ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/25sep1309

Real-Time Streaming of Call Detail Records to HDFS: An End-to-End Big Data Pipeline Using Kafka Connect, Apache Airflow, and Apache Spark

Germain Uwiringiyedata[1]

[1]Master's in Big Data Analytics Adventist University of Central Africa

Publication Date: 2025/09/29

Abstract: The rapid expansion of telecommunications services produces enormous quan- tities of Call Detail Records (CDRs), requiring real-time ingestion, storage, and analysis to support billing operations and fraud detection systems, and network op- timization. paper presents an end-to-end, containerized big data pipeline Call Detail Records (CDRs) are generated as high-volume event streams that require low-latency ingestion, durable storage, and dependable analytics. This paper presents an end-to- end, containerized big data pipeline that integrates Apache Kafka, Kafka Connect, Hadoop Distributed File System (HDFS), PySpark, and Apache Airflow within a reproducible Docker environment. Unlike conventional batch-oriented approaches, the proposed architecture demonstrates low-latency ingestion, fault-tolerant storage, and scalable processing of high-throughput CDR streams. Experimental results show zero delivery loss at 25 records per second (RPS), balanced partition throughput, and immediate analytical readiness, with roaming traffic analysis and cell-level usage statistics produced in seconds. The work contributes a practical reference model for telecom streaming pipelines, highlighting the advantages of containerized deployment, automated orchestration, and reproducible analytics, and it outlines directions for scaling and production integration.

Keywords: *Kafka; Kafka Connect; HDFS; PySpark; Airflow; Docker; Streaming; CDR.*

List of Acronyms

CDR	Call Detail Record
HDFS	Hadoop Distributed File System
RPS	Records Per Second
DAG	Directed Acyclic Graph
API	Application Programming Interface
SQL	Structured Query Language
IoT	Internet of Things

How to Cite: Germain Uwiringiyedata (2025) Real-Time Streaming of Call Detail Records to HDFS: An End-to-End Big Data Pipeline Using Kafka Connect, Apache Airflow, and Apache Spark. *International Journal of Innovative Science and Research Technology*, 10 (9), 2064-2071. https://doi.org/10.38124/ijisrt/25sep1309

I. INTRODUCTION

The exponential growth of telecommunications services has led to the massive generation of Call Detail Records (CDRs)—metadata describing voice calls, SMS, mobile money transactions, and internet usage. These records are indispensable to operators and service providers, underpinning billing, customer behavior analysis, fraud detection, network optimization, and regulatory compliance. Given their volume, velocity, and variety, managing CDRs demands big data technologies that can capture, process, and analyze streams in near real time.

Traditional enterprise data warehouses (e.g., Oracle, Teradata, IBM Netezza) were designed for structured, batch-oriented analytics. While effective for periodic reporting, they are ill-suited to the velocity and scale of modern telecom data streams due to high latency, limited scalability, and elevated infrastructure costs [2, 3, 4, 1]. To address these challenges, we propose a cost-effective, scalable streaming data pipeline built on open-source frameworks. In this architecture, Apache Kafka serves as the distributed messaging backbone to handle continuous CDR flows with high throughput and fault tolerance, while Kafka Connect provides a standardized integration layer that

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/25sep1309

lands records into the Hadoop Distributed File System (HDFS) for durable, scalable storage.

Once raw data is available in HDFS, Apache Spark—via PySpark—enables advanced transformations, enrichment, and aggregations over CDRs, supporting both batch and streaming analytics. This capability facilitates timely insights into network usage patterns, anomalies, and customer trends. To orchestrate, schedule, and monitor pipeline compo- nents, Apache Airflow supplies workflow automation, dependency management, and fault recovery, ensuring that each stage—from ingestion to processing and storage—operates reliably and transparently.

To simplify deployment and ensure portability, all components are containerized using Docker. Containers eliminate environment-configuration drift and promote consistent behavior across development, testing, and production. By combining Kafka, HDFS, PySpark, Airflow, and Docker into a unified streaming pipeline, telecom operators can realize near-real-time analytics at scale to support decision-making and improve operational efficiency.

This work presents the architecture, implementation, and empirical evaluation of the pipeline. We show how integrating modern big data frameworks within a containerized environment yields a practical platform for telecom data engineering and analytics, with applications extending beyond CDRs to other domains requiring high-throughput, low- latency processing.

➢ *This Work is Designed to Provide a Practical, Reproducible Reference Architecture for Streaming CDR Data Pipelines. Specifically, it Offers:*

- A Docker Compose–based deployment designed for rapid setup, bundling Kafka, Kafka Connect (HDFS sink), Hadoop HDFS (NameNode, DataNodes, YARN), Airflow, and Kafka-UI into a single reproducible environment.
- A Kafka Connect configuration designed to roll files deterministically (via flush.size) into HDFS directories partitioned by topic and Kafka partition, reducing small-file overhead and ensuring predictable storage layouts.
- Airflow DAGs designed to automate key operational tasks including topic creation, connector (re)configuration, health and lag monitoring, and verification of successful data landing.
- A PySpark/Jupyter workflow designed for schema discovery, data compaction, and baseline CDR analytics on landed files, enabling extensible analysis and downstream modeling.

II. LITERATURE REVIEW

➢ *From Traditional Warehouses to Streaming Data Platforms*

For many years, enterprise analytics relied heavily on centralized data warehouses such as Oracle, Teradata, and IBM Netezza. These systems were optimized for structured data and batch-oriented Extract–Transform–Load (ETL) processes, enabling robust business reporting. However, their architecture introduced high latency, as data was typically refreshed on hourly or daily schedules. In telecommunications, where Call Detail Records (CDRs) are produced continuously and at scale, such systems cannot meet near real-time requirements. This limitation motivated the transition toward distributed, log-centric, and horizontally scalable platforms capable of handling semi-structured streaming data.

➢ *Kafka as a High-Throughput Streaming Backbone*

Apache Kafka has emerged as a leading platform for large-scale event ingestion. Its de- sign—based on distributed commit logs and partitioned topics—provides high throughput, durability, and fault tolerance. Kafka allows producers and consumers to operate inde-pendently, making it suitable for telecom environments where CDRs need to be delivered simultaneously to billing systems, analytics engines, and fraud detection pipelines. Its offset management and replication features ensure reliability even in the presence of node failures.

➢ *Kafka Connect for Automated Ingestion into HDFS*

Kafka Connect provides a standardized framework for integrating Kafka with external systems through pluggable connectors. The HDFS Sink Connector enables partition-aware file writing directly into Hadoop Distributed File System (HDFS). This eliminates the need for custom ingestion scripts while ensuring reliability through mechanisms such as the write-ahead log (WAL) and recovery support. For telecom operators, this reduces operational complexity while ensuring CDRs are durably persisted for downstream analysis.

➢ *HDFS as a Scalable Data Lake Substrate*

HDFS was designed for storing and processing massive datasets by replicating blocks across clusters of commodity hardware. Its fault-tolerant architecture, based on NameN- ode–DataNode separation, provides both durability and scalability. For streaming telecom workloads, HDFS serves as a cost-effective landing zone for raw CDRs, enabling historical retention and compatibility with both batch and real-time processing frameworks.

➢ *Stream Processing in Apache Spark*

Apache Spark has evolved from Resilient Distributed Datasets (RDDs) to provide both batch and stream processing. Spark Streaming introduced micro-batch processing, while Structured Streaming unified batch and streaming workloads under a declarative API.

Features such as event-time semantics, watermarks, and stateful aggregations allow telecom engineers to derive insights such as call durations, anomaly detection, and customer segmentation from CDR streams with minimal latency.

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/25sep1309

➢ *Workflow Orchestration with Apache Airflow*

Apache Airflow is widely adopted for orchestrating data workflows through Directed Acyclic Graphs (DAGs). Its scheduling, retry, and monitoring capabilities make it well suited to telecom streaming pipelines, where ingestion failures or delays must be resolved quickly. Airflow provides transparency and reproducibility by allowing complex multi-stage workflows (e.g., ingestion, compaction, validation) to be expressed programmatically.

➢ *Containerization with Docker*

Docker has revolutionized deployment by packaging applications and dependencies into portable containers. Unlike traditional virtual machines, containers are lightweight and provide consistent execution environments across development and production. In telecom streaming pipelines, Docker Compose allows complex stacks—Kafka, HDFS, Spark, and Airflow—to be deployed reproducibly, reducing operational overhead and ensuring portability.

➢ *Telecom CDR Analytics Patterns*

CDRs support key telecom processes including billing, fraud prevention, capacity planning, and customer analytics. A common architecture involves: (i) ingesting events into Kafka, (ii) persisting to HDFS through Kafka Connect, (iii) processing via Spark for enrichment and aggregation, (iv) orchestrating workflows with Airflow, and (v) serving curated datasets to BI dashboards or machine learning systems. This layered pipeline balances real-time responsiveness with long-term analytical flexibility.

➢ *Synthesis and Research Gaps*

The reviewed systems converge on a "lake-first" architecture where Kafka ensures scalable ingestion, HDFS provides durable storage, Spark enables unified processing, Airflow delivers orchestration, and Docker ensures reproducibility. Despite these advances, open challenges remain, such as achieving exactly-once guarantees across heterogeneous systems, managing late or out-of-order events, and handling evolving schemas in regulated telecom environments. Moreover, there is limited published work demonstrating containerized, telecom-specific streaming architectures. This gap motivates the present study, which integrates these technologies in a reproducible Docker-based environment tailored for CDR analytics.

III. METHODOLOGY

➢ *System Architecture*

The proposed pipeline is designed as an end-to-end streaming framework that ingests synthetic Call Detail Records (CDRs) into Apache Kafka, persists them in HDFS via Kafka Connect, performs compaction and query operations with PySpark, and orchestrates lifecycle tasks through Apache Airflow. All services are containerized with Docker and interconnected via a user-defined bridge network (hadoop net), ensuring reproducibility and portability.

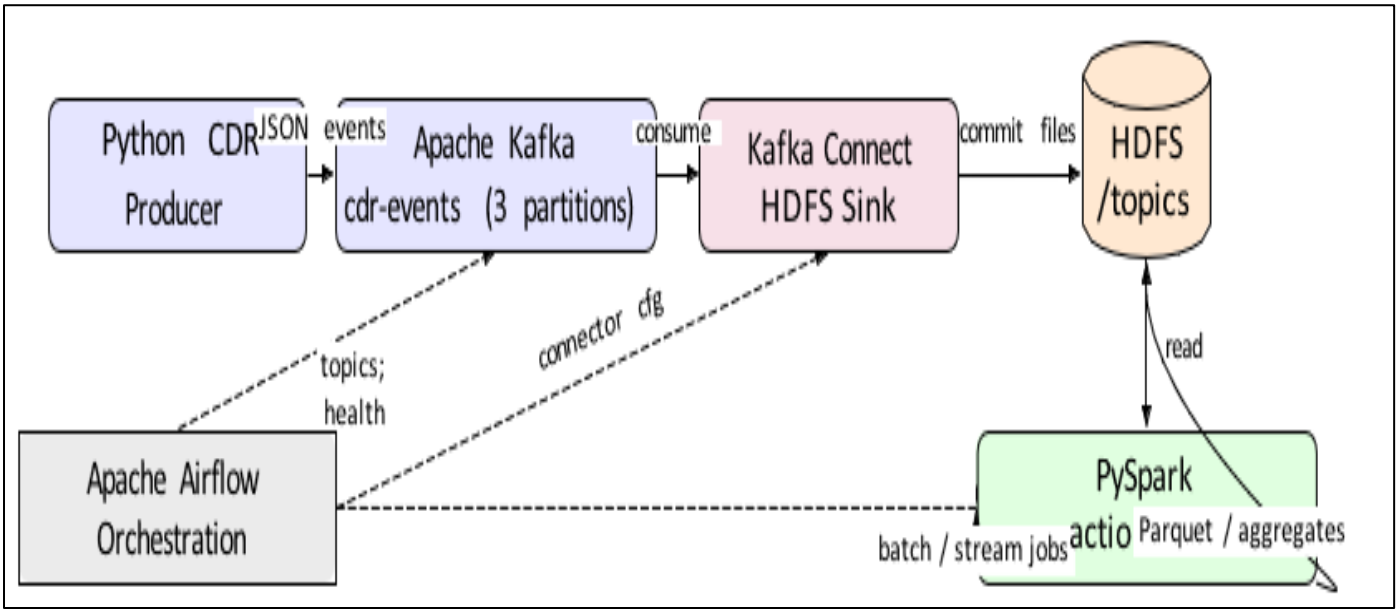

Fig 1 End-to-End CDR Streaming Pipeline with Kafka Ingestion, HDFS Landing, PySpark Analytics, and Airflow Orchestration.

➢ *Core Components*

- Kafka 3.7 (KRaft): single broker hosting the topic cdr-events with three partitions for parallelism.
- Kafka Connect 7.6.1: distributed worker configured with the *HDFS Sink* plugin to land CDR data into HDFS.
- HDFS 3.3.6: one NameNode and two DataNodes with a writable root directory (/topics) serving as the data lake.
- Airflow 2.9.3: webserver and scheduler (LocalExecutor) managing orchestration, retries, and task dependencies.
- PySpark 3.5: employed for data compaction (Parquet

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/25sep1309

conversion) and exploratory analytics, optionally accessed via a Jupyter UI.

➢ *Environment*

The system runs on a single Windows host with Docker Desktop (WSL2 backend). Con-tainer images are explicitly version-pinned: bitnami/kafka:3.7, confluentinc/cp-kafka-connect:7. local/hadoop:3.3.6, apache/airflow:2.9.3, and a PySpark/Jupyter image. Servicenames (e.g., namenode, kafka) resolve automatically via Docker DNS.

➢ *Data & Workload*

Synthetic JSON-based CDRs are generated using a Python producer at a configurable request rate (RPS). Records cover both:

- Voice events: call id, caller, callee, duration seconds, call result.
- Data events: app, bytes up, bytes down.

All events share common fields such as event time, operator, cell id, tower id, roaming, and cost rwf. To evaluate robustness, the workload may also inject malformed records, which are redirected to a Dead Letter Queue (DLQ).

➢ *Messaging & Sink Configuration*

We ingest all CDRs to a single Kafka topic (cdr-events) with three partitions (RF=1 for this single-host demo). Kafka Connect (HDFS Sink) consumes the topic and writes line-delimited JSON to HDFS, with tasks.max matched to partitions and size-only rotation. We sweep flush.size to study latency–file size trade-offs while keeping rotate.interval.ms=0. Schemaless JSON is used throughout, and a DLQ is enabled to quarantine malformed records. Table 1 summarizes the exact settings used in all runs.

Table 1 Core Connector & Topic Settings.

Kafka topic:	**cdr-events**
Partitions	3 (balances producer load; enables up to three parallel sink tasks). Replication factor 1 (single-host demo).
Payload	Line-delimited JSON.
Kafka Connect:	**HDFS Sink**
Target	hdfs.url=hdfs://namenode:9000, root topics.dir=/topics. Parallelism tasks.max=3 (match number of Kafka partitions).
Batching / rotation	flush.size swept [200, 2000] to study latency vs. file size; rotate.interval.ms=0 (size-only rotation).
Converters	Schemaless JSON via value.converter=org.apache.kafka.connect.json.JsonConver and value.converter.schemas.enable=false.
Format	format.class=io.confluent.connect.hdfs.json.JsonFormat (line-delimited JSON files).
Partitioning / layout	Default partitioner ⇒partition={i} directories per Kafka parti- tion; downstream PySpark compacts by event date.
Error handling	errors.tolerance=all; DLQ topic cdr-events-dlq with headers and logging enabled.

➢ *Operational Notes.*

- Throughput vs. Latency: Larger flush.size values reduce HDFS overhead but increase ingestion latency.
- Scaling: tasks.max should not exceed the number of partitions; higher partition counts are needed for greater parallelism.
- Permissions: write access to /topics must be granted to the Connect user inside the container to avoid task failures.
- Monitoring: connector/task status is tracked via Connect REST APIs, while consumer lag is observed in Kafka UI. Lag stability indicates balanced flush/rotation thresholds.

➢ *Orchestration with Airflow*

Pipeline execution is orchestrated by a single idempotent DAG, cdr delivery check, composed of four PythonOperator tasks (Fig. **??**):

- check connector running — validates that the Kafka Connect HDFS sink (hdfs-sink-cdr) exists and is running, reconfiguring if needed.
- Count kafka produced — computes record counts and consumer lag for the cdr-events topic, storing results in XCom.

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/25sep1309

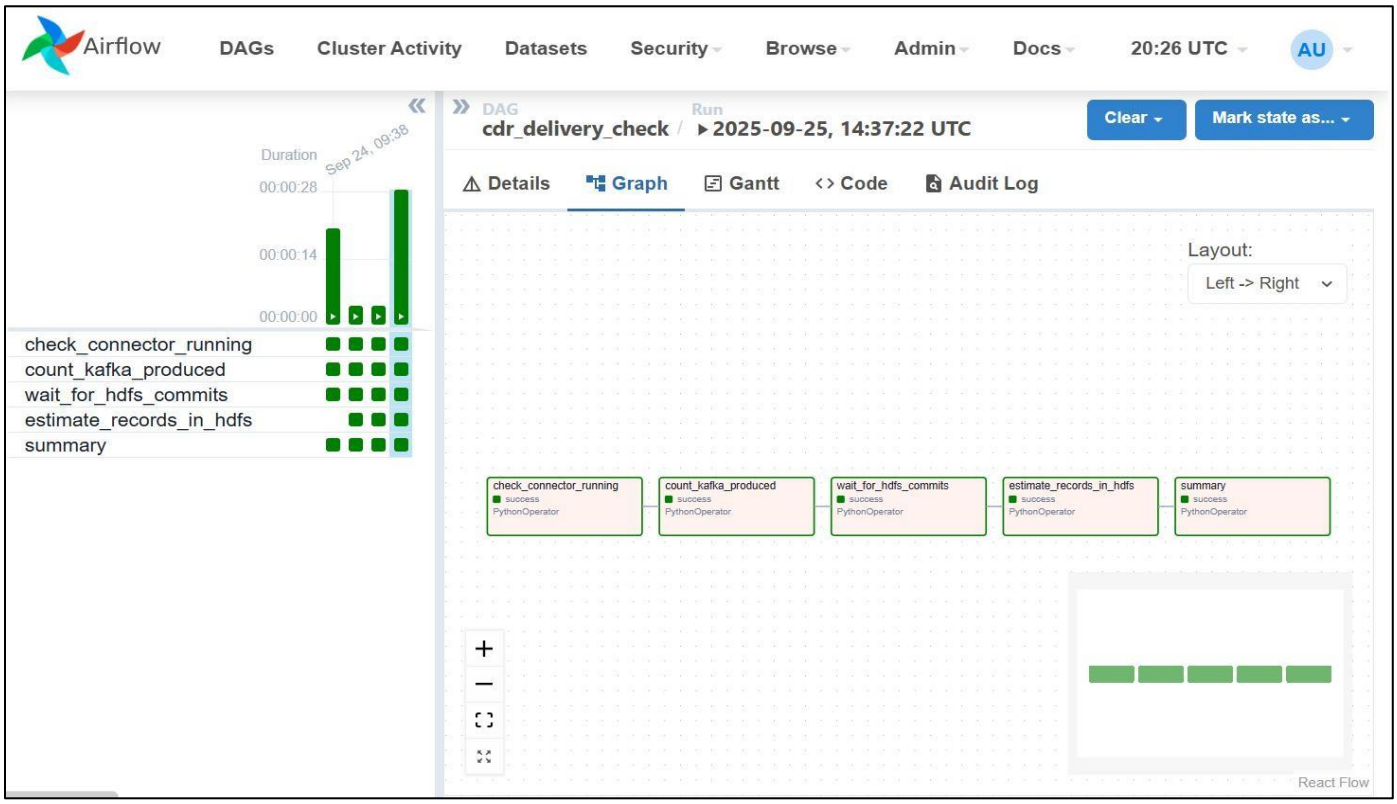

Fig 2 Airflow DAG Schematic for CDR Delivery Check.

- wait for hdfs_commits — waits until committed, non-empty files appear in HDFS under /topics/cdr-events/partition={i}, confirming successful flush.
- summary — compiles metrics including throughput (records/s), file counts and sizes, and latency estimates (p50, p95), saving outputs for visualization.

IV. RESULTS

➢ *Operational*

Connector health. The HDFS sink (hdfs-sink-CDR) consistently reported RUNNING. The consumer group maintained three members (matching the three Kafka partitions) in a stable state throughout the runs.

Ingestion behavior. With tasks.max=3 and size-only rotation (rotate.interval.ms=0), consumer lag remained at or near zero for a producer rate of approximately 25 RPS. Committed files appeared continuously per partition with monotonically increasing offset ranges, indicating steady ingestion and deterministic rotation.

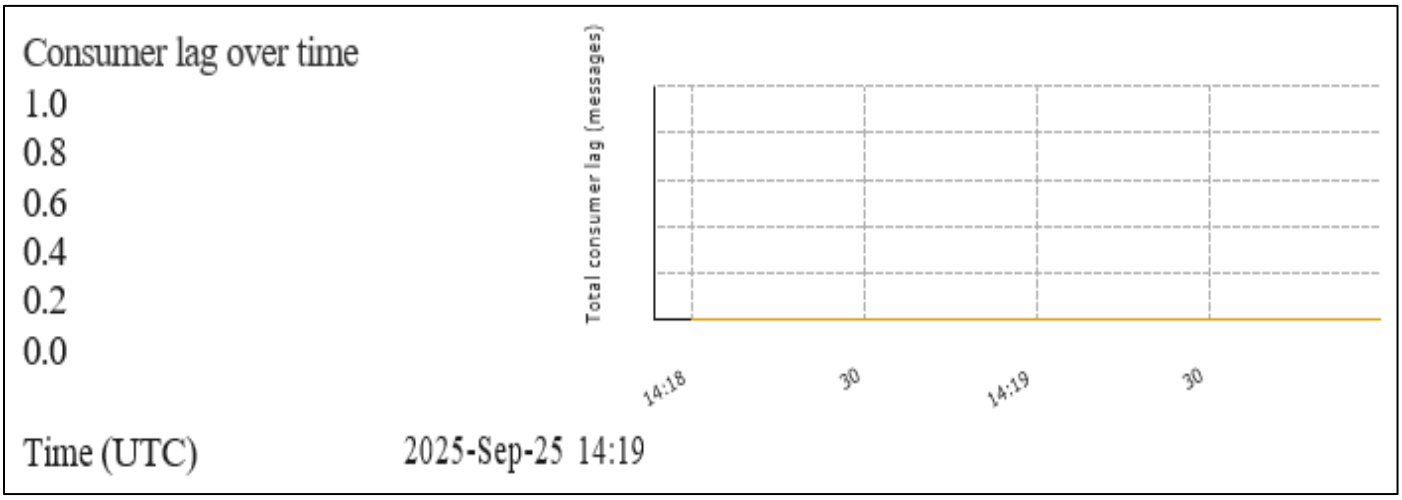

Fig 3 Consumer Lag Over Time.

Interpretation of Fig. 3. Consumer lag remains at 0 messages throughout the run, indicating the HDFS Sink keeps pace with production at ~25 RPS. The absence of spikes or drift shows no backlog accumulation, steady per-partition commits, and that the pipeline is ingestion-bound (not consumer-limited) at this load.

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/25sep1309

➢ *Storage.*

The HDFS NameNode UI indicated 2.00 healthy DataNodes (Hadoop 3.3.6) with low utilization. A recursive listing (hdfs dfs -ls -R /topics/cdr-events) showed steady growth of the per-partition directories and no orphaned +tmp artifacts, indicating clean file closure and commit behavior by the HDFS Sink. The aggregate footprint from hdfs dfs-count -v /topics/cdr-events is summarized in Table 2: 31,420.00 committed objects totaling 110.18 MiB (115,528,793.00 B), with an average object size of 3,676.92 B and approximately 10,473.33 files per partition. Because this demonstration uses a replication factor of 1, reported bytes reflect single-replica storage.

Table 2 Storage Footprint Under /Topics/CDR-Events.

Path	Files Total [B]	Total [MiB]	Avg [B]	Files/part.
/topics/cdr-events	31,420.00115,528,793.00	110.18	3,676.92	10,473.33

➢ *Per-Partition Throughput Balance*

We aggregated committed records per Kafka partition over the last 6 hours to check for distribution skew with tasks.max=3. Ideally the three bars are similar; large gaps indicate partition skew or uneven task capacity. Figure 4 shows near-uniform throughput across partitions.

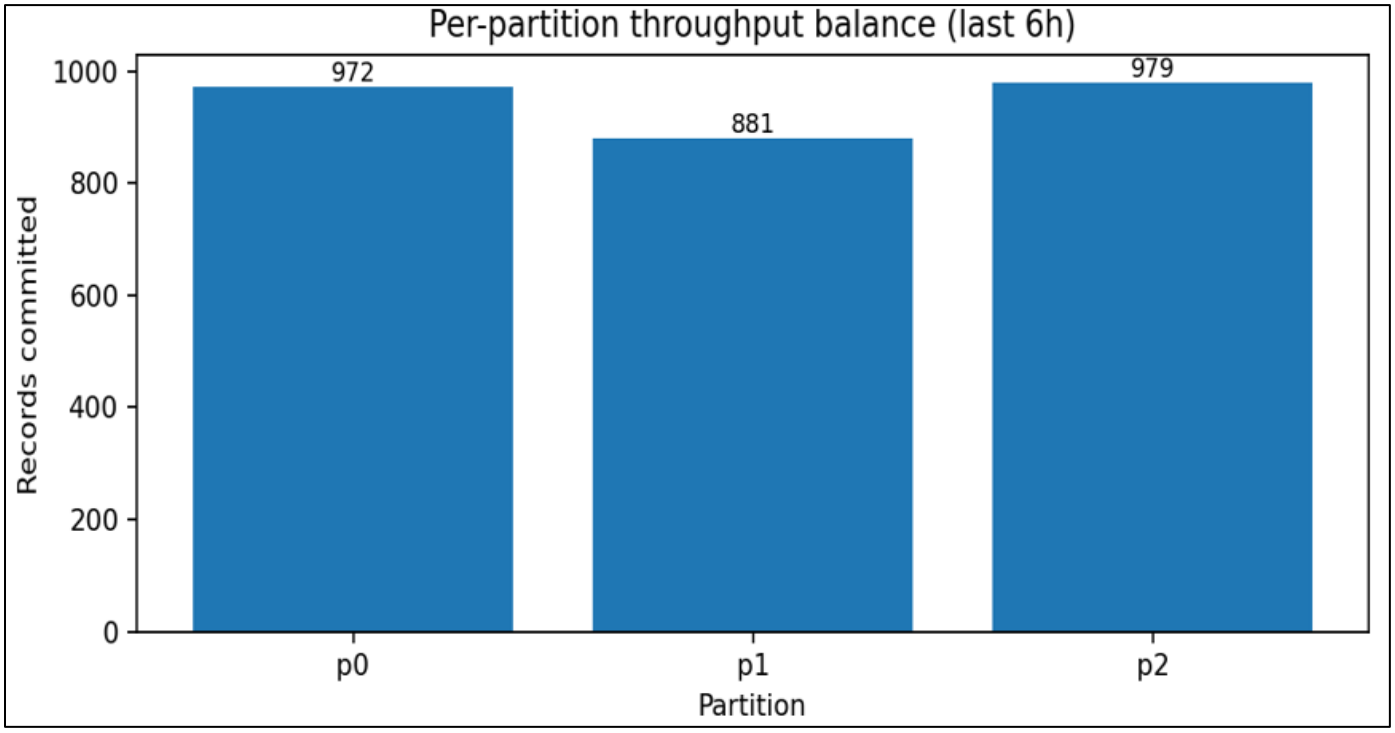

Fig 4 Per-Partition Throughput Balance (Records Committed in the Last 6 Hours). Over the last 6 h the Committed Records Per Partition Were **p0** = 972, **p1** = 881, **p2** = 979 (Fig. 4). Throughput is Nearly Even; **p1** is ∼9% Below p0/p2, which is Typical Short-Window Variance (Timing, Retries, Message size) and does not Indicate Skew or Backpressure. We Would Only Investigate if an Imbalance Persisted Above **20**%.

➢ *Delivery Integrity (Produced vs. Landed)*

Table 3 summarizes recent runs of cdr delivery check. In each run, the number of records produced–matched those landed in HDFS (delivery gap = 0.00%), demonstrating lossless end-to-end ingestion. These results indicate that the Kafka → Kafka Connect (HDFS Sink) → HDFS path is keeping pace with the producer at the tested loads, and that the configured three-way parallelism (three topic partitions) prevents file commit anomalies.

Table 3: Produced vs. landed counts for the latest DAG runs of cdr delivery check.

Run Id	Produced	Landed	Diff	Gap (%)
manual___ 2025-09-25T22:52:17.769125+00:00	1,611	1,611	0	0.00%
manual___ 2025-09-25T14:37:22.639462+00:00	7,122	7,122	0	0.00%

➢ *Analytical Samples*

We computed light-weight aggregates in PySpark to validate downstream usability. Table 4 shows that roaming traffic is stable at about 3.2–3.3% of total CDRs across the three study days. Table 5 lists the most active cells by event volume together with distinct-caller estimates, illustrating how the landed data can support capacity or anomaly analyses at the cell level.

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/25sep1309

Table 4 Daily Roaming Share Computed as Roaming Events/total. The Spark Output Included a NULL Date Row with Zero Counts; it is Omitted Here.

Date	Roaming Events	Total Events	Roaming Share
2025-09-22	3,124	94,803	3.30%
2025-09-24	1,523	47,455	3.21%
2025-09-25	2,667	80,116	3.33%

Table 5 Most Active Cells by Event Count with Distinct-Caller Estimates.

cell_id	events	unique_callers
cell-0308	717	264
cell-0366	678	262
cell-0228	674	262
cell-0312	667	252
cell-0053	660	241

V. CONCLUSION

We implemented an end-to-end CDR streaming pipeline using a Python producer (Kafka), a Kafka Connect HDFS sink, HDFS storage, PySpark analytics, and Apache Airflow orchestration. At a target load of ∼25 RPS, the consumer lag remained effectively zero, the connector stayed healthy, and recent runs showed a 0% delivery gap between produced and landed records. Partition throughput was well balanced across the three topic partitions, confirming parallelism end-to-end. With flush.size=10 the system performs deterministic, size-only rotation; this yields many small files (≈31k files, ≈110 MiB total, ≈3.7 KiB average) that are still read interactively by PySpark. Illustrative aggregates— including daily roaming share (≈ 3.2–3.3%) and top-cell activity—were computed within seconds, indicating that the landing layout is immediately queryable.

➢ *Future Work*

- Integrate Schema Registry; adopt Avro/Parquet at ingestion and enforce schema contracts.
- Scale to multi-broker Kafka and replication factor ≥ 2; benchmark higher RPS and larger partition counts.
- Enable authentication/authorization (SASL/SSL; Kerberos) and RBAC for production parity.
- Replace raw folders with a table format (Delta Lake/Iceberg/Hudi) to gain ACID properties and time travel.
- Introduce near-real-time transforms (Spark Structured Streaming or Flink) and formal- ize DLQ monitoring/alerting.
- Automate compaction and file-sizing policies; add latency/throughput dashboards in Airflow.

ACKNOWLEDGMENT

First, I thank the Almighty God for the gift of life, strength, and guidance throughout this work. I am deeply grateful to my supervisor, Dr. Thulani Nyandeni, for clear direction, timely reviews, and constant encouragement from design to final writing.

I also thank the lecturers of the Master's in Big Data Analytics at Adventist University of Central Africa. Your courses, labs, and feedback provided the skills and confidence to build and evaluate this pipeline.

My heartfelt thanks go to my family for their love, prayers, patience, and practical support. You're understanding during long hours of study made this project possible.

REFERENCES

[1]. J. Kreps, N. Narkhede, and J. Rao, "Kafka: A Distributed Messaging System for Log Processing," in *NetDB (co-located with SIGMOD)*, 2011. [Online]. Available: https://notes.stephenholiday.com/Kafka.pdf

[2]. K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *MSST*, IEEE, 2010. [Online]. Available: https://pages.cs.wisc.edu/~akella/CS838/F15/838-CloudPapers/hdfs.pdf

[3]. M. Zaharia, T. Das, H. Li, *et al.*, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," in *SOSP*, 2013. [Online]. Available: https://people.csail.mit. edu/matei/papers/2013/sosp spark streaming.pdf

[4]. M. Armbrust, T. Das, J. Torres, *et al.*, "Structured Streaming: A Declara- tive API for Real-Time Applications in Apache Spark," in *SIGMOD*, 2018. doi: 10.1145/3183713.3190664. [Online]. Available: https://people.eecs.berkeley.edu/~matei/papers/2018/sigmod structured streaming.pdf

[5]. Apache Software Foundation, "Apache Airflow Documentation (Stable)," 2025. [On- line]. Available: https://airflow.apache.org/docs/apache-airflow/stable/index.html

[6]. Apache Software Foundation, "Airflow Scheduler," 2025. [Online]. Available: https: //airflow.apache.org/docs/apache-airflow/stable/administration-and-deployment/scheduler.html

ISSN No: -2456-2165 https://doi.org/10.38124/ijisrt/25sep1309

[7]. Docker, Inc., "Docker Overview," 2024. [Online]. Available: https://docs.docker.com/ get-started/docker-overview/
[8]. Docker, Inc., "What is a container?" 2025. [Online]. Available: https://docs.docker. com/get-started/docker-concepts/the-basics/what-is-a-container/
[9]. Confluent, Inc., "HDFS 3 Sink Connector for Confluent Platform," 2025. [Online]. Available: https://docs.confluent.io/kafka-connectors/hdfs3-sink/current/overview. html
[10]. Cloudera, " HDFS Sink Connector (Cloudera Runtime 7.3.1)," 2023. [On- line]. Available: https://docs.cloudera.com/runtime/7.3.1/kafka-connect/topics/ kafka-connect-connector-hdfs-sink.html
[11]. W. H. Inmon, *Building the Data Warehouse*, 4th ed. Wiley, 2005.
[12]. R. Kimball and M. Ross, *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd ed. Wiley, 2013.
[13]. N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Real-Time Data Systems*. Manning Publications, 2015.
[14]. M. Chen, S. Mao, and Y. Zhang, "Big Data: A Survey," *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, 2014. Springer.