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.

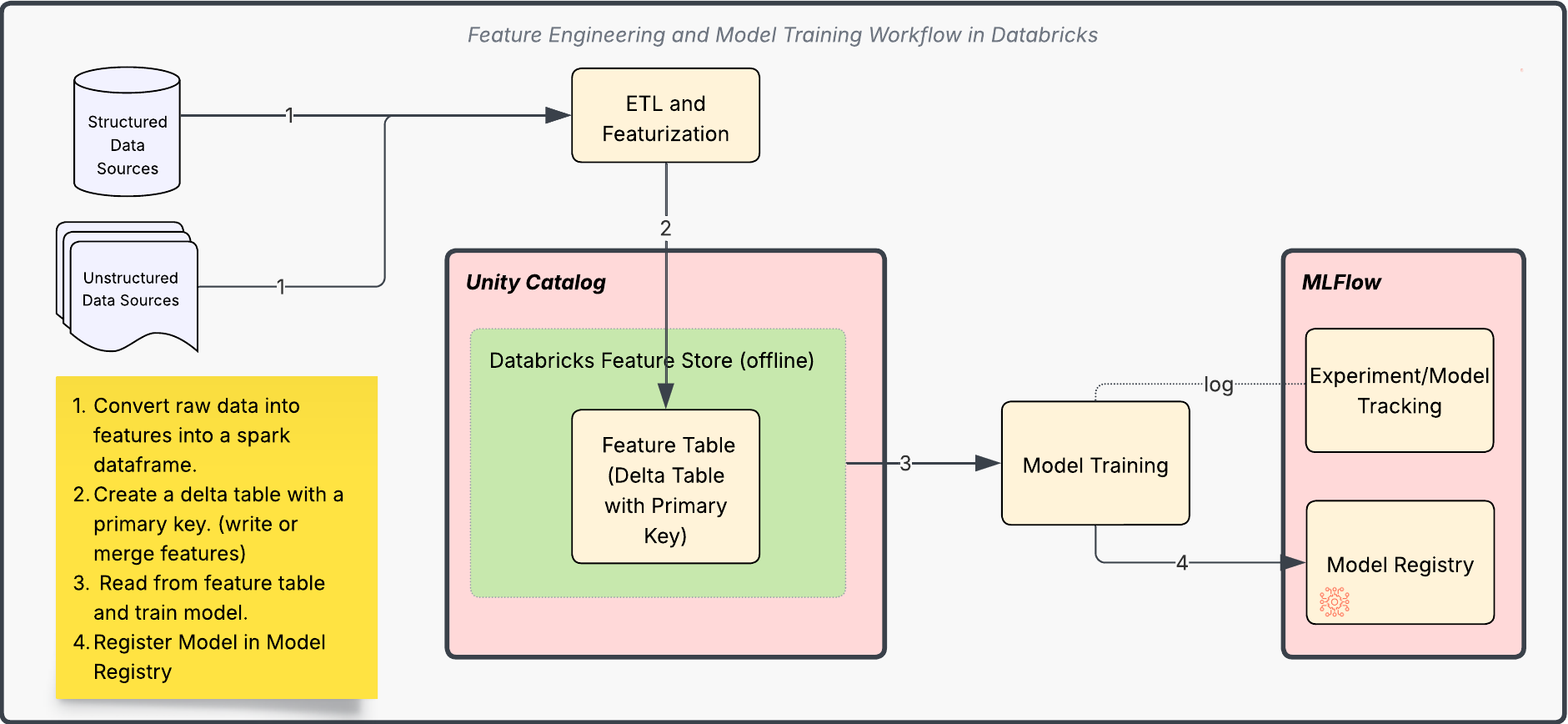
## Feature Management

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 | 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 | It is a high-performance, scalable solution for serving feature data to online applications and real-time machine learning models. | [Doc reference](https://docs.databricks.com/aws/en/machine-learning/feature-store/concepts#online-feature-store) |
| Feature Serving endpoints | 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.*



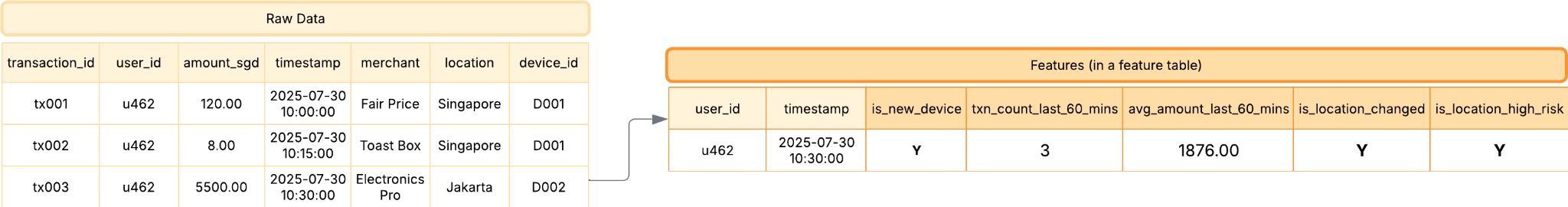
### Feature Engineering

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. |
| --- |

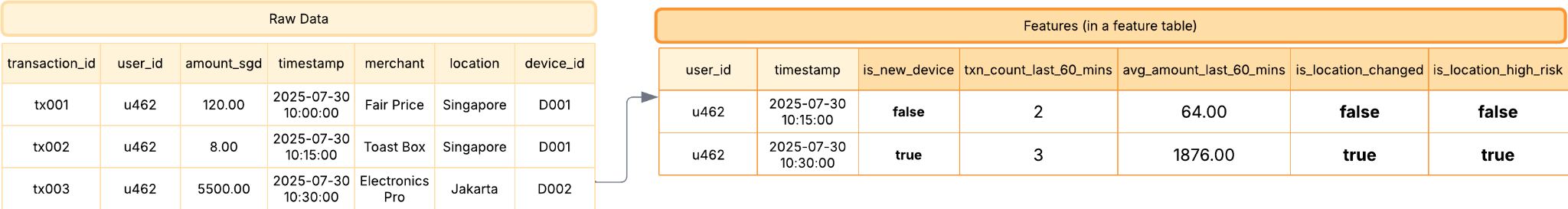
#### TimeSeries Feature Table

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 timeseries 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

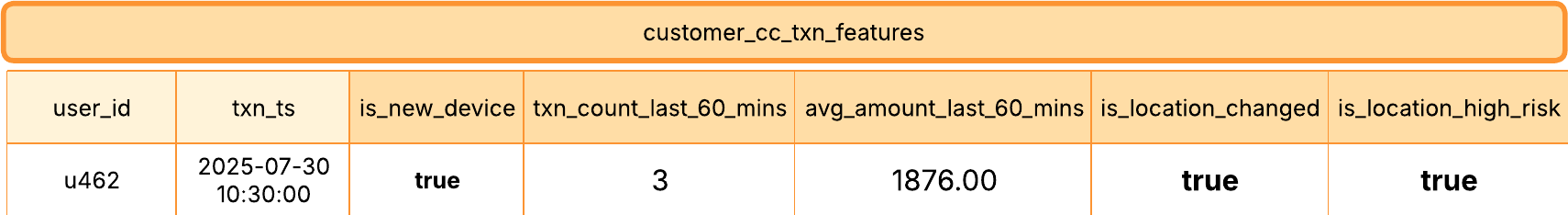
A feature table is a delta table with additional metadata w.r.t features to support model training and inference.

| ***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 use as a feature table, then 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))

### Online Feature Store

<TODO: Add relevant documentation>

## Model Training and Deployment

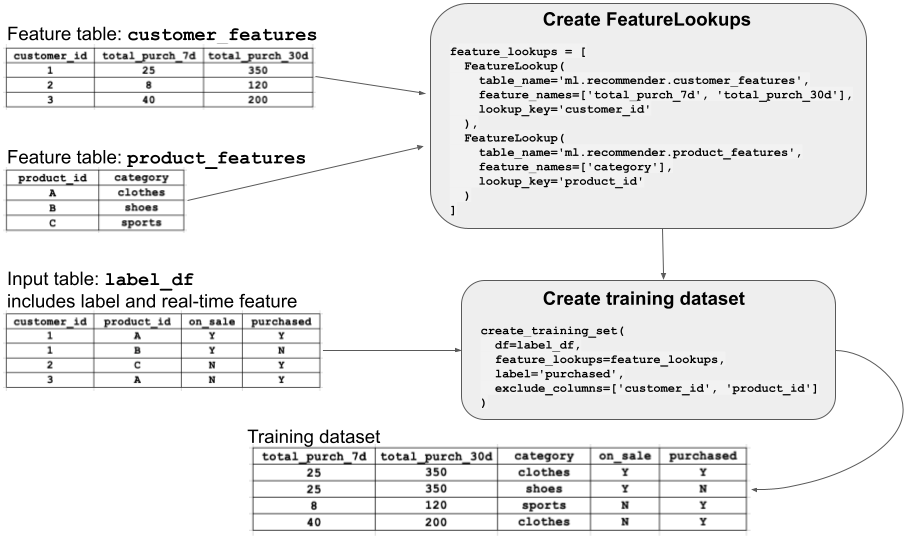
### Overview

This section outlines the process of model training and deployment in Azure Databricks while using tracking, logging, monitoring and registration in Databricks managed MLFlow.

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

### Feature Lookup

Azure Databricks Feature Store provides a centralized repository for feature definitions. We can retrieve pre-computed features for training and inference using `[FeatureLookup](https://docs.databricks.com/aws/en/machine-learning/feature-store/concepts#featurelookup)`.



In the above diagram, we want to train a model using features from two tables, i.e. customer\_features and product features. We create the FeatureLookup by specifying the name of the table and the feature columns that we want to lookup along with the lookup key to use when joining features with label\_df to create the training dataset.

Sample code outlined below:

from databricks.feature\_engineering import FeatureEngineeringClient, FeatureLookup

# The model training uses two features from the 'customer\_features' feature table and

# a single feature from 'product\_features'

feature\_lookups = [

FeatureLookup(

table\_name='ml.recommender\_system.customer\_features',

feature\_names=['total\_purchases\_30d', 'total\_purchases\_7d'],

lookup\_key='customer\_id'

),

FeatureLookup(

table\_name='ml.recommender\_system.product\_features',

feature\_names=['category'],

lookup\_key='product\_id'

)

]

fe = FeatureEngineeringClient()

# Create a training set using training DataFrame and features from Feature Store

# The training DataFrame must contain all lookup keys from the set of feature lookups,

# in this case 'customer\_id' and 'product\_id'. It must also contain all labels used

# for training, in this case 'rating'.

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

df=training\_df,

feature\_lookups=feature\_lookups,

label='rating',

exclude\_columns=['customer\_id', 'product\_id']

)

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

### Model Training

MLFlow is recommended for tracking experiments, model training and parameters.

For more details in MLFlow please refer to the databricks documentation [here](https://learn.microsoft.com/en-gb/azure/databricks/mlflow/).

Sample code for model training and tracking using MLFlow is 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)

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

fe.log\_model(

model=model,

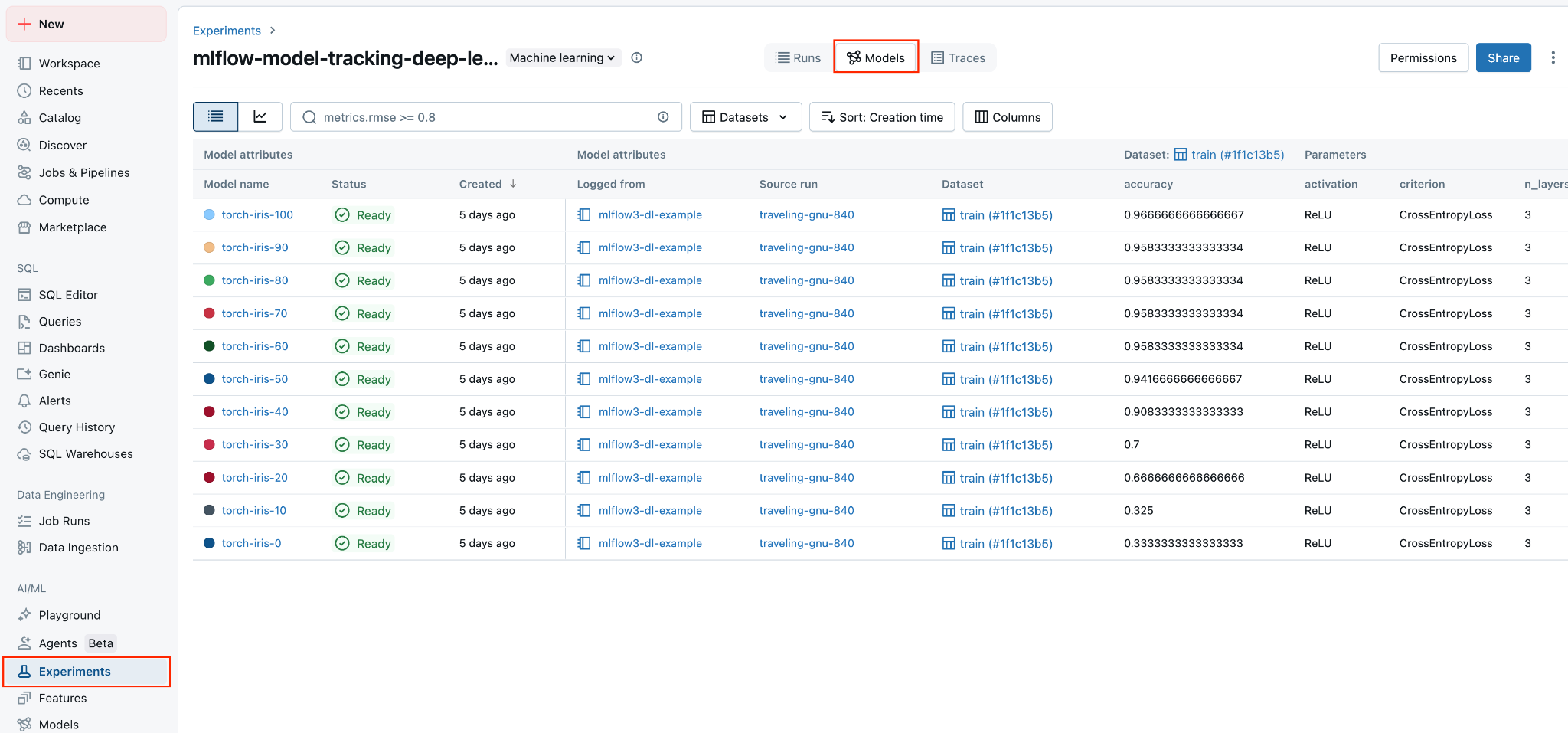
artifact\_path="model\_packaged",

flavor=mlflow.lightgbm,

training\_set=training\_set,

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

)

After the model training is run, you may track your experiment (for each training run in mlflow) as shown below:  


## Model Serving

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**.

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)

### 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

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.

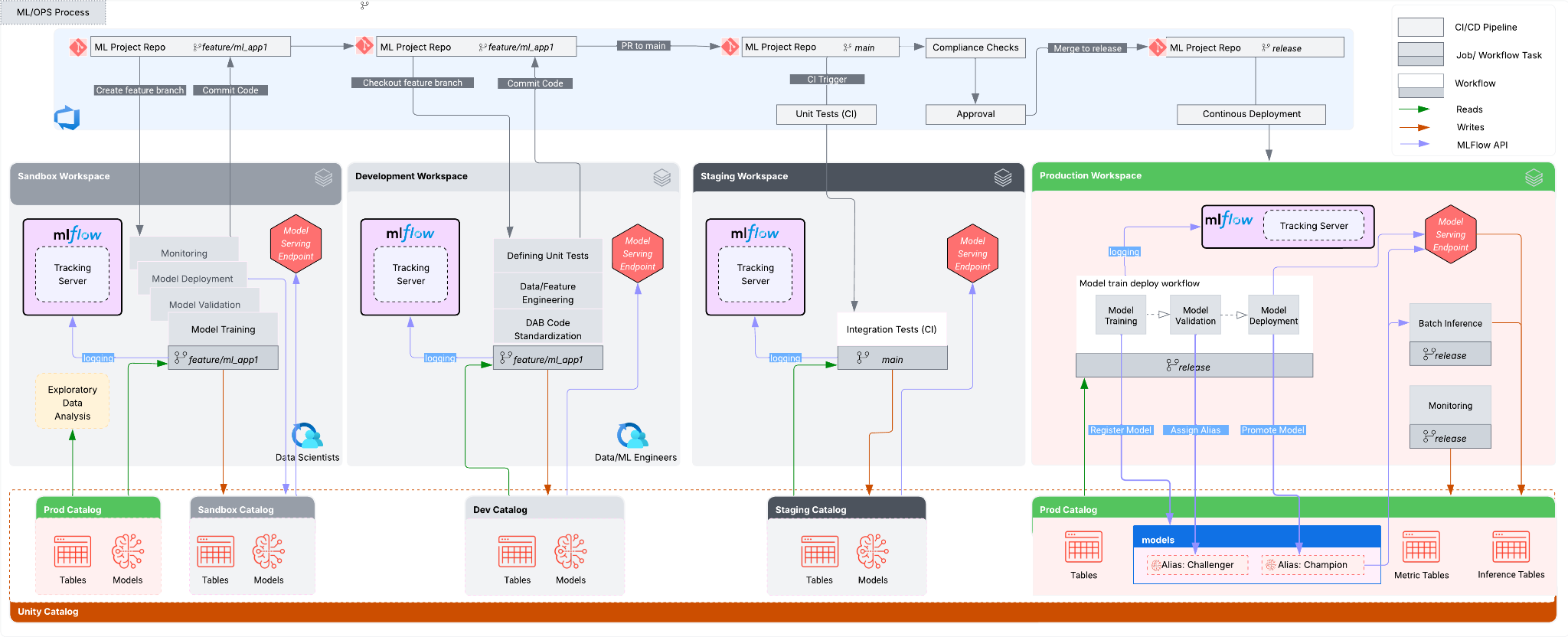
## Monitoring

<ToDo>

## 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

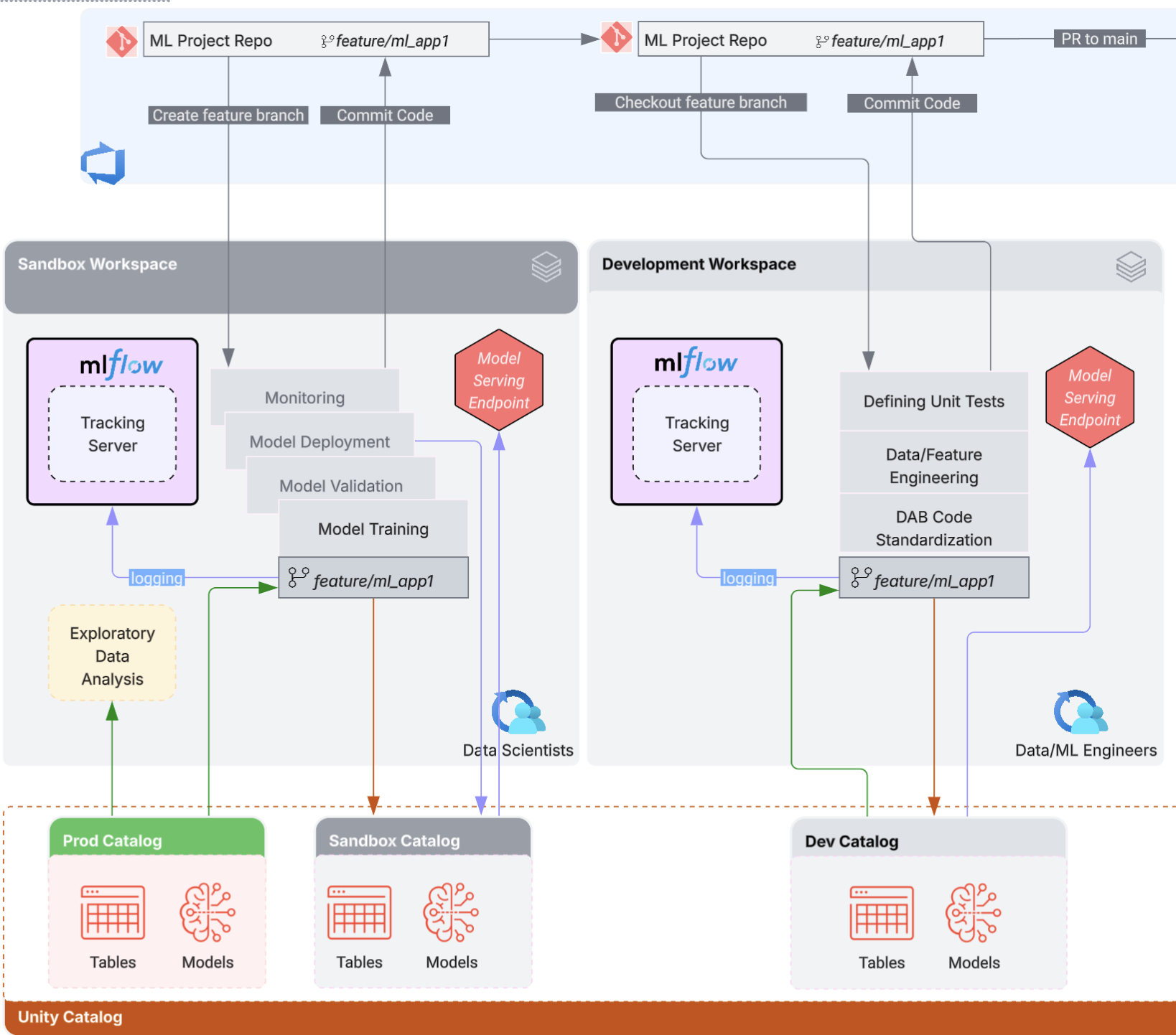
This section outlines the components identified for ML/OPS and outlines how they would be deployed across all environments, including Sandbox.

| ***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. | 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 lifecyle 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.

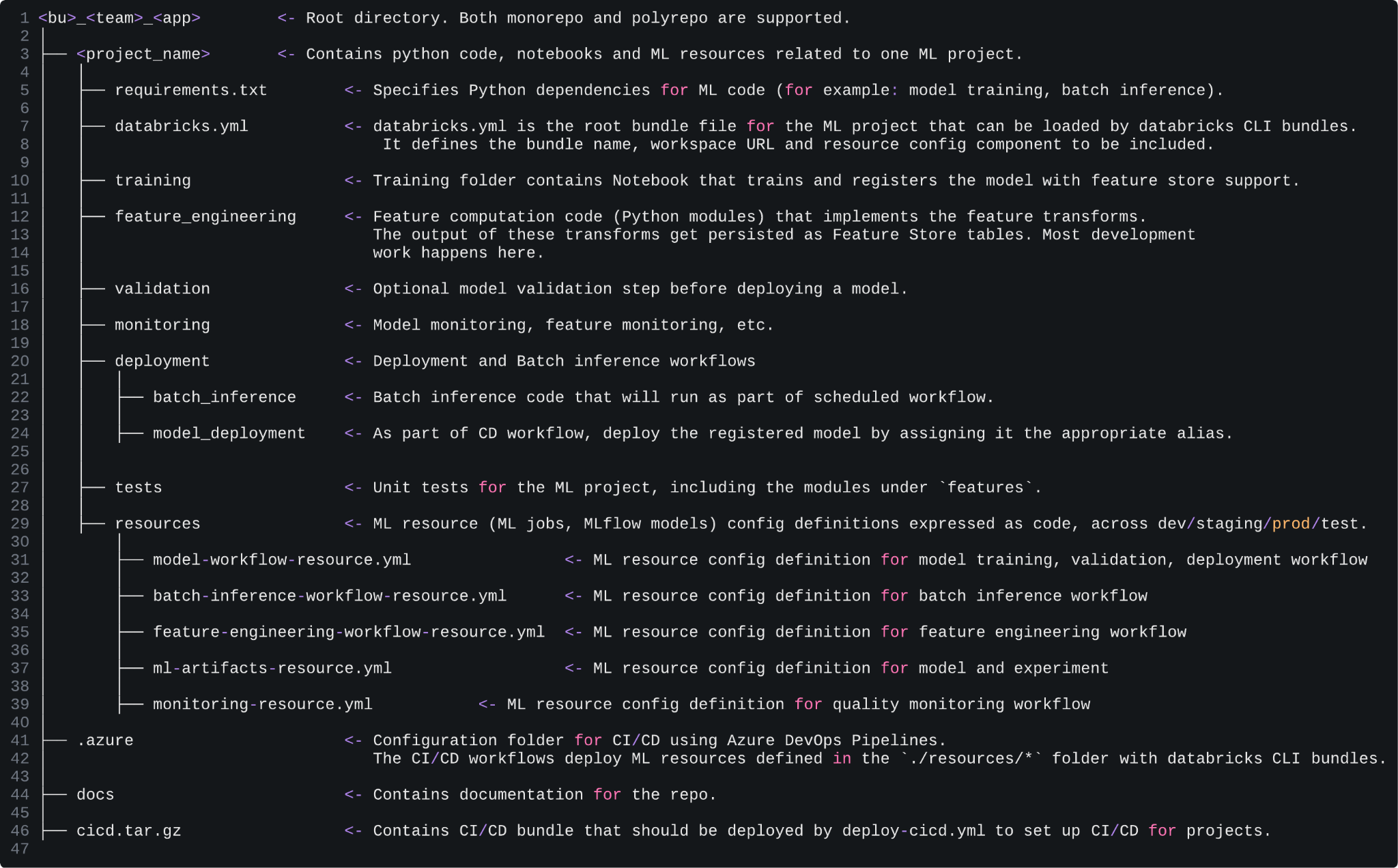
#### ML Project folder structure

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>

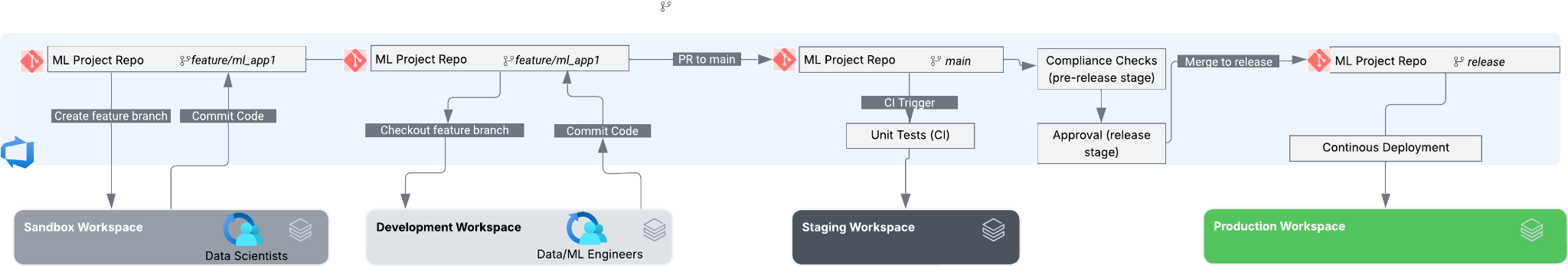
### Deployment Lifecycle: Azure Devops Pipelines

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 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) | * Staging environment shall be used for unit tests prior to deployment and also for running Integration Tests. * Unit Tests (CI) would be use-case specific and to be determined by the developers for their specific codebase or scenario. * 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. | Note:In the production environment, Data Scientists may need read-only access to:   * View test results and job logs. * Inspect model artifacts. * Check production pipeline status. * Review monitoring tables.   This would allow the data scientists to identify and debug issues in production and compare the performance of newer models vs models currently in production. |

# 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);

