Highlighting Apache Spark: Distributed Processing Engine

Apache Spark is a powerful, unified analytics engine for large-scale data processing. It excels at both batch processing and real-time streaming analytics, making it a cornerstone of modern data platforms. With its rich APIs in Python (PySpark), Scala, Java, and R, Spark allows engineers to perform complex transformations, aggregations, and machine learning tasks on massive datasets.

This guide will demonstrate basic and advanced use cases of Apache Spark, leveraging your **Advanced Track** local environment setup.

Reference: This guide builds upon the concepts and setup described in **Section 4.2. Core Technology Deep Dive** of the **Core Handbook** and the **Progressive Path Setup Guide Deep-Dive Addendum**.

Basic Use Case: Streaming ETL from Kafka to Delta Lake

Objective: To demonstrate how Spark Structured Streaming can consume real-time data from Kafka topics, apply a basic ETL process (e.g., parsing, schema enforcement), and write the results to a Delta Lake table in MinIO.

Role in Platform: Power the real-time ingestion pipeline, transforming raw event streams into structured, queryable data in the data lakehouse.

Setup/Configuration (Local Environment - Advanced Track):

1. **Ensure all Advanced Track services are running:** docker compose up --build -d from your project root.
2. **Verify Spark is accessible:** Check Docker logs for the spark container to ensure it's healthy. Access Spark History Server at http://localhost:18080.
3. **Ensure Kafka topics are initialized:** Confirm raw_financial_transactions and raw_insurance_claims topics exist (from onboard.sh or manual commands).
4. **simulate_data.py is running:** Ensure data is continuously being sent to your FastAPI and then published to Kafka.
5. **Spark Consumer Script:** You will use the pyspark_jobs/streaming_consumer.py script. This script defines the logic for consuming from Kafka and writing to Delta Lake.
   Example pyspark_jobs/streaming_consumer.py (conceptual, as referenced in previous docs):
   ```
   # pyspark_jobs/streaming_consumer.py
   import sys
   from pyspark.sql import SparkSession
   from pyspark.sql.functions import col, from_json
   from pyspark.sql.types import StructType, StringType, FloatType, TimestampType,
   ```

```
MapType

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",
"org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0,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_consumer.py <kafka_topic> <kafka_broker>
<delta_output_path>")
        sys.exit(-1)

    kafka_topic = sys.argv[1]
    kafka_broker = sys.argv[2]
    delta_output_path = sys.argv[3]

    spark = create_spark_session(f"KafkaToDeltaStream_{kafka_topic}")
    spark.sparkContext.setLogLevel("WARN") # Reduce verbosity of Spark logs

    # Define schema for the incoming Kafka message value (financial/insurance data)
    # This schema should match the data structure produced by FastAPI
    # For simplicity, using a generic schema, in reality you'd have specific ones
    # for financial_transactions and insurance_claims
    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("claim_id", StringType(), True) \
        .add("policy_number", StringType(), True) \
        .add("claim_amount", FloatType(), True) \
        .add("claim_type", StringType(), True) \
        .add("claim_status", StringType(), True) \
```

```
        .add("customer_id", StringType(), True) \
        .add("incident_date", StringType(), True)

    # Read from Kafka as a streaming DataFrame
    kafka_df = (spark.readStream
            .format("kafka")
            .option("kafka.bootstrap.servers", kafka_broker)
            .option("subscribe", kafka_topic)
            .option("startingOffsets", "latest") # Start consuming new messages
            .load())

    # Parse the value column (which contains the JSON message)
    # Add metadata for debugging (topic, offset, timestamp)
    parsed_df = kafka_df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING) as
json_value",
                            "topic", "partition", "offset", "timestamp") \
        .select(from_json(col("json_value"), data_schema).alias("data"),
            col("topic"), col("partition"), col("offset"),
col("timestamp").alias("kafka_timestamp")) \
        .select("data.*", "topic", "partition", "offset", "kafka_timestamp")

    # Define checkpoint location for fault tolerance and exactly-once processing
    checkpoint_location = f"{delta_output_path}/_checkpoints"

    # Write the processed data to Delta Lake
    query = (parsed_df.writeStream
          .format("delta")
          .outputMode("append") # Append new data to the Delta table
          .option("checkpointLocation", checkpoint_location) # Required for streaming
writes
          .start(delta_output_path))

    print(f"Spark Structured Streaming job for topic '{kafka_topic}' started, writing to:
{delta_output_path}")
    print(f"Checkpoint location: {checkpoint_location}")

    query.awaitTermination() # Keep the job running until terminated

    spark.stop()
```

Steps to Exercise:

1. **Ensure data generation:** Verify python3 simulate_data.py is running in the background.
2. **Submit Financial Streaming Job:** In a new terminal, submit the Spark job for financial

data.

```
docker exec -it spark spark-submit \
   --packages
org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0,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_consumer.py \
   raw_financial_transactions kafka:29092 s3a://raw-data-bucket/financial_data_delta
```

3. **Submit Insurance Streaming Job:** In another new terminal, submit the Spark job for insurance data.

```
docker exec -it spark spark-submit \
   --packages
org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0,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_consumer.py \
   raw_insurance_claims kafka:29092 s3a://raw-data-bucket/insurance_data_delta
```

Observe the console output for both jobs; they will show Spark logging and progress.

Verification:

- **MinIO Console (http://localhost:9001):** Navigate to raw-data-bucket. You should see two growing directories: financial_data_delta and insurance_data_delta. Each will contain .parquet files (the actual data) and a _delta_log directory (Delta Lake transaction log), confirming continuous data ingestion.
- **Spark History Server (http://localhost:18080):** You should see two active streaming applications. Click on them to inspect their progress, input/output rates, and micro-batch statistics.
- **Grafana (http://localhost:3000):** On the "Kafka Overview" or "Health Dashboard," observe that the consumer lag for both raw_financial_transactions and raw_insurance_claims topics remains low and stable, indicating that Spark is efficiently consuming messages as they arrive.

Advanced Use Case 1: Complex Batch Transformation & Data Quality

Objective: To demonstrate Spark's capability for complex batch transformations, including data cleansing, enrichment (e.g., joining with a reference table in PostgreSQL), and applying data quality rules before writing to a curated zone.
Role in Platform: Refine raw data into high-quality, consumable datasets for analytics and machine learning.
Setup/Configuration:

1. **Ensure Basic Use Case is running:** raw-data-bucket/financial_data_delta is populated.
2. **PostgreSQL reference data:** Assume your PostgreSQL main_db has a merchant_lookup table with merchant_id and merchant_category (can be populated manually or via a setup script).
3. **Spark Batch Transformation Script:** Create pyspark_jobs/batch_transformations.py to handle these steps.
   Example pyspark_jobs/batch_transformations.py (conceptual):

```
# pyspark_jobs/batch_transformations.py
import sys
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, trim, lower, coalesce, lit,
current_timestamp
from pyspark.sql.types import StringType, FloatType, TimestampType, LongType
from delta.tables import DeltaTable

def create_spark_session(app_name):
    """Helper function to create a SparkSession with Delta Lake and PostgreSQL
packages."""
    return (SparkSession.builder.appName(app_name)
            .config("spark.jars.packages",
"io.delta:delta-core_2.12:2.4.0,org.postgresql:postgresql:42.6.0")
            .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
            .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")
            .getOrCreate())

def run_transformation(spark, raw_path, curated_path, pg_host, pg_port, pg_db,
pg_user, pg_password):
    """Performs batch transformation, enrichment, and data quality checks."""
    print(f"Reading raw data from: {raw_path}")
    df_raw = spark.read.format("delta").load(raw_path)

    # 1. Data Cleansing
```

```
   df_cleaned = df_raw.withColumn("account_id", trim(col("account_id"))) \
                  .withColumn("currency", lower(col("currency")))

   # 2. Data Enrichment (Joining with PostgreSQL lookup table)
   # Assuming a merchant_lookup table in PostgreSQL
   print(f"Connecting to PostgreSQL at {pg_host}:{pg_port}/{pg_db} for merchant
lookup...")
   df_merchant_lookup = spark.read \
      .format("jdbc") \
      .option("url", f"jdbc:postgresql://{pg_host}:{pg_port}/{pg_db}") \
      .option("dbtable", "merchant_lookup") # Assuming 'merchant_lookup' table
      .option("user", pg_user) \
      .option("password", pg_password) \
      .load()

   df_enriched = df_cleaned.join(
      df_merchant_lookup,
      df_cleaned["merchant_id"] == df_merchant_lookup["merchant_id"],
      "left" # Use left join to keep all financial transactions
   ).select(df_cleaned["*"], coalesce(df_merchant_lookup["category"],
lit("UNKNOWN")).alias("enriched_category")) # Example enrichment

   # 3. Data Quality Checks (Simple example: flagging invalid amounts)
   df_quality_checked = df_enriched.withColumn(
      "is_amount_valid",
      when(col("amount").isNull() | (col("amount") <= 0), False).otherwise(True)
   ).withColumn("processing_timestamp", current_timestamp()) # Add processing
timestamp

   # Define schema for the curated table (conceptual)
   curated_schema = StructType() \
      .add("transaction_id", StringType()) \
      .add("timestamp", StringType()) \
      .add("account_id", StringType()) \
      .add("amount", FloatType()) \
      .add("currency", StringType()) \
      .add("transaction_type", StringType()) \
      .add("merchant_id", StringType(), True) \
      .add("category", StringType(), True) \
      .add("enriched_category", StringType(), True) \
      .add("is_amount_valid", StringType()) \
      .add("processing_timestamp", TimestampType())
```

```
    # Select columns in the order of curated_schema and cast if necessary
    df_final = df_quality_checked.select([col(c.name).cast(c.dataType) for c in
curated_schema.fields])


    print(f"Writing curated data to: {curated_path}")
    # Write to Curated Delta Lake (using DeltaTable for upserts if needed, otherwise
standard write)
    if DeltaTable.isDeltaTable(spark, curated_path):
        # Example: Simple overwrite for batch, or merge for SCD type operations
        df_final.write.format("delta").mode("overwrite").option("overwriteSchema",
"true").save(curated_path)
    else:
        df_final.write.format("delta").mode("overwrite").option("overwriteSchema",
"true").save(curated_path) # Changed to overwrite for simple batch runs

    print("Batch transformation complete.")

if __name__ == "__main__":
    if len(sys.argv) != 3: # raw_path, curated_path
        print("Usage: batch_transformations.py <raw_delta_path> <curated_delta_path>")
        sys.exit(-1)

    raw_path = sys.argv[1]
    curated_path = sys.argv[2]

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

    # Get PostgreSQL connection details from environment variables (set in
docker-compose.yml for 'spark' service)
    PG_HOST = "postgres" # Service name in docker-compose
    PG_PORT = "5432"
    PG_DB = "main_db"
    PG_USER = "user"
    PG_PASSWORD = "password"

    # Create a dummy merchant_lookup table in PostgreSQL if it doesn't exist
    # This would typically be part of your database migration
    try:
        conn_str = f"jdbc:postgresql://{PG_HOST}:{PG_PORT}/{PG_DB}"
        df_dummy = spark.createDataFrame([(1, "Electronics"), (2, "Groceries"), (3,
"Entertainment")], ["merchant_id", "category"])
```

```
        df_dummy.write.format("jdbc") \
            .option("url", conn_str) \
            .option("dbtable", "merchant_lookup") \
            .option("user", PG_USER) \
            .option("password", PG_PASSWORD) \
            .mode("overwrite") \
            .save()
        print("Dummy 'merchant_lookup' table ensured in PostgreSQL.")
    except Exception as e:
        print(f"Could not ensure dummy merchant_lookup table: {e}. Assuming it exists or
will be created.")

    run_transformation(spark, raw_path, curated_path, PG_HOST, PG_PORT, PG_DB,
PG_USER, PG_PASSWORD)
    spark.stop()
```

Steps to Exercise:

1. **Stop streaming jobs:** Stop the previously submitted Spark streaming jobs (Ctrl+C in their terminals, or docker compose stop spark then docker compose start spark). This is important so the batch job can have a consistent snapshot of the raw data.
2. **Submit Batch Job:** In a new terminal, submit the batch_transformations.py job.
```
docker exec -it spark spark-submit \
    --packages io.delta:delta-core_2.12:2.4.0,org.postgresql:postgresql:42.6.0 \
    --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/batch_transformations.py \
    s3a://raw-data-bucket/financial_data_delta \
    s3a://curated-data-bucket/financial_data_curated_batch
```
3. **Monitor:** Observe the console output of the spark-submit command.

Verification:

- **MinIO Console (http://localhost:9001):** Navigate to curated-data-bucket. You should see a new financial_data_curated_batch directory containing .parquet files and _delta_log.
- **Spark History Server (http://localhost:18080):** The completed batch job should appear. Inspect its details, including the data read and written.
- **Data Content (Conceptual Query):** If you can query the Delta table (e.g., using spark-sql from inside the spark container), verify the enriched_category and

is_amount_valid columns are correctly populated.

```
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, currency, enriched_category, is_amount_valid
FROM delta.\`s3a://curated-data-bucket/financial_data_curated_batch\` LIMIT 10;"
```

Advanced Use Case 2: Machine Learning Integration & Feature Engineering

Objective: To demonstrate how Spark can be used for feature engineering on curated data and conceptually apply a machine learning model, highlighting the analytical capabilities of the platform.
Role in Platform: Prepare and serve high-quality features for ML models, and perform large-scale inference.
Setup/Configuration:

1. **Ensure financial_data_curated_batch is populated:** From Advanced Use Case 1.
2. **ML Script:** Create pyspark_jobs/ml_model_inference.py (as provided in the previous Data Platform Usage Guide). This script will read curated data, perform basic feature engineering (e.g., using VectorAssembler), and conceptually apply an ML model.
   Example pyspark_jobs/ml_model_inference.py (as used before):

```
# pyspark_jobs/ml_model_inference.py
import sys
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression # Example ML library
from pyspark.sql.functions import col

def create_spark_session(app_name):
  """Helper function to create a SparkSession with Delta Lake and MLlib 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) != 2:
        print("Usage: ml_model_inference.py <curated_delta_path>")
        sys.exit(-1)

    curated_path = sys.argv[1]
    spark = create_spark_session("ML_Inference_Example")
    spark.sparkContext.setLogLevel("WARN")

    print(f"Reading curated data from: {curated_path}")
    try:
        df = spark.read.format("delta").load(curated_path)
        df.printSchema()
        df.show(5, truncate=False)

        # --- Feature Engineering ---
        # For demonstration, let's create a 'feature' vector from 'amount' and
'is_amount_valid'
        # In a real scenario, this would involve more complex feature selection and
transformation
        feature_columns = ["amount"]
        if "is_amount_valid" in df.columns: # Conditionally add if exists from previous step

feature_columns.append(col("is_amount_valid").cast("double").alias("is_amount_valid_n
umeric"))
            # If "is_amount_valid" is boolean, cast to double for VectorAssembler

        assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
        feature_df = assembler.transform(df)
        print("Schema after feature engineering:")
        feature_df.printSchema()

        # --- Conceptual ML Model Application ---
        # This part is conceptual as we don't have a trained model.
        # In a real scenario, you would load a pre-trained model:
        # from pyspark.ml.classification import LogisticRegressionModel
        # model = LogisticRegressionModel.load("path/to/your/trained_model")
        # predictions = model.transform(feature_df)
        # predictions.show()

        # For demonstration, we'll just print some summary statistics of the features
        print("Summary of features for ML:")
```

```
        feature_df.select("features").show(5, truncate=False)
        # You could save this prepared feature set for later training/inference
        #
feature_df.write.format("delta").mode("overwrite").save("s3a://ml-features-bucket/finan
cial_features")

        print(f"Successfully read data from {curated_path} and conceptually prepared for
ML.")
        print("In a real scenario, an ML model would now be applied or trained on these
features.")

    except Exception as e:
        print(f"Error reading curated data for ML or during feature engineering: {e}")
        print("Ensure batch_transformations.py has run and populated the
curated-data-bucket path.")

    spark.stop()
```

Steps to Exercise:

1. **Submit ML Script:** In a new terminal, submit the ml_model_inference.py 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/ml_model_inference.py \
  s3a://curated-data-bucket/financial_data_curated_batch
```

2. **Monitor:** Observe the console output for the spark-submit command.

Verification:

- **Console Output:** The script should successfully read from the curated Delta Lake and print the schema with the new features column (a vector of your chosen numerical features). It should also print a summary of the features. This demonstrates Spark's role in preparing data for ML.
- **Spark History Server (http://localhost:18080):** A new completed job for "ML_Inference_Example" will appear.

Advanced Use Case 3: Data Lineage Tracking and

Schema Enforcement (via Spline & Delta Lake)

Objective: To explicitly demonstrate how Spark, combined with Delta Lake, enforces schema, and how Spline automatically captures and visualizes the lineage of Spark transformations.
Role in Platform: Ensure data quality, provide a single source of truth for schema, and enable full transparency of data flow for governance and debugging.
Setup/Configuration:

1. **Ensure Spline and OpenMetadata are running** (Advanced Track setup).
2. **Ensure streaming_consumer.py is running** (Basic Use Case).
3. **Introduce Schema Drift in Producer:** Modify simulate_data.py to temporarily introduce a schema change that *violates* the expected schema for financial data. For example, change amount from a float to a string for a few messages. *Then, revert it back quickly to avoid excessive errors for other jobs.*
   - **Original (in simulate_data.py):** "amount": round(random.uniform(1.0, 10000.0), 2),
   - **Temporary change:** "amount": "invalid_amount_string",
   - **Revert back to original immediately after testing this step.**
4. **Ensure Spark Streaming job is running with mergeSchema disabled for this test:** This is to explicitly show a *failure* if schema enforcement is strict. If mergeSchema is always on, it will adapt. For this demo, let's assume streaming_consumer.py uses outputMode("append") without mergeSchema for a moment, *or* you are running a specific test job without it.

Steps to Exercise:

1. **Trigger Schema Violation (Temporary):**
   - Modify simulate_data.py to send a few messages with amount as a string instead of a float.
   - Run simulate_data.py for a very short period (e.g., 5-10 seconds) with this modification.
   - **Immediately revert simulate_data.py back to its correct schema for amount (float)!** This is crucial to avoid continuous errors.
2. **Observe Spark Job Behavior:**
   - Watch the logs of your financial_transactions Spark streaming job.
   - **Expected (if mergeSchema is OFF):** Spark will likely throw an error (e.g., AnalysisException: Cannot write unknown type string into float type column ...) and the job might fail or restart, indicating strict schema enforcement.
   - **Expected (if mergeSchema is ON):** Spark will likely append data, potentially creating a new column for the string if it detects a new type, or handling nulls if it's a type coercion issue. This demonstrates the flexibility of mergeSchema.
   - **Re-enable/Restart Spark:** Ensure your Spark streaming job is running correctly again (e.g., docker compose restart spark if it crashed, or re-submit with correct mergeSchema options if you were toggling them).
3. **Access Spline UI:** http://localhost:8081.

4. **View Lineage:**
   - Locate the Spark job that processes raw_financial_transactions to financial_data_delta.
   - Click on the job to view its lineage graph.
   - **Utilize:** Observe the input (Kafka topic), the Spark transformation node, and the output (Delta Lake table). Spline will capture the schema of both input and output, and detail the operations performed (e.g., from_json, select, write). This visualizes the schema flow through the pipeline.
5. **Access OpenMetadata UI:** http://localhost:8585.
6. **Verify Schema & Lineage in Catalog:**
   - Search for your financial_data_delta table.
   - Go to its **Schema tab**. Verify the current schema matches what Spark is writing.
   - Go to its **Lineage tab**. OpenMetadata should pull the lineage from Spline, providing an end-to-end view of the data's journey, including column-level lineage if Spline captured it.
   - **Utilize:** This unified view in OpenMetadata demonstrates how governance teams can audit data flow, understand schema evolution, and pinpoint potential data quality issues by tracing data back to its source transformation.

This use case strongly emphasizes how Spark, Delta Lake, and Spline/OpenMetadata collaborate to provide robust data quality, schema management, and transparent data lineage within your platform.