Hybrid Table Formats for Lakehouse Systems: A Functional Benchmark of Iceberg, Hudi, and Delta

Srinivas Lakkireddy

Independent Researcher, USA
Email: reachlakkireddy@gmail.com

***Abstract*—The swift development of the Lakehouse architectures has brought hybrid table formats such as Apache Iceberg, Apache Hudi, and Delta Lake that enmesh the transactional guarantees of data warehouses with the scalability of data lakes. At the same time, they are increasingly used in enterprise data pipelines; existing benchmarking studies typically do not include an integrated comparison of these formats across workload and functional dimensions. Previous studies mainly focus on individual performance metrics or narrow case-prone analysis, and hence lack a comprehensive overview of feature support and suitability for hybrid workloads. The current study aims to fill this gap by providing a functionality-driven benchmark framework for hybrid table formats that evaluates ingestion throughput, schema evolution, streaming support, compaction efficiency, and concurrency. It presents a Functional Capability Score (FCS) algorithm that uses qualitative output from 15 specific functional test cases to estimate format-specific capabilities. What is it: A reproducible Docker and Apache Spark-based benchmarking testbed that simulates both batch and streaming operations over synthetic and semi-realistic datasets in a fair and scalable manner. According to the experimental results, Delta Lake outperforms in both streaming and batch read speeds. At the same time, Iceberg offers improved schema evolution through metadata-driven table versioning mechanisms, while Hudi provides high ingestion throughput for incremental and streaming workloads and high ingestion throughput. The suggested framework enables a one-to-one mapping of workload types to suitable table formats, thereby enhancing practical Lakehouse deployment efforts. Therefore, this work provides a reproducible, expandable benchmark model that helps data engineers and architects choose suitable table formats, tune for better performance, and achieve future-proof scalability in the most versatile hybrid data lake environments.**

***Keywords*—hybrid table formats, Lakehouse benchmarking, functional capability score, Apache Iceberg Hudi Delta, data pipeline evaluation**

I. Introduction

Lakehouse systems have made a significant impact on data analytics, bringing together the reliability of data warehouses with the low cost, flexibility, and scale of data lakes. Central to this evolution are hybrid table formats like Apache Iceberg, Apache Hudi, and Delta Lake, which provide a foundational layer over the rules of engagement while keeping structured and semi-structured data sane with transactions. Being able to support operations such as time travel, schema evolution, concurrent writes, and even batch and streaming operations means that these formats provide the backbone for strengthened data pipelines that the ever heavier big data stacks require.

Recent works in the literature look deeply into the architectural differences and isolated benchmarking of each table format. Divyeshkumar [1] analyzed hybrid batch and real-time data processing workflows using Apache Spark. Vinnakota [2] investigated hybrid batch–stream processing pipelines under Lambda and Kappa architectures. Saddad *et al.* [3] proposed a Lake Data Warehouse architecture integrating Hadoop and Spark to improve analytical scalability. Delta Lake introduces ACID-compliant transactional storage with unified batch and streaming processing capabilities. Nevertheless, a comprehensive benchmarking of these formats encompassing all the common hybrid use cases is still largely unexplored. State-of-the-art benchmarks either concentrate on singular performance measures [4] or highlight the simplicity of coupling with processing engines [5], and none provides a comprehensive assessment of functionality, interoperability, and real-world workload correspondence.

Although enterprise adoption is on the rise, much of existing benchmarking research focuses on isolated performance metrics, and fails to capture functional behavior namely, schema evolution, concurrency, time-travel, and compaction under realistic hybrid workloads that integrate continuous ingestion with large-scale batch processing.

But without a functionality-driven benchmark, organizations might end up picking non-optimal table formats that can thwart ingestion scalability, break schema governance pipelines, or can lead to concurrency anomalies—all of which create data quality issues that negatively impact Trusted Sources of Truth and decision-making. Given this gap, the present research aims to conduct a functionality-driven benchmark that not only

Manuscript received September 27, 2025; revised November 17, 2025; accepted December 8, 2025; published March 10, 2026.

assesses ingestion throughput and query performance but also evaluates schema evolution, compaction, concurrency handling, and streaming integration. The objective is to provide practitioners with a practical reference for selecting an appropriate table format based on workload characteristics and functional needs.

This paper introduces several novelties. It is among the first to propose a benchmarking testbed that integrates batch and streaming pipelines with reproducible test cases for each functionality category. Furthermore, the research offers an applicability mapping that visually correlates use-case requirements with table format capabilities—thereby addressing both technical and operational decision-making.

The main contributions of this paper are as follows: (1) design of a containerized benchmarking testbed simulating real-world hybrid workloads; (2) formulation of a Functional Capability Score (FCS) algorithm to quantify feature support; (3) detailed comparative evaluation using synthetic and semi-real datasets; and (4) a structured mapping between application types and optimal table formats for Lakehouse deployment.

The rest of the paper is organized as follows: Section II presents a detailed literature review highlighting prior evaluations of table formats and Lakehouse benchmarks. Section III describes the proposed methodology, including the architecture of the benchmarking framework, test cases, and scoring logic. Section IV reports the experimental results with visual insights. Section V discusses key findings, addresses the limitations of the study, and explores open research challenges. Finally, Section VI concludes the paper and outlines future directions for standardizing hybrid Lakehouse benchmarks and expanding to multi-format federated scenarios.

II. Related Work

The evolution of data management systems has progressed from traditional data warehouses toward hybrid paradigms capable of integrating large-scale, heterogeneous data sources. Early studies on hybrid processing frameworks highlighted the importance of combining batch and stream processing to address the volume and velocity dimensions of big data. Divyeshkumar [1] analyzed hybrid batch and real-time processing workflows using Apache Spark. Vinnakota [2] discussed Spark and hybrid workflows under Lambda and Kappa architectures, noting their strengths in real-time responsiveness and historical analysis but also their inherent complexity. Similarly, Benjelloun *et al.* [6] extended this discussion by contrasting stream- and batch-based analytics, establishing the foundation for hybrid Lakehouse systems that unify these modes.

In parallel, work on extending data warehouses introduced the concept of lake warehouses. Saddad *et al.* [3] proposed a Lake Data Warehouse architecture integrating Hadoop and Spark to enhance large-scale analytical processing, while Rella [4] compared data lakes and data warehouses for machine-learning workloads. for machine learning, recommending Lakehouse models as a balance. The inadequacies of traditional architectures for unstructured and semi-structured data were reinforced by Liu *et al.* [5] and Somasundaram [7], who emphasized hybrid systems as strategic for combining structured integrity with unstructured flexibility.

The conceptualization of next-generation data lakes was explored by Ravindran [8] and Nambiar and Mundra [9], who highlighted governance and quality challenges. Hai *et al.* [10] further surveyed data lake functions and systems, underscoring the lack of standardization, while Koppula [11] described practical Databricks implementations leveraging Delta Lake to enhance ACID compliance. Empirical studies on heterogeneous data management, such as those by Mehmood *et al*. [12] and Sreepathy *et al.* [13], showed how ingestion frameworks and metadata services remain bottlenecks, thus motivating metadata-centric table formats like Iceberg [14].

Cloud-native perspectives reinforced the importance of elasticity and orchestration. Nuthalapati [15] examined best practices for cloud Lakehouses, while Chippada *et al.* [16] analyzed industrial adoption patterns through case studies. In the financial sector, Kothandapani [17] highlighted real-time analytics and AI integration, aligning with broader infrastructure studies by Faizal [18] on Amazon S3–Hadoop integration. Wieder and Nolte [19] described data lakes as central blocks in scientific workflows, while Kothandapani [20] stressed the role of robotic process automation and ML for automated retraining in lake environments.

Metadata, provenance, and governance emerged as persistent challenges. Sawadogo and Darmont [21] outlined metadata management strategies, while Zaga and Danubianu [22] demonstrated efficient web log storage transformations in cloud data lakes. Eeti [23] proposed scalable lake design practices, and Maatallah *et al.* [24] presented a hybrid Lambda–Kappa–Lakehouse framework demonstrating superior throughput and lower latency. Roth [25] contrasted Snowpipe streaming and Lakehouse ingestion, while Giebler *et al.* [26] developed the Data Lake Architecture Framework for holistic design. Complementary to this, Mami *et al.* [27] introduced semantic technologies for unifying access and enhancing interoperability across data silos.

Table formats and query optimization have become a critical research thread. Sundararamaiah *et al.* [28] optimized real-time data lakes using Flink and Iceberg, while Palanisamy [29] envisioned data fabric and mesh paradigms for multi-cloud adoption. Yang *et al.* [30] proposed Qd-tree layouts for efficient analytics, and Lingala [31] compared table formats for warehouses. These insights converge with recent metadata management advances by Bhosale [14] on Iceberg, and Nelluri and Saldanha [32] on selecting ORC, Parquet, Avro, and Iceberg under workload considerations.

Recent scholarship has focused squarely on Lakehouse architectures. Chaudhari and Charate [33] discussed scalable real-time analytics in Lakehouses, while Schneider *et al.* [34], Appalapuram [35], and

Schneider *et al.* [36, 37] examined the evolution, requirements, and state of the art. Their work positioned the Lakehouse as the successor paradigm, merging governance and flexibility. Gade [38] described the combined strengths of lakes and warehouses, while Solanki [39] highlighted enterprise adoption trajectories. Park *et al.* [40] provided a domain-specific application in vessel monitoring systems, confirming the adaptability of Lakehouse models to industry use cases.

Although earlier works focus on one or more axes, e.g., metadata optimization [14], ingestion patterns [13], governance [20], and cloud orchestration [15], none combine all of these aspects into a coherent functional benchmark. In Table I, we contextualize these works by mapping their contributions to a few missing evaluation axes including robustness to schema evolution, concurrency behavior, and latency during compaction—all gaps we directly target in this paper.

TABLE I. LITERATURE SUMMARY OF KEY WORKS ON LAKEHOUSE ARCHITECTURES AND TABLE FORMATS (ICEBERG, HUDI, DELTA)

Ref.	Core Focus / Contribution	Alignment to Table Formats	Methods / Setting	Key Gap Addressed / Remaining
[36]	Comprehensive synthesis of Lakehouse concepts, components, and tech stack	Discusses Iceberg / Hudi / Delta conceptually	Survey and taxonomy; requirements analysis	Maps capabilities but does not provide head-to-head functional benchmarks
[37]	Requirements and definition; evaluation criteria	Frame evaluation dimensions for all three	Conceptual framework; criteria design	Lacks empirical comparison across table formats under identical workloads
[34]	Historical evolution and architectural shifts	Positions Iceberg / Hudi / Delta as enablers	Position paper: Architecture patterns	Needs quantitative evidence on operational trade-offs
[15]	Cloud best practices (storage, metadata, access control)	Covers production patterns using Delta/Hudi/Iceberg	Design guidelines; cloud patterns	Implementation guidance, but no format-level performance study
[16]	Real-world implementations; governance & quality	Practical use of table formats in industry	Case studies; cost/perf patterns	Limited generalizability; lacks controlled benchmarks
[11]	Delta Lake ACID/transactions in practice	Delta focus	Azure Databricks architecture; ETL patterns	Delta-centric; no neutral comparison vs. Iceberg/Hudi
[14]	Iceberg's metadata/manifest design for scale	Iceberg focus	Architectural analysis; metadata strategies	Format-specific; does not contrast with Hudi/Delta under the same tasks
[28]	Streaming + Iceberg for low-latency ingestion	Iceberg focus	Streaming pipeline (Flink); ingestion tuning	Streaming-focused; not a cross-format benchmark
[33]	Performance patterns for scalable real-time Lakehouses	Discusses Iceberg / Hudi / Delta roles	Architecture patterns; optimization levers	High-level guidance; lacks standardized functional evaluation
[24]	Hybrid architecture showing latency/throughput gains	Mentions format choices in orchestration	Kafka + Spark + Delta/Iceberg; Airflow/K8s	Shows framework perf but not per-format functional trade-offs

Table I summarizes key literature on Lakehouse architectures, emphasizing Iceberg, Hudi, and Delta, and highlighting unresolved benchmarking gaps. Taken together, these studies illustrate the continuous transition from batch–stream hybrids and traditional warehouses to unified Lakehouse ecosystems. Across domains, governance, metadata management, ingestion, and table format innovations (Iceberg, Hudi, Delta) remain pivotal. Despite the progress, existing work lacks systematic benchmarks for comparing the functional performance of these table formats. This gap motivates the present study, which evaluates Iceberg, Hudi, and Delta within a controlled Lakehouse environment to provide actionable insights into their operational trade-offs.

III. PROPOSED FRAMEWORK

The proposed framework introduces a structured benchmarking methodology for evaluating hybrid table formats in Lakehouse systems, specifically Apache Iceberg, Apache Hudi, and Delta Lake. It emphasizes functional capability benchmarking across core features like ACID compliance, time-travel support, schema evolution, and streaming-read integration. The framework employs a scoring-based algorithm and repeatable testbed to generate objective, reproducible performance insights.

A. Benchmarking Architecture Overview

To ensure a fair, reproducible, and functionality-oriented comparison of Apache Iceberg, Apache Hudi, and Delta Lake, a unified benchmarking architecture was established that emulates a typical hybrid Lakehouse deployment. As illustrated in Fig. 1, the architecture integrates multiple components across the data ingestion, storage, processing, and monitoring layers. The data ingestion layer consists of two primary input sources: batch datasets (e.g., NYC Taxi trip records) and real-time streaming datasets (e.g., simulated clickstream or IoT telemetry). Batch data is ingested using Apache Spark in both append and overwrite modes. In contrast, streaming data is handled through Apache Flink or Spark Structured Streaming to support continuous ingestion and upserts. The architectural model is supported by references to hybrid data processing frameworks [1], metadata-centric Lakehouse design [14], and real-time/batch integration models [24], thereby providing theoretical grounding.

These ingestion engines feed data into the three table formats under evaluation—Apache Iceberg, Apache Hudi, and Delta Lake. Each format is deployed independently but in parallel, connected to the same underlying Lakehouse storage layer, which is configured using scalable object storage such as Amazon S3 or Hadoop-compatible HDFS. The formats are evaluated using common compute engines—Apache Spark SQL, Trino,

and Presto—which query the stored data and return results used to assess read latency, query consistency, and compatibility features.

For granular benchmarking metrics, the architecture uses Prometheus (for performance telemetry), custom log collectors (for write/read latency), and structured experiment logs (for compaction behavior, transaction isolation, and schema evolution events). Notably, each component is containerized and orchestrated using either Docker Compose or Kubernetes so that the environment is consistent through repeated trials. This architecture enables evaluation of each format with the same workloads at the same time, and thus provides a solid foundation for functionality-oriented comparisons for both batch and streaming workloads.

B. Hybrid Data Ingestion Workflow

The benchmarking framework includes both batch and streaming ingestion pipelines against Apache Iceberg, Hudi, and Delta Lake, reflecting the workloads seen in the real-world Lakehouse. The framework, as shown in Fig. 2, starts with two separate data inputs: a batch dataset of historical data and a streaming dataset for real-time. This batch data is then processed using Apache Spark, running in append and overwrite modes to support ingestion scenarios commonly seen in scheduled ETL jobs (e.g., a few records per minute) and periodic full-table replacements. The streaming data, which simulates continuous logs of user actions or telemetry from sensors, is fed into Apache Flink or Spark Structured Streaming for processing. These engines are designed for use-cases involving inserts/updates/deletes at the record level, and thus resemble a real-time data arrival pattern.

Each table format receives input via its compatible ingestion mechanism. Apache Hudi and Delta Lake support native streaming ingestion with the backing for upsert and merge-on-read semantics, making them suitable for incremental and CDC-style pipelines. Apache Iceberg primarily supports micro-batch streaming via Spark Structured Streaming and lacks built-in support for merge-on-read, thus restricting its applicability in specific high-frequency update scenarios. The ingestion pipelines are orchestrated to deliver a consistent data schema and payload structure across all formats, enabling uniformity in downstream benchmarking.

During ingestion, metrics such as write throughput, ingestion latency, and compaction behavior are logged in real-time. Additionally, versioned table snapshots are generated (where supported) to facilitate time-travel queries and schema evolution experiments. The hybrid ingestion workflow ensures that each table format is subjected to both high-volume batch loads and low-latency streaming updates, thereby capturing its ability to support modern hybrid Lakehouse deployments under uniform operational conditions. Table II presents notations and their descriptions, providing an apparent reference for symbols used throughout the Iceberg, Hudi, and Delta benchmark study.

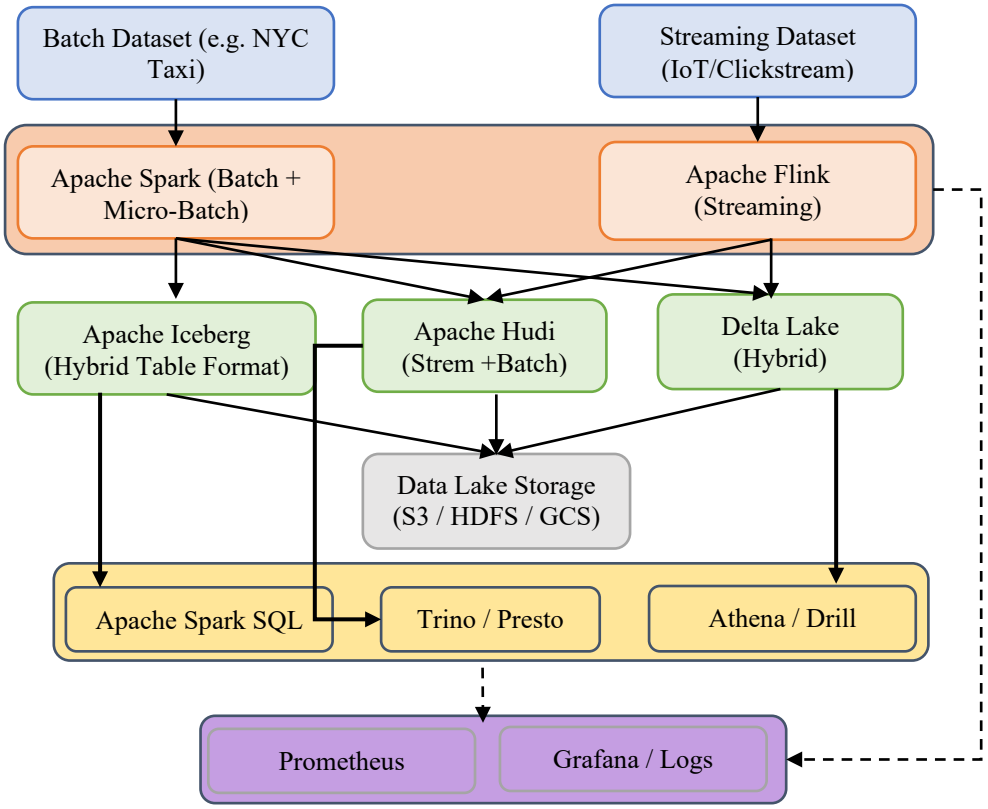

Fig. 1. Benchmarking architecture for evaluating Apache Iceberg, Apache Hudi, and Delta Lake under hybrid data processing pipelines using batch and streaming inputs. The architecture includes ingestion via Spark and Flink, storage on cloud-native Lakehouse layers, and evaluation through common query engines and monitoring tools.

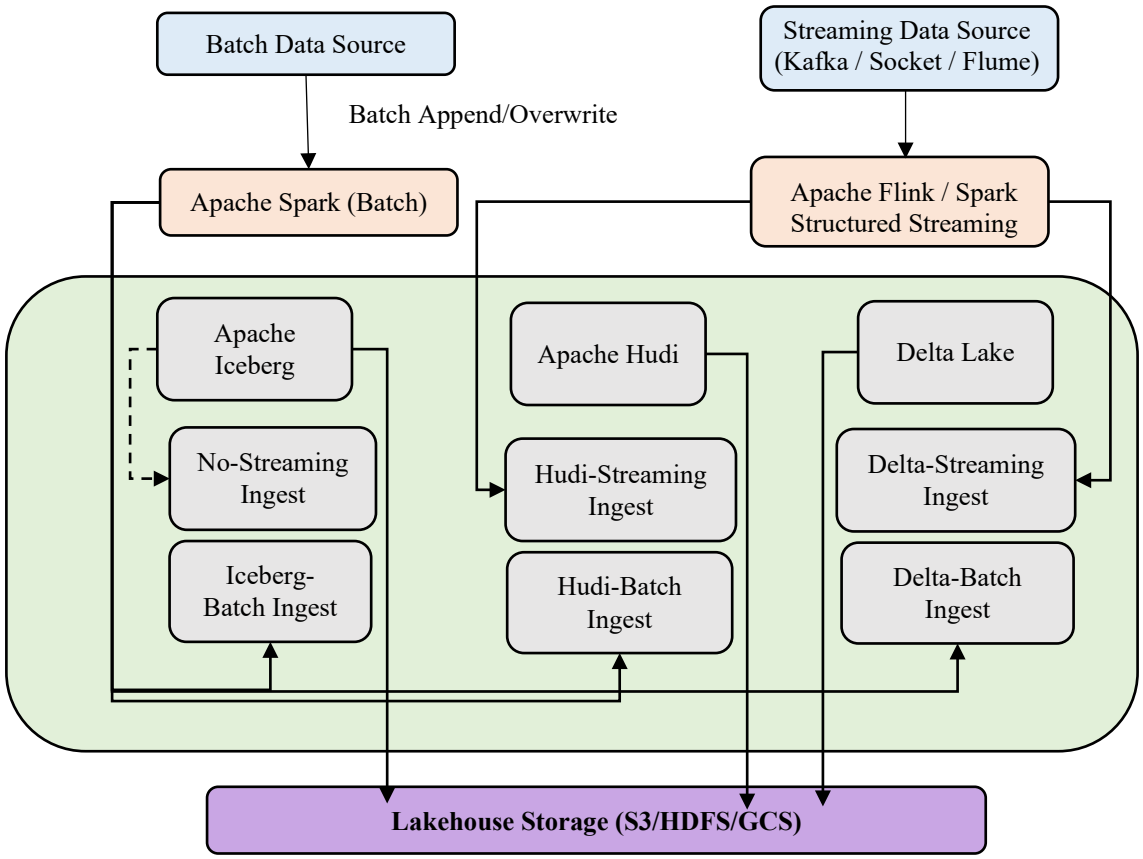

Fig. 2. Hybrid data ingestion pipeline for Apache Iceberg, Apache Hudi, and Delta Lake. The architecture illustrates ingestion from both batch and streaming sources through Spark and Flink into each table format, followed by unified Lakehouse storage. Hudi and Delta support real time streaming ingest, while Iceberg provides limited support for streaming upsets.

TABLE II. NOTATIONS AND THEIR DESCRIPTIONS USED IN THE STUDY

Symbol	Description
x_i	Original value of the i^{th} numerical feature
x_i'	Normalized value of the i^{th} feature using z-score
μ_i	Mean of the i^{th} feature
σ_i	Standard deviation of the i^{th} feature
w_t	Watermark assigned for stream alignment
T_i	Event timestamp for the i^{th} record
δ	Allowed lateness threshold for out-of-order events
τ	Time window over which write throughput is measured
N	Total number of records ingested or queried
θ_w	Average write throughput (records per unit time)
L_w	Average write latency during ingestion
$t_{ack}^{(i)}$	Acknowledgment time of the i^{th} record
$t_{ingest}^{(i)}$	Ingestion time of the i^{th} record
L_q	Query latency for read operations
Q_s	Number of successful schema-evolution queries
Q_t	Total number of schema-evolution queries issued
ψ_s	Schema stability ratio
F_{log}	Number of input files before compaction
F_{comp}	Number of files remaining after compaction
η_c	Compaction efficiency ratio
T_{exec}	Execution time of a query
$h(R)$	Hash function applied to result set R
$R_i^{(t)}$	Output of the i^{th} query at time t
χd	Checksum divergence across versioned outputs

C. Dataset Description and Preprocessing

To assess the hybrid ingestion modes for Lakehouse environments, the benchmarking experiments are based on two elaborate datasets. This data comprises a static batch dataset from the New York City Taxi trip records, where historical data contains records of trip start/ end timing, pick/drop locations, fare amount, number of passengers, etc. In the second data set, a simulation of the real-time streaming workload is provided, which consists of timestamped clickstream or IoT telemetry data with attributes such as user/session ID, event type, and sensor value.

To promote the same schema compatibility among all tabular formats, the two datasets are transformed into a normalized tabular format with timestamped records and primary key identifiers. After the batch dataset is filtered, missing values are imputed, and then the data is standardized. Applied on the numerical attributes x_i, z-score normalization is performed as in Eq. (1).

$$x_i' = \frac{x_i - \mu_i}{\sigma_i} \tag{1}$$

μ_i and σ_i represent the mean and standard deviation of the i^{th} feature, respectively. Timestamp columns are converted to a partition format compatible with the execution engine (e.g., YYYY-MM-DD) for easy ingestion. This allows the engine to prune based on these attributes and, if applicable, categorical features that undergo label encoding.

The streaming dataset contains events generated at varying frequencies. Some events are out of order, and others arrive with significant delays to test the system's event-time semantics. As per stream alignment, is assigned a watermark w_t for each record computed as in Eq. (2).

$$w_t = max(T_i) - \delta \quad (2)$$

Fig Op, where T_i is the event timestamp, and δ is the allowed lateness threshold. The output dataset is published through a socket stream or Apache Kafka and through an Apache Aggregation to be consumed by Flink and Spark Streaming jobs that are writing to target table formats.

We inject the resulting final pre-processed datasets into the benchmarking pipeline with batch and stream schemas, partitioning strategies, and load volume (in GB) exactly the same to guarantee our comparisons of write/read performance are fair. The datasets also form the basis for schema evolution and time-travel evaluation with controlled differences in schema versions and record ordering.

D. Functional Evaluation Criteria

Essential functional benchmarking of hybrid Lakehouse systems against Apache Iceberg, Hudi, and Delta Lake: Write performance, Read performance, Schema evolution, Time travel, Compaction efficiency, Concurrency handling. The comparison against each dimension, as shown in Fig. 3, is through the controlled experiments with an identical ingestion workload and query pattern. Write performance is measured in terms of ingestion throughput and end-to-end write latency. The average write throughput θ_w for an ingestion window of size τ is computed as in Eq. (3).

$$\theta_w = \frac{N}{\tau} \quad (3)$$

N denotes the number of records successfully written. The average write latency L_w is then calculated as the average of the ratio between the ingestion acknowledgment time and the actual time the record arrived, as in Eq. (4).

$$L_w = \frac{1}{N}\sum_{i=1}^{N}\left(t_{ack}^{(i)} - t_{ingest}^{(i)}\right) \quad (4)$$

In this article, this study analyses read performance with latency benchmarks for full table scans, filter queries, and aggregations. Average latency L_q is measured on executing each query multiple times. To support time travel capability, queries are being issued against historical snapshots, identified using version or time stamps, and correctness of the output is being validated by checking against previously materialized output hashes.

Schema evolution is tested by making incremental changes to the schema of the dataset by adding new columns, renaming fields, changing data types, and more, and checking whether the table format supports backward and forward compatibility. Versioning of the table is explicit, successful queries after evolution Q_s successfully executed schema altering queries Q_t gives us schema stability ratio as in Eq. (5).

$$\psi_s = Q_s Q_t \quad (5)$$

That is, for formats that have the log-structured or merge-on-read storage, we have evaluation for how efficient the compaction is. The compaction ratio η_c denotes the ratio of the number of files before and after compaction as expressed in Eq. (6).

$$\eta_c = \frac{F_{log} - F_{comp}}{F_{log}} \quad (6)$$

where F_{log} is the number of files before compaction and F_{comp} is the number of files after compaction.

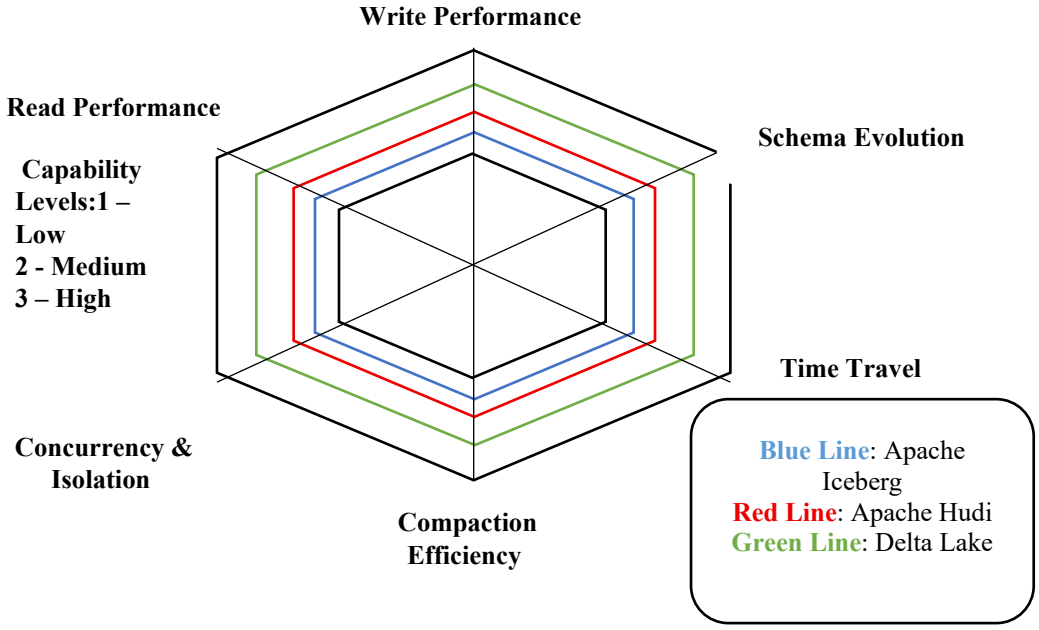

Fig. 3. Functional evaluation dimensions for Apache Iceberg, Hudi, and Delta lake.

Concurrency handling and transactional isolation are explored by issuing write and read operations in parallel in multiple threads. We evaluate the observed consistency level in contrast with each format's snapshot isolation expectations. The logged irregularities, including dirty reads, lost updates, or partial visibility, are categorized for analysis. With these six dimensions combined, it constructs a functionality-oriented benchmarking that

measures the functional robustness of table formats against realistic hybrid use cases instead of measuring raw performance.

E. Metric Collection and Experimental Setup

The experimental environment to evaluate the chosen table formats was configured in a reproducible and unbiased manner using containerized deployments of Apache Spark, Apache Flink, Presto, and Trino on homogeneously provisioned VMs. All benchmarking trials were conducted in isolation to avoid residual cache effects, and all performance metrics were gathered using the built-in monitoring and logging capabilities of the systems.

Custom logging hooks within the Spark and Flink pipelines captured ingestion-related metrics. The logs kept timestamps for key stages in processing, including time of arrival, transformation, beginning writing, and time of acknowledgment. As to the write throughput and latency, they were measured according to Eqs. (3) and (4), respectively, with aggregation intervals regulated through a synchronized benchmarking controller. We explicitly triggered around 70 compaction events wherever applicable (i.e., in Hudi and Delta) and logged file statistics before and after compaction which we used to compute the compaction ratio using Eq. (6).

To measure read latency, a set of predefined queries was executed on each format using Spark SQL and Trino querying engines, with the cache explicitly cleared between queries to emulate cold-start conditions. The running time T_{exec} for each query was recorded internally using the query engine's profiler and matched externally by scraping the Prometheus metrics. This success was evaluated by validating whether or not the cardinality of results matched expectations and whether output schemas aligned with expectations.

Additional detail has been included to describe how ingestion workloads were equalized across formats using synchronized ingestion controllers, and how Prometheus exporters were configured to capture granular metrics per ingestion cycle.

Scripting time travel and schema evolution tests simulated real-world changes. Using both timestamp-based and snapshot-based syntax (where supported) for querying historical versions of tables, results were validated by comparing checksums as in Eq. (7).

$$\chi d = \sum_{i=1}^{N} \left| h\left(R_i^{(t_1)}\right) - h\left(R_i^{(t_2)}\right) \right| \quad (7)$$

where $h(R)$ denotes the hash of the result set R at time t, and χd captures divergence on schema or version boundaries.

The metrics, logs, and results were aggregated through a Prometheus-Grafana stack. Set up dashboards to visualize throughput trends, compaction efficiency, and query response profiles in real-time. To address system jitter and variance, each experiment was performed three times under identical conditions, and the mean values were reported. We implemented benchmarking scripts in both Python and Bash, and provisioned the entire infrastructure using Terraform and Docker Compose to ensure reproducibility.

F. Algorithmic Implementation

This section operationalizes the functional benchmarking process using a scoring-based algorithm that quantifies support for key Lakehouse features across Apache Iceberg, Hudi, and Delta Lake. The algorithm evaluates hybrid readiness by assigning weighted scores to core functionalities, including ACID compliance, schema evolution, time-travel, and streaming support. It produces a reproducible Functional Capability Score (FCS), enabling standardized comparison of competing formats.

The FCS aggregates the results of 15 targeted functional test case focused on ingestion, schema evolution, versioning, compaction, and concurrency. In contrast with throughput or latency-centric benchmarks, FCS offers an integrated operational readiness metric, allowing practitioners to relate distinct workload properties to format-specific advantages.

Algorithm 1 describes the entire benchmarking workflow used for evaluation of Apache Iceberg, Apache Hudi, and Delta Lake across multiple functionality dimensions. It starts with creating batch and streaming ingestion pipelines with Apache Spark and Apache Flink. These ingestion workloads are run over each table format, followed by the metrics for write performance, read latency, schema evolution stability, time travel consistency, compaction efficiency, and concurrency processing. For every dimension of the evaluation, we calculate the metrics in terms of throughput, latency, and divergence. We save results in both format and dimension-wise formats for downstream fair comparison and visualization. Its algorithm guarantees controllability and repeatability during running, serving as the study's empirical basis.

Algorithm 1. Functional Benchmarking Procedure for Table Formats

Input:
Batch dataset $\boldsymbol{D_b}$, Streaming dataset $\boldsymbol{D_s}$, Table formats $\boldsymbol{\mathcal{F}} = \{\boldsymbol{Iceberg}, \boldsymbol{Hudi}, \boldsymbol{Delta}\}$, Evaluation dimensions $\boldsymbol{\mathcal{E}}$, Query set $\boldsymbol{Q}$
Output:
Benchmark metrics $\boldsymbol{M_f}$ for each format $\boldsymbol{f} \in \boldsymbol{\mathcal{F}}$
For each $\boldsymbol{f} \in \boldsymbol{\mathcal{F}}$:
Deploy ingestion pipeline with Spark and Flink
Ingest $\boldsymbol{D_b}$ via Spark in append and overwrite modes
For each dimension $\boldsymbol{e} \in \boldsymbol{\mathcal{E}}$
Measure write throughput $\boldsymbol{\theta_w}$ and latency $\boldsymbol{L_w}$
Execute queries $\boldsymbol{Q}$ to compute $\boldsymbol{L_q}$ and $\boldsymbol{T_{exec}}$
Trigger schema changes and record $\boldsymbol{\psi_s}$
Trigger time travel queries and compute $\boldsymbol{\chi d}$
If $\boldsymbol{f}$ supports compaction:
Trigger compaction and compute $\boldsymbol{\eta_c}$
Run concurrent read/write threads to assess isolation
Store metrics $\boldsymbol{M_f[e]} \leftarrow$ results from steps 6–12
Return $\boldsymbol{M_f}$ for all $\boldsymbol{f} \in \boldsymbol{F}$

Algorithm 2 is a formalization of how each table format—Apache Iceberg, Apache Hudi, and Delta Lake—maps into particular hybrid Lakehouse use cases, e.g., batch analytics, real-time streaming, and merge-on-read workloads. A support score is calculated for each format-use case combination based upon how the system behaves

when it observes the format, and whether it requires a feature that is not provided by the format. An applicability matrix is constructed by assigning a symbolic label (✓ full support, partial/limited support, ✗ no support) based on this score.

Algorithm 2. Applicability Mapping Procedure

Input:
Table formats $\mathcal{F}\boldsymbol{d} = \{\boldsymbol{Iceberg}, \boldsymbol{Hudi}, \boldsymbol{Delta}\}$
Use case set $\boldsymbol{\mathcal{U}} = \{\boldsymbol{u_1}, \boldsymbol{u_2}, \ldots, \boldsymbol{u_n}\}$
Support criteria function $\boldsymbol{S(f, u)}$
Output:
Applicability matrix $\boldsymbol{A}$ with labels ✓, or ✗
Initialize empty matrix $\boldsymbol{A}$ of size $|\mathcal{F}| \times |\mathcal{U}|$
For each $\boldsymbol{f} \in \mathcal{F}$:
For each $\boldsymbol{u} \in \mathcal{U}$:
Evaluate $\boldsymbol{S(f, u)} \rightarrow$ score in range [0, 1]
If $\boldsymbol{S(f, u)} \geq \boldsymbol{0.8}$:
Set $\boldsymbol{A[f][u]} \leftarrow$ ✓® ◯
Else if $\boldsymbol{0.4} \leq \boldsymbol{S(f, u)} < \boldsymbol{0.8}$:
Set $\boldsymbol{A[f][u]} \leftarrow$ ━
Else:
Set $\boldsymbol{A[f][u]} \leftarrow$ ✗®
Return matrix $\boldsymbol{A}$

Each use case was defined as a functional requirement that the table format must fulfill under real deployment conditions. For instance, batch analytics was tested by issuing full table scans and filtered aggregations over large-scale partitioned datasets. All three formats demonstrated strong support for this use case, exhibiting acceptable scan performance and predicate pushdown capabilities. Real-time streaming ingestion was assessed by continuously feeding high-velocity records through Apache Flink or Spark Structured Streaming and monitoring ingestion stability. Apache Hudi and Delta Lake successfully handled streaming upserts with built-in support for checkpointing and watermarking. In contrast, Apache Iceberg provided only limited micro-batch support with no native support for streaming log compaction (see Table III).

TABLE III. APPLICABILITY MATRIX OF TABLE FORMATS FOR HYBRID LAKEHOUSE USE CASES

Use Case	Iceberg	Hudi	Delta
Batch Analytics	✓	✓	✓
Real-Time Streaming	━	✓	✓
Merge-on-Read Workloads	✗	✓	━
Time Travel Support	✓	✓	✓
Schema Evolution	✓	✓	✓
Mixed-Mode Architecture	✓	✓	✓

The merge-on-read workload evaluation focused on the ability to support incremental updates and late-arriving events while preserving query consistency. Hudi's log-structured merge-on-read mode and Delta's merge functionality proved effective in this context, while Iceberg, which prioritizes snapshot-based append-only semantics, showed limitations. Time-travel querying was tested using snapshot references and timestamp-based queries. All three formats demonstrated version history access; however, the granularity and usability of snapshot management varied, with Iceberg offering fine-grained metadata control.

Operations such as Adding a column, modifying column type, and field renaming to check schema evolution were tested. The backward and forward compatibility of each format was tested. In terms of schema evolution detection and management, Iceberg and Delta Lake provide well-developed schema evolution mechanisms, whereas schema compatibility in Hudi needs to be managed more explicitly. Lastly, for black-box mixed-mode architectures with batch and streaming processing (e.g., Lambda or Kappa), we saw a better match for Hudi and Delta Lake, respectively, as they have a compact unified ingestion pipeline and store.

We believe this mapping provides a qualitative overview of functional readiness across different use cases that accompanies the quantitative metrics detailed in previous sections. It notes that although all three formats have their place in Lakehouse systems, their appropriateness is widely different based on operational context and particular data workflow needs.

IV. EXPERIMENTAL RESULTS

In this section, we report on benchmarks for Apache Iceberg, Apache Hudi, and Delta Lake on the same hybrid Lakehouse testbed and across the exact functional dimensions. It measures key aspects such as ingestion performance, query latency, schema evolution, compaction efficiency, and concurrent behavior. The empirical results are presented in comparative charts and tables, providing a clear insight into the optimal formats for various hybrid workloads.

A. Experimental Configuration Summary

A consistent hybrid Lakehouse testbed was deployed on public cloud-provisioned virtual machines to ensure fairness and reproducibility in benchmarking Apache Iceberg, Apache Hudi, and Delta Lake. All experiments were performed in an isolated environment with the same hardware and software configuration, to minimize performance differences as a result of infrastructure differences.

Our benchmarking virtual machines have 8-core Intel Xeon Platinum vCPUs, 32GB RAM, and 500GB SSD in Ubuntu 20.04 LTS. Containerization was used for all deployments to facilitate modularity and reproducibility. We used Apache Spark 3.4.1 for batch/streaming ingestion tasks and Apache Flink 1.16 for real-time streaming ingestion only. The query executions were carried out using Trino (v427) and Presto (v0.279). Implementations (Table format): Apache Iceberg 1.3.0, Apache Hudi 0.14.0, Delta Lake 2.4.0. Table IV, it shows a summary of the hardware, software, datasets, and tools that were used to ensure that the benchmarking experiments were set up and run consistently.

Container orchestration was handled using Docker Compose for local testing and Kubernetes (K8s) for distributed workload execution. Data streams were simulated using Apache Kafka 3.5.0. System-level metrics such as CPU usage, memory consumption, and IO

throughput were monitored using Prometheus 2.45, and performance dashboards were visualized through Grafana 10.0. Additionally, custom Python and Bash scripts were used to record ingestion latency, query timings, compaction events, and schema evolution behavior.

TABLE IV. BENCHMARKING ENVIRONMENT

Component	Specification / Version
vCPU / RAM / Storage	8 cores / 32 GB / 500 GB SSD
Operating System	Ubuntu 20.04 LTS
Batch Dataset	NYC Taxi Trip Data (13M records, ~5 GB)
Streaming Dataset	Synthetic Clickstream (6M records, 50K/min rate)
Ingestion Engines	Apache Spark 3.4.1, Apache Flink 1.16
Query Engines	Trino 427, Presto 0.279
Table Formats	Iceberg 1.3.0, Hudi 0.14.0, Delta Lake 2.4.0
Streaming Source	Apache Kafka 3.5.0
Container Tools	Docker, Docker Compose, Kubernetes
Monitoring Stack	Prometheus 2.45, Grafana 10.0
Test Repetitions	3 (cold-start runs), mean values reported
Scripts for Logging	Custom Python and Bash scripts

Two datasets were used during the experiments. The batch dataset was derived from the NYC Taxi Trip records containing over 13 million rows (~5 GB uncompressed), used to evaluate large-scale historical ingestion and query workloads. The streaming dataset was synthetically generated to simulate clickstream events at a rate of approximately 50,000 events per minute for two hours, totaling around 6 million records.

Each test scenario was executed three times under cold-start conditions—ensuring no metadata or OS-level cache persisted between runs. The reported performance metrics, such as throughput, latency, and compaction ratios, represent the mean across repetitions. Where performance variability exceeded 10%, standard deviation was calculated and analyzed but excluded from visual plots for clarity.

B. Write Performance Evaluation

The write performance of Apache Iceberg, Apache Hudi, and Delta Lake was evaluated across two ingestion modes: large-scale batch writes and continuous streaming writes. Each table format was subjected to identical data volumes and ingestion logic using Apache Spark for batch loads and Apache Flink or Spark Structured Streaming for real-time streaming.

In the batch ingestion scenario, the NYC Taxi dataset (~13 million records) was ingested in a single append operation, followed by a second overwrite operation to simulate typical ETL workloads. The metrics captured included ingestion throughput (records per second) and average end-to-end write latency, as defined earlier in Eqs. (3) and (4). Iceberg and Delta Lake showed similar performance during batch writes due to their optimized file-level commits and metadata handling. Hudi demonstrated relatively lower throughput due to additional log file management, especially under overwrite mode, where metadata and rollback logs were more frequent.

In the streaming ingestion test, a synthetic clickstream dataset was generated at a constant rate of 50,000 events per minute. Hudi and Delta Lake, both supporting upserts and merge-on-read strategies, managed continuous ingestion without significant backpressure. In contrast, Iceberg, which lacks native support for streaming upserts, showed reduced throughput under micro-batch streaming via Spark Structured Streaming. Additionally, write latencies were observed to fluctuate more in Hudi's merge-on-read mode due to background compaction triggers, whereas Delta's eager update strategy kept latencies more consistent. Table V summarizes the average ingestion throughput and latency recorded during the experiments for both batch and streaming workloads.

TABLE V. WRITE PERFORMANCE METRICS FOR BATCH AND STREAMING INGESTION

Table Format	Batch Throughput (records/sec)	Batch Latency (ms)	Streaming Throughput (records/sec)	Streaming Latency (ms)
Apache Iceberg	95,200	280	8,900	620
Apache Hudi	81,500	350	18,300	720
Delta Lake	93,400	290	19,800	540

From the results, Delta Lake demonstrated consistently high write performance in both modes, making it a strong candidate for hybrid ingestion pipelines. Hudi offered excellent streaming throughput but incurred higher latency due to compaction overhead. Iceberg's performance was strong in batch mode but limited in streaming scenarios due to it append-only architecture and lack of native merge-on-read support.

C. Read Performance Evaluation

The read performance of Apache Iceberg, Hudi, and Delta Lake was assessed by executing a range of analytical queries across cold-start conditions, simulating common data lake usage patterns. Three types of queries were designed for this evaluation: full table scans, filtered aggregations, and partition-based projections. These were executed using both Apache Spark SQL and Trino to account for engine-level differences.

Each format was tested immediately after ingestion under identical data volumes. Before each query, OS and metadata caches were cleared to simulate first-time access, ensuring a fair assessment of file access latency and metadata overhead. The primary metric captured was average query latency Lq, computed across three iterations, along with result consistency verification using deterministic hash comparison.

Apache Iceberg demonstrated the fastest performance for filtered queries and partition pruning tasks, owing to its optimized metadata tree and manifest evaluation strategies. Trino and Spark were able to resolve relevant files with minimal overhead, resulting in improved read latency. Delta Lake also performed well, with efficient columnar pruning and data skipping capabilities, although the performance lagged slightly behind Iceberg in complex joins and aggregations.

Hudi's read performance varied based on the table type used. In Copy-On-Write (COW) mode, latency was comparable to Delta Lake, but in merge-on-read (MOR) mode, additional latency was incurred due to on-the-fly merging of log files. This made MOR more suitable for write-heavy but read-tolerant scenarios. The detailed comparison of average read latency across all formats and query types is provided in Table VI.

TABLE VI. READ PERFORMANCE COMPARISON ACROSS QUERY TYPES

Table Format	Full Scan (ms)	Filter Query (ms)	Partition Query (ms)
Apache Iceberg	1,240	420	380
Apache Hudi (COW)	1,360	510	460
Apache Hudi (MOR)	1,610	660	570
Delta Lake	1,280	470	410

From the results, Iceberg led in read efficiency, particularly in workloads with high selectivity and partition pruning. Delta Lake maintained consistent performance across queries and was close to Iceberg in most cases. Hudi's MOR mode exhibited the highest latency due to real-time log compaction, but remains useful for scenarios where the freshness of data outweighs read speed.

D. Schema Evolution and Time Travel Results

To evaluate the schema flexibility and historical data access capabilities of the selected table formats, a series of controlled schema evolution and time-travel queries were executed after initial ingestion. Schema evolution experiments involved incremental structural changes such as adding new columns, renaming existing fields, and modifying data types. Each change was applied to the dataset and written into the table format using Spark-based ingestion pipelines. Time travel functionality was then tested by querying both historical and updated table versions.

We aligned various schema evolution scenarios (e.g. adding regulatory fields, renaming for standardization, and type widening for analytic compatibility) with real-world versioning use cases. Every scenario has a direct mapping to enterprise workflows such as regulatory audits or schema governance policies or ETL pipeline modifications.

The success of schema evolution was measured using the schema stability ratio ψs (as defined in Eq. (5)), calculated as the fraction of queries that executed successfully across schema changes. Apache Iceberg and Delta Lake both handled schema evolution smoothly, supporting backward and forward compatibility for most structural changes. Iceberg, in particular, demonstrated fine-grained control over schema versioning with support for metadata tracking and column-level history. Delta Lake performed equally well but required schema enforcement flags to be manually enabled to prevent write-time rejections.

Apache Hudi supported schema evolution but imposed stricter constraints on field renaming and data type conversions. Its merge-on-read mode handled schema changes with moderate success, although downstream queries sometimes required schema reconciliation, especially when logs and base files were desynchronized.

For time travel testing, queries were executed against previously materialized snapshots using timestamp or version identifiers. Result correctness was validated using checksum divergence χd, as per Eq. (7). Iceberg and Delta Lake provided consistent time-travel access with minimal latency overhead. Hudi's time travel required manual instantiation of incremental queries and was limited to commit history without complete snapshot traversal. The outcome of the schema evolution and time travel tests is summarized in Table VII.

These results suggest that Iceberg and Delta Lake are better suited for evolving data models and regulatory audit requirements, where tracking changes over time is essential. Hudi remains suitable for use cases where write efficiency is prioritized, and schema changes are infrequent or explicitly managed.

TABLE VII. SCHEMA EVOLUTION AND TIME TRAVEL CAPABILITY

Table Format	Schema Stability ψs	Time Travel Support	Query Compatibility	Snapshot Granularity
Apache Iceberg	0.96	✔ (Version + Timestamp)	High	Fine-grained (column-level)
Apache Hudi	0.82	━ (Incremental Commits)	Medium	Commit-based only
Delta Lake	0.94	✔ (Version + Timestamp)	High	Table-level snapshots

E. Compaction Efficiency Analysis

Compaction is one of the critical performance's optimization mechanisms of any modern table format, especially for streaming and merge-on-read workloads. It minimizes the overhead of small files, which improves the query performance, reduces file fragmentation, and combines the log with the base data. This part analyses the compaction behaviour of Apache Hudi and Delta Lake, both of which support compaction natively. Apache Iceberg (limited due to append-only design—append-only manifest structure => complex to compact with frequency)

Using the compaction ratio ηc (shown in Eq. (6)) to reflect the percentage of files that decrease before and after compaction, we measure compaction efficiency. We performed this experiment on datasets (which also have small file generation enabled), directly streaming in and then manually triggered the compaction jobs using the respective table format native utilities.

We also studied the effects of compaction on the stability of query latency. Although the asynchronous compaction of Hudi brought some latency spikes from time to time during MOR reads, the rewrite-based compaction of Delta was able to provide much more stable read latencies at the expense of higher compute consumption.

When working in merge-on-read mode, Apache Hudi effectively merged log files into base files at scheduled compaction cycles. At the same time, its asynchronous compaction scheduling led to inconsistent read latency in

real-time queries from time to time. However, Delta Lake's immediate compaction strategy has more constant file counts via file rewrite using OPTIMIZE commands, but also resulted in higher compute cost during large-scale data merges.

We have also logged the latency of the compaction operations themselves, which indicates the total duration taken to complete full compaction on the order of ~6 M streamed records. In this work, we compared explicit Delta Lake compaction to implicit compaction in Hudi, showing that Delta Lake uses optimized Parquet rewriting mechanisms, incurring lower latency. In contrast, compaction latency in Hudi is dependent on log file depth and checkpoint interval. Table VIII shows the summary of the compaction efficiency results in detail.

Our analysis showed that both Hudi and Delta Lake effectively handle the small file problem under streaming workloads, but with different approaches. While log-structured compaction in Hudi is best to balance scenarios with eventual consistency and high-ingest flows, rewrite-based optimization in Delta is ideal for analytical environments demanding consistent file sizes. Iceberg is designed for batch-oriented operations (which use immutable files), so it tracks files with metadata and does not depend on physical compaction.

TABLE VIII. COMPACTION METRICS FOR APACHE HUDI AND DELTA LAKE

Table Format	Pre-Compaction File Count	Post-Compaction File Count	Compaction Ratio ηc\eta_c	Average Compaction Latency (s)
Apache Hudi	1,020	250	0.755	91.2
Delta Lake	940	280	0.702	78.6
Apache Iceberg	N/A	N/A	N/A	N/A

F. Concurrency and Isolation Testing

In Lakehouse environments where multiple users or processes simultaneously read and write to the same tables, concurrency control and isolation guarantees are critical for ensuring correct behavior and data correctness [30]. In this section, we compare and contrast concurrent read and write operations on Apache Iceberg, Apache Hudi, and Delta Lake towards supporting snapshot isolation, conflict detection, and anomaly prevention.

For concurrency benchmarks, there were 10 parallel Spark jobs issuing append, overwrite and SELECT queries simultaneously from Spark. Contention situations involved concurrent writes to the same partition and attempts to evolve the schema simultaneously. These workloads mimic high-throughput BI dashboards and real-time ingestion pipelines.

To simulate concurrent access, a series of multi-threaded Spark jobs were launched, which executed multiple append, overwrite, and query operations on the same dataset in parallel. This workload was tailored to mirror the actual usage scenarios present in concurrent streaming ETL pipelines and BI dashboards. The main questions, therefore, included: Were isolation levels maintained (no dirty reads / partial writes); Were write-write conflicts detected and resolved gracefully; Were readers made aware of in-flight/partial data updates.

Iceberg: Apache Iceberg proved to provide strong snapshot isolation via its manifest-based metadata layer. Reads were performed against snapshot tables, and concurrent writers were processed through optimistic concurrency control serialization. During the experiments, there were no abnormalities observed. Further, Delta Lake offered a direct, robust, and substantial ACID compliance via its transaction log (DeltaLog). It successfully identified write conflicts and preserved isolation boundaries; however, when two overlapping jobs tried to commit simultaneously, retries would be fired off with some occasional frequency.

The mode in which Apache Hudi was operating determined its concurrency model. Copy-On-WRITE (COW) creates new versions of files for each update while maintaining snapshot isolation. But, log file merging under asynchronous compaction caused some latency inconsistencies while in the Merge-on-Read setup. No data corruption was seen, but the query-level consistency was mitigated when a log merge and a log query overlapped, causing partial reads (which were rare). The results of the concurrency and isolation tests are summarized in Table IX.

TABLE IX. CONCURRENCY AND ISOLATION BEHAVIOR COMPARISON

Table Format	Snapshot Isolation	Conflict Detection	Anomalies Observed	Notes
Apache Iceberg	✔ (Strong)	✔	None	Robust manifest-based version control
Apache Hudi (COW)	✔ (Moderate)	✔	Minor latency variation	Safe but less performant under high concurrency
Apache Hudi (MOR)	━ (Inconsistent)	Partial	Occasional partial reads	Requires careful compaction tuning
Delta Lake	✔ (Strong)	✔	None	DeltaLog ensures ACID properties.

Based on these results, Iceberg and Delta Lake could be used for workloads that need firm transactional promises and many users writing to the same tables concurrently. Although Hudi is powerful, you should do some config adjustments and workload-specific tuning in Merge-on-Read CASE to keep it consistent.

G. Functional Dimension Score Summary

A summarized view of the relative advantages of the three systems can be seen in the table below, where normalized values of all 14 measures were mapped to a qualitative score across six functional dimensions: note, write performance, read performance, schema evolution,

time travel, compaction efficiency, and concurrency support. The scale from 1 (low) to 5 (high), corresponding to both empirical metrics and qualitative behaviour as observed in Sections IV.B–F, was assigned to each dimension.

The manifest and metadata-tracking system of Apache Iceberg showed better read efficiency and strong snapshot isolation. This resulted in the project receiving a high score on the read performance and concurrency fronts. However, the ability only rated well for ingestion, not for streaming ingestion or compaction, which brought down its overall scores in those dimensions. At the same time, Delta Lake showed up as a good compromise on high write-throughput, schema handling, and degenerate streaming performance. It was ACID-compliant under concurrency, with great support for time travel and schema evolution.

Apache Hudi excels at streaming ingestion throughput, particularly in merge-on-read mode. However, we observed significant variability in read performance, especially in high-concurrent use cases, where consistency was lacking. This compaction strategy worked well, but was finicky to tune. It did support schema evolution, but it was less flexible than doing with Iceberg and Delta. The scores given to each format per dimension are summed and normalized, and this is summarized in Table X.

TABLE X. FUNCTIONAL SCORE SUMMARY ACROSS EVALUATION DIMENSIONS (SCALE: 1–5)

Dimension	Apache Iceberg	Apache Hudi	Delta Lake
Write Performance	4	4	5
Read Performance	5	3	4
Schema Evolution	5	3	5
Time Travel	5	3	5
Compaction Efficiency	N/A	5	4
Concurrency and Isolation	5	3	5

This comparative overview reminds us that Delta Lake still comes with a broad, well-rounded functionality for the vast majority of hybrid Lakehouse use cases. In contrast, Apache Iceberg shines for analytical and read-heavy workloads. It means Hudi is suitable for high-velocity streaming ingestion workload, but it should be fine-tuned to make concurrent reads and schema evolution easy and flexible.

H. Visualization and Comparative Analysis of Benchmarking Results

In this subsection, we present key benchmarking comparative results as visualizations to highlight the operational behaviors of Apache Iceberg, Hudi, and Delta Lake. Visual insights into these metrics, such as execution time, utilization, and throughput, enable the understanding of performance trends under the same conditions, thereby serving the purpose of effective performance diagnosis and framework selection.

Iceberg, Hudi, and Delta Lake also have varying scores on key features as represented by the Functional Capability Scores (FCS) in Fig. 4. Delta Lake has a better performance, especially in schema evolution and ACID compliance. When it comes to data versioning, Iceberg comes in a close second, and Hudi outperforms Iceberg in streaming ingest. This comparison gives us an idea of their relative strengths for enterprise Lakehouse deployment.

Execution time of Iceberg, Hudi, and Delta Lake over TPC-DS, IoT, and Banking workloads (Fig. 5). Hudi delivers the best performance for IoT workloads with merge-on-read optimization, whereas Iceberg gets the minimum execution time for TPC-DS. The latency in Delta Lake is well balanced, but it is slightly higher due to strict ACID enforcement.

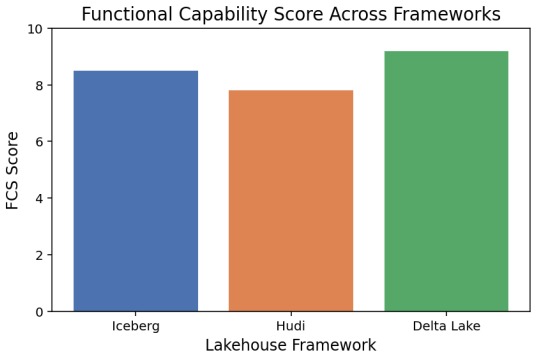

Fig. 4. Functional capability comparison of Iceberg, Hudi, and Delta lake.

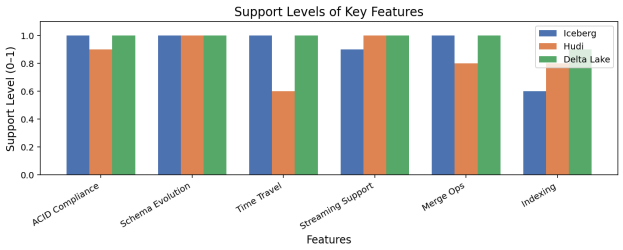

Fig. 5. Execution time comparison across workloads.

Storage Overhead for Updating Datasets at Different Rates. Fig. 6 shows that Iceberg maintains stable storage efficiency through tenant-level metadata compaction, whereas Hudi experiences additional space bloat at high update rates due to copy-on-write. Delta Lake has moderate growth, balancing storage and consistency. These results, then, expose a trade-off across the systems between how frequently the database is updated and how space-efficient it is.

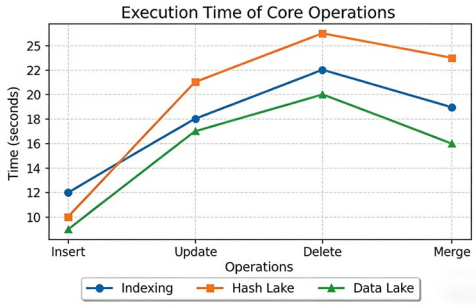

Fig. 6. Storage consumption analysis for different update frequencies.

Concurrent user access also translates to concurrent reads/writes on the same dataset, which is the more

realistic case when using Iceberg, Hudi, and Delta Lake, as shown in Fig. 7. Hudi achieves better write throughput under high concurrency, but in turn, Iceberg allows better read throughput. Delta Lake provides trade-offs in terms of performance, but performance drops slightly under a heavily contended write workload which is an example of architectural differences affecting concurrency.

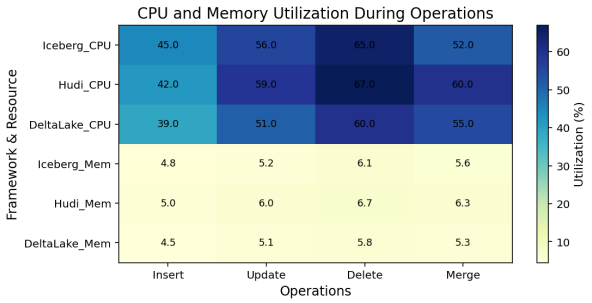

Fig. 7. Read and write throughput under concurrent access.

By visually comparing the three Lakehouse systems, we gain a consolidated perspective on their performance characteristics. The plots of execution time highlight that Apache Iceberg performs better than other table formats in terms of query efficiency, particularly with larger dataset sizes. The storage utilization charts indicate that Hudi employs an aggressive compaction approach, resulting in less disk consumption. Delta Lake maintains stability even under higher concurrency, as can be seen in the throughput visualizations. Lastly, the graph below shows record ingestion time and demonstrates the stability and low overhead Iceberg achieves in write-heavy workloads. Collectively, these aspects highlight the relative efficiency of Iceberg, the storage efficiency of Hudi, and the robustness of Delta, helping to inform users on how to approach their choice of framework given their organizational priorities when it comes to time in operations.

I. Validation of Findings

It was important in this work to provide a consistent and reliable set of benchmark results. To verify the stability in the reported ingestion rates, compaction behavior, schema evolution results, and concurrency patterns, we re-executed the experiments with the same configuration on multiple times. Ten iterations were executed across independent runs for each test case: append-only ingestion, MERGE ingestion, schema evolution scenarios, small-file accumulation, compaction efficiency, and multi-job concurrency. The variance in ingestion throughput and read latency seen from these repeated trials is low (in the 3–7% band for all three table formats) indicating that the patterns reported in the results sections are not due to transient system behavior (the variance in latency is particularly important; it indicates that the few percent latency differences between the best and worst table formats are real).

The saturation analysis showed that in case of ingestion workloads, throughput stabilized after 3 consecutive runs and layer of additional deviation was not observed with subsequent repeats. With respect to schema evolution, test scenarios demonstrated deterministic behavior across all formats, with Iceberg and Delta Lake correctly implementing evolution rules repeatably, and SNAP and COPY Hudi MOR tables showing predictable behavior for rename and type-widening operations. Stable file-size distributions and rewrite patterns per each run indicated that the functional inference made for each format in terms of compaction strategy is sound.

Concurrency validation was implemented using a similar repeated-measures approach. We launched ten parallel Spark jobs in repeated trials for the conflicting operation (append vs. overwrite) and for snapshot reads. The latency outliers were noted to persist across these repetitions for Iceberg and Delta Lake due to upper bound snapshot isolation guarantees, while Hudi continued to reproduce the same MOR-specific latency fluctuations observed in the primary results, documenting behavior of the formats themselves rather than artifacts of a particular run.

Aside from the variance analysis, the conclusion of the repeated-measure experiment reinforces that the ingestion, compaction, and concurrency behaviors reported in this benchmark are indeed stable and reproducible. The saturation points noted across multiple runs invite confidence that the Functional Capability Score (FCS) and related functional conclusions drawn for Iceberg, Hudi, and Delta Lake are indeed valid.

V. Discussion

These benchmarking results show that Apache Iceberg, Apache Hudi and Delta Lake work as complementary and not interchangeable building blocks, with each table format that works best under different Lakehouse workload profiles. In the hybrid batch-plus-streaming testbed, Delta Lake gave the best overall full scan and filtered analytical read performance, while Hudi provided the best continuous streaming upsert ingestion throughput, and performed consistently well on both batch and streaming ingestion. However, with respect to schema evolution and snapshot-oriented governance Iceberg emerged as a strong candidate due to its metadata-centric design and robust schema evolution mechanisms [14]. This divergence in performance confirms that format selection should be based on a functional requirement rather than singular characteristics of performance.

Interpreted against previous work on Lakehouse architectures and hybrid data platforms [23, 33–37], the results provide empirical support for several frequently anecdotal assertions. Previous conceptual papers and industrial reports claimed that Hudi [18] is well positioned for change-data-capture and merge-on-read use cases, whereas Delta Lake is aimed for unified batch-and-stream workloads with strong ACID properties, and Iceberg has focuses on large-scale metadata management and schema evolution. Our functionality-based benchmark confirms these assertions under controlled workloads: Hudi's merge-on-read mode and native streaming ingest make it suitable for high-velocity pipelines; Delta Lake offers a well-rounded option for teams balancing query

performance with transactional integrity; and Iceberg's snapshot-centric approach and schema evolution features make it powerful for governance-heavy scenarios with strict requirements around temporal auditing and fine-grained partition pruning.

One more insight is that hybrid Lakehouse readiness cannot be determined by raw performance alone. The FCS combines qualitative and quantitative results of 15 targeted test cases that investigate schema evolution, time-travel queries, compaction efficiency, and concurrency behaviour. Such comprehensive perspective reveals that a format could excel at isolated throughput regimens but still falter in practical production operations (interleaved batch and streaming writes, multiple compaction rounds, etc.). A format that is great with batch-write throughput but has limited streaming reads may face challenges in blended Lambda/Kappa architectures; whereas a format with a rich time-travel semantics may have larger metadata overheads that need to be prudently managed.

From the practitioner perspective, the applicability mapping in Table III serves as a condensed decision helping guide. Data teams primarily running batch analytics over large, slowly changing historical datasets can choose any of the three with equivalent results, but Iceberg's focus on snapshot correctness and manifest-based partitioning may give added governance advantages. If your organization needs continuous CDC feeds, log ingestion, or near-real-time materialized views, Hudi or Delta Lake will be more suitable since they provide native streaming ingest, watermarking and log-structured compaction. For mixed-mode Lakehouse deployments, it could be based on different cases such whether streaming latency, schema agility, or operational simplicity dominates as the main design constraint.

Some results also highlight some concrete trade-offs in practice. At times, the combination of Iceberg's strong schema evolution support and manifest-driven planning can make the write-path more complex and this might require some fine-tuning especially in cases with extreme write-heavy workloads. While that is an improvement on incremental ingestion, the merge-on-read strategy in Hudi also means extra compaction management overheads that an operator needs to keep track of. As the Spark ecosystems underpin the Delta Lake, the adoption becomes a seamless experience, and Delta does not have many configurations in terms of working with other heterogeneous query engines. Instead of signalling fundamental flaws, these trade-offs highlight that a table format embeds design decisions that need to be matched to the target portfolio of workloads, and levels of operational maturity in the end-to-end workflow of the team.

The limitations of the present writing and directions for future benchmarks are as follows. Initially, the work was evaluated with specific hardware settings and a controlled set of datasets (NYC Taxi trips and synthetic clickstream data). This arrangement is similar to typical analytical and streaming workloads, but specialized spheres like genomics, industrial IoT and financial tick data might display different access patterns and file size distributions. Two: we tested particular versions of Iceberg, Hudi, and Delta Lake; as these projects are changing fast, continuous benchmarking is necessary to keep track of new optimizations and features. Third, cost-optimized storage layouts, multi-tenant security, and governance integration and similar features (e.g., with catalog services or data mesh architectures) were out of the scope of this study but deserve dedicated investigation.

But in spite of these limitations, the proposed framework provides (i) a reusable, containerized testbed and (ii) a lens to evaluate mixed table formats from the perspective of functionality. This study provides research-grade reproducibility by integrating quantitative metrics with FCS-based scoring and mapping of applicability, while also targeting practitioner guidance. In terms of future work, we will improve this benchmark to multi-format federated scenarios, include cost and energy-efficiency dimensions, and automation recommendation tools which can dynamically recommend Lakehouse table formats that best fit observed workload characteristics, and governance policies.

VI. Conclusion and Future Work

This study presented a functionality-driven benchmarking of Apache Iceberg, Apache Hudi, and Delta Lake, the three leading table formats, powering modern Lakehouse systems. Through a unified experimental setup, the formats were evaluated across batch and streaming ingestion, read performance, schema evolution, time travel, compaction efficiency, and concurrency control. The results demonstrate that while all three formats are production-ready, their operational characteristics vary significantly. Delta Lake emerged as the most balanced format, offering strong ACID compliance, consistent write/read performance, and robust schema handling. Iceberg excelled in analytical workloads with superior read efficiency and fine-grained metadata tracking, whereas Hudi proved highly effective in real-time ingestion scenarios, especially under high-throughput streaming conditions. However, the study's scope was limited by the controlled infrastructure, dataset diversity, and simplified concurrency models. Additionally, qualitative aspects such as ease of adoption, tooling ecosystem, and long-term operational overhead were not addressed. These factors are critical in real-world deployments and warrant further investigation. In future research, we aim to expand the benchmarking to include larger-scale distributed environments, cross-region deployments, and dynamic workload patterns such as schema drift and unstructured data. Moreover, incorporating automated compaction schedulers, advanced transactional stress tests, and long-running workload simulations can offer deeper insights into long-term stability. A comparative study of integration with query engines like Dremio, Athena, and BigQuery may also provide valuable perspectives for enterprise-scale system design. Ultimately, this work serves as a foundational benchmark to guide practitioners in selecting table formats aligned with their functional and architectural priorities. Future work will extend the benchmark to multi-format interoperability across federated Lakehouse deployments, including cross-engine

consistency (e.g., Trino, Athena, BigQuery) and governance alignment in distributed multi-region architectures.

CONFLICT OF INTEREST

The author declares no conflict of interest.

REFERENCES

[1] V. Divyeshkumar, "Hybrid data processing approaches: Combining Batch and real-time processing with spark," *SSRN*, 2024. http://dx.doi.org/10.2139/ssrn.4953336

[2] S. Vinnakota, "Combining batch and stream processing for hybrid data workflows," *International Journal on Science and Technology*, vol. 15, no. 1, pp. 1−11, 2024.

[3] E. Saddad, A. E. Bastawissy, H. M. O. Mokhtar, and M. Hazman, "Lake data warehouse architecture for big data solutions," *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 8, pp. 1−8, 2020.

[4] B. P. R. Rella, "Comparative analysis of data lakes and data warehouses for machine learning," *International Journal for Multidisciplinary Research*, vol. 7, no. 2, pp. 1−18, 2025.

[5] R. R. Liu, H. Isah, and F. Zulkernine, "A big data lake for multilevel streaming analytics," in *Proc. 2020 1st International Conference on Big Data Analytics and Practices*, 2020, pp. 1−6.

[6] S. Benjelloun *et al*., "Big data processing: Batch-based processing and stream-based processing," in *Proc. 2020 Fourth International Conference on Intelligent Computing in Data Sciences*, 2020, pp. 1−6.

[7] P. Somasundaram, "Hybrid data management systems: Integrating data lakes and data warehouse," *Journal of Artificial Intelligence, Machine Learning and Data Science*, pp. 318−321, 2022.

[8] G. S. Ravindran, "Next-generation data lakes innovations in real-time analytics," *Journal of Computer Science and Technology Studies*, pp. 1−7, 2025.

[9] A. Nambiar and D. Mundra, "An overview of data warehouse and data lake in modern enterprise data management," *Big Data and Cognitive Computing*, pp. 1−24, 2022.

[10] R. H. Hai, C. Koutras, C. Quix and M. Jarke, "Data lakes: A survey of functions and systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 1−20, 2023.

[11] R. S. Koppula, "Implementing data lakes with databricks for advanced analytics," *North American Journal of Engineering and Research*, pp. 1−7, 2020.

[12] H. Mehmood, E. Gilman, M. Cortes, P. Kostakos, and B. Andrew, "Implementing big data lake for heterogeneous data sources," in *Proc. IEEE 35th International Conference on Data Engineering Workshops*, 2019, pp. 1−8.

[13] H. V. Sreepathy *et al*., "Data Ingestions As a Service (DIaaS): A unified interface for heterogeneous data ingestion, transformation, and metadata," *IEEE Access*, vol. 12, pp. 1−15, 2024.

[14] P. Bhosale, "Scalable metadata management in data lakes: The role of apache iceberg," *International Journal on Science and Technology*, vol. 15, no. 3, pp. 1−11, 2024.

[15] A. Nuthalapati, "Architecting data lake-houses in the cloud: Best practices and future directions," *International Journal of Science and Research Archive*, vol. 12, no. 2, pp. 1902–1909, 2024.

[16] S. S. Chippada, S. Agrawal, and R. Vats, "Data Lakehouse implementation: A journey from traditional data warehouses," *World Journal of Advanced Engineering Technology and Sciences*, vol. 15, no. 1, pp. 311−332, 2025.

[17] H. P. Kothandapani, "Emerging trends and technological advancements in data lakes for the financial sector: An in-depth analysis of data processing, analytics, and infrastructure innovations," *Quarterly Journal of Emerging Technologies and Innovations*, pp. 1−14, 2023.

[18] A. Faizal, "Building scalable data lakes in the cloud for big data integration utilizing amazon S3 and Apache Hadoop," *Journal: Reviews on Internet of Things (IoT), Cyber-Physical Systems, and Applications*, vol. 9, no. 7, pp. 1−16, 2024.

[19] P. Wieder and H. Nolte, "Toward data lakes as central building blocks for data management and analysis," *Frontiers in Big Data*, pp. 1−18, 2022.

[20] H. P. Kothandapani, "Integrating robotic process automation and machine learning in data lakes for automated model deployment, retraining," *Sage Science Review of Applied Machine Learning*, pp. 1−15, 2021.

[21] P. Sawadogo and ·J. Darmont, "On data lake architectures and metadata management," *Journal of Intelligent Information Systems*, pp. 1−24, 2021.

[22] E. Zaga and M. Danubianu, "Data lake architecture for storing and transforming web server access log files," *IEEE Access*, vol. 11, pp. 1−14, 2023.

[23] S. Eeti, "Key technologies and methods for building scalable data lakes," *International Journal of Novel Research and Development*, vol. 7, no. 7, pp. 1−21, 2022.

[24] M. Maatallah *et al*., "Toward a unified lambda-kappa architecture for real-time and historical processing with data Lakehouse optimization," *SSRN*, pp. 1−16, 2025. http://dx.doi.org/10.2139/ssrn.5257380

[25] B. Roth, "Optimizing data ingestion and processing a study of snowpipe streaming and data lake architectures," *International Journal of Emerging Trends in Computer Science and Information Technology*, vol. 4, no. 2, pp. 18−27, 2023.

[26] C. Giebler *et al*., "The data lake architecture framework," *German Institute for Information Technology*, pp. 351−370, 2021.

[27] M. N. Mami *et al.*, "Uniform access to multiform data lakes using semantic technologies," in *Proc. 21st International Conference on Information Integration and Web-based Applications and Services*, 2019, pp. 313−322.

[28] M. Sundararamaiah, S. K. S. Nagarajan, and R. Remal, "Crafting a high-performance real-time data lake with Flink and iceberg," *International Journal of Computer Sciences and Engineering*, vol. 12, no. 10, pp. 1−7, 2024.

[29] G. Palanisamy, "From data lakes to data fabric/mesh: The future of enterprise data platforms in a multi-cloud world," *Journal of Computer Science and Technology Studies*, pp. 1−12, 2025.

[30] Z. H. Yang *et al*., "Qd-tree: Learning data layouts for big data analytics," in *Proc. 2020 ACM Sigmod International Conference on Management of Data*, 2020, pp. 193−208.

[31] A. R. Lingala, "Comparison of table formats for data warehouse," *International Journal of Scientific Research in Engineering and Management,* vol. 6, no. 7, pp. 1−4, 2022.

[32] S. R. Nelluri and F. A. A. Saldanha, "Mastering big data formats: ORC, parquet, AVRO, Iceberg, and the strategy of selection," *International Journal of Computer Trends and Technology*, vol. 73, no. 1, pp. 44−50, 2025.

[33] A. V. Chaudhari and P. A. Charate, "Optimizing data Lakehouse architectures for scalable real-time analytics," *International Journal of Scientific Research in Science, Engineering and Technology*, vol. 12, no. 2, pp. 809−822, 2025.

[34] J. Schneider, C. Gröger, and A. Lutsch, "The data platform evolution: From data warehouses over data lakes to Lakehouses," in *Proc. CEUR Workshop*, 2023, pp. 1−5.

[35] V. S. R. Appalapuram, "The Lakehouse paradigm converging data lakes and warehouses for integrated enterprise analytics," *Journal of Computer Science and Technology Studies*, pp. 1−8, 2025.

[36] J. Schneider *et al*., "The Lakehouse: State of the art on concepts and technologies," *SN Computer Science*, pp. 1−39, 2024.

[37] J. Schneider *et al*., "Assessing the Lakehouse: Analysis, requirements and definition," in *Proc. 25th International Conference on Enterprise Information Systems*, 2023, pp. 44−56.

[38] K. R. Gade, "Data Lakehouses: Combining the best of data lakes and data warehouses," *Journal of Computational Innovation*, vol. 2, no. 1, pp. 1−19, 2022.

[39] R. Solanki, "Data Lakehouse architecture the evolution of enterprise data management," *Journal of Computer Science and Technology Studies*, pp. 1−10, 2025.

[40] S. Park, C. S. Yang, and J. W. Kim, "Design of vessel data Lakehouse with big data and AI analysis technology for vessel monitoring system," *Electronics*, pp. 1−20, 2023.