Streaming Data Lakehouse Integration: Moving from Kafka Topics to Cloud Data Lakes in AWS & Azure via Confluent Connectors

Girish Rameshbabu

Customer Success Technical Architect
girish.prasad.23@gmail.com

Abstract:
The emergence of the Data Lakehouse—fusing the scale of cloud object storage (AWS S3, Azure ADLS Gen2) with the transactional governance of a data warehouse—necessitates a robust mechanism for handling high-velocity event streams from Apache Kafka. This paper addresses the architectural challenge of reliably bridging Kafka topics to Lakehouse formats (Delta Lake, Apache Hudi, Apache Iceberg) while ensuring three critical constraints are met: consistency (ACID), timeliness (low latency), and schema integrity. We propose and evaluate two primary ingestion patterns utilizing Confluent Sink Connectors: Direct Micro-Batch Ingestion (Pattern 1) and Stream Processor-Aided Ingestion (Pattern 2). The analysis demonstrates that Pattern 2, which incorporates an intermediate stream processor for inline deduplication and enrichment, is superior for high-churn/Change Data Capture (CDC) workloads. Furthermore, we establish that transactional consistency and graceful schema evolution are maintained through the layered cooperation of the Confluent Schema Registry (enforcing data contracts pre-landing) [11] and the Lakehouse Table Format (implementing atomic commits post-landing) [4]. The approach confirms the architectural portability of this streaming integration strategy across both AWS and Azure cloud ecosystems, providing a standardized blueprint for real-time analytics and machine learning applications.

I. INTRODUCTION

1.1 Motivation and Background

The architecture of data storage and analytics has undergone a significant paradigm shift driven by the imperative for real-time decision-making [1], [2]. Historically, enterprises relied on a bifurcated system: Data Warehouses (DW) provided high-performance structured queries and ACID compliance for Business Intelligence (BI), while Data Lakes (DL), leveraging affordable cloud object storage (e.g., AWS S3 and Azure Data Lake Storage Gen2), stored vast volumes of raw, multi-structured data for machine learning (ML) and exploration.

This duality created critical operational and analytical bottlenecks. As detailed by Jain et al. [7], the separation of these systems resulted in data redundancy, increased governance complexity, and stale insights. Specifically, the necessity of periodic batch processing meant that insights derived from transactional DWs lagged real-world events, undermining the potential of the high-velocity, continuous data streams managed by platforms like Apache Kafka. The problem of data lakes becoming "data swamps" without proper governance further motivated the need for a new architecture [3].

The Data Lakehouse emerged to resolve this tension, fusing the flexibility and cost-efficiency of the data lake with the transactional and governance features of the data warehouse [4], [5]. The core tenet of the Lakehouse, as defined by its core enabling technologies, is to bring fundamental DW properties—specifically Atomicity, Consistency, Isolation, and Durability (ACID)—and schema enforcement directly onto cloud object storage [4], [5].

1.2 Problem Statement: Bridging the Stream-to-Lakehouse Gap

While the foundational use case of streaming data from Kafka topics into object storage for analytics is well-established in reference architectures [9], [10], the integration with the modern transactional Lakehouse introduces new complexity. Apache Kafka serves as the de facto standard for event streaming, ensuring reliable, high-throughput log delivery. However, the objective of this work is not merely to land raw data (the traditional DL approach) but to land transactionally consistent data ready for immediate consumption (the Lakehouse approach) [6].

The primary architectural challenge lies in efficiently and reliably translating the high-velocity, continuous event stream from Kafka topics into the transactional upserts and optimized file layouts required by Lakehouse formats (Delta Lake, Apache Hudi, Apache Iceberg). This translation process must simultaneously satisfy three architectural constraints:

1. Consistency: Ensuring the eventual consistency writes from a Kafka Sink Connector are reconciled by the Lakehouse framework to maintain ACID properties.
2. Timeliness: Minimizing end-to-end latency (from Kafka produce time to query-ready Lakehouse snapshot) while avoiding the storage query performance degradation associated with the small file problem.
3. Schema Management: Guaranteeing data contract integrity using the Confluent Schema Registry to safely govern schema evolution across the boundary between the mutable stream and the structured Lakehouse table.

This paper proposes and evaluates design patterns utilizing Confluent Connectors as the core integration mechanism. We analyze how Connector configuration and optional stream processing layers can be strategically deployed to address these constraints across multi-cloud environments (AWS S3 and Azure Data Lake Storage Gen2), thereby realizing a truly integrated Streaming Data Lakehouse architecture.

II. BACKGROUND AND RELATED WORK

To establish a transactional streaming pipeline from an event backbone to cloud-native storage, the following architectural components must be understood.

2.1 The Apache Kafka and Confluent Ecosystem

Apache Kafka serves as the distributed, fault-tolerant commit log for event streaming, providing ordered, immutable, and durable storage for messages in partitions. For integrating Kafka with external data stores, the Kafka Connect framework is utilized. Kafka Connect is an integral part of the data ingestion and egress strategy, running specialized processes called Sink Connectors to stream data from Kafka topics into a target system.

The Confluent S3 Sink Connector [12] and the Azure Data Lake Storage Gen2 (ADLS Gen2) Sink Connector [13] are critical components of this work. Their function is to poll records from specified Kafka topics, buffer them into files (often Parquet or Avro), and commit these files to the respective cloud object storage. Key configurations for these connectors, such as flush. Size, time-based file rotation, and partitioning strategies, directly influence the trade-off between timeliness (low latency) and the creation of optimized, large files necessary for efficient Lakehouse querying. Both connectors support exactly-once semantics within the Kafka Connect framework, ensuring records are processed once, although the final transactional integrity in the Lakehouse is governed by the table format itself.

2.2 Confluent Schema Registry and Data Contract Enforcement

A key vulnerability of the traditional data lake is the risk of schema drift and data corruption, commonly known as "garbage in, garbage out." The Confluent Schema Registry addresses this by providing a centralized repository for managing and validating schemas (typically Avro or JSON Schema) used in Kafka messages [11].

The Schema Registry enforces a strict data contract between producers and consumers. When a producer attempts to write data with a new schema, the Registry checks it against defined compatibility rules (e.g., BACKWARD, FORWARD, or FULL) before acceptance [11].

- Backward Compatibility: Ensures that the new schema can read data written by the previous schema version (critical for consumers reading historical data in the Lakehouse).
- Forward Compatibility: Ensures that the previous schema version can read data written by the new schema (critical for legacy consumers).

By utilizing the Schema Registry, the Sink Connector is guaranteed to ingest data that conforms to a versioned schema, enabling the downstream Lakehouse formats to safely manage schema evolution by updating only their metadata layer rather than rewriting the underlying data files.

2.3 The Data Lakehouse: ACID and Open Formats

The Lakehouse architecture is fundamentally enabled by open-source table formats that overlay cloud object storage, introducing database-like transactional guarantees. This mechanism directly resolves the consistency challenge faced when streaming data lands in an eventual-consistency storage system like S3 or ADLS Gen2 [4], [7]. The three leading formats are:

Format	Transaction Mechanism	Key Streaming Features
Delta Lake	JSON-based transaction log; Optimistic Concurrency Control (OCC).	Strong integration with Spark; supports MERGE INTO (upserts) and Change Data Feed (CDF).
Apache Hudi	Timeline metadata; supports two table types: Copy-on-Write (COW) and Merge-on-Read (MOR).	Optimized for Change Data Capture (CDC) workloads; supports partial updates.
Apache Iceberg	Snapshot-based metadata tree (Metadata, Manifest List, Manifests); Catalog integration.	Decouples physical file layout from logical table definition; robust schema and partition evolution [7].

These formats achieve transactional integrity by using a metadata layer (either a transaction log or snapshot files) to track the data files belonging to a table at any given moment. A commit operation is atomic: the new set of data files is only exposed to readers once the metadata update is successfully written and validated. This is how the Lakehouse ensures read isolation and the ability to perform crucial data maintenance operations like compaction, indexing, and time-travel, all while receiving continuous event streams from Kafka. Further architectural context is provided in [8].

This established foundation demonstrates that while Kafka provides reliable transport, the Lakehouse formats are the mechanisms that fulfill the requirements for consistency and advanced schema management on cloud storage. Comparisons of the core designs are detailed in [7], [15], [16].

III. PROPOSED DESIGN PATTERNS AND ARCHITECTURE

The integration of high-velocity data streams from Kafka into the transactional Lakehouse (AWS S3/Azure ADLS Gen2) requires strategic design to balance low latency with data consistency and file optimization. We propose two distinct patterns based on the complexity of the data processing needs.

3.1 Pattern 1: Direct Sink Connector Ingestion (Micro-Batch)

The simplest and highest-throughput pattern relies on a direct Confluent Sink Connector pulling data from a raw Kafka topic and committing files to the Lakehouse. This pattern is suitable for simple append-only data streams (e.g., IoT sensor logs, web clickstreams) where there is no need for inline deduplication or complex lookups.

The connector is configured for rapid file rotation (low rotation.interval.ms) and small batch sizes (flush.size) to minimize end-to-end latency, addressing the timeliness challenge. While this aggressively writes data to cloud storage, it generates numerous small files. The Lakehouse table format (Delta, Hudi, or Iceberg) must then utilize asynchronous compaction services to merge these small files into larger, query-optimized files, thus addressing the small file problem post-ingestion.

3.2 Pattern 2: Stream Processor-Aided Ingestion

For streams that require complex transformations, data quality enforcement, or handling of Change Data Capture (CDC) records (which demand upserts and deletes), Pattern 1 is insufficient. Pattern 2 introduces an intermediate stream processing layer—utilizing Kafka Streams, ksqlDB, or Spark Structured Streaming—to curate the data stream before it is committed to the Lakehouse.

This architectural flow (Raw Topic $\rightarrow$ Processor $\rightarrow$ Curated Topic $\rightarrow$ Sink Connector) is necessary for fulfilling the consistency requirement for high-churn data [9], [10], [14]:

1. Deduplication and Keying: The stream processor uses state stores to identify and remove duplicate events or to ensure records are uniquely keyed by a business identifier. This is critical for data quality before upserting into the Lakehouse.
2. Enrichment and Aggregation: Data can be enriched by joining the stream with reference data (e.g., geo-lookup, customer segmentation) or aggregated (e.g., calculating a running sum of transactions per minute).
3. Pre-Compaction Optimization: For formats like Apache Hudi or Delta Lake, the processor can handle the complex logic of applying updates and deleting records before the final write. This approach is superior for CDC workloads, where pre-processing the log of changes simplifies the subsequent atomic merge operation performed by the Lakehouse format [14].

By writing a curated, high-quality, and deduped stream to an intermediary Kafka topic, the Sink Connector operates on an ideal input, significantly improving the Lakehouse's efficiency in managing transactional updates and reducing the burden on its internal compaction services.

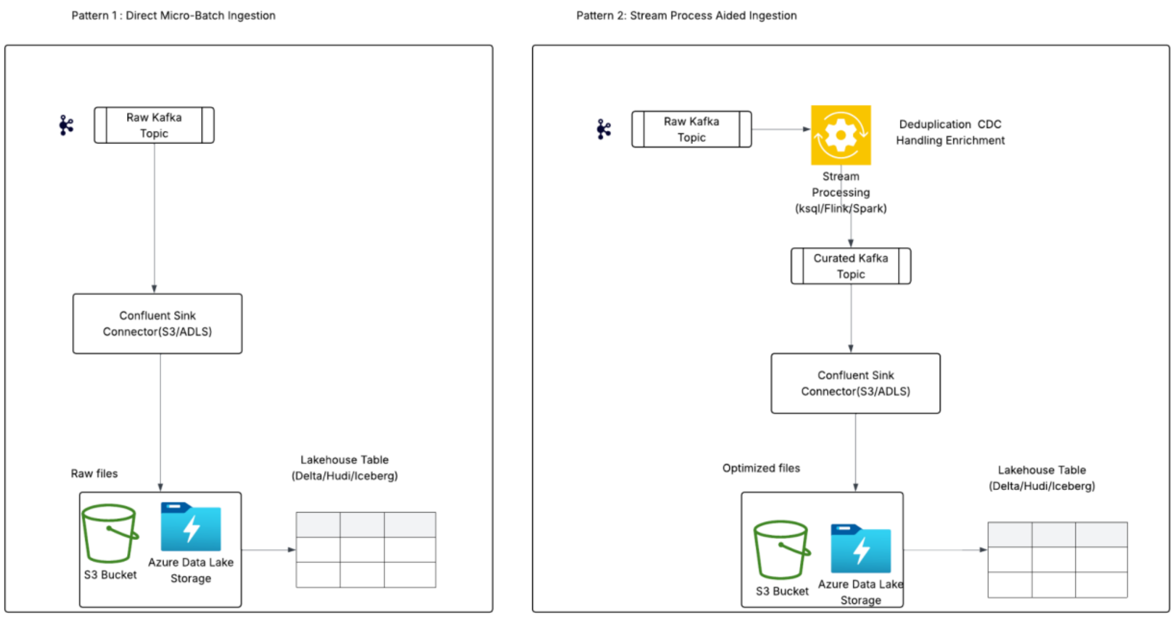
Pattern 1 : Direct Micro-Batch Ingestion
Raw Kafka Topic
Confluent Sink Connector(S3/ADLS)
Raw files
Lakehouse Table (Delta/Hudi/Iceberg)
S3 Bucket
Azure Data Lake Storage
Pattern 2: Stream Process Aided Ingestion
Raw Kafka Topic
Deduplication CDC Handling Enrichment
Stream Processing (ksql/Flink/Spark)
Curated Kafka Topic
Confluent Sink Connector(S3/ADLS)
Optimized files
Lakehouse Table (Delta/Hudi/Iceberg)
S3 Bucket
Azure Data Lake Storage

Figure1: Comparison of Ingestion Patterns. Pattern 1 (left) demonstrates direct micro-batching for append-only logs. Pattern 2 (right) introduces an intermediate processing layer for CDC and deduplication prior to Lakehouse commitment.

3.3 Multi-Cloud Architecture Visualization

The proposed architecture utilizes the Confluent Connect framework to provide a unified data egress layer from a central Apache Kafka cluster to distinct cloud storage targets: AWS S3 and Azure Data Lake Storage Gen2 (ADLS Gen2). This design establishes Kafka as the single source of truth for the event stream while supporting multi-cloud data distribution for redundancy, compliance, or geo-specific analytics.

The connectors are deployed in a fan-out configuration [17], [19]:

- The Confluent S3 Sink Connector [12] is configured with an AWS Identity and Access Management (IAM) role to securely write Parquet or Avro files into a specified S3 bucket.
- The ADLS Gen2 Sink Connector [13] is similarly configured using an Azure Service Principal to write to the designated container in ADLS Gen2.

The key to multi-cloud transactional consistency is ensuring that the same message offsets (or transactional boundaries) are used when committing files to both S3 and ADLS Gen2. This guarantees that both Lakehouse instances—regardless of whether they are running Delta Lake on Databricks/AWS or Iceberg on Azure Synapse—reflect an identical snapshot of the Kafka topic state for a given offset.
This centralized ingestion strategy, coupled with the transactional metadata layer of the Lakehouse formats, enables robust, consistent, and low-latency data availability across heterogeneous cloud storage environments.

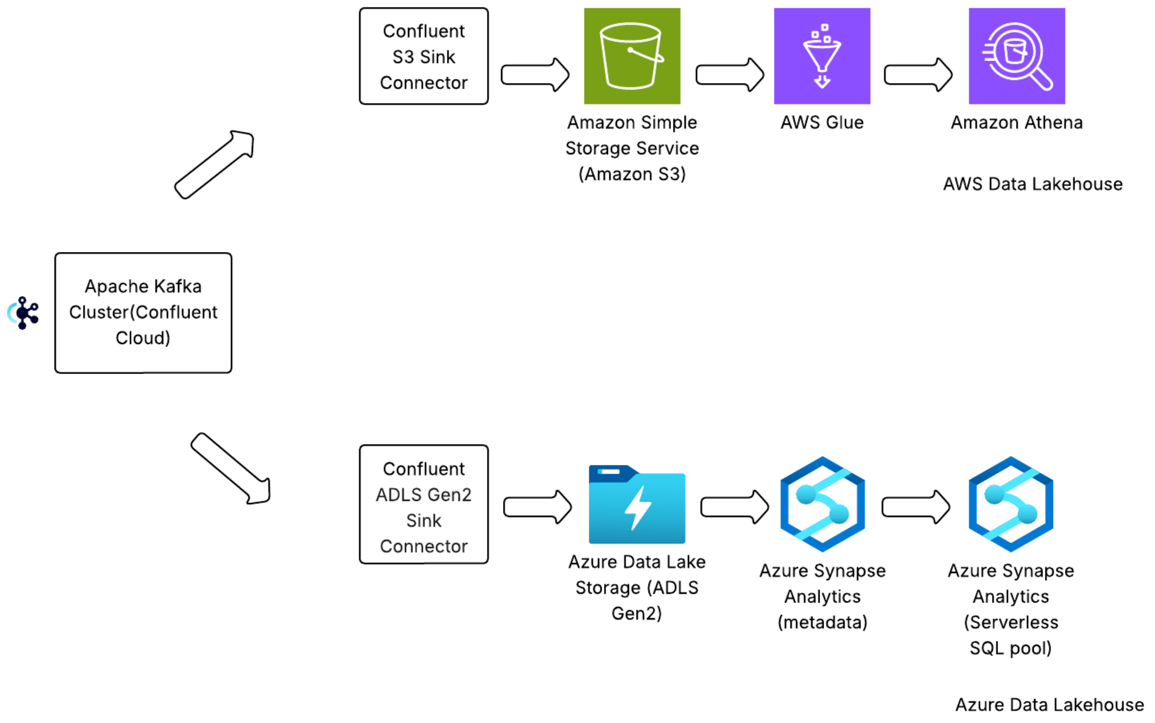
Confluent S3 Sink Connector
Amazon Simple Storage Service (Amazon S3)
AWS Glue
Amazon Athena
AWS Data Lakehouse
Apache Kafka Cluster(Confluent Cloud)
Confluent ADLS Gen2 Sink Connector
Azure Data Lake Storage (ADLS Gen2)
Azure Synapse Analytics (metadata)
Azure Synapse Analytics (Serverless SQL pool)
Azure Data Lakehouse

Figure2: Multi-Cloud Streaming Data Lakehouse Architecture. A unified Kafka integration layer performs a fan-out distribution, ensuring transactional consistency across both AWS and Azure ecosystems simultaneously.

IV. ADDRESSING KEY ARCHITECTURAL CHALLENGES

The successful fusion of a high-velocity Kafka stream with a query-optimized Data Lakehouse is determined by how three key architectural challenges are systematically mitigated.

4.1 Schema Enforcement and Evolution

Maintaining data integrity is paramount, especially when handling heterogenous data sources flowing through Kafka. The Lakehouse ensures schema integrity through a layered approach:

1. Schema Contract Enforcement (Pre-Landing): The Confluent Schema Registry acts as a mandatory gatekeeper. Producers must register and adhere to an official schema (e.g., Avro, JSON Schema) before publishing. This guarantees that all data arriving at the Sink Connector is valid and tagged with a Schema ID. The Sink Connector (e.g., ADLS Gen2 Connector) uses this ID to read and serialize the data, ensuring the files committed to storage are structurally correct [11], [20].
2. Schema Evolution Management (Post-Landing): When an upstream schema changes (e.g., adding a non-nullable field), the Schema Registry ensures the change adheres to the compatibility rules (e.g., BACKWARD). The Lakehouse table format then takes over, handling the evolution purely in the metadata layer [4]. For example, a new column is added to the table definition without rewriting

existing Parquet/Avro data files, which will present a null value for that field upon query. This mechanism eliminates the need for expensive, bulk data rewriting and ensures continuous data availability.

4.2 Achieving Transactional Consistency (ACID)

Cloud object storage (S3/ADLS Gen2) is an eventually-consistent file store, meaning concurrent operations can lead to inconsistent reads. The Sink Connector's role is simply to write files; it does not inherently guarantee transactional consistency. This critical function is delegated entirely to the open Lakehouse Table Format layer (Delta, Hudi, or Iceberg).
These formats implement ACID properties through a separate, dedicated metadata management system [4], [7], [8]:

- Atomicity and Isolation: Every write operation initiated by a Kafka micro-batch is treated as a single, atomic transaction that generates a new snapshot of the table. A transaction is only deemed successful if the updated metadata file (which references the new data files) is successfully written. Readers query the latest, committed snapshot reference provided by the table's central Catalog (e.g., AWS Glue, Hive Metastore). If a write fails mid-process, the previous, valid snapshot remains available for readers, guaranteeing isolation and preventing corrupted reads.
- Time Travel and Rollback: By storing a complete, sequential history of these transactional snapshots in the metadata log, the Lakehouse provides time-travel capabilities. Users can query the table as it existed at a specific timestamp or offset, offering crucial benefits for auditing, debugging, and simple rollback to a previous valid state in case of data pipeline errors.

4.3 Timeliness, Latency, and File Optimization

Balancing low latency (timeliness) with efficient query performance is the fundamental architectural trade-off in streaming ingestion. Frequent, low-latency writes create many small files, which drastically degrade the performance of query engines (e.g., Spark, Athena) due to excessive metadata calls and file opening overhead. The optimization requires careful tuning of the Confluent Sink Connector parameters [12], [13], [18], [25]:

Parameter	Function	Effect on Timeliness	Effect on File Size
flush.size	Maximum records per file.	Lower value $\rightarrow$ Better timeliness	Lower value $\rightarrow$ Smaller files
rotation.interval.ms	Time until a file is closed and committed.	Lower value $\rightarrow$ Better timeliness	Lower value $\rightarrow$ Smaller files

To achieve optimal performance, the connector must be configured for micro-batching (low rotation.interval.ms and flush.size) to ensure data freshness. However, this necessitates asynchronous, parallel compaction jobs, managed by the Lakehouse format itself, to run continuously or periodically. These compaction services merge the numerous small files into optimal-sized files (typically 128 MB or larger) to maintain read performance for analytics and machine learning applications. Thus, the solution to the small file problem is not in the connector's settings but in the cooperative operation between the aggressive micro-batch ingestion (Sink Connector) and the asynchronous file maintenance (Lakehouse Compaction) [18].

V. DISCUSSION AND COMPARATIVE ANALYSIS

The evaluation of the proposed streaming ingestion patterns relies on both quantitative performance metrics and the qualitative fit within the chosen cloud ecosystem. This section discusses these criteria, highlighting the operational distinctions between AWS and Azure tooling.

5.1 Performance Metrics and Lakehouse Behavior

Evaluating the Streaming Data Lakehouse requires shifting focus from simple Kafka throughput to metrics that reflect end-to-end data quality and query efficiency. The performance is highly dependent on the behavior of the underlying Lakehouse format (Delta, Hudi, Iceberg) as it handles the micro-batched file commits from the Sink Connector.

The critical metrics for comparative analysis include:

- Write Latency (Timeliness): The duration from a record being published to Kafka to the point it is committed and available for querying in the Lakehouse catalog. This metric is primarily controlled by the Connector's rotation.interval.ms and the Lakehouse format's metadata commit time. Concepts are detailed in [1], [2].
- Query Performance: Measured by the time taken for analytical queries (BI/ML) to execute. This is inversely proportional to the effectiveness of compaction, as excessive small files lead to high read amplification and poor performance. Benchmarks are provided in [7], [5].
- Compaction Overhead: The cost and processing time required by the asynchronous services to merge small files. Formats like Apache Hudi's Merge-on-Read (MOR) trade higher write latency for lower query overhead, while Iceberg's structure can be sensitive to small file creation without dedicated compaction [7].

Pattern 2 (Stream Processor-Aided) generally exhibits superior data consistency and query performance for high-churn/CDC workloads because the costly operations (deduplication, key lookups) are performed in the stream processor rather than asynchronously in the Lakehouse, minimizing the complexity of the transactional merge operation.

5.2 Multi-Cloud Tooling Comparison

The final architecture must integrate seamlessly with the native tooling provided by the target cloud providers (AWS and Azure). The choice of cloud environment dictates the specific services used for cataloging, governance, and running the primary query workload over the data ingested by the Confluent Connectors [3].

The multi-cloud integration architecture can be summarized by mapping the functional layers:

Functional Layer	**AWS (Amazon Web Services)**	**Azure (Microsoft Azure)**
Object Storage (DL)	Amazon S3 (S3 Standard, Infrequent Access)	Azure Data Lake Storage Gen2 (ADLS Gen2)
Kafka Bridge	Confluent S3 Sink Connector [12]	Confluent ADLS Gen2 Sink Connector [13]
Data Catalog & Governance	AWS Glue Catalog, AWS Lake Formation	Azure Synapse Analytics (Metadata), Azure Purview
Serverless SQL Query	Amazon Athena, Amazon Redshift Spectrum	Azure Synapse Serverless SQL Pool
Managed Spark/Lake Compute	AWS EMR, Amazon MSK (for Kafka Connect)	Azure Databricks, Azure Synapse Apache Spark Pool

AWS traditionally relies on AWS Glue for cataloging and Athena for serverless query over S3 [23], providing a decoupled, service-oriented ecosystem. Azure provides a more integrated Lakehouse experience through Azure Synapse Analytics (which houses serverless SQL and Spark pools) and Azure Databricks [24]. The ability to use managed Confluent Connectors within Azure further solidifies this integration [19].

The integration point for the Confluent Connectors remains consistent: they commit serialized Avro/Parquet files to the respective object storage, and the Cloud Catalog (Glue or Synapse) is then responsible for reading the Lakehouse format's metadata log to expose the transactional tables to the query engines. The overall architectural principles of schema enforcement and transactional integrity remain constant, demonstrating the portability of the open Lakehouse formats across cloud platforms.

VI. CONCLUSION AND FUTURE WORK

6.1 Conclusion

This paper addressed the critical architectural challenge of reliably bridging the gap between high-velocity event streams originating from Apache Kafka and the transactional, governed environment of the cloud Data Lakehouse (AWS S3 and Azure ADLS Gen2).

We demonstrated that the successful implementation of a Streaming Data Lakehouse hinges on a layered architectural strategy:

1. Ingestion Flexibility: Two design patterns were proposed—Direct Sink Connector Ingestion for high-throughput, append-only streams, and Stream Processor-Aided Ingestion (Pattern 2) for complex Change Data Capture (CDC) and high-churn workloads requiring pre-deduplication and enrichment.
2. Transactional Integrity: The core requirements of Consistency and Schema Evolution are satisfied not by the connectors themselves, but by the cooperative interaction of the Confluent Schema Registry (enforcing data contracts pre-landing) and the Lakehouse Table Format (implementing ACID properties via snapshot metadata and asynchronous compaction post-landing).
3. Portability: The use of open standards (Kafka, Confluent Connect, Delta/Hudi/Iceberg) ensures architectural portability, with the differences between AWS (Glue, Athena) and Azure (Synapse, Databricks) confined to the tooling used for cataloging and query execution over the consistent storage layer.

Pattern 2 emerged as the superior solution for achieving maximum data quality and consistency in business-critical use cases, despite the added operational complexity of maintaining the stream processing layer.

6.2 Future Work

The convergence of streaming and transactional storage opens several avenues for future research and development, particularly concerning the seamless integration of analytics and machine learning:

- Unified Streaming Feature Stores: The Lakehouse is ideally positioned to serve as the unified storage layer for both streaming and batch data, enabling the development of integrated Feature Stores [4], [5]. Future work should focus on optimizing the Kafka-to-Lakehouse pipeline for immediate low-latency reads by ML models, ensuring that streaming features (e.g., rolling averages, real-time counts) generated in Kafka Streams/ksqlDB can be efficiently materialized into Lakehouse formats (e.g., Hudi MOR tables) to support online and offline model training and serving [22].
- Low-Latency Query Layers over Lakehouse Tables: While current query engines like Athena and Synapse Serverless SQL are effective, their performance on minute-level micro-batches can still introduce latency. Further research is needed to evaluate and optimize technologies (such as specialized indexing layers or incremental query engines) that can provide sub-second latency directly over the highly-compacted and versioned Lakehouse tables, thereby closing the gap between the stream processing layer and the analytical consumption layer [9], [10].
- Automated Multi-Cloud Connector Governance: Developing advanced governance tooling to automatically synchronize IAM/RBAC permissions and encryption keys across AWS and Azure for Confluent Sink Connectors, ensuring a zero-trust environment where data movement policies are uniformly applied across heterogeneous cloud boundaries.

REFERENCES:

[1] Langseth, M. (2007). “Real-Time Data Warehousing: Challenges and Solutions.” *Journal of Database Management*.
[2] S Bouaziz et al. “From Traditional Data Warehouse To Real Time Data Warehouse, Research Gate
[3] “Architecting Data Lake-Houses in the Cloud: Best Practices and Future Directions,” *International Journal of Scientific Research in Architecture* (IJSRA), 2024.

[4] Armbrust, M., et al. (2021). “Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics.” *CIDR*.
[5] Harby, R., et al. (2024). “Data Lakehouse: A Survey and Experimental Study.”
[6] “The Data Lakehouse: Data Warehousing and More.” *arXiv*, 2023.
[7] Jain, P., et al. (2023). “Analyzing and Comparing Lakehouse Storage Systems.” *CIDR*.
[8] “Spatial Big Data Architecture: From Data Warehouses and Data Lakes to the LakeHouse.”
[9] “Kafka-based Architecture in Building Data Lakes for Real-time Data Processing,” *International Journal of Computer Applications* (IJCA), 2023.
[10] “A Data Lake Architecture Using Kafka,” *International Journal of Advanced Research in Science, Engineering and Technology* (IJARSET), 2019.
[11] Confluent Schema Registry. “Schema Evolution and Compatibility” documentation and best-practices article.
[12] Confluent. Amazon S3 Sink Connector overview and configuration reference.
[13] Confluent. Azure Data Lake Gen2 Sink Connector documentation.
[14] Wähner, S. (2022). “Serverless Kafka in a Cloud-native Data Lake Architecture.” *Blog*.
[15] lakeFS. “Hudi vs Iceberg vs Delta Lake: Detailed Comparison.”
[16] Onehouse. “Apache Hudi vs Delta Lake vs Apache Iceberg: Lakehouse Feature Comparison.”
[17] IBM. “Kafka Connect to S3 Source & Sink” reference architecture.
[18] Community Q&A and guides on tuning flush.size / rotation.interval.ms / rotate.schedule.interval.ms for S3 connectors.
[19] Microsoft Learn. “Use Confluent Connectors in Azure (preview) – Azure Native Integrations.”
[20] “Kafka Schema Evolution: A Guide to the Confluent Schema Registry.”
[21] ADLS Gen2 connector configuration docs for record-count-based chunking.
[22] Dremio. “State of the Data Lakehouse 2024” survey.
[23] AWS Glue/Athena documentation and comparison guides.
[24] Azure Synapse and ADLS Gen2 docs, plus community discussions on “Azure equivalent of S3/Glue/Athena.”
[25] Confluent S3 Sink Connector configuration reference for all batching/rotation properties.