Highlighting Apache Kafka: Distributed Streaming Platform

Apache Kafka is the backbone of real-time data ingestion and streaming applications within your enterprise data platform. It acts as a highly scalable, fault-tolerant, and durable message broker, decoupling data producers from consumers and enabling a wide array of real-time use cases.
This guide will demonstrate basic and advanced use cases of Apache Kafka, 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: High-Throughput Decoupled Data Ingestion

Objective: To demonstrate Kafka's fundamental role in receiving high volumes of disparate data (financial transactions and insurance claims) from the FastAPI ingestor and holding it durably for asynchronous consumption.
Role in Platform: Serve as the central streaming hub, ensuring data resilience and decoupling producers from consumers, preventing backpressure issues.
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 Kafka is accessible:** Check Docker logs for the kafka and zookeeper containers to ensure they are healthy.
3. **Ensure Kafka topics are initialized:** As per onboard.sh or manual commands, confirm raw_financial_transactions and raw_insurance_claims topics exist.

Steps to Exercise:

1. Start Data Generator:
   Open a new terminal session in your project root and execute the simulate_data.py script. This script sends mock financial and insurance data to your FastAPI endpoints, which then publish to Kafka.
   python3 simulate_data.py

   Let this run in the background.
2. Verify Data Ingestion (Kafka Console Consumers):
   Open two separate 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

Verification:

- **Console Output:** Both kafka-console-consumer terminals should continuously display JSON messages, confirming that FastAPI is successfully publishing data to the respective Kafka topics and Kafka is receiving and storing them.
- **FastAPI Logs:** Check the fastapi_ingestor container logs (docker compose logs fastapi_ingestor) for messages indicating successful publishing to Kafka.

Advanced Use Case 1: Real-time Fan-out to Multiple Consumers

Objective: To demonstrate Kafka's ability to deliver the same data stream to multiple, independent consumer groups without affecting each other's progress, enabling diverse real-time applications.
Role in Platform: Enable real-time dashboards, fraud detection systems, and other concurrent consumers from a single, canonical data stream.
Setup/Configuration:

1. **Basic Use Case completed:** Ensure data is actively flowing into both raw_financial_transactions and raw_insurance_claims Kafka topics (i.e., simulate_data.py is running).
2. **Spark Streaming Consumers are active:** Ensure your Spark streaming jobs are consuming from these topics (as set up in the Data Platform Usage Guide, Section 3).

Steps to Exercise:

1. Start a new "Dashboard" Consumer (Financial Data):
   Open a new terminal and start a Kafka console consumer for financial transactions, but assign it a different consumer group ID than your Spark job's consumer group.
   docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw_financial_transactions --group real-time-dashboard-financial --from-beginning

2. Start a new "Audit" Consumer (Insurance Data):
   Open another new terminal and start a Kafka console consumer for insurance claims, with a unique group ID.
   docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw_insurance_claims --group audit-log-insurance --from-beginning

3. **Observe all Consumers:** Keep an eye on the output of these new consumers, as well as the logs/metrics of your existing Spark streaming jobs (e.g., via Grafana's Kafka

consumer lag panel).

Verification:

- **All Consumers Receiving Data:** Both new console consumers should be actively receiving financial and insurance transaction messages, respectively.
- **Independent Progress:** Crucially, the Kafka consumer lag for your Spark jobs (visible in Grafana) should remain low and unaffected by the introduction of these new console consumers. This demonstrates that each consumer group maintains its own offset and consumes independently from the same topic.
- **Data Consistency:** All consumers receive the exact same sequence of messages within a partition, showcasing Kafka's immutable log.

Advanced Use Case 2: Data Retention, Replay, and Disaster Recovery Preparedness

Objective: To demonstrate Kafka's capability to retain messages for a configurable period, allowing for data replay from arbitrary points, which is crucial for reprocessing, debugging, and disaster recovery scenarios.

Role in Platform: Provide a durable, replayable source of truth for streaming data, reducing data loss RPO (Recovery Point Objective).

Setup/Configuration:

1. **Basic Use Case completed:** Ensure simulate_data.py is running and ingesting data into Kafka.
2. **Identify a Kafka Topic:** For this example, we'll use raw_financial_transactions.
3. **Note Current Offset:** Before stopping Spark, check a recent offset.

Steps to Exercise:

1. Simulate Spark Job Failure/Redeployment:
   Pause your financial Spark streaming job container (or stop the corresponding Airflow DAG run) to simulate a failure or a need for a full reprocessing.
   docker compose pause spark # If one Spark service handles both. If separate, pause only the relevant part.

   Observe in Grafana: The Kafka consumer lag for raw_financial_transactions will start to increase significantly.
2. **Let Data Accumulate:** Allow simulate_data.py to continue running for a few minutes while Spark is paused, letting messages accumulate in Kafka.
3. Simulate New/Restarted Spark Job (Replay from Earliest):
   Unpause Spark (if paused) or submit a new Spark job instance for financial data, configured to read from the earliest offset (or a specific historical offset if you noted one earlier).
   docker compose unpause spark # If you paused it
   # Or, if restarting a new instance/job, ensure its startingOffsets is set to 'earliest'
   # Example for new consumer reading from earliest for replay
   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_reprocessed \
  --startingOffsets earliest # <-- Key for replay
```

Verification:

- **Grafana:** Observe the Kafka consumer lag for raw_financial_transactions rapidly decreasing as the (restarted/new) Spark job consumes the accumulated backlog and older messages.
- **MinIO:** If you wrote to a new path (financial_data_delta_reprocessed), you'll see new data accumulating, including historical data that was reprocessed. This validates Kafka's durable storage and replay capabilities.

Advanced Use Case 3: Partitioning Strategy & Scalability

Objective: To understand how Kafka partitioning directly influences throughput and consumer parallelism, demonstrating a core scaling mechanism.
Role in Platform: Optimize data distribution for maximum throughput and enable highly parallel consumption.
Setup/Configuration:

1. **Initial Setup:** Your raw_financial_transactions topic likely has 3 partitions (as initialized by onboard.sh).
2. **Prepare Scalable Consumer (Conceptual):** Your Spark streaming job is inherently capable of parallel consumption across partitions.

Steps to Exercise:

1. **Monitor Baseline:**
   - Ensure simulate_data.py is running at a *high rate* (e.g., DELAY_SECONDS = 0.01 in simulate_data.py).
   - Monitor the raw_financial_transactions Kafka topic's consumer lag in Grafana. Observe the Spark job's CPU utilization in Grafana. This is your baseline performance with the current partition count.
2. Increase Topic Partitions (Dynamically):
   While it's generally best practice to set partition counts at topic creation, Kafka allows

increasing them dynamically (though not decreasing).
docker exec -it kafka kafka-topics --bootstrap-server localhost:29092 --alter --topic raw_financial_transactions --partitions 6

Note: Increasing partitions only benefits new messages. Existing messages remain in their original partitions. For full benefit, you might re-ingest data or create a new topic with more partitions.

3. Adjust Spark Parallelism (Conceptual/Manual):
   If your Spark job is constrained by partitions (i.e., less Spark executors/cores than partitions), it won't fully utilize the new partitions. Conceptually, you would:
   - Stop the current Spark job.
   - Increase the number of Spark executors or cores allocated to your financial data processing job (in docker-compose.yml or the spark-submit command).
   - Restart the Spark job.
4. **Observe Performance under Increased Partitions/Parallelism:**
   - Continue running simulate_data.py at a high rate.
   - Monitor Grafana.

Verification:

- **Kafka Topic Description:** docker exec -it kafka kafka-topics --bootstrap-server localhost:29092 --describe --topic raw_financial_transactions (Expected: Output shows 6 partitions).
- **Grafana Metrics:** With sufficient producer load and increased Spark parallelism, you should observe:
  - Potentially higher overall throughput for the raw_financial_transactions topic.
  - Better distribution of load across Spark executors (if you could monitor individual executor metrics).
  - The Kafka consumer lag remaining stable or reducing faster under heavy load, indicating that the increased parallelism is effectively consuming the data.

This demonstrates how strategic Kafka partitioning, combined with corresponding consumer parallelism, is a key lever for scaling your streaming data pipelines.