Data Platform Usage Guide: Ingestion, Processing & Observation

This guide provides a practical walkthrough on how to use your local enterprise-ready data platform. It covers the core steps for data ingestion, processing, and how to observe the data flow and system health, specifically demonstrating handling of disparate financial and insurance data. This version introduces additional complexity to demonstrate the system's flexibility in handling different data types and its scalability, as well as integrating with Machine Learning, showcasing serverless ETL with Lambdas, and explicitly demonstrating data lineage.
Before you begin, ensure your local environment is fully set up following the **"Progressive Path Setup Guide Deep-Dive Addendum"**, especially the **Advanced Track** which includes all components.

1. Starting the Platform (Advanced Track)

To ensure all components are running, use the docker compose command from your project root.

1. **Navigate to your project root:**
   cd /path/to/your/data-ingestion-platform

2. Bring up all services (Advanced Track):
   This command will start FastAPI, PostgreSQL, MinIO, Zookeeper, Kafka, Spark, Airflow, MongoDB, OpenMetadata, Spline, Grafana, Grafana Alloy, and cAdvisor.
   docker compose up --build -d

   Wait a few minutes for all services to become healthy. You can check their status with docker compose ps and docker compose logs -f.
3. Perform initial Kafka topic setup (if not done by onboard.sh):
   Ensure both financial and insurance topics exist.
   docker exec -it kafka kafka-topics --create --topic raw_financial_transactions --bootstrap-server kafka:29092 --partitions 3 --replication-factor 1 --if-not-exists
   docker exec -it kafka kafka-topics --create --topic raw_insurance_claims --bootstrap-server kafka:29092 --partitions 3 --replication-factor 1 --if-not-exists

2. Ingesting Disparate Data (FastAPI & Kafka)

You will use the simulate_data.py script to generate mock financial and insurance data and send it to your FastAPI ingestor. FastAPI will then publish this data to their respective Kafka topics.

1. Run the data generator:

Open a new terminal session in your project root and execute the simulate_data.py script. This script will continuously send data to your FastAPI endpoints.
python3 simulate_data.py

You should see console output indicating data being sent (e.g., "Sent financial transaction...", "Sent insurance claim..."). Let this script run in the background.

2. Verify data in Kafka:
   You can manually inspect Kafka topics to ensure data is arriving. Open two new terminal sessions.
   - **Financial Data Consumer:**
     docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw_financial_transactions --from-beginning
   - **Insurance Data Consumer:**
     docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw_insurance_claims --from-beginning

You should see a continuous stream of JSON messages in both consumer terminals.

3. Processing Data (Spark Structured Streaming)

Now, you will start the Spark Structured Streaming jobs to consume data from Kafka and write it as Delta Lake files into MinIO (your simulated S3 data lake). Since you have two types of data, you will likely have two separate streaming jobs or a single job designed to handle both.

1. Submit Spark Streaming Job for Financial Data:
   Open another new terminal session and run the following command to submit your financial streaming consumer job. This command assumes your streaming_consumer.py is designed to take topic and output path as arguments.

```
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
```

2. Submit Spark Streaming Job for Insurance Data:
   Open yet another new terminal session and submit the insurance streaming consumer

job.

```
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
```

Let both Spark jobs run. They will continuously consume from their respective topics.

3. **Verify data in MinIO (Raw Zone):**
    - Access the MinIO Console: http://localhost:9001 (login minioadmin/minioadmin).
    - Navigate to the raw-data-bucket.
    - You should see two new directories: financial_data_delta and insurance_data_delta, each containing .parquet files and a _delta_log directory. This confirms your Spark jobs are writing data.

4. Alternative Processing: Lightweight ETL with AWS Lambda (Conceptual)

While not running locally via Docker Compose, it's crucial to understand how AWS Lambda functions would fit into this architecture for specific ETL tasks, typically for lightweight, event-driven transformations.

Concept: Imagine a Lambda function that triggers whenever a new file lands in your raw-data-bucket (MinIO/S3), or a message arrives on a specific Kafka topic. This Lambda then performs a simple transformation (e.g., anonymization, minor field mapping, data enrichment from a small lookup table) and writes the result to a different S3 location or a database.

Role in Platform:

- **Event-Driven:** Ideal for immediate processing of individual records or small files.
- **Cost-Effective:** Pay-per-execution, suitable for intermittent or bursty workloads.
- **Scalable:** Automatically scales to meet demand without managing servers.

Conceptual Workflow:

1. **Event Source:** New JSON file in raw-data-bucket/financial_data_delta (S3 trigger), or a Kafka message on a specific topic.
2. **Lambda Function:**
    - Reads the event/file content.
    - Performs a simple, in-memory transformation (e.g., adds a processing_timestamp, masks sensitive fields).

    - Writes the transformed data to curated-data-bucket/lambda_processed_data (S3) or inserts into a PostgreSQL table.
3. **Observability:** Lambda's execution metrics (invocations, errors, duration) would automatically feed into CloudWatch (equivalent to Grafana metrics for local Lambda simulation).

Local Simulation:
While a live Lambda can't run in Docker Compose, you can locally test Lambda functions using AWS SAM CLI.

- **Step 1: Define a SAM template:** In your fastapi_app or a new lambda_etl_app directory, you'd define template.yaml for your Lambda function.
- **Step 2: Implement Lambda logic:** Write your Python code for the transformation.
- **Step 3: Test locally:** Use sam local invoke or sam local start-api to test your Lambda locally, simulating S3 events or Kafka messages.

This demonstrates how agile, serverless components handle specific ETL needs, complementing the batch and streaming capabilities of Spark.

5. Orchestration with Airflow

Your platform includes Apache Airflow for orchestrating complex workflows. In the Advanced Track, you should have DAGs defined in airflow_dags/ that trigger your Spark jobs, or even metadata ingestion tasks.

1. **Access Airflow UI:** http://localhost:8080 (login admin/admin).
2. **Unpause and Trigger DAGs:**
    - Locate DAGs like financial_data_processing_dag and insurance_data_processing_dag (or whatever names you've given them).
    - Toggle their status to "On" (unpause).
    - Manually trigger a run for each DAG (click the "Play" icon).
3. **Monitor DAG Runs:** Observe the Graph View and Task Logs for each DAG run.
    - Expected: The DAGs should execute tasks, including Spark job submissions, successfully. Look for green boxes and "success" messages in logs. This demonstrates Airflow's ability to manage complex, multi-component pipelines.

6. Observing the System & Data Governance (Grafana, OpenMetadata, Spline)

The Advanced Track heavily emphasizes observability and governance. These tools provide insights into system health, data flow, and metadata, giving you full control and understanding of your data landscape.

6.1. Grafana for Real-time Monitoring & Performance Metrics

Grafana dashboards visualize metrics from various components (FastAPI, Kafka, Spark, Docker containers via cAdvisor), providing real-time insights into system performance.

1. **Access Grafana UI:** http://localhost:3000 (initially anonymous or configure admin

user).

2. **Explore Dashboards:**
    - Navigate to pre-provisioned dashboards (e.g., "Health Dashboard", "Kafka Overview", "Spark Jobs").
    - **Observe:**
        - **FastAPI:** Request rates (RPS), latency for both financial and insurance ingestion endpoints.
        - **Kafka:** Producer and consumer throughput for raw_financial_transactions and raw_insurance_claims. Crucially, monitor Kafka consumer lag for your Spark jobs. It should remain low and relatively stable, indicating Spark is keeping up.
        - **Spark:** Resource utilization (CPU, memory) of the Spark container, job completion times, and processed record counts.
        - **Docker Container Metrics (via cAdvisor):** Overall CPU/memory usage for all running Docker services.
    - **Demonstration:** Introduce a heavy load (e.g., increase DELAY_SECONDS in simulate_data.py to 0.01 or less, or run multiple instances of it) and observe how metrics like latency or Kafka lag respond. This shows the system's capacity and helps identify bottlenecks.

6.2. Spline for Automated Data Lineage

Spline automatically captures and visualizes data lineage for your Apache Spark jobs. This is critical for understanding data origins, transformations, and impacts of changes.

1. **Access Spline UI:** http://localhost:8081.
2. **View Lineage:**
    - You should see entries for your financial and insurance Spark jobs (from Section 3.2, or triggered by Airflow in Section 5).
    - Click on an entry (e.g., "Kafka to Delta Lake - Financial Data") to see its **detailed lineage graph**.
    - **Utilize:** Examine the graph to trace the path of financial data from the raw_financial_transactions Kafka topic, through the Spark processing job, to the financial_data_delta table in MinIO. This visual confirmation is invaluable for auditing, debugging, and understanding data transformations.
    - **Explore Details:** Click on specific nodes (e.g., a dataset or a transformation) within the graph to view metadata such as schema, columns, and applied operations.

6.3. OpenMetadata for Centralized Data Catalog & Active Governance

OpenMetadata acts as a centralized data catalog, consolidating metadata from all your data assets, making them discoverable and governable.

1. **Access OpenMetadata UI:** http://localhost:8585.
2. **Explore Data Assets:**
    - Once your Airflow DAGs for OpenMetadata ingestion have run (these should be

part of your airflow_dags and periodically run), you should see your Kafka topics (raw_financial_transactions, raw_insurance_claims), Delta Lake tables (e.g., financial_data_delta, insurance_data_delta), PostgreSQL tables, and potentially FastAPI endpoints listed.
    - **Search for Assets:** Use the search bar (e.g., search for "financial claim" or "insurance policy").
    - **Inspect Asset Details:** Click on a Kafka topic or Delta Lake table.
        - You should see its **schema**, **sample data**, and crucially, a **"Lineage" tab** that provides detailed lineage information (often integrated from Spline). This shows where the data came from and where it flows.
        - Explore the **"Profiler" tab** (if configured) for data quality metrics and column-level statistics.
    - **Utilize for Governance:**
        - **Add Descriptions:** Click the "Edit" icon next to an asset's description or column description and add a meaningful business description. This directly impacts data discoverability and understanding for other users.
        - **Add Tags:** Apply relevant tags (e.g., PII, Financial, Sensitive) to assets or individual columns. This demonstrates how data can be classified for governance and security purposes.
        - **Assign Owners:** Assign team members or groups as owners to data assets, clarifying responsibility.

7. Exercising the System with Robust Disparate Data

Beyond just verifying data flow, these steps demonstrate the platform's robustness and flexibility, including preparing data for machine learning.

1. **Introduce Schema Evolution:**
    - Modify src/models/financial_transaction.py (or the generate_financial_transaction function in simulate_data.py) to add a new, *optional* field (e.g., is_flagged: bool = False).
    - Restart simulate_data.py.
    - Ensure your financial Spark streaming job is configured with mergeSchema option for Delta Lake writes.
    - **Verify:** Observe in MinIO that the Delta Lake schema for financial_data_delta has evolved without breaking. Query the Delta table with Spark to see the new column. Old records will have null for this column, new ones will have data. This shows the system's ability to handle evolving data schemas gracefully.
2. **Simulate Backpressure and Recovery:**
    - While simulate_data.py is running (generating high volume), temporarily pause the Spark container: docker compose pause spark.
    - **Observe:** Watch the Kafka consumer lag in Grafana for both financial and insurance topics. It should rapidly increase as Spark stops consuming.
    - After a minute or two, resume the Spark container: docker compose unpause

spark.
    - **Observe:** The Kafka consumer lag should start to decrease, demonstrating Spark's ability to catch up on accumulated backlog and the system's resilience.
3. **Stress Testing (using Locust):**
    - Stop your simulate_data.py script.
    - Start Locust: locust -f locust_fastapi_ingestor.py --host http://localhost:8000
    - Access Locust UI at http://localhost:8089.
    - **Start a new test** with high user concurrency (e.g., 50-100 users, 10 spawn rate) and a short hatch rate.
    - **Observe in Grafana:** Watch FastAPI RPS and latency, and Kafka consumer lag for both topics. This demonstrates how the system performs under sustained heavy load. You can identify potential bottlenecks (e.g., FastAPI's ability to handle concurrent requests, Kafka's ability to buffer, Spark's processing speed).
4. Integrating with Machine Learning (Conceptual):
   The curated Delta Lake serves as a high-quality data source for machine learning tasks. While a full ML model training/deployment isn't run locally, you can conceptually demonstrate the integration.
   Concept: A Spark job (or a separate Python script with pyspark installed locally) can read from your curated Delta Lake table, prepare features, and then apply a pre-trained ML model or perform a simple training step.
   Conceptual Steps:
    - **Prepare Curated Data:** Ensure your batch_transformations.py (or similar) runs to create curated-data-bucket/financial_data_curated. This data would typically be cleaned, aggregated, and modeled for ML.
    - **Run a conceptual Spark ML script:** Imagine a pyspark_jobs/ml_model_inference.py script.

```
# Conceptual pyspark_jobs/ml_model_inference.py
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression # Or any other ML model

def create_spark_session(app_name):
    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__":
    spark = create_spark_session("ML_Inference_Example")
    curated_path = "s3a://curated-data-bucket/financial_data_curated" # Point to
your curated data
```

```
    # Simulate reading curated data
    # In a real scenario, this would be a well-defined feature set
    try:
        df = spark.read.format("delta").load(curated_path)
        df.show(5, truncate=False)

        # Conceptual ML preparation
        # Assuming 'amount' is a feature for a simple model
        assembler = VectorAssembler(inputCols=["amount"], outputCol="features")
        feature_df = assembler.transform(df)

        # Simulate a pre-trained model (or a simple training)
        # For demonstration, we'll just show structure
        # model = LogisticRegressionModel.load("path/to/pretrained_model")
        # predictions = model.transform(feature_df)
        # predictions.show()

        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.")

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

    spark.stop()
```

- **Run the script:** docker exec -it spark spark-submit pyspark_jobs/ml_model_inference.py
- **Verify:** The script successfully reads from the curated Delta Lake, demonstrating that the data platform is providing high-quality, consumable data for advanced analytics and ML applications.

By performing these steps, you will gain a profound understanding of how your enterprise-ready data platform operates, its capabilities in handling diverse data, its resilience, and its comprehensive observability features.