Deep Dive: ML Tooling in the Platform

This document delves into how your enterprise-ready data platform provides a robust foundation for Machine Learning (ML) workloads. A successful ML initiative relies heavily on high-quality, accessible data and efficient processes for feature engineering, model training, and inference. Your platform, with its integrated components, is designed to support the entire ML lifecycle, from raw data ingestion to delivering predictions.

1. Core Concepts: Data Readiness for ML

At its heart, any ML process requires structured, clean, and consistent data. Your data platform facilitates this through:

- **Curated Data Lakehouse (Delta Lake on MinIO):** This is the primary storage layer for ML. Data is transformed, cleaned, and enriched here, providing a reliable and versioned source of truth for features and labels. The ACID properties of Delta Lake ensure data quality and reproducibility for ML experiments.
- **Feature Engineering with Apache Spark:** Spark's distributed processing capabilities are ideal for transforming raw data into features at scale, handling large volumes and complex logic.
- **Operational Databases (PostgreSQL, MongoDB):** These can serve specific roles, such as storing lookup tables for real-time feature enrichment, or even acting as a simple "feature store" for frequently accessed, low-latency features.
- **Data Lineage (Spline, OpenMetadata):** Understanding where your ML features come from and how they were transformed is critical for debugging, reproducibility, and compliance. OpenMetadata, enriched by Spline, provides this transparency.
- **Observability (OpenTelemetry, Grafana Alloy, Grafana):** Monitoring the health and performance of your ML pipelines, including data quality, model inference latency, and drift, is crucial for MLOps.

2. ML Lifecycle Stages in the Platform

Let's map the key stages of an ML lifecycle to how they are supported by your platform components:

- **Data Ingestion (FastAPI, Kafka):** Raw data (e.g., transaction events) enters the platform via FastAPI, is buffered by Kafka, and lands in the raw data lake (MinIO/Delta Lake). This forms the basis for feature extraction.
- **Feature Engineering (Apache Spark, Delta Lake):** Raw data is transformed and aggregated into features. Spark processes these large datasets, and the results are stored in curated Delta Lake tables.
- **Model Training (Apache Spark MLlib / External ML Frameworks):** While a full ML training cluster setup is beyond the scope of this local environment, Spark MLlib can perform distributed training. For more complex models (e.g., deep learning), data can be exported from Delta Lake to specialized training environments.

- **Model Inference (Apache Spark Structured Streaming / FastAPI + PostgreSQL/MongoDB):**
  - **Batch/Streaming Inference:** Spark Structured Streaming can load trained models and apply them to new incoming data streams in near real-time.
  - **Real-time Inference (APIs):** A separate FastAPI service (or an extension of the existing one) can serve real-time predictions, often backed by pre-computed features in a database or a dedicated feature store.
- **Model Monitoring (OpenTelemetry, Grafana, OpenMetadata):**
  - **Operational Monitoring:** OpenTelemetry and Grafana track the performance and resource utilization of inference services and feature engineering pipelines.
  - **Data/Model Drift:** Data quality checks on features and output predictions (tracked via OpenMetadata profiling) can help detect data drift or concept drift.

3. Interactive How-Tos: Leveraging ML Tooling

Let's explore some practical examples of how ML workflows can be built using your platform.

Basic Use Case: Feature Engineering with Spark

Objective: To demonstrate how Apache Spark can be used to perform basic feature engineering, transforming raw financial transactions into aggregated features suitable for an ML model (e.g., daily transaction counts per account, average amount).
Role in Platform: Create derived features from raw or intermediate data, making it ready for model training and inference.
Setup/Configuration (Local Environment - Advanced Track):

1. **Ensure all Advanced Track services are running:** docker compose up --build -d. This includes spark and minio.
2. **Ensure raw-data-bucket/financial_data_delta is populated:** Your streaming_consumer.py job should have written some data to this path.
3. **Create a PySpark script for feature engineering:** In pyspark_jobs/, create feature_engineering_job.py.

```
# pyspark_jobs/feature_engineering_job.py
import sys
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, sum, avg, to_date, lit, current_timestamp
from pyspark.sql.types import StructType, StringType, FloatType, TimestampType
from delta.tables import DeltaTable # Ensure DeltaTable import if writing to Delta

def create_spark_session(app_name):
    """Helper function to create a SparkSession with Delta Lake packages."""
    return (SparkSession.builder.appName(app_name)
        .config("spark.jars.packages", "io.delta:delta-core_2.12:2.4.0")
        .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
        .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")
        .getOrCreate())

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: feature_engineering_job.py <input_delta_path>
<output_delta_path>")
        sys.exit(-1)

    input_delta_path = sys.argv[1]
    output_delta_path = sys.argv[2]

    spark = create_spark_session("FinancialFeatureEngineering")
    spark.sparkContext.setLogLevel("WARN")

    print(f"Reading raw financial data from: {input_delta_path}")
    try:
        df_raw = spark.read.format("delta").load(input_delta_path)
        df_raw.printSchema()
        df_raw.show(5, truncate=False)
    except Exception as e:
        print(f"Error reading input Delta Lake: {e}. Ensure data exists at
{input_delta_path}")
        spark.stop()
        sys.exit(-1)

    print("Performing feature engineering: daily aggregates per account...")
    # Convert timestamp string to actual timestamp, then to date
    df_features = df_raw.withColumn("transaction_date", to_date(col("timestamp"),
"yyyy-MM-dd'T'HH:mm:ss'Z'")) \
                    .groupBy("account_id", "transaction_date") \
                    .agg(
                        count(col("transaction_id")).alias("daily_transaction_count"),
                        sum(col("amount")).alias("daily_total_amount"),
                        avg(col("amount")).alias("daily_average_amount")
                    ) \
                    .withColumn("feature_created_at", current_timestamp())

    print("Schema of engineered features:")
    df_features.printSchema()
    df_features.show(5, truncate=False)

    # Write the engineered features to a new curated Delta Lake table
    print(f"Writing engineered features to: {output_delta_path}")
    df_features.write.format("delta") \
            .mode("overwrite") \
            .option("overwriteSchema", "true") \
            .save(output_delta_path)
    print("Feature engineering job completed.")

    spark.stop()
```

Steps to Exercise:

1. **Ensure raw-data-bucket/financial_data_delta has data.** If not, run python3 simulate_data.py and then trigger streaming_consumer.py from your financial_data_lake_pipeline Airflow DAG.
2. **Submit the Spark Feature Engineering Job:**
   ```
   docker exec -it spark spark-submit \
     --packages io.delta:delta-core_2.12:2.4.0 \
     --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
     --conf
   spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
     --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
     --conf spark.hadoop.fs.s3a.access.key=minioadmin \
     --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
     --conf spark.hadoop.fs.s3a.path.style.access=true \
     /opt/bitnami/spark/jobs/feature_engineering_job.py \
     s3a://raw-data-bucket/financial_data_delta \
     s3a://curated-data-bucket/financial_features_daily_account
   ```
3. **Monitor Spark Job:** Observe the console output for Spark logs, confirming the reading, aggregation, and writing process.
4. **Verify Features in MinIO:** Access the MinIO Console (http://localhost:9001). Navigate to curated-data-bucket/financial_features_daily_account/. You should see new Delta Lake files.
5. **Query Engineered Features via Spark SQL:**
   ```
   docker exec -it spark spark-sql \
     --packages io.delta:delta-core_2.12:2.4.0 \
     --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
     --conf
   spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
     --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
     --conf spark.hadoop.fs.s3a.access.key=minioadmin \
     --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
     --conf spark.hadoop.fs.s3a.path.style.access=true \
     -e "SELECT * FROM
   ```

```
delta.\`s3a://curated-data-bucket/financial_features_daily_account\` LIMIT 10;"
```

Verification:

- **Spark Job Completion:** The Spark job executes successfully, indicated by the console output.
- **MinIO Contents:** The financial_features_daily_account Delta Lake table is created with new .parquet files and _delta_log.
- **Spark SQL Query:** The query results show aggregated data, confirming that the raw transaction data has been transformed into daily account-level features.

Advanced Use Case 1: Model Training with PySpark MLlib (Conceptual)

Objective: To conceptually demonstrate how a Spark job could train a simple ML model (e.g., Logistic Regression for fraud detection) using PySpark MLlib on the curated features.
Role in Platform: Perform large-scale, distributed model training directly on your data lakehouse data, leveraging Spark's optimized algorithms.
Setup/Configuration:

1. **Ensure curated features are available:** The financial_features_daily_account Delta Lake table (from the previous Basic Use Case) should be populated.
2. **Create a PySpark script for model training:** In pyspark_jobs/, create model_training_job.py.

```
# pyspark_jobs/model_training_job.py
import sys
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lit, rand, monotonically_increasing_id
from pyspark.ml.feature import VectorAssembler, StandardScaler # For feature
preprocessing
from pyspark.ml.classification import LogisticRegression # Example ML algorithm
from pyspark.ml import Pipeline
from delta.tables import DeltaTable # Ensure DeltaTable import

def create_spark_session(app_name):
  """Helper function to create a SparkSession with Delta Lake packages."""
  return (SparkSession.builder.appName(app_name)
      .config("spark.jars.packages", "io.delta:delta-core_2.12:2.4.0")
      .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
      .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")
      .getOrCreate())

if __name__ == "__main__":
  if len(sys.argv) != 3:
      print("Usage: model_training_job.py <input_features_path> <model_output_path>")
      sys.exit(-1)

  input_features_path = sys.argv[1]
  model_output_path = sys.argv[2]

  spark = create_spark_session("MLModelTraining")
  spark.sparkContext.setLogLevel("WARN")

  print(f"Reading engineered features from: {input_features_path}")
  try:
      # For demonstration, let's assume a 'label' column exists for training.
      # In a real scenario, this 'label' would come from historical labeled data.
      df_features = spark.read.format("delta").load(input_features_path) \
                          .withColumn("id", monotonically_increasing_id()) # Add unique ID for
partitioning

      # Simulate a 'label' column (e.g., 'is_fraud' or 'high_risk') for demo purposes
      # In a real scenario, this label would come from your actual labeled dataset.
      df_labeled_features = df_features.withColumn("label",
when(col("daily_total_amount") > 5000 and rand() < 0.2, 1.0).otherwise(0.0))

      df_labeled_features.printSchema()
      df_labeled_features.show(5, truncate=False)

  except Exception as e:
      print(f"Error reading input features Delta Lake: {e}. Ensure data exists at
{input_features_path}")
      spark.stop()
      sys.exit(-1)

  # Prepare features for MLlib
  # Assume numerical features are 'daily_transaction_count', 'daily_total_amount',
'daily_average_amount'
  feature_columns = ["daily_transaction_count", "daily_total_amount",
"daily_average_amount"]

  # Assemble features into a vector
  assembler = VectorAssembler(inputCols=feature_columns,
outputCol="raw_features")

  # Scale features (optional, but good practice for many ML algorithms)
  scaler = StandardScaler(inputCol="raw_features", outputCol="features",
withStd=True, withMean=False)

  # Train a Logistic Regression model (example classifier)
  lr = LogisticRegression(labelCol="label", featuresCol="features", maxIter=10)

  # Create a Pipeline to chain preprocessing and model training
  pipeline = Pipeline(stages=[assembler, scaler, lr])

  print("Training ML model...")
  # Split data into training and test sets (conceptual)
  (trainingData, testData) = df_labeled_features.randomSplit([0.8, 0.2], seed=42)

  # Fit the model
  model = pipeline.fit(trainingData)
  print("Model training completed.")

  # Evaluate the model (conceptual)
  # predictions = model.transform(testData)
  # evaluator = BinaryClassificationEvaluator(labelCol="label")
  # auc = evaluator.evaluate(predictions)
  # print(f"Area under ROC: {auc}")

  # Save the trained model
  print(f"Saving trained model to: {model_output_path}")
  model.save(model_output_path)
  print("Model saved.")

  spark.stop()
```

Steps to Exercise (Conceptual):

1. **Ensure financial_features_daily_account is populated.**
2. **Submit the Spark Model Training Job:**

```
docker exec -it spark spark-submit \
  --packages io.delta:delta-core_2.12:2.4.0 \
  --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
  --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
  --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
  --conf spark.hadoop.fs.s3a.access.key=minioadmin \
  --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
  --conf spark.hadoop.fs.s3a.path.style.access=true \
  /opt/bitnami/spark/jobs/model_training_job.py \
  s3a://curated-data-bucket/financial_features_daily_account \
```

s3a://models-bucket/financial_fraud_model

3. **Monitor Spark Job:** Observe the console output for messages indicating model training and saving.
4. **Verify Model in MinIO:** Check http://localhost:9001 in the models-bucket/financial_fraud_model/ path. You should see Spark MLlib's serialized model files.

Verification:

- **Spark Job Completion:** The job runs and indicates "Model saved."
- **MinIO Contents:** The specified MinIO path contains the serialized Spark MLlib model, demonstrating the platform's capability to store trained models.

Advanced Use Case 2: Model Inference with Spark Structured Streaming

Objective: To demonstrate how a trained ML model (from MinIO) can be loaded by a Spark Structured Streaming job and applied to new incoming data (e.g., from Kafka or a raw Delta table) for real-time or near-real-time inference.
Role in Platform: Deploy ML models to production for continuous scoring and anomaly detection on live data streams.
Setup/Configuration:

1. **Ensure trained model is available:** s3a://models-bucket/financial_fraud_model should be populated (from previous use case).
2. **Ensure raw data source is active:** raw-data-bucket/financial_data_delta should be continuously receiving data from Kafka via streaming_consumer.py or you can adjust to read directly from Kafka if preferred.
3. **Create a PySpark script for streaming inference:** In pyspark_jobs/, create streaming_inference_job.py.

```
# pyspark_jobs/streaming_inference_job.py
import sys
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, from_json, current_timestamp
from pyspark.sql.types import StructType, StringType, FloatType, TimestampType, BooleanType
from pyspark.ml import PipelineModel # To load the saved pipeline model
from pyspark.ml.feature import VectorAssembler, StandardScaler # Necessary for schema of loaded model
from pyspark.ml.classification import LogisticRegression # Necessary for schema of loaded model

def create_spark_session(app_name):
    """Helper function to create a SparkSession with Delta Lake and Kafka packages."""
    return (SparkSession.builder.appName(app_name)
        .config("spark.jars.packages", "io.delta:delta-core_2.12:2.4.0")
        .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
        .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")
        .getOrCreate())

if __name__ == "__main__":
    if len(sys.argv) != 4:
        print("Usage: streaming_inference_job.py <input_delta_path> <model_path>
<output_delta_path>")
        sys.exit(-1)

    input_delta_path = sys.argv[1]
    model_path = sys.argv[2]
    output_delta_path = sys.argv[3]

    spark = create_spark_session("StreamingModelInference")
    spark.sparkContext.setLogLevel("WARN")

    print(f"Loading trained model from: {model_path}")
    try:
        loaded_model = PipelineModel.load(model_path)
        print("Model loaded successfully.")
    except Exception as e:
        print(f"Error loading model from {model_path}: {e}. Ensure model exists and is
valid.")
        spark.stop()
        sys.exit(-1)

    # Define schema for the incoming streaming data (should match source raw data)
    data_schema = StructType() \
        .add("transaction_id", StringType(), True) \
        .add("timestamp", StringType(), True) \
        .add("account_id", StringType(), True) \
        .add("amount", FloatType(), True) \
        .add("currency", StringType(), True) \
        .add("transaction_type", StringType(), True) \
        .add("merchant_id", StringType(), True) \
        .add("category", StringType(), True) \
        .add("is_flagged", BooleanType(), True)


    print(f"Reading streaming data from: {input_delta_path}")
    # Read from Delta Lake as a streaming DataFrame
    # For a truly 'real-time' demo, this could read directly from Kafka
    streaming_df = (spark.readStream
                .format("delta")
                .option("startingOffsets", "latest") # Start consuming new data
                .load(input_delta_path))

    # Basic feature preparation (must match preprocessing in training pipeline)
    # The model expects a 'features' column as input, created by
VectorAssembler/StandardScaler
    # In a real scenario, you would extract/compute these features from the incoming
data
    # based on the model's requirements. For this simple demo, we'll recreate the feature
vector
    # structure expected by the loaded model.

    feature_columns = ["amount"] # Simplified feature for quick demo.
                            # In real life, this would be daily_transaction_count etc.
                            # and would require complex stateful processing or lookup for live
data.

    # To make this example runnable with simple raw data, we will just pass 'amount' as
the feature.
    # In a true scenario, you'd apply the same feature engineering logic as the training
job.
    # This requires more complex stateful streaming or a feature store.
    # For now, let's just make sure the 'features' vector is created from available data.

    # If your model was trained on 'daily_transaction_count', 'daily_total_amount',
'daily_average_amount',
    # you'd need to compute these in a streaming fashion or join with a feature store.
    # For simplicity of this demo, we use a single numerical feature present in raw data.

    # IMPORTANT: The schema of the DataFrame passed to `model.transform` must
exactly
    # match the input schema expected by the `PipelineModel`.
    # This means the 'raw_features' and 'features' columns created by the assembler and
scaler
    # in the training pipeline must be recreated here.
    # For a demo, let's simplify to directly use 'amount' as the input feature for now,
    # assuming the model can handle it, or adjust the model training to use simple
features.
```

```
  # Let's assume the model was trained on a single feature 'amount' for simplicity of
streaming demo
  # (This is a simplification for a real-world complex model)
  assembler = VectorAssembler(inputCols=["amount"], outputCol="features") # Adjust
to actual features used in training

  df_with_features = assembler.transform(streaming_df)

  print("Applying model inference to streaming data...")
  # Apply the loaded model to the streaming data
  predictions_df = loaded_model.transform(df_with_features) \
                     .withColumn("inference_timestamp", current_timestamp()) \
                     .withColumn("prediction_score", col("probability")[1]) # Get score for
positive class


  # Select relevant columns for the output
  output_df = predictions_df.select(
    col("transaction_id"),
    col("timestamp"),
    col("account_id"),
    col("amount"),
    col("currency"),
    col("transaction_type"),
    col("prediction").alias("is_fraud_prediction"), # Binary prediction (0 or 1)
    col("prediction_score"), # Probability score for fraud
    col("inference_timestamp")
  )

  # Define checkpoint location for fault tolerance
  checkpoint_location = f"{output_delta_path}/_checkpoints"

  # Write predictions to a new Delta Lake table
  query = (output_df.writeStream
      .format("delta")
      .outputMode("append") # Append new predictions
      .option("checkpointLocation", checkpoint_location)
      .option("mergeSchema", "true") # Enable schema evolution
      .start(output_delta_path))

  print(f"Spark Structured Streaming inference job started, writing predictions to:
{output_delta_path}")
  print(f"Checkpoint location: {checkpoint_location}")
```

```
    query.awaitTermination()
    spark.stop()
```

Steps to Exercise:

1. **Ensure financial_fraud_model is in MinIO.**
2. **Ensure raw-data-bucket/financial_data_delta is receiving data.** (Run simulate_data.py and the streaming_consumer.py or the financial_data_lake_pipeline DAG).
3. **Submit the Spark Streaming Inference Job:**
```
docker exec -it spark spark-submit \
    --packages io.delta:delta-core_2.12:2.4.0 \
    --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
    --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
    --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
    --conf spark.hadoop.fs.s3a.access.key=minioadmin \
    --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
    --conf spark.hadoop.fs.s3a.path.style.access=true \
    /opt/bitnami/spark/jobs/streaming_inference_job.py \
    s3a://raw-data-bucket/financial_data_delta \
    s3a://models-bucket/financial_fraud_model \
    s3a://predictions-bucket/financial_fraud_predictions_stream
```

4. **Monitor Spark Job:** Observe the console output for messages confirming model loading and stream processing.
5. **Verify Predictions in MinIO:** Access http://localhost:9001. Navigate to predictions-bucket/financial_fraud_predictions_stream/. You should see new Delta Lake files.
6. **Query Predictions via Spark SQL:**
```
docker exec -it spark spark-sql \
    --packages io.delta:delta-core_2.12:2.4.0 \
    --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension \
    --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
    --conf spark.hadoop.fs.s3a.endpoint=http://minio:9000 \
    --conf spark.hadoop.fs.s3a.access.key=minioadmin \
    --conf spark.hadoop.fs.s3a.secret.key=minioadmin \
    --conf spark.hadoop.fs.s3a.path.style.access=true \
    -e "SELECT transaction_id, amount, is_fraud_prediction, prediction_score,
inference_timestamp FROM
delta.\`s3a://predictions-bucket/financial_fraud_predictions_stream\` LIMIT 10;"
```

Verification:

- **Spark Streaming Job:** The job runs continuously, processing new data and applying the model.
- **MinIO Contents:** The financial_fraud_predictions_stream Delta Lake table is continuously updated with predictions.
- **Spark SQL Query:** The query results show the original transaction data augmented with is_fraud_prediction and prediction_score, demonstrating successful real-time model inference.

Advanced Use Case 3: Simple Feature Store Integration (Conceptual with PostgreSQL)

Objective: To conceptually demonstrate how PostgreSQL can act as a lightweight "feature store" for pre-computed, low-latency features that can be quickly retrieved by real-time inference services (e.g., another FastAPI application).
Role in Platform: Serve pre-computed, frequently accessed features for low-latency online inference, decoupling feature computation from real-time serving.
Setup/Configuration (Conceptual Discussion):

1. **Feature Computation (Spark Job):** A batch Spark job (similar to feature_engineering_job.py) would compute features and then *write them to PostgreSQL* instead of (or in addition to) Delta Lake.
   - Example: Daily aggregated features per account_id could be stored in a PostgreSQL table.
2. **PostgreSQL Table for Features:**
   ```
   CREATE TABLE IF NOT EXISTS daily_account_features (
     account_id VARCHAR(255) PRIMARY KEY,
     feature_date DATE,
     daily_transaction_count INTEGER,
     daily_total_amount NUMERIC,
     daily_average_amount NUMERIC,
     last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP
   );
   ```
   Spark would MERGE INTO this table for upserts.
3. **Real-time Inference Service (FastAPI):** A new or existing FastAPI endpoint would receive an account_id and query PostgreSQL for its latest features.
   Example FastAPI endpoint (conceptual):
   ```
   # fastapi_app/app/main.py (conceptual addition for real-time inference)
   # ... other imports ...
   import psycopg2 # For PostgreSQL connection
   from pydantic import BaseModel
   from typing import Dict, Any
   ```

```
# Assuming pg_conn is initialized on startup as in Starter Track example
# ...

class AccountFeaturesRequest(BaseModel):
    account_id: str

class AccountFeaturesResponse(BaseModel):
    account_id: str
    features: Dict[str, Any]
    status: str

@app.post("/get-account-features/", response_model=AccountFeaturesResponse,
tags=["ML Features"])
async def get_account_features(request: AccountFeaturesRequest):
    if not pg_conn:
        raise HTTPException(status_code=503, detail="Database connection not
available.")

    try:
        with pg_conn.cursor() as cur:
            cur.execute(
                """
                SELECT daily_transaction_count, daily_total_amount, daily_average_amount,
feature_date
                FROM daily_account_features
                WHERE account_id = %s
                ORDER BY feature_date DESC LIMIT 1;
                """,
                (request.account_id,)
            )
            result = cur.fetchone()

        if result:
            features = {
                "daily_transaction_count": result[0],
                "daily_total_amount": float(result[1]), # Convert Decimal to float
                "daily_average_amount": float(result[2]),
                "feature_date": result[3].isoformat()
            }
            return AccountFeaturesResponse(account_id=request.account_id,
features=features, status="found")
        else:
            return AccountFeaturesResponse(account_id=request.account_id, features={},
status="not_found")
    except Exception as e:
        print(f"Error fetching features from PostgreSQL: {e}")
        raise HTTPException(status_code=500, detail=f"Failed to fetch features: {e}")
```

Steps to Exercise (Conceptual/Discussion):

1. **Discuss Feature Generation and Load to PG:**
   - Explain how the feature_engineering_job.py could be modified to write its output (df_features) to the daily_account_features PostgreSQL table using Spark's JDBC connector (similar to how batch_transformations.py reads from PG).
2. **Discuss Real-time Feature Lookup:**
   - Explain how an API call to /get-account-features/ with an account_id would trigger a quick lookup in the PostgreSQL daily_account_features table.
   - Emphasize the low latency for retrieving pre-computed features, ideal for augmenting real-time inference requests.
3. **Explain the "Online" vs. "Offline" Feature Store Concept:**
   - **Online Feature Store (PostgreSQL in this case):** Optimized for low-latency reads during online inference.
   - **Offline Feature Store (Delta Lake in MinIO):** Optimized for large-scale batch reads during model training and batch inference.
   - The platform supports both paradigms.

Verification (Conceptual):

- **Architectural Understanding:** Demonstrate how the platform can support a simple feature store pattern using existing components, providing a clear path for more sophisticated MLOps patterns. This highlights the flexibility of the platform to integrate different data access patterns for ML.

4. Integrations and Future Enhancements for MLOps

While your current platform provides core ML capabilities, a full MLOps (Machine Learning Operations) setup would involve further integrations:

- **MLflow:** For tracking ML experiments (parameters, metrics, models), managing model versions, and deploying models.
- **Kubeflow / Airflow for Orchestration:** Complex ML pipelines (data prep, training, evaluation, deployment) can be orchestrated by Airflow. For Kubernetes-native environments, Kubeflow Pipelines would be an option.
- **Model Registry:** A centralized repository for managing the lifecycle of ML models, from development to production deployment.
- **Monitoring Model Drift:** Extending Grafana dashboards with custom metrics and alerts to detect concept drift (model performance degrades due to changes in data distribution) or data drift (changes in input feature distributions). OpenMetadata's profiling capabilities are a good starting point for data quality monitoring that feeds into

drift detection.
- **Automated Retraining:** Setting up Airflow DAGs to automatically trigger model retraining when performance degrades or data drift is detected.
- **CI/CD for ML Models:** Applying the existing CI/CD principles (from IaC & CI/CD Recipes addendum) to ML models, ensuring reproducible builds and automated deployment of new model versions.

Your current platform provides the essential data infrastructure (ingestion, storage, processing, basic observability) that is prerequisite for implementing these advanced MLOps practices.

This concludes the deep dive into ML Tooling in your platform.