Low-Level Design Document (LLD)
Project Name: Data Streaming (Kafka – NiFi – HDFS -Airflow– Hive Data Pipeline)
Prepared By: Marwa Mansour
Date: 11-Feb-2026

Code Details:

Producer Application

- Class Name: **KafkaAvroProducerApp**
- Responsibility:
  - Generates 1000 UserEvent records
  - Serializes records using Avro
  - Registers schema in Confluent Schema Registry
  - schema.registry.url=http://schema-registry:8081
  - Publishes records to Kafka topic: user-events

Consumer Application

- Class Name: **KafkaAvroConsumerApp**
- Responsibility:
  - Consumes messages from Kafka topic user-events
  - Deserializes using KafkaAvroDeserializer
  - Prints GenericRecord output for validation

- **Topic Configuration**
- • Topic Name: user-events
- • Offset Reset: earliest
- • Message Count Tested: 1000 records

Schema Registration:

Avro Schema

```
{
  "type": "record",
  "name": "UserEvent",
  "namespace": "com.example.avro",
  "fields": [
   { "name": "user_id", "type": "int" },
   { "name": "name", "type": "string" },
   { "name": "email", "type": "string" },
   { "name": "event_type", "type": "string" },
   { "name": "event_time", "type": "long" }
  ]
}
```

KafkaAvroProducerApp Code To Generate 1000 Record:

```java
package com.example;
import com.example.avro.UserEvent;
import org.apache.kafka.clients.producer.*;
import io.confluent.kafka.serializers.KafkaAvroSerializer;
import java.util.Properties;
public class KafkaAvroProducerApp {

  public static void main(String[] args) {

    String bootstrapServers = "localhost:9092";
    String schemaRegistryUrl = "http://localhost:8081";
    String topic = "user-events";

    Properties props = new Properties();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class.getName());
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class.getName());
    props.put("schema.registry.url", schemaRegistryUrl);

    Producer<String, UserEvent> producer = new KafkaProducer<>(props);

    for (int i = 1; i <= 1000; i++) {

      UserEvent userEvent = UserEvent.newBuilder()
          .setId(i)
          .setName("User" + i)
          .setEmail("user" + i + "@example.com")
          .setEventTime(System.currentTimeMillis())
          .build();

      ProducerRecord<String, UserEvent> record =
          new ProducerRecord<>(topic, String.valueOf(userEvent.getId()), userEvent);

      producer.send(record, (metadata, exception) -> {
        if (exception != null) {
          exception.printStackTrace();
        } else {
          System.out.println("Sent record " + userEvent.getId() +
              " to partition " + metadata.partition() +
              " with offset " + metadata.offset());
        }
      });
    }

    producer.flush();
    producer.close();

    System.out.println("All 1000 messages sent successfully!");
  }
}
```

KafkaAvroConsumerApp To consumer data record from producer:

```
package com.example;
import org.apache.kafka.clients.consumer.*;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import org.apache.avro.generic.GenericRecord;
import java.time.Duration;
import java.util.Arrays;
import java.util.Properties;

public class KafkaAvroConsumerApp {
  public static void main(String[] args) {
    Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9092");
    props.put("group.id", "test-group");
    props.put("key.deserializer", KafkaAvroDeserializer.class.getName());
    props.put("value.deserializer", KafkaAvroDeserializer.class.getName());
    props.put("schema.registry.url", "http://localhost:8081");
    props.put("specific.avro.reader", "false");

    KafkaConsumer<String, GenericRecord> consumer = new KafkaConsumer<>(props);
    consumer.subscribe(Arrays.asList("user-events"));

    while (true) {
      ConsumerRecords<String, GenericRecord> records = consumer.poll(Duration.ofMillis(1000));
      for (ConsumerRecord<String, GenericRecord> record : records) {
        System.out.println(record.value());
      }
    }
  }
}
```

NiFi Flow Detailed Design (Production Level)

Flow Name:

kafka_nifi_hdfs_hive

Purpose

This NiFi flow is designed to consume real-time streaming data from Kafka, validate and enrich the records, persist them in HDFS in a structured format, and provide full observability through structured logging and failure handling mechanisms.

The flow follows a controlled success/failure routing strategy to ensure reliability, data integrity, and production-grade monitoring.

ConsumeKafkaRecord_2_6

Component Role

Acts as the ingestion entry point of the streaming pipeline.

- **Functional Responsibilities**
- Subscribes to Kafka topic: user-events
- Deserializes Avro messages using configured Record Reader
- Converts Kafka records into NiFi FlowFiles
- Commits offsets only after successful processing (consumer group controlled)

Relationship	Action
success	Routes valid records to UpdateAttribute
parse.failure	Routes malformed or schema-invalid records to logging processor

Production Considerations

- Consumer group configured for controlled offset tracking
- Back pressure enabled to prevent memory overflow
- Supports horizontal scaling when deployed in NiFi cluster

---

2. Parse Failure Handling (LogAttribute – Failure Path)

Purpose

Captures malformed or schema-incompatible records.

Observability

This ensures early detection of schema incompatibility or producer-side data issues.

UpdateAttribute

Component Role

Prepares the FlowFile for structured storage.

Functional Responsibilities

- Generates dynamic filename if required
- Adds metadata attributes for lineage tracking

Design Rationale

- Enables dynamic HDFS directory structure.

PutHDFS

Component Role

Handles distributed storage into HDFS.

Functional Responsibilities

- Writes records in Parquet format
- Stores data under directory structure
- Ensures compatibility with Hive external table

Target Directory Structure

/data/user_events/ user_events_<timestamp>.parquet

Relationships

Relationship	Action
success	Routed to Success LogAttribute
failure	Routed to Failure LogAttribute

Production Considerations

- HDFS replication factor ensures durability
- Supports large-scale distributed storage
- Fault-tolerant write mechanism

End-to-End Execution Lifecycle

1. Kafka message ingestion
2. Schema-based deserialization
3. Validation routing (success vs parse failure)
4. Attribute enrichment
5. Parquet file creation
6. HDFS persistence
7. Success/Failure logging

Reliability & Enterprise Controls

✓ Controlled Offset Management

Offsets committed only after successful processing.

✓ Schema Governance

Strict Avro validation prevents malformed ingestion.

✓ Observability

Logging implemented at ingestion, transformation, and storage stages.

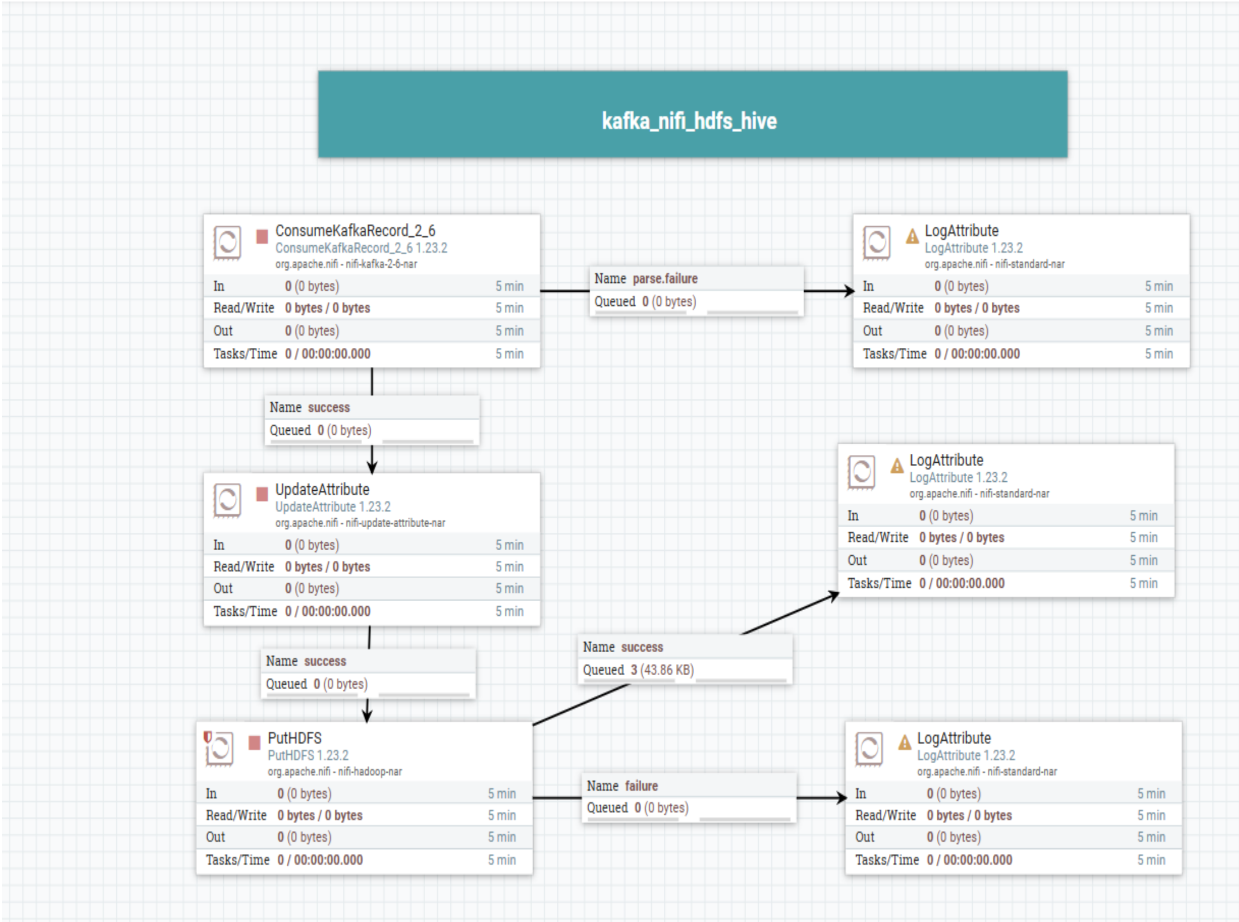
✓ Failure Isolation

Corrupted records are isolated and do not impact healthy data flow.

Production Readiness Summary

This NiFi flow design ensures:

- High reliability
- Clear data lineage
- Strong schema enforcement
- Distributed scalable storage
- Enterprise-grade monitoring
- Fault-tolerant streaming ingestion



Connection Details

- **Kafka:**
  Broker: kafka:29092
  Topic: user-events
  Schema Registry: http://schema-registry:8081
- **NiFi:**
  - URL: http://localhost:8060
  - Processor: ConsumeKafkaRecord_2_6
  - Record Reader: AvroReader
  - Record Writer: ParquetRecordSetWriter
- **HDFS:**
  - Path: /data/user_events
- **Hive:**
  - Database: user_events_db

DDLs (Hive Tables)

- **External Table (Parquet)**

```
CREATE EXTERNAL TABLE user_events_db.user_events_parq (
 id INT,
 name STRING,
 email STRING,
 event_time BIGINT
)
STORED AS PARQUET
LOCATION '/data/user_events';
```

- **Managed Table (Parquet)**

```
CREATE TABLE user_events_prod.user_events_parquet_managed (
 id INT,
name STRING,
email STRING,
event_time BIGINT
)
STORED AS PARQUET
TBLPROPERTIES ("parquet.compression"="SNAPPY");
```

- **Load data into managed table :**
- **Method 1:**

  LOAD DATA INPATH '/data/user_events' INTO TABLE user_events_managed;
- **Methode 2 (loading data from external table into managed table):**

  INSERT INTO TABLE user_events_parquet_managed

  SELECT * FROM user_events_db.user_events_parq;

Data Loading & Archiving Logic

1 Overview

This section describes the detailed implementation of the data ingestion process from HDFS to Hive managed table using Apache Airflow.

The pipeline ensures:

- Only new files are loaded into the Hive managed table.
- Previously processed files are archived.
- Data is stored in Parquet format with Snappy compression.
- Duplicate loading is prevented.

---

2 HDFS Directory Structure

/data/user_events/ → Incoming files

/data/user_events/archive/ → Archived processed files

---

3 Hive Configuration

- Database: user_events_prod
- Managed Table: user_events_parquet_managed
- Storage Format: Parquet
- Compression: Snappy

Table type: Managed Table(id , name , email, event_time)

Location: Managed by Hive (default warehouse path)

---

4 Processing Logic

Step 1: Detect New Files

Airflow checks the HDFS incoming directory.

Step 2: Load to Hive Managed Table

New files are inserted into:

user_events_prod.user_events_parquet_managed

Using:

- Parquet format
- Snappy compression

Step 3: Archive Processed Files

After successful load:

- Files are moved from:
- /data/user_events/

To:

/data/user_events/archive/

This ensures:

- Idempotent processing
- No duplicate ingestion
- Clear separation between new and processed data

---

5 DAG Implementation

DAG Name: user_events_kafka_nifi_hadoop_hive

Main Task:

- load_and_archive

Execution Flow:

1. Load new data into Hive
2. Move processed files to archive
3. Log execution status

---

6 Error Handling

- If Hive load fails → files are NOT moved to archive.
- If archive step fails → DAG retries based on retry configuration.
- Logs are stored in Airflow logs directory.

Airflow Dag Code:

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime, timedelta
# Paths and Hive table names
HDFS_PATH = "/data/user_events"
ARCHIVE_PATH = "/data/user_events/archive"
HIVE_DB = "user_events_prod"
HIVE_TABLE = "user_events_parquet_managed"

# Default arguments for the DAG
default_args = {
    'owner': 'airflow',
    'start_date': datetime(2026, 2, 16),
    'retries': 1,
    'retry_delay': timedelta(minutes=2),
}

# Define the DAG
with DAG(
    dag_id="user_events_kafka_nifi_hadoop_hive",
    default_args=default_args,
    schedule_interval="*/5 * * * *",
    catchup=False,
    max_active_runs=1,
    tags=['hive', 'hdfs', 'archive'],
) as dag:

    # Task: Load new files into managed Hive table and archive them
    load_and_archive = BashOperator(
        task_id="load_and_archive",
        bash_command=f"""
        set -e
        echo "Creating archive directory if not exists..."
        hdfs dfs -mkdir -p {ARCHIVE_PATH}

        # Count new files
        file_count=$(hdfs dfs -ls {HDFS_PATH} 2>/dev/null | grep -v '/archive' | grep 'user-
events-.*\\.parquet$' | wc -l)

        if [ "$file_count" -eq 0 ]; then
```

```
    echo "No new files to process"
    exit 0
  fi

  echo "Found $file_count new file(s)"

  temp_ext_table="ext_user_events_temp_$(date +%s)"
  echo "Creating temporary external Hive table: $temp_ext_table"

  hive -e "
  USE {HIVE_DB};
  SET hive.stats.autogather=false;
  SET hive.compute.query.using.stats=false;

  CREATE EXTERNAL TABLE $temp_ext_table (
    id INT,
    name STRING,
    email STRING,
    event_time BIGINT
  )
  STORED AS PARQUET
  LOCATION '{HDFS_PATH}';

  INSERT INTO TABLE {HIVE_TABLE}
  SELECT t.id, t.name, t.email,t.event_time
  FROM $temp_ext_table t
  LEFT JOIN {HIVE_TABLE} h ON t.id = h.id
  WHERE h.id IS NULL;

  DROP TABLE $temp_ext_table;
  "
  echo "? Data loaded to managed table successfully"
  # Archive processed files
  for file in $(hdfs dfs -ls {HDFS_PATH} | awk '{{print $8}}' | grep 'user-events-
.*\\.parquet$'); do
    filename=$(basename "$file")
    hdfs dfs -mv "$file" "{ARCHIVE_PATH}/$filename"
    echo "? Archived: $filename"
  done
  echo "? All files archived successfully"
 """,
```