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
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
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
NFR-05	Interoperability	The system should be able to provide: • Support for open standards • Plug and play for tools like MLFlow
NFR-06	Maintainability	The system should be able to provide: • Easy Upgrades, patching and system health checks • Modular architecture for faster upgrades
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	xxxx	xxxx	xxx

AD-01	xxxx
Category	xxxx
Description	
FRs/NFRs Addressed	
Options	**Option - 1**: **Option - 2**:
Justification	**Pros/Cons of Option-1:** ➕. ➕ ➖ ➖ ➖ **Pros/Cons of Option-2:** ➕ ➕ ➖
Databricks Recommendation	

Decision	

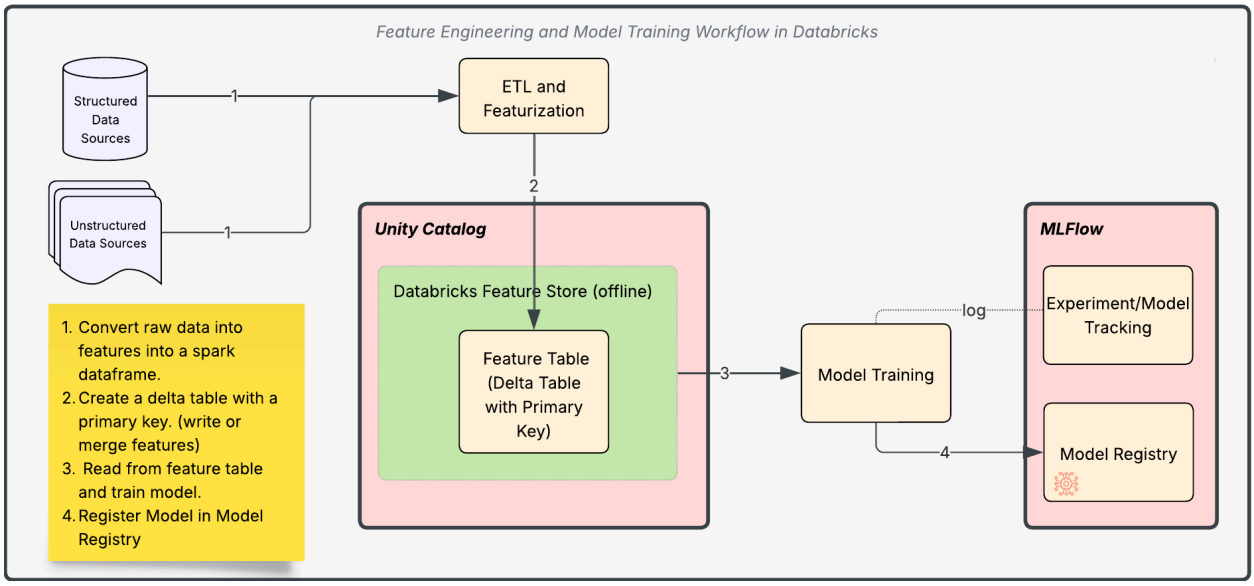
Design

Overview

This document 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. **CI/CD Process**: Outlines the CI/CD process for the end to end lifecycle.

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



Feature Management

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

Glossary of Terms

Term	Definition	Links
Feature Engineering	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	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	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
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
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

A feature store acts as a centralized repository where data scientists can find, share, and manage features. Its primary benefit is ensuring that the code used for computing feature values remains consistent between model training and inference

Features are organized into feature tables, and each feature table is backed by a Delta table.

Pre-requisites

- **Unity Catalog Metastore:** Your workspace **must be enabled for Unity Catalog**
- **Databricks Runtime:** Feature Engineering in Unity Catalog **requires Databricks Runtime 13.2 or above,** with the `TIMESERIES` keyword **requiring Databricks Runtime 13.3 LTS or above**

Creating Feature Tables

Create a feature table in Unity Catalog (ref)

You can use any Delta table with a primary key constraint as a feature table.

```
SQL
CREATE TABLE ml.recommender_system.customer_features (
  customer_id int NOT NULL,
  feat1 long,
  feat2 varchar(100),
  CONSTRAINT customer_features_pk PRIMARY KEY (customer_id)
);
-- To create a time series feature table, add a time column as a primary key
column and specify the TIMESERIES keyword.
CREATE TABLE ml.recommender_system.customer_features (
  customer_id int NOT NULL,
  ts timestamp NOT NULL,
  feat1 long,
  feat2 varchar(100),
  CONSTRAINT customer_features_pk PRIMARY KEY (customer_id, ts TIMESERIES)
);

-- Create a feature table in Unity Catalog with Lakeflow Declarative Pipelines
CREATE MATERIALIZED VIEW customer_features (
  customer_id int NOT NULL,
  feat1 long,
  feat2 varchar(100),
  CONSTRAINT customer_features_pk PRIMARY KEY (customer_id)
) AS SELECT * FROM ...;
```

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

If an existing Delta table does not have a primary key constraint, you can create one as follows:

```
SQL
-- 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)

Any streaming table or materialized view in Unity Catalog with a primary key can be a feature table

```
SQL
CREATE OR REFRESH MATERIALIZED VIEW existing_live_table(
  id int NOT NULL PRIMARY KEY,
  ...
) AS SELECT ...
```

Use an existing view in Unity Catalog as a feature table (ref)

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). A "simple SELECT view" means it's 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`.

Note: Feature tables backed by views do not appear in the Features UI and cannot be published to online stores

```
SQL
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;
```

General Criteria for Creating Feature Tables

- **Backed by Delta Table:** Feature tables are organized as feature tables and are **backed by a Delta table** and additional metadata
- **Primary Key:** Every feature table **must have a primary key**. This primary key can consist of one or more columns
    - If an existing Delta table does not have a primary key, you must update it by setting primary key columns to `NOT NULL` and then adding the primary key constraint using `ALTER TABLE` DDL statements
- **Supported Data Types**
    - The data types in the Delta table must be supported by Feature Engineering in Unity Catalog. Supported PySpark data types include `IntegerType`, `FloatType`, `BooleanType`, `StringType`, `DoubleType`, `LongType`, `TimestampType`, `DateType`, `ShortType`, `ArrayType`, `BinaryType`, `DecimalType`, `MapType`, and `StructType` (`ref`)
- **Namespace:** Feature tables are accessed using a three-level namespace: `<catalog-name>.<schema-name>.<table-name>`. You must have appropriate privileges (`CREATE CATALOG`, `USE CATALOG`, `CREATE SCHEMA`, `USE SCHEMA`, `CREATE TABLE`)
- **Performance-wise for time series feature table**: it's recommended to enable liquid clustering on the primary key and timestamp columns. (ref)

Feature Serving Endpoints

<TODO>

Limitations

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)
- **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)
  - **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)
  - **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)
- **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)
- **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)

Model Training and Inference

- **Table and Function Count:** A model can use at most **50 tables and 100 functions for training** (ref)
- A maximum of 100 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)

Model Training and Deployment

<outline>

- Feature Lookup/ Feature Function/ Model Training
- MLFlow experiments, model registration and model lifecycle

Model Serving

<outline>
- Batch Inference
- Real Time Serving

CI/CD Process

<outline>
- repeatable folder structure across projects for standardisation
- as per our MLOps Stacks DABs, have ML Engineers and DS own defined portions of the code base
- designing a clear git ops workflow for moving models from dev to staging to prod (unless this is already covered by UC design)
- Deployment process/ADO Pipeline

Appendix

Liquid Clustering (ref)

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

```sql
-- 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

```sql
-- 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);
```