Exploratory Data Analysis and Model training and validation by Data Scientists and Data/ML Engineers (optional). * Assumption here is that each team has their own ADO Repository. * A *feature* branch shall be created from the ML Project ADO Repo. * Once exploratory data analysis, model training and validation is complete, the data scientists shall push their model training code to the ADO Repo *feature* branch. * This will be picked up by the Data/ML engineers for adding supporting code (feature engineering pipelines, unit tests, validation, monitoring etc) and for model training code standardization in the Development environment. | Note: It is recommended to deploy code across environments instead of models. For a better understanding around this approach, please refer to the databricks documentation [here](https://docs.databricks.com/aws/en/machine-learning/mlops/mlops-workflow#deploy-code-not-models).  Note: It is common for Data Scientists to test multiple algorithms and hyperparameters for testing/tuning a model for a given scenario or use case. This would happen in the sandbox environment and only the top performing or viable option(s) would be deployed to the production environment. |
| Development | * Development environment primarily to be used for creating feature engineering pipelines, standardizing model training code, defining unit tests, validation and monitoring. * Developer to pull the *feature* branch from the ADO Repo and push their changes back into the feature branch. * The developer then creates a *Pull Request* to merge the code into the *main* branch. * This pull request (merge to main) triggers the ADO CI/CD pipeline, for deployment to Dev, Stage and Prod. |  |
| [Staging](https://docs.databricks.com/aws/en/machine-learning/mlops/mlops-workflow#staging-stage) | * Prior to deployment to Staging, a CI stage containing unit tests would be triggered. Unit Tests (CI) would be use-case specific and to be determined by the developers for their specific codebase or scenario. * The staging environment shall be used for running Integration Tests. * Integration tests (CI) to run the entire flow , i.e. feature engineering, model training, model validation, model deployment, inference and monitoring. * If an ML Application with real-time inference is being deployed, it is recommended to test the serving infrastructure in the Staging environment. This would involve creating a model-serving endpoint and deploying the model. * Certain trade-offs may be made in case the model is large and costly to train, i.e. training the model with a lesser amount of data and/or running the training for fewer iterations depending upon the use-case. * Once the CI Tests are complete, a release branch would be created or alternatively a release would be pushed to an artifactory (i.e. JFrog) and then trigger the deployment to Production. |  |
| [Production](https://docs.databricks.com/aws/en/machine-learning/mlops/mlops-workflow#production-stage) | * As mentioned previously, it is recommended to deploy code and re-run the entire workflow in the production environment. The workflow would include the steps outlined below. * Feature Engineering: Compute Features and register them as feature tables. * Model Training: Using the feature tables from the production catalog, re-run the model training, evaluation and registration steps.   + Training/Tuning: The Model, logs and parameters would be tracked in the production MLFlow Server.   + Evaluation: Model Quality is evaluated by running the inference on the test data from the train-test split (i.e. held out data) and the results of these tests are logged into MLFlow. The test would include evaluation metrics specified by the data scientists (may consist of custom code).   + Registration: After model training is complete, the model is saved in Unity Catalog as a registered model version in the specified path in the production catalog. * Validation: The validation checks may include the following   + format/metadata validations   + Performance evaluation based on selected data slices (i.e. for specific date range like last month etc)   + Compliance checks (i.e. tags, documentation, or any other org specific checks)   + If the model does not pass the validation checks the process exits and data scientists can be notified automatically   + If the model passes the validation checks, then a `Challenger` alias shall be assigned to the model version in unity catalog. * Deployment: This may be specific to the scenario/use case. A typical workflow would proceed as follows:   + If there is no existing production model (Champion), the *Challenger* to be evaluated against a business heuristic.   + If there exists a *Champion* (existing production model), then an offline comparison would be performed between both models (*Champion* and *Challenger*) and the results to be tracked in MLFlow.   + In case the Challenger model version performs better in the comparison, it becomes the `Champion`.   + Batch Inference Pipelines to be setup to read from the model `Champion` alias.   + Model Serving Endpoints to be updated with the new model. |  |

# Appendix

## [Liquid Clustering](https://docs.databricks.com/en/delta/clustering.html) ([ref](https://docs.google.com/document/d/1qS1lGUg-dfRU8cnOH2NuBKQGjTpkJG1zkdDaKl5J78c/edit?tab=t.0))

An adaptive partitioning strategy that can adjust to the distribution of data. By dynamically reshaping partitions as data grows or changes, it ensures that data retrieval remains optimal over time. Compared to Z-order, Liquid Clustering offers **incremental optimization**.

This means that each time you run OPTIMIZE, it only processes newly ingested data. If there’s no new data, the operation becomes a no-op, avoiding unnecessary rewrites.

As a result, it reduces write overhead significantly while maintaining query performance, leading to lower overall costs.

### Optimal Use Case

* Tables often filtered by high cardinality columns.
* Tables with significant skew in data distribution.
* Tables that grow quickly and require maintenance and tuning effort.
* Tables with concurrent write requirements.
* Tables with access patterns that change over time.
* Tables where a typical partition key could leave the table with too many or too few partitions.

### Enable liquid clustering

To enable liquid clustering, add the CLUSTER BY phrase to a table creation statement

-- Create an empty Delta table

CREATE TABLE table1(col0 INT, col1 string) CLUSTER BY (col0);

-- Using a CTAS statement

CREATE EXTERNAL TABLE table2 CLUSTER BY (col0) -- specify clustering after table name, not in subquery

LOCATION 'table\_location'

AS SELECT \* FROM table1;

-- Enable liquid clustering on an existing unpartitioned Delta table

ALTER TABLE <table\_name>

CLUSTER BY (<clustering\_columns>)

-- To remove clustering keys

ALTER TABLE table\_name CLUSTER BY NONE;



Enable or disable automatic liquid clustering

-- Create an empty table.

CREATE OR REPLACE TABLE table1(column01 int, column02 string) CLUSTER BY AUTO;

-- Enable automatic liquid clustering on an existing table,

-- including tables that previously had manually specified keys.

ALTER TABLE table1 CLUSTER BY AUTO;

-- Disable automatic liquid clustering on an existing table.

ALTER TABLE table1 CLUSTER BY NONE;

-- Disable automatic liquid clustering by setting the clustering keys

-- to chosen clustering columns or new columns.

ALTER TABLE table1 CLUSTER BY (column01, column02);

