em
lyon
business
school

Jia Xin Tang Zhi

Promotion 2024

Master of Science in Data Science & Artifitial Intelligence

Design and Optimization of a Cloud-Based Transactional Data Lake for Evolving Data Models

Date of Defense: June 2025.
Supervisor: Prof. Dr. Benoit LOEILLET
Department: Operations, Data & Artificial Intelligence

Declaration form

I declare that I have personally prepared this master thesis and that it has not in whole or in part been submitted for any other degree or qualification. Nor has it appeared in whole or in part in any textbook, journal or any other document previously published or produced for any purpose. The work described here is my own, carried out personally, unless otherwise stated. All sources of information, including quotations, are acknowledged by means of reference, both in the final reference section, and at the point where they occur in the text.

Jia Xin Tang Zhi

Date: 28/02/2025

Jia Xin Tang Zhi

“Data is the new oil.”
— *Clive Humby*

Contents

List of Figures

List of Tables

Abstract

The rapid expansion of data-driven applications has heightened the demand for scalable and transactional data management solutions. Traditional S3-based data lakes, while effective for large-scale storage, often struggle with maintaining consistency and supporting real-time processing. This study explores the design and implementation of a cloud-based transactional data lake, integrating the strengths of data lakes and data warehouses to optimize both storage and query efficiency. By leveraging open table formats such as Apache Iceberg, this research evaluates their ability to provide ACID compliance, schema evolution, and incremental processing. The study follows an iterative deployment approach, continuously assessing the impact of technological choices on scalability, reliability, and performance while benchmarking storage performance under Copy-on-Write (COW) and Merge-on-Read (MOR) paradigms to evaluate overall system efficiency. By implementing a robust data pipeline from ingestion to final storage, the work demonstrates how modern architectures enhance real-time adaptability while ensuring transactional integrity. Findings indicate that S3-based architectures face challenges in managing transactions efficiently, whereas modern open table formats mitigate these limitations through its customized storage strategies, enabling cost-effective, high-performance cloud-based data lakes. The insights from this study contribute to the advancement of scalable data solutions, providing practical guidelines for organizations seeking to optimize their data infrastructures for evolving analytical and operational demands.

Chapter 1
Introduction

The exponential growth of data generation in the digital era has revolutionized the way businesses, governments, and research institutions manage and process information. From social media interactions to IoT devices collecting real-time sensor data, the unprecedented velocity and volume of information being produced every second necessitates new methodologies for storage, retrieval, and analysis. The rise of Big Data has unlocked immense opportunities to derive meaningful insights that drive innovation and informed decision-making. However, managing such vast amounts of data requires balancing multiple factors, including storage scalability, transactional consistency, and analytical efficiency.

In response to these challenges, modern data architectures have evolved from traditional databases and data warehouses to more flexible and scalable solutions such as data lakes and, more recently, data lakehouses. While S3-based data lakes offer cost-effective storage for raw, unstructured, and semi-structured data, they often lack robust governance and transactional capabilities, leading to inefficiencies the risk of data swamps (Hai et al., 2023), where large amounts of unstructured data are stored but are difficult to process. The data lakehouse paradigm, combining the scalability of data lakes with the transactionality integrity of data warehouses, has emerged a transformative solution. This study proposes the development a lakehouse leveraging modern lakehosue frameworks, particularly Apache Iceberg, which plays a pivotal role in enabling this architectural shift, offering enhanced data management functionalities to support both analytical and operational workloads.

1.1 Context and Background

Industries such as finance, healthcare, manufacturing, and industrial IoT increasingly dependent on real-time data processing, necessitating scalable and transactional architectures capable of efficiently managing continuous data streams. The widespread adoption of data lakehouses is largely driven by their ability to support schema evolution, version control, and concurrent query execution, making them ideal for environments that require both batch and streaming analytics.

One of the foremost challenges in modern Big Data ecosystems extends beyond mere data volume to encompass its inherent complexity. The diversity of data formats, sources, and structures underscores the inadequacy of simple storage solutions, as traditional methods often prove costly and difficult to scale under the constant influx of new data. Instead, systems must be designed for efficient querying, schema flexibility, and real-time adaptability.

As explained by Blinowski et al. (2022), two primary approaches are commonly employed to address scalability challenges. **Vertical scaling**, the traditional method, expands system capacity by upgrading existing hardware. Although effective, this approach becomes increasingly cost-prohibitive and eventually reaches physical limitations.

In contrast, **horizontal scaling** distributes resources across multiple interconnected nodes, offering greater flexibility for system expansion. This method has gained popularity and now serves as the foundation of many modern data solutions. However, despite its advantages, horizontal scaling introduces new challenges, particularly in transaction management. Unlike vertically scaled databases, which natively support ACID transactions, horizontally distributed architectures often lack built-in transactional consistency, complicating the maintenance of data integrity across nodes (Blinowski et al., 2022). Addressing these constraints is essential for designing scalable, high-performance, and reliable data systems.

1.2 Research Objectives

The primary objective of this study is to design a scalable and transactional cloud-based data lakehouse architecture that addresses the challenges of Big Data environments. To achieve this, Apache Iceberg will be implemented as a central component, bridging the gap between traditional data lakes and data warehouses.

Despite the growing demand for scalable and transactional data architectures, industry adoption remains constrained by implementation complexity and performance trade-offs, underscoring the significance of this research. The field of data management has undergone a profound transformation, particularly in domains where real-time analytics and historical data processing must coexist seamlessly. This convergence necessitates storage solutions that can handle high-velocity data ingestion, transactional updates, and complex querying at scale.

The effectiveness of transactional data lakehouses is fundamentally governed by their underlying storage strategies, which dictate how data is ingested, updated, and queried. Optimizing these mechanisms is critical to ensuring performance, consistency, and reliability in large-scale distributed data ecosystems.

At the core of this discussion lie two competing storage paradigms, each presenting distinct trade-offs in terms of scalability, latency, and transactional integrity: **Copy-on-Write (CoW)** and **Merge-on-Read (MoR)**. The selection of either paradigm significantly impacts system performance, directly affecting storage efficiency, throughput processing, and the overall adaptability of data lakehouse architectures in large-scale deployments.

This study conducts an empirical evaluation of these two strategies within *Apache Iceberg*, benchmarking their trade-offs under varying operational conditions, including high-frequency ingestion, schema evolution, and mixed transactional-analytical workloads. To further strengthen Iceberg's capabilities for large-scale transactional data lakes, this research will examine core functionalities such as schema evolution, partitioning, metadata pruning, snapshot tracking, time travel, and data rollback mechanisms. Particular emphasis will be placed on analyzing the performance of *Cow* and *MoR* in high-ingestion environments with concurrent queries, measuring query response time, ingestion throughput, compaction overhead, and storage footprint.

Beyond performance analysis, a key objective is to design and optimize an ETL pipeline for Iceberg, leveraging an AWS-based architecture tailored for IoT data ingestion and querying. Given the prevalence of semi-structured data streams in industrial applications, we will focus on schema evolution, metadata governance, and efficient partitioning strategies. To achieve this, both real-world and synthetic IoT workloads collected from a monitoring simulator will be used to evaluate ingestion efficiency, transformation processes, and query performance. This research will:

- Implement AWS S3 as a storage layer and AWS Glue for metadata management.
- Analyze key deployment aspects, including schema enforcement, snapshot tracking, and metadata pruning, to improve system stability and efficiency.
- Bridge theoretical discussions on Iceberg's capabilities with practical deployment strategies, providing actionable guidelines for cloud architects and data engineers.

Since data lifecycle management is critical for cost efficiency and maintaining query performance, this paper will also examine compaction strategies, indexing techniques, snapshot retention policies, and rollback mechanisms. By systematically assessing these aspects, this research aims to enhance Iceberg's scalability and reliability, strengthening its role in the future of high-performance transactional data lakes.

Chapter 2

Literature Review

2.1 Evolution of Data Management Systems

2.1.1 Evolution from Industry 4.0 to 5.0

The industrial sector has undergone a transformation through the adoption of digital technologies, reshaping how organizations monitor, optimize, and automate operations. Industry 4.0 marked a shift toward data-driven manufacturing, integrating IoT sensors, AI-powered analytics, and real-time monitoring systems to support predictive maintenance and operational decision-making. As this phase matured, Industry 5.0 emerged with a stronger emphasis on human-AI collaboration, sustainability, and adaptive production systems. Managing the increasing volume, velocity, and variety of data from these advancements requires selecting appropriate infrastructures, whether deployed on-premise or in cloud environments. The following sections outline the core technological shifts in Industry 4.0 and 5.0, their reliance on IoT data, and the architectural considerations involved in managing transactional data lakehouses.

2.1.1.1 Industry 4.0: The Digital Transformation of Manufacturing

Industry 4.0 introduced digitized, autonomous, and interconnected manufacturing systems, fundamentally altering industrial processes. One of the key elements in this transition is the deployment of cyber-physical systems (CPS), which integrate computational intelligence with physical manufacturing processes (Kalsoom et al., 2021). CPS bridges the gap between operational technology (OT) and information technology (IT), allowing factories to adapt dynamically to real-time conditions. The adoption of digital twins further strengthens this transition by simulating different manufacturing scenarios, allowing organizations to mitigate risks, refine workflows, and identify bottlenecks in production (Berardi et al., 2023a).

A core technological driver of Industry 4.0 is the proliferation of IoT sensor networks, which continuously capture and transmit data for real-time analysis. These sensors play a crucial role in predictive maintenance, reducing unplanned downtime and allowing industries to identify anomalies before they lead to system failures. By combining sensor-generated data with automated control systems, manufacturing processes become more responsive to demand fluctuations, disruptions, and system inefficiencies (Pech et al., 2021). Robotics and AI-driven automation further enhance operational agility, with machines adapting to changing conditions based on pattern recognition and anomaly detection algorithms (Chaudhary et al., 2024).

Beyond automation, artificial intelligence, machine learning, and big data analytics have revolutionized how manufacturers process vast amounts of data to refine production workflows. Mohamed (2023) highlights how AI-powered predictive analytics assists in anticipating equipment failures, optimizing resource allocation, and improving supply chain coordination. Through data-driven insights, organizations can refine production schedules, reduce material waste, and minimize energy consumption. AI-based process control ensures self-regulating industrial systems that continuously adjust parameters to align with performance targets and sustainability goals.

The success of Industry 4.0 relies on interoperability, as manufacturing environments require seamless data exchange across multiple layers of industrial infrastructure. The integration of Industrial IoT (IIoT), cloud platforms, and edge analytics facilitates this transition, allowing manufacturers to centralize and analyze data from diverse sources. Cloud platforms support long-term storage and large-scale analytics, while edge computing processes time-sensitive data closer to the source, reducing network congestion and latency. By adopting these technologies, Industry 4.0 has laid the foundation for a more adaptive, data-driven industrial ecosystem (Zhou et al., 2024).

2.1.1.2 Industry 5.0: Human-AI Collaboration

While Industry 4.0 focused on automation, data integration, and system intelligence, Industry 5.0 shifts attention toward collaboration between humans and AI to create more adaptable and sustainable production systems. This evolution reintroduces human expertise into decision-making processes, leveraging

AI to augment human capabilities rather than replace them. The approach seeks to balance technological advancements with considerations such as energy conservation, waste reduction, and personalized manufacturing (Cheok et al., 2024).

A significant advancement in Industry 5.0 is the introduction of Explainable AI (XAI), which addresses the lack of transparency in conventional AI models. Many industrial AI systems operate as *black boxes*, making decisions that lack interpretability. XAI allows manufacturers to understand and validate AI-generated recommendations, fostering trust and reliability in automated processes. This is particularly relevant for safety-critical applications, compliance-driven industries, and high-precision manufacturing, where understanding why an AI model made a particular decision is essential (Khan et al., 2024). Another defining feature of Industry 5.0 is the expansion of human-centric automation, where AI-powered systems complement rather than replace human labor. The adoption of collaborative robots (cobots), adaptive manufacturing systems, and AI-assisted user interfaces enhances human workers' ability to manage complex production environments. This approach supports greater customization in manufacturing, addressing demand for personalized products without compromising production throughput (Chaudhary et al., 2024).

The transition from fully autonomous factories to AI-assisted, human-driven production enables industries to refine decision-making while maintaining flexibility, sustainability, and accountability. Industry 5.0 represents a shift toward more ethical, transparent, and socially responsible industrial ecosystems, ensuring that technological advancements serve both economic and environmental objectives.

2.1.1.3 IoT and Sensorization in Industrial Environments

Building upon the advancements in human-AI collaboration characteristic of Industry 5.0, the integration of Internet of Things (IoT) technologies has further transformed industrial operations. Sensors that measure parameters such as temperature, vibration, pressure, and flow provide valuable insights for predictive maintenance, energy optimization, and overall operational enhancements. However, challenges persist in ensuring reliable connectivity and managing low-latency data processing, especially in settings that demand immediate decision-making.

To address latency and connectivity issues, industries are adopting hybrid architectures that combine edge computing and cloud platforms. Edge computing involves processing time-sensitive data or near the data source, which allows for faster response times and reduces network congestion. This approach is particularly beneficial for applications requiring rapid decision-making and minimal latency. Meanwhile, cloud computing offers centralized storage and advanced analytics, supporting long-term trend analysis and large-scale AI model training. Chaudhary et al. (2024) explains that by balancing edge and cloud computing, organizations can achieve both immediate responsiveness and scalable analytical capabilities.

This harmonious integration of IoT with edge and cloud computing not only enhances operational efficiency but also exemplifies the collaborative ethos of Industry 5.0, where human expertise is augmented by advanced technologies to create adaptable and sustainable production systems.

2.1.2 Transition from Data Warehouses and Data Lakes to Lakehouses

In the evolving landscape of enterprise data management, organizations have traditionally relied on data warehouses and data lakes to handle their information needs. Data warehouses offer structured storage with strong transactional integrity and optimized query performance, making them suitable for business intelligence applications. They often come with high costs and limited flexibility in handling unstructured data. On the other hand, data lakes provide scalable storage for a variety of data types, including unstructured data, which is ideal for machine learning and exploratory analytics. Yet, the lack of ACID compliance and schema enforcement in data lakes can lead to data inconsistency and governance challenges.

The rise of lakehouse architectures represents a major breakthrough in merging the benefits of data warehouses and data lakes while addressing their respective limitations. Lakehouses integrate features such as schema evolution, ACID transactions, and real-time processing within a flexible, cloud-native framework. This integration supports time-travel analytics, compliance auditing, and reproducible data science workflows, enabling organizations to maintain data integrity while accommodating diverse analytics workloads. Consequently, lakehouses are particularly well suited for AI-driven applications, business intelligence, and high-volume industrial data processing (Lorica et al., 2025).

As industries increasingly depend on real-time streaming data, federated analytics, and cross-functional data sharing, the lakehouse model offers a centralized yet adaptable foundation that unifies transactional and analytical processing. This evolution facilitates scalable integration of structured and semi-structured data, positioning lakehouses as a cornerstone of next-generation enterprise data architectures.

2.1.2.1 Data Storage Abstractions: Data Warehouse, Data Lake, and Lakehouse

Selecting an appropriate data storage architecture becomes crucial for supporting analytics, governance, and scalability. Three primary models, Data

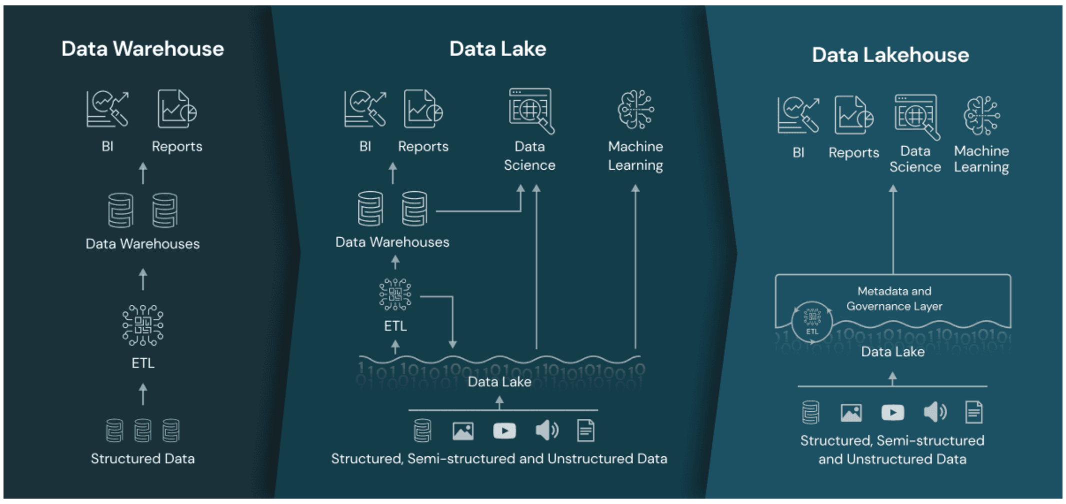


Figure 1: Data Storage Abstractions (Lorica et al., 2020).

Warehouses, Data Lakes, and Data Lakehouses, each offer different trade-offs in query performance, schema enforcement, and transactional capabilities. Each model, as shown in Figure 1, each offer distinct advantages and trade-offs in terms of query performance, schema enforcement, and transactional capabilities (Lorica et al., 2025). Understanding these differences is essential for organizations to determine the most suitable approach for storing, processing, and analyzing their data.

2.1.2.2 Data Warehouse

Data warehouses are optimized for structured, high-performance analytical workloads, following a schema-on-write approach. In this model, data undergoes preprocessing, transformation, and optimization before storage, enabling fast, complex queries refined for Online Analytical Processing (OLAP). This design facilitates rapid aggregations, filtering, and indexing, making it ideal for business intelligence (BI), reporting, and structured data analysis (Serra, 2024).

Platforms like Amazon Redshift, Google BigQuery, and Snowflake exemplify traditional data warehouses, offering robust query optimization and ACID transaction support to ensure strong data consistency. However, their requirement for structured data formats limits adaptability to semi-structured and unstructured datasets (Sá et al., 2024). The rigid schema enforcement can pose challenges when dealing with evolving data structures, necessitating careful planning for schema modifications. Likewise, the computational and storage costs associated with data warehouses are generally higher, particularly for large-scale industrial datasets.

While data warehouses excel at structured analytical processing, they lack flexibility in integrating diverse, unstructured data sources. As industrial applications increasingly utilize semi-structured formats (e.g., JSON, Avro, Parquet) and streaming data, organizations are exploring more adaptable storage architectures to meet these evolving needs (Reis & Housley, 2022a).

2.1.2.3 Data Lake

According to IBM (2024), data lakes emerged as a response to the limitations of traditional data warehouses, offering low-cost, schema-on-read storage capable of handling structured, semi-structured, and unstructured data. Unlike warehouses, data lakes store raw data without predefined schemas, allowing greater flexibility for machine learning, and exploratory data analysis. Common data lake implementations include Apache Hadoop, Amazon S3, and Azure Data Lake, which leverage distributed file systems for handling large-scale datasets. By deferring schema enforcement to query time, data lakes accommodate evolving data structures, making them highly adaptable for IoT sensor data, log files, and real-time event streams.

Despite these advantages, data lakes face several challenges related to transactional integrity. The absence of ACID compliance and schema evolution can lead to data swamps, where poorly managed data accumulates without clear governance. Complementarily, query speeds in data lakes are often slower than data warehouses, particularly when working with large datasets without indexing or partitioning strategies. Without proper metadata management and data cataloging, retrieving relevant data can become increasingly complex (IBM, 2024).

2.1.2.4 Data Lakehouse

Data lakehouses unify the flexibility of data lakes with the structured querying capabilities of data warehouses, supporting both transactional and analytical processing. Built on open table formats such as Apache Iceberg, Apache Hudi, and Delta Lake, they incorporate ACID transactions, schema evolution, and time-travel capabilities to address challenges related to data consistency, governance, and version control. This allows organizations to store raw, semi-structured, and structured data while optimizing query performance.

Kujawski (2023) argues that adopting columnar storage formats, e.g., Parquet, Avro, and metadata tracking, lakehouses improve data retrieval speeds while remaining adaptable to evolving industrial datasets. They also support real-time analytics, machine learning pipelines, and federated queries. Platforms such as Databricks Lakehouse and AWS Lake Formation integrate structured and unstructured data processing, enabling industries to analyze complex datasets without migrating between separate storage systems.

The same author mentions another core strength of lakehouses is their ability to eliminate data silos by centralizing disparate datasets within a unified platform. Many organizations struggle with fragmented data environments, where CRM systems, lead tracking databases, and behavioral analytics rely on isolated data warehouses. This fragmentation leads to inefficiencies, duplication, and missed insights. A lakehouse architecture consolidates these sources, supporting real-time updates, schema evolution, and scalable processing within a single framework. This integration enhances data accessibility, improves operational decision-making across industries, and establishes a cost-effective foundation for modern data-driven workflows.

Feature	**Data Warehouse**	**Data Lake**	**Data Lakehouse**
Data	Structured	Structured, Semi-structured, Unstructured	Structured, Semi-structured, Unstructured
Processing	OLAP (Online Analytical Processing)	Batch processing, Real-time processing	Batch processing, Real-time processing
Storage	Optimized for structured data storage	Optimized for raw data storage	Optimized for both raw and structured data storage
Schema	Schema-on-write	Schema-on-read	Combines schema-on-read and schema-on-write approaches
Integration	ETL (Extract, Transform, Load)	ELT (Extract, Load, Transform)	Primarily ELT, may also support ETL
Use Cases	Business intelligence, reporting, analytics	Data exploration, machine learning, analytics, data science	Hybrid use cases of data warehouse and data lake

Table 1: Comparison of storage architectures.

2.2 Data Lakehouse Architecture

Transactional data lakes, commonly structured as lakehouses, incorporate open table formats that introduce ACID transactions, schema evolution, and indexing mechanisms to refine data organization, consistency, and query execution. Unlike traditional data lakes, which store raw data without transactional integrity, lakehouses centralize structured and semi-structured datasets, improving data versioning, schema enforcement, and long-term storage efficiency. These architectures address challenges associated with fragmented data environments by integrating multiple sources into a unified transactional layer.

2.2.1 Components of a Lakehouse

A lakehouse architecture unifies disparate technology into a cohesive architecture. It typically consists of five layers that work in unison to create a scalable data ecosystem capable of handling large datasets while supporting real-time and batch processing workflows:

1. The **Ingestion Layer** gathers and transfers data from various sources, including IoT devices, real-time streaming systems, batch pipelines, and external databases. Designed to handle high-velocity data streams, it reserves the original structure and format while processing large volumes of continuously generated information.

2. The **Storage Layer** manages large-scale, persistent data storage, and typically relies on distributed object stores. This layer is responsible for handling various data formats, such as Parquet, ORC, or Avro, while applying compression, indexing, and partitioning techniques to optimize retrieval and reduce storage overhead.

3. The **Metadata Layer** functions as the data catalog for every object in the storage layer, tracking table schemas, versioning, partitioning, and snapshot histories. It supports schema evolution, indexing, and partition pruning, all essential for optimizing query performance.

4. The **Application programming interface (API) Layer** facilitates secure and programmatic access to data. This layer handles low-latency query execution, event-driven data access, and RESTful integration, streamlining interactions between the lakehouse and various consumer platforms.

5. The **Consumption Layer** converts raw and processed data into actionable insights through visualization, reporting, and analytics platforms. This layer integrates with business intelligence (BI), machine learning (ML) and other data science tools to extract meaningful patterns, evaluate system performance, and drive strategic initiatives.

This modular design from IBM (2024) allows organizations to manage structured, semi-structured, and unstructured data while maintaining query performance across diverse workloads. As Industry 4.0 and 5.0 drive an unprecedented surge in data complexity, storage architectures must strike a balance between performance, governance, and adaptability. Data warehouses offer structured storage but impose rigid schema constraints, limiting their applicability in evolving industrial environments. Data lakes, while accommodating raw data in various formats, present challenges related to consistency and query speed. Lakehouses provide a middle ground by incorporating transactional capabilities with schema evolution, in both real-time analytics and batch processing. As industries increasingly shift to cloud-based infrastructures, transitioning to lakehouse architectures presents a path toward optimizing data pipelines and addressing modern computational demands.

Volume
Variability
Velocity
6 Vs of Big Data
Value
Variety
Veracity

Figure 2: The six principles of Big Data.

2.3 Data Ingestion techniques

In the context of data ingestion within a lakehouse architecture, understanding the six V's of Big Data is crucial for data collection, transfer, and organization of data for analysis and operational decision-making (Chen, 2024; GeeksforGeeks, 2023). The approach adopted dictate how information flows from edge devices to cloud infrastructures. Figure 2 display these dimensions:

1. **Volume** pertains to the massive amounts of data generated continuously from various sources, such as IoT devices, social media platforms, and transactional systems. Managing this vast volume necessitates scalable storage solutions and efficient processing capabilities.

2. **Velocity** refers to the speed at which data is produced and needs to be processed. In industrial settings, data streams in real-time from sensors and devices, requiring prompt ingestion and analysis to facilitate timely decision-making.

3. **Variety** encompasses diverse data types and formats, including structured data (e.g., relational databases), semi-structured data (e.g., XML, JSON files), and unstructured data (e.g., text, images, videos). Effective ingestion processes must accommodate this heterogeneity to integrate and analyze data comprehensively.

4. **Veracity** addresses the trustworthiness and quality of data. Ingestion systems must implement validation and cleansing mechanisms to ensure

the accuracy and reliability of data, as poor data quality can lead to flawed analyses and insights.

5. **Value**. Beyond the sheer accumulation of data, the primary goal is to extract meaningful insights that can drive business value. Efficient data ingestion lays the foundation for analytics that can uncover patterns, optimize operations, and informed strategic decisions.

6. **Variability** relates to the inconsistencies and fluctuations in data flow rates and formats. Data ingestion systems must be adaptable to handle these variations, ensuring consistent performance and data integrity despite changing data characteristics.

Furthermore, Reis and Housley (2022) delineates that ingestion methods can be broadly categorized into batch and streaming processes, each configured to specific industrial requirements. Batch ingestion processes data collected over a fixed period, reducing computational overhead by aggregating records before transfer. This method is ideally suited for historical analysis, periodic reporting, and machine learning model training, where immediate updates are not critical. In contrast, streaming ingestion processes data in real-time, making it essential for anomaly detection, event-driven analytics, and industrial automation. While streaming ingestion captures real-time system fluctuations, it requires infrastructure capable of managing continuous data flow without disruptions.

2.2.3.1 IoT Data Transmission and Communication Protocols

Once the nature of the data stream, whether batch or real-time, has been established, the next crucial step is selecting an appropriate transmission protocol. The choice of communication protocol determines how IoT devices transfer data to cloud environments, directly influencing factors such as latency, bandwidth consumption, and data integrity. Many IoT data streams rely on publish-subscribe (pub/sub) architectures, which facilitate asynchronous, decoupled communication between devices and cloud services (Saleh et al., 2024). The following middleware protocols are widely adopted in industrial and enterprise IoT environments:

- **Apache Kafka** is designed for high-throughput, event-driven architectures, making it suitable for large-scale ingestion and real-time event processing. Its ability to handle massive data streams efficiently integrates well with cloud-based analytics services, making it particularly advantageous for predictive maintenance and automated decision-making (Apache Software Foundation, 2024).

- **MQTT (Message Queuing Telemetry Transport)** is a lightweight protocol for low-bandwidth, high-latency networks. Alshammari (2023) conceptualized its publish-subscribe model makes it particularly ideal for telemetry, remote monitoring, and edge computing, where frequent status updates must be transmitted without excessive resource consumption."

- **RabbitMQ** is a message broker that supports multiple messaging patterns, including publish-subscribe and point-to-point queuing.It is beneficial in scenarios requiring guaranteed message delivery, message queuing, and transactional processing, particularly in hybrid cloud environments where IoT data must undergo intermediate processing before ingestion into a data lakehouse (Toshev, 2015).
- **HTTP/REST** follows a request-response communication model and is widely used for client-server interactions. However, it is less efficient for continuous telemetry compared to protocols like MQTT or Kafka. Traditionally, HTTP/REST is employed for device configuration, command transmission, and periodic data exchanges rather than real-time event streaming (Alshammari, 2023)

Each protocol fulfills a distinct IoT ingestion. MQTT is favoured for resource-constrained environments, Kafka excels in large-scale real-time analytics, RabbitMQ is suited for transactional processing and message durability, and HTTP/REST remains effective for structured interactions and control commands. The selection of the most appropriate protocol depends on the specific operational constraints and performance requirements of the system.

Beyond these protocols, according to Berardi et al. (2023) industrial IoT ecosystems often integrate digital twin technologies, where real-world assets are connected to virtual models via data provider components. The authors propose that these components communicate using a variety of IoT and IIoT protocols, including MQTT, HTTP/REST, and Open Platform Communications Unified Architecture (OPC UA).

Although OPC UA is fundamental protocol in industrial automation, facilitating secure, structured data exchange between machines, control systems, and cloud platforms, its detailed examination falls outside scope of this discussion. Nevertheless, its mention remains relevant due to its widespread adoption in smart manufacturing and IIoT infrastructures, particularly as a framework for interoperability, standardized communication, and the integration of digital twins within industrial control networks (Berardi et al., 2023b).

2.4 Distributed Computing Frameworks for Data Lake Processing

After the ingestion layer is deployed, data lakes typically store both structured and unstructured data generated by IoT devices. To process this data effectively, compute strategies must balance ingestion throughput, query performance, and long-term data retention. Cloud-based data lakehouses rely on distributed computing frameworks to support scalable batch and real-time processing, each optimized for specific workloads by balancing latency, throughput, and analytical complexity to meet diverse computational demands

The approach taken directly impacts data transformation, analytics speed, and storage efficiency, making it a key factor in industrial and business applications. Data processing follows two main paradigms:

- **ETL (Extract, Transform, Load)** processes data prior to storage, ensuring structured, pre-optimized datasets. Ideally, for traditional data warehouses that rely on *schema-on-write*, it optimizes query performance but introducing ingestion delays due to preprocessing overhead.
- **ELT (Extract, Load, Transform)** ingests raw data first and applies transformations on demand, leveraging scalable compute resources to provide greater flexibility. This paradigm is prevalent in modern data lakehouses utilizing *schema-on-read*, enabling faster data ingestion, real-time analytics, and schema evolution at the expense of increased storage requirements and computational overhead.

Our industrial case implements the ETL approach to enhance data quality, ensuring that data is cleansed, transformed, and validated before storage. This process reinforces data consistency, facilitates structured analytics, and optimizes query performance, critical aspects for our simulator's batch processing workloads. By applying transformations before ingestion, we reduce processing complexity and ensure that datasets are readily available for structured analysis.

Furthermore, distributed computing frameworks such as *Apache Spark* and *Apache Flink* enhance both batch and real-time processing capabilities in large-scale environments respectively. The choice between ETL and ELT ultimately depends on factors such as data volume, latency constraints, and computational resources. Table 2 provides a comparative analysis of these compute engines, highlighting their processing capabilities, latency, primary use cases, and compatibility with data lakehouse frameworks.

2.4.1 Apache Spark

Apache Spark is a distributed, in-memory computing framework designed for batch analytics, machine learning, and large-scale data transformations. In the article, Rajpurohit et al. (2023) states that *Spark* is widely used in data lakehouses due to its ability to process massive datasets across multiple nodes efficiently, leveraging a fundamental data abstraction that enables

Feature	**Apache Spark**	**Apache Flink**
Processing Type	Batch (with some streaming support)	Real-time stream processing
Use Cases	ETL, Machine Learning, Batch Analytics	Event-Driven Analytics, Anomaly Detection
Latency	Higher (minutes to hours)	Lower (milliseconds to seconds)
Best for	Data transformation, ML pipelines, historical analysis	IoT data streams, fraud detection, monitoring
Integration	Delta Lake, Apache Iceberg, Apache Hudi	Delta Lake, Apache Iceberg, Apache Hudi, Apache Kafka, AWS Kinesis, MQTT

Table 2: Comparison of Apache Spark and Apache Flink.

fault-tolerant distributed computations, known as **Resilient Distributed Datasets (RDDs)**. This design not only enhances Spark's ability to handle large-scale transformations and aggregations but also optimizes machine learning pipelines and batch analytics. It seamlessly integrates with modern lakehouse storage formats, including *Apache Iceberg*, *Delta Lake*, and *Apache Hudi*, which enhance data consistency, transactional reliability, and schema evolution. Moreover, Spark supports SQL-based querying, streaming analytics, and AI workloads, consolidating diverse analytical capabilities within a single framework (Šestak & Vovk, 2023).

2.4.2 Apache Flink

Apache Flink is designed for low latency stream processing, making it preferable for continuous data ingestion, real-time anomaly detection, and event-driven analytics in edge-based monitoring systems. Its ability to process event streams within milliseconds enables efficient tracking of device performance metrics (e.g., CPU, memory, disk I/O) and rapid identification of anomalies in AI model inferences. Flink supports event-time processing, handling out-of-order and late-arriving data, which is crucial for distributed edge environments where network conditions may introduce delays. Additionally, Flink is optimized for real-time transformations, such as data enrichment, anomaly detection, and predictive maintenance in resource-constrained edge scenarios. It integrates seamlessly with Apache Kafka, AWS Kinesis, and MQTT, facilitating end-to-end streaming lakehouse architectures for storing, analyzing, and correlating real-time and historical device performance data (Kalogerakis & Magoutis, 2023).

Nevertheless, given our emphasis on batch processing and scheduled data ingestion rather than continuous real-time analytics, we determined that Flink's capabilities did not align with the specific needs of our simulator. This makes Flink a powerful solution for monitoring AI workloads on edge devices while ensuring scalability and fault tolerance in distributed environments.

2.4.3 Open Table Formats

The evolution of data lake architectures has necessitated the development of open table formats—structured frameworks that enhance data management. These formats address fundamental challenges such as incremental data ingestion, metadata scalability, and compatibility across multiple processing engines. This section examines three of the most relevant open table formats for our industrial simulator: Delta Lake, Apache Hudi, and Apache Iceberg, evaluating their design principles, strengths, and trade-offs in the context of batch processing, historical data analysis, and metadata management. Table 3 presents a comparative summary of these formats, outlining their key features and applicability to our system's requirements.

Feature	Apache Iceberg	Apache Hudi	Delta Lake
Evolution	Supports schema amd partition evolution.	Allows schema modifications, but not partitioning changes.	Enforces strict schema consistency, not partition evolution.
ACID Transactions	Strong multi-writer consistency.	Ensures consistency during real-time ingestion.	Log-based ACID compliance.
Query Performance	Optimized for batch analytics.	Optimized for ingestion and upserts, but may introduce read overhead.	Optimized using Delta Logs for faster queries.
Metadata Management	Efficient pruning with optimized metadata structures.	Metadata overhead can increase with frequent modifications.	Delta Logs introduce scalability challenges.
CoW vs. MoR	Supports both.	Supports both.	Only supports CoW (emerging for MoR).
Streaming Capabilities	Not designed for real-time streaming, but supports incremental processing.	Best suited for real-time ingestion workloads.	Optimized for streaming with Apache Spark but lacks native support for other streaming engines.
Engine Compatibility	Has the widest variety of integrations, e.g. Spark, Flink, Snowflake, etc.	Primarily optimized for Spark and Flink.	Primarily optimized for Spark; limited support for other engines.

Table 3: Comparison of Open Table Formats: Apache Iceberg, Apache Hudi, and Delta Lake (Dremio, 2022; Lake, 2023).

2.4.3.1 Delta Lake

Delta Lake, developed by Databricks, extends Apache Spark by introducing ACID compliance, schema enforcement, and time travel. By leveraging log-based storage, it maintains transactional integrity, supports schema modifications while allowing rollback capabilities through its structured Delta Logs, which meticulously track every operation to preserve data integrity across updates (Databricks, 2024). Despite these advantages, Delta Lake was excluded from this study due to several limitations that can impact its applicability in broader analytics ecosystems. Its tight coupling with Apache Spark restricts compatibility with other compute engines such as Trino and Flink (Delta-IO, 2025), as well as BI tools like Grafana, potentially limiting its flexibility in heterogeneous processing environments. The reliance on Delta Logs also introduces metadata scaling challenges, particularly when dataset size increases, impacting performance in large-scale cloud-native infrastructures (Gopalan, 2022). These considerations warrant further evaluation when comparing open table formats for scalable and efficient data lake architectures.

2.4.3.2 Apache Hudi

Apache Hudi, initially developed by Uber (Chandar, 2017), focuses on incremental data processing in data lakes by optimizing upserts, deletions, and schema evolution. Unlike traditional batch-oriented frameworks, Hudi is designed for low-latency ingestion, enabling rapid data updates while maintaining transactional consistency. Its built-in indexing mechanisms reduce I/O overhead, accelerating updates and deletes while improving query execution efficiency (Hudi, 2021). As a competing open table format to Apache Iceberg, Hudi is often evaluated for its strengths in real-time data processing and frequent schema modifications. Its approach to metadata management and transactional integrity presents an alternative design philosophy, making it relevant for comparison in this study to assess potential advantages in specific workloads.

2.4.3.3 Apache Iceberg

Apache Iceberg, originally developed by Netflix, addresses the inefficiencies of Apache Hive in managing large-scale analytical databases (Ho, 2022). It introduces a more efficient metadata layer that enables features such as schema evolution, ACID transactions, and time travel, making it particularly well-suited for batch analytics and long-term data retention. Unlike traditional data lakes, Iceberg provides partition pruning and snapshot-based versioning, significantly enhancing query performance while ensuring strong transactional consistency. Given its scalability, query optimization, and ability to allow schema evolution without requiring manual rewriting or data migration, Iceberg was chosen for this study. It integrates seamlessly with distributed data processing engines such as Spark or Flink (Iceberg, 2024a). Its ability to manage large analytical workloads efficiently makes it an optimal choice for batch processing and historical data analysis, ensuring high-performance query execution while maintaining strong data consistency.

Apache Iceberg Architecture
Ho (2022) was among the first to provide an in-depth explanation of the multilayered structure of Apache Iceberg, as illustrated in Figure 3, highlighting its metadata management and data organization strategy. The architecture is divided into two primary layers:

- **Metadata Layer**. This layer orchestrates table metadata management, including schema definitions, historical snapshots, and the current state of the dataset by maintaining structured records of changes over time.
  - **Metadata Files**. These files store critical information about snapshots and historical versions, facilitating time-travel queries and rollback operations.
  - **Manifest List** serves as an indexing mechanism, tracking the relevant data files associated with each snapshot. This structure enables Apache Iceberg to optimize query execution by pruning irrelevant data, improving read performance.
  - **Manifest Files** contain references to individual data files and deleted files within a snapshot. This file-based tracking allows efficient garbage collection and enhances data pruning techniques, reducing unnecessary I/O operations.
- **Data Layer**. The data layer is responsible for storing the actual dataset in optimized columnar formats such as Apache Parquet. It supports compression, indexing, and efficient query execution, ensuring that analytical workloads can scale effectively without performance degradation.

This layered approach exemplifies Apache Iceberg's ability to deliver transactionally consistent, high-performance data lakes by combining advanced metadata management with scalable data storage techniques. By leveraging these architectural principles, Iceberg enables cost-efficient, scalable, and performant analytical processing across distributed environments.

2.4.4 Storage: Read and Write Strategies

After determining the appropriate communication protocol, the next aspect of a lakehouse architecture is the strategy for handling read and write opera-

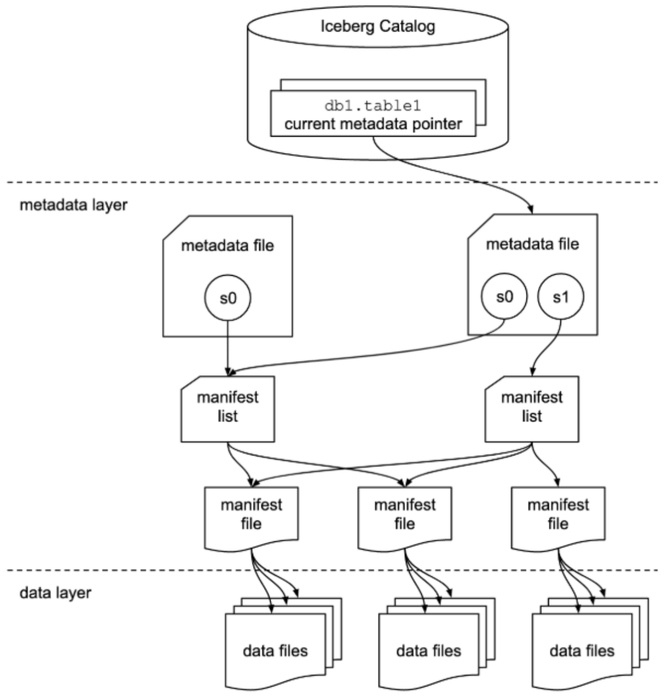


Figure 3: Apache Iceberg Metadata and Data Layer Architecture (Iceberg, 2024b).

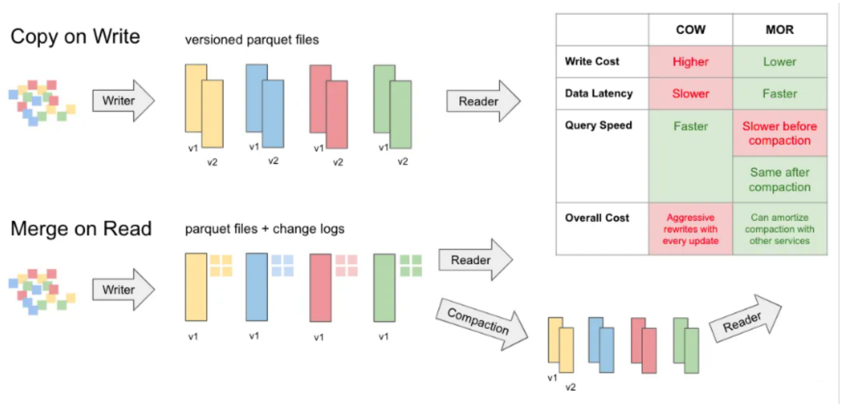
tions within transactional data lakes. These moderns architectures incorporate advanced data modification techniques to balance transactional consistency, query efficiency, and data ingestion speed. The selection between strategies depends on workload-specific constraints, including update frequency, query latency tolerance, and storage overhead considerations. Understanding these trade-offs is fundamental for designing a storage layer that aligns with the broader system's performance and scalability requirements.

2.4.4.1 Copy-on-Write (CoW) Strategy

The *Copy-on-Write (CoW)* approach rewrites entire data files when updates occur, ensuring strong data consistency while reducing query latency. This method suits batch analytics, where structured, compact datasets improve query execution speed. However, the primary drawback is its high write latency, as rewriting files during each update incurs significant I/O overhead. This characteristic makes CoW less suitable for high-ingestion environments, particularly those requiring low-latency data availability for real-time processing.

2.4.4.2 Merge-on-Read (MoR) Strategy

The *Merge-on-Read (MoR)* approach prioritizes write efficiency and ingestion speed by storing changes as delta logs rather than immediately rewriting entire files. These changes are later merged during query execution, reducing write overhead and enabling low-latency access to fresh data. MoR is particularly advantageous for real-time streaming applications, as it allows high-frequency updates while deferring computational costs to the read stage. However, this strategy can introduce query performance trade-offs, as merging records at read time may increase latency.



	COW	MOR
Write Cost	Higher	Lower
Data Latency	Slower	Faster
Query Speed	Faster	Slower before compaction
		Same after compaction
Overall Cost	Aggressive rewrites with every update	Can amortize compaction with other services

Figure 4: Copy-on-Write (CoW) and Merge-on-Read strategies overview.

Figure 4 illustrates the trade-offs between Copy-on-Write (CoW) and Merge-on-Read (MoR) approaches in data processing. CoW prioritizes query speed by rewriting entire files upon modification, providing strong data consistency and facilitating batch analytics, though at the expense of higher write latency. In contrast, MoR optimizes ingestion speed by deferring data compaction until read time, thereby reducing immediate write costs but introducing query overhead before compaction occurs.

2.5 Governance and Metadata Optimization for Iceberg in AWS

This section examines different metadata strategies used in AWS, evaluating traditional approaches and contrasting them with more scalable alternatives that incorporate Apache Iceberg and AWS Glue. Through this comparative analysis, AWS Glue emerges as the most efficient and adaptable solution for Iceberg-based metadata management in cloud-native environments.

2.5.1 Limitations of Traditional Metadata Strategies

Before the adoption of *AWS Glue*, various metadata management approaches were employed in Iceberg-based environments. However, these solutions often introduced performance bottlenecks, limited scalability, and integration challenges, making them less effective in modern cloud deployments. The following sections examine the key limitations of these traditional strategies.

2.5.1.1 Apache Hive Metastore (HMS)

Apache Hive Metastore (HMS), originally designed for Hadoop-based data lakes, faces significant challenges in cloud-native environments, particularly when managing Iceberg tables at scale. As datasets grow, HMS experiences performance degradation, leading to slow metadata retrieval and inefficiencies in high-frequency query environments. Furthermore, HMS lacks native support for time-travel queries, limiting its ability to efficiently retrieve historical data compared to *AWS Glue*, which seamlessly integrates with *Athena*, *Redshift Spectrum*, and *EMR* for such operations. Another critical drawback is its reliance on dedicated infrastructure, as HMS does not provide serverless capabilities, making it less adaptable to modern, elastic cloud architectures.

2.5.1.2 AWS Lake Formation

AWS Lake Formation strengthens fine-grained access control and security policies, offering a robust governance framework. However, it presents challenges when handling Iceberg metadata due to several architectural limitations. Unlike *AWS Glue*, *Lake Formation* is not Iceberg-native, as it is primarily optimized for AWS-native formats such as *Parquet* and *Delta Lake*, making seamless integration with Iceberg more complex. It also lacks dynamic schema tracking and versioning, which are critical for Iceberg's transactional consistency. Additionally, fine-grained access control mechanisms introduce higher query overhead, leading to performance trade-offs, particularly in high-frequency query scenarios. Another limitation is *Lake Formation*'s restricted schema evolution capabilities. Due to its strong governance features, *AWS Lake Formation* is most suitable for highly regulated industries where data security takes precedence over query performance and schema evolution.

2.5.1.3 Apache Nessie

Apache Nessie introduces a Git-like versioning system for metadata, allowing branching and rollback capabilities that facilitate multi-user metadata management. While this approach is innovative, *Nessie*'s ecosystem remains less mature than *AWS Glue*, limiting its adoption in large-scale Iceberg deployments. One of the platform's primary limitations is that it is still evolving and lacks production-level optimizations needed for managing large transactional lakehouses. Additionally, *Nessie* offers limited cloud-native support, requiring manual setup and dedicated infrastructure management, whereas *AWS Glue* is fully managed within AWS. Another key drawback is *Nessie*'s limited support for query engine integrations, making it less versatile than *AWS Glue*, which seamlessly connects with AWS-native analytics services.

2.5.1.4 AWS Glue

AWS Glue is as a fully managed metadata catalog and schema registry within AWS, addressing the limitations of traditional metadata management approaches. Unlike *HMS*, *Lake Formation*, and *Nessie*, *Glue* provides automated schema discovery and versioning, allowing Iceberg tables evolve dynamically, partition management and indexing, while maintaining query compatibility for services such as *Athena*, *Redshift Spectrum*, and *EMR*, as well as snapshot tracking and metadata pruning, enabling time-travel queries while reducing unnecessary data scans. Furthermore, its classification and tagging features improve data discoverability. This integration with AWS-native services establishes *Glue* as the leading solution for centralizing schema enforcement, transactional consistency, and metadata retrieval, making it a scalable and efficient choice for Iceberg-based data lakehouses.

This chapter has demonstrated how the progressive evolution of data management systems has culminated in the emergence of modern architectural paradigms such as the data lakehouse, a transformative approach that harmonizes large-scale data processing with robust governance frameworks. The integration of efficient data ingestion pipelines and distributed computational frameworks provides the foundational infrastructure necessary for high-performance analytics at scale. Complementing these capabilities, advanced metadata management systems, particularly Apache Iceberg's implementation within AWS ecosystems, deliver the critical governance layer that ensures data reliability, auditability, and transactional integrity.

The confluence of these technological advancements creates a powerful operational substrate that effectively reconciles the traditionally competing demands of analytical scalability and ACID-compliant transactionality. This synthesis represents a significant milestone in enterprise data architecture, offering organizations, a path to achieve both operational flexibility and governance rigor.

Chapter 3

Methodology

This chapter presents the methodology employed in developing our data pipeline architecture from design to deployment. The architecture is built on principles of replicability and adaptability, designed not just for immediate deployment success but for seamless integration across diverse industrial ecosystems. This modular approach: (1) promotes sustainability and scalability, (2) minimizes redundant development through reusable components, (3) incorporates clear implementation guidelines, and (4) accounts for cost considerations and system limitations.

Our evaluation framework thoroughly examines each architectural component through:

- **Simulation-based Testing**. Utilizing our custom-developed simulator that collects system/hardware metrics.
- **Comparative Technology Assessment**. Benchmarking batch processing performance between Apache Hudi and Iceberg.
- **Core Functional Analysis**. Validating the performance of essential system capabilities.

The workflow initiates with IoT devices generating telemetry data, emulating industrial sensor networks in an edge computing environment. This raw data stream undergoes structured ingestion and persistent storage, establishing a reliable foundation for downstream processing. A meticulously designed ETL pipeline then transforms and loads the data into a transactional data lake, preserving atomicity and reliability throughout the data lifecycle as illustrated in Figure 5.

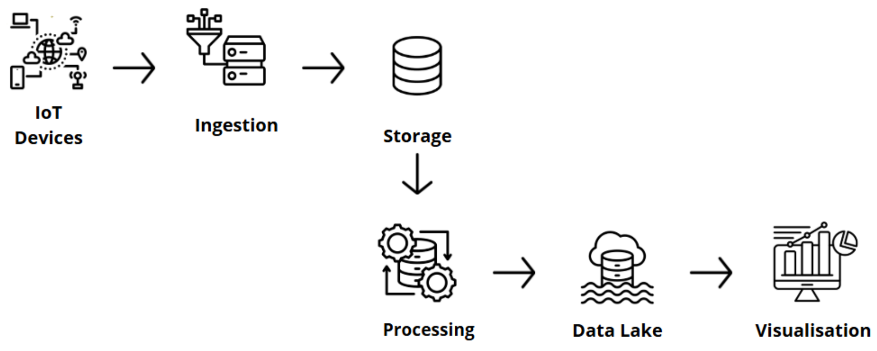


Figure 5: End-to-End solution workflow.

1. **IoT Devices**

   The data pipeline begins with IoT devices that generate real-time telemetry data, including CPU usage, memory usage, network I/O, and disk I/O. These devices simulate industrial sensors, logging critical operational metrics.

2. **Ingestion**

   Raw data is collected from IoT sources and transferred to an ingestion system. This stage ensures that the incoming data is continuously or periodically captured for further processing.

3. **Storage**

   Once ingested, the data must be stored in an appropriate infrastructure. This is often achieved using distributed file systems or cloud-based storage solutions in our case.

4. **Processing and Transformation**

   In this phase, raw data undergoes cleaning, normalization, aggregation, and enrichment through ETL (Extract, Transform, Load) processes. This step converts raw data into structured, usable formats.

5. **Data Lake**

   Processed data is then stored in a Data Lake, which provides scalability, transactional support, and schema evolution capabilities. This architecture allows for efficient querying and retrieval while supporting analytical workloads.

6. **Visualization**

Finally, insights are made available to end users or downstream systems through interactive dashboards, business intelligence tools, and visual reports, helping decision-makers interact with data and extract actionable intelligence.

The ultimate goal is a robust, context-agnostic architecture that balances technical sophistication with practical transferability, enabling seamless adoption across heterogeneous industrial domains while providing comprehensive performance insights through rigorous comparative testing.

The subsequent sections present a comprehensive examination of our proposed workflow and explore the distinct contributions of data engineers, data scientists, and data analysts at each phase of the pipeline, demonstrating how their specialized expertise collectively enables the system's success.

3.1 Data Lakehouse Architecture Design

The proposed architecture adopts a modular paradigm that optimally supports data ingestion, storage, processing, and consumption from both simulated edge devices and streaming sources. This design is composed of four core modules, each addressing a specific phase of data management:

- **Ingestion Module**. Responsible for capturing and storing raw data from IoT devices or real-time streaming sources.
  - **IoT Core and Data Sources**: Simulates IoT devices or external streaming sources that generate raw telemetry data, including performance metrics, activity logs, and other relevant monitoring information.
  - **Storage Layer**: The ingested data is first stored in Amazon S3, which serves as the primary data repository. To optimize queries and improve processing efficiency, the data is cleaned and stored in Parquet and ORC formats, allowing faster reads and reduced storage costs.
- **Compute Module**. Handles data transformation, processing, and query execution.
  - **Data Processing**: Data transformations occur using AWS Glue ETL and Apache Spark, depending on the workload type. This layer supports both batch and streaming processing, making it adaptable to various data analysis scenarios.
  - **Metadata Management**: The AWS Glue Data Catalog manages metadata, including discovery, versioning, and schema evolution tracking. It maintains snapshots and table structures in Apache Iceberg, ensuring query consistency and optimization.

- **API Module**. Facilitates query execution and data accessibility.
  - **Query Execution**: Amazon Athena is integrated into the architecture to allow serverless, SQL-based querying over structured data stored in Apache Iceberg and Parquet formats. This enhances analytical capabilities and reduces computation overhead.
- **Front-End Module**. Focuses on data visualization and consumption, for real-time monitoring and analytics.
  - **Visualization and Analytics Tools**: Processed data is made available for analysis using Grafana and other Business Intelligence (BI) tools. These platforms enable users to perform queries, real-time monitoring, and reporting based on processed insights.

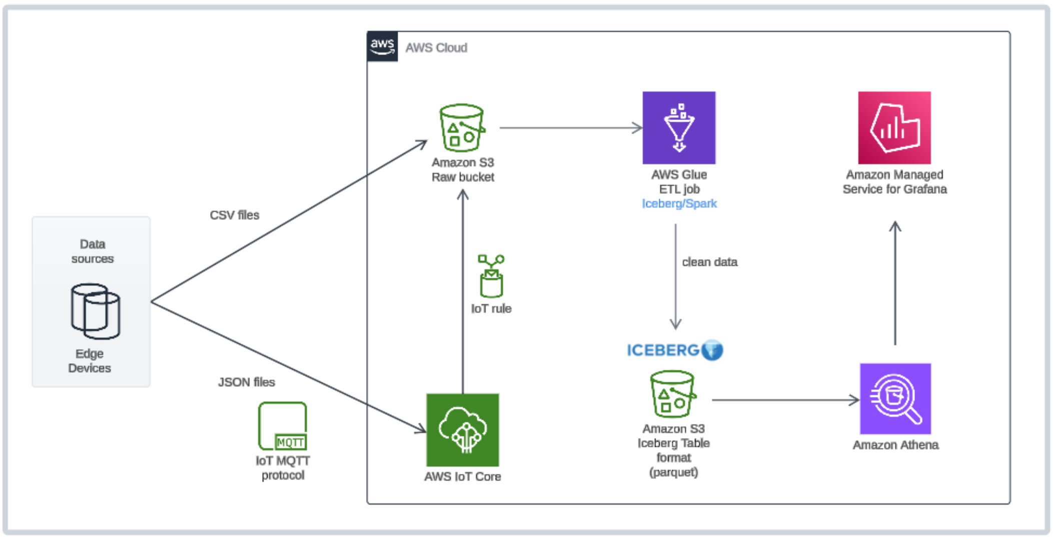


Figure 6: Transactional data lake architecture.

In the following sections, we will examine the structured framework of our proposed Data Lakehouse architecture, as illustrated in Figure 6. Our analysis will explore its ingestion, storage, processing, and consumption layers, as well as the advanced capabilities that make it a unified and scalable system.

3.1.1 Services used in the architecture

a) Networking: VPC and Subnet

The networking layer forms the foundation of the architecture, working as the outer boundary that encapsulates all internal services within a secure and controlled AWS Cloud environment.

- **Amazon VPC (Virtual Private Cloud)** serves as the isolated network environment where all cloud resources are provisioned, ensuring a secure and structured communication between services. It encapsulates the entire data architecture within a protected network boundary. Through customizable IP address ranges, subnets, and routing configurations, the VPC controls how traffic flows between components, maintaining both security and flexibility across the architecture.

- **Subnets.** The VPC is subdivided into private and public subnets to optimize security and accessibility:
  - **Private subnets** host critical backend services such as Glue ETL jobs, metadata storage (AWS Glue Data Catalog), and transactional data storage (S3 with Iceberg tables), preventing direct external exposure.
  - **Public subnets** are used for services that require controlled external interaction, such as IoT Core, which receives telemetry data from external edge devices.

The configuration of this network segmentation within AWS enforces that data remains secure while enabling high availability and fault tolerance across AWS Availability Zones.

b) Identity and Access Management (IAM) and Policies

Within the AWS ecosystem, Identity and Access Management (IAM) acts as the core security layer that governs access control across all services. It enforces strict permissions based on roles, allowing each component of the architecture to interact only with the resources it needs. IAM prevents unauthorized access and minimizes the risk of data breaches or accidental data exposure. Following the principle of least privilege, each AWS service —such as Glue, S3, Athena, and IoT Core— is assigned a custom role with only the permissions necessary for its operation. S3 bucket policies are configured to allow access exclusively to those services or users explicitly authorized, ensuring secure handling of both raw and processed data. This role-based security model not only enforces access boundaries but also contributes to the overall efficiency and trustworthiness of the system.

c) Amazon Simple Storage Service (S3)

Storage is not merely about keeping information, it is about accessibility, efficiency, and longevity. Amazon Simple Storage Service (S3) is more than just an object storage container, it serves as the primary data repository where incoming sensor data was preserved in its original format for flexibility in downstream processing. AWS Glue was later used to transform and optimize the raw data into columnar formats like Parquet, enabling faster queries and reducing storage costs.

d) AWS Glue

Before analysis can begin, data must be cleaned, normalized, and structured. AWS Glue automates this process through Extract, Transform, and Load (ETL) pipelines, converting noisy, inconsistent IoT sensor data into a reliable, high-quality dataset. Whether processing batch workloads or real-time streams, Glue adapts seamlessly, offering a flexible and scalable solution.

The Glue Data Catalog further enhances this system by providing robust metadata management. Serving as a blueprint for data relationships and version control for Apache Iceberg tables. It integrates effortlessly with Amazon Athena, enabling SQL-based querying across vast datasets without complex indexing. By eliminating manual data wrangling, AWS Glue delivers a serverless, scalable ETL solution that fits seamlessly into the broader data ecosystem.

e) AWS IoT Core

AWS IoT Core operates as the central ingestion gateway, serving as a real-time bridge between edge devices and the cloud. By facilitating seamless data flow, robust security, and intelligent automation, it ensures efficient system wide communication. Given the critical need for secure telemetry transmission, IoT Core delivers a scalable, low-latency MQTT broker, enabling edge devices to publish performance metrics with minimal overhead.

When implementing this service we must consider how provisioning can scale to accommodate large fleets of devices while maintaining security and operational efficiency. Provisioning refers to the process of securely registering and configuring a device so it can authenticate and communicate with cloud services.

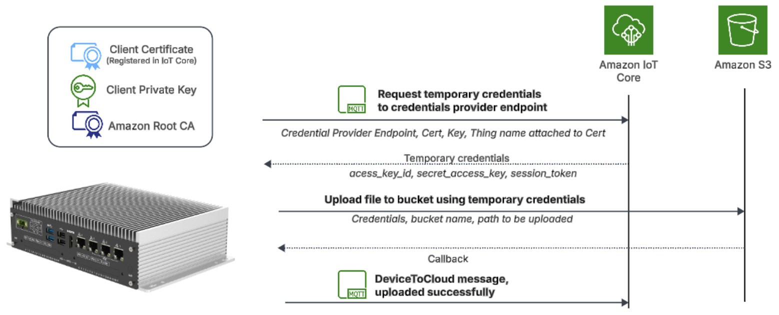

Client Certificate
(Registered in IoT Core)
Client Private Key
Amazon Root CA
Amazon IoT Core
Amazon S3
Request temporary credentials to credentials provider endpoint
Credential Provider Endpoint, Cert, Key, Thing name attached to Cert
Temporary credentials
acess_key_id, secret_access_key, session_token
Upload file to bucket using temporary credentials
Credentials, bucket name, path to be uploaded
Callback
DeviceToCloud message, uploaded successfully

Figure 7: Provisioning mechanism.

To prevent unauthorized access, the AWS IoT Core provisioning workflow involves devices authenticating using X.509 client certificates (signed by the Amazon Root CA) and private keys to securely obtain temporary AWS credentials through IAM role assumption, as shown in Figure 7. These credentials enable devices to upload data to designated Amazon S3 buckets while being constrained by fine-grained IoT policies. The entire process is validated through a `DeviceToCloud` MQTT callback mechanism that confirms successful data transmission and maintains end-to-end visibility.

f) Amazon CloudWatch

In any complex system, visibility is paramount; without it, issues go undetected, performance degrades, and inefficiencies accumulate. Amazon CloudWatch serves as the centralized monitoring and logging system, orchestrating observability, alerting, and automation within a unified framework.

AWS Glue ETL jobs, IoT Core message streams, and Athena queries generate logs that CloudWatch stores, indexes, and analyzes. These logs provide detailed insights into errors, query execution times, and data processing anomalies, serving as a historical record that aids in troubleshooting and optimization. Rather than waiting for issues to escalate, CloudWatch enables customizable alarms that trigger alerts when system performance deviates from expected behavior. These alarms empower administrators to take immediate action, whether it's scaling resources, debugging failed queries, or addressing network congestion.

g) Amazon Athena

Amazon Athena is a serverless, interactive query service that enables users to analyze structured datasets stored in Amazon S3 using standard SQL. Unlike traditional database systems that require manual provisioning, indexing, and performance tuning, Athena operates on-demand, executing queries directly on stored data.

Athena simplifies data exploration and analysis while reducing operational complexity and costs by eliminating the need for dedicated query engines or persistent databases. It also integrates with AWS Glue Data Catalog, leveraging metadata to optimize queries and enable schema evolution. This makes it an ideal solution for ad-hoc analysis, report generation, and performance monitoring.

h) Amazon Managed Service for Grafana

This service acts as the visual intelligence layer, bridging the gap between analytics and informed decision-making. Converting data into actionable insights is as crucial. At the final stage of the pipeline, AMG transforms structured data into interactive visualizations, helping engineers, analysts, and decision-makers monitor trends, detect anomalies, and optimize system performance at a glance.

Unlike self-hosted Grafana instances, AMG provides a fully managed, scalable solution that seamlessly integrates with AWS services such as Athena, CloudWatch, and IoT Core, allowing real-time dashboards to reflect live system metrics. With customizable panels, advanced alerting mechanisms, and role-based access controls, it facilitates collaborative monitoring and operational efficiency while eliminating the overhead of infrastructure management.

3.2 Development Environment

We have selected AWS as our cloud service provider to implement our project due to its widespread adoption and recognition as a leading platform within the company where I completed my internship. This decision gives access to a reliable infrastructure and an ecosystem of cutting-edge services that align with the scope of our industrial simulator. The development environment is structured to support monitoring data collection running on simulated edge devices within Virtual Machines (VMs). The implementation follows a modern Lakehouse architecture, which allows seamless ingestion, structured storage, and efficient management of performance-related metrics.

3.2.1 Ingestion and Storage

The industrial simulator acts as the main data source, replicating real-world edge computing environments where IoT devices generate telemetry data. The simulation setup consists of three devices, `Device-8C16G`, `Device-4C8G`, and `Device-2C4G`, as shown in Figure 8. Each device is assigned different workloads according to its hardware configuration, as detailed in Table 4. These simulated workloads, or function costs, can be classified as low, medium, or high.

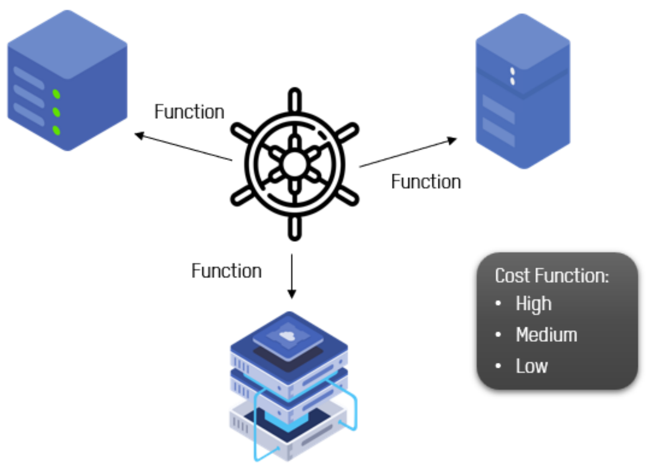


Figure 8: Industrial simulator concept.

Once the simulator is developed, this fleet of devices generates structured logs in CSV format, which include their respective device IDs and timestamps in UTC format, as well as essential resource utilization indicators such as CPU usage, memory consumption, disk I/O, and network activity. The collected logs are first stored in their raw form within an Amazon S3 bucket, maintaining the original CSV format. This raw storage layer is used as a repository for historical data before any transformation or processing occurs.

Another S3 bucket will store the processed clean data converted into parquet format to be managed as Iceberg tables. This transformation optimizes query performance and enables schema evolution, which is crucial for handling different device variations over time.

Devices	**Configurations**	**Workloads**
Device-8C16G	8 cores, 16 GB RAM	Ideal for intensive tasks
Device-4C8G	4 cores, 8 GB RAM	Suitable for mixed tasks
Device-2C4G	2 cores, 4 GB RAM	Best for testing or light tasks

Table 4: Device configurations and recommended workloads

Initially, an AWS object that functions as a logical entity integrating physical items with the cloud had to be built using the AWS IoT Core service. To achieve this, a secure and reliable connection was required, along with a set of certificates, and private and public keys. Once this entry point is configured with AWS, those function charges, along with the corresponding device ID and durations, are published as a payload in JSON format, following the schema in Figure 9.

```
payload = {
    "device_id": device_id,
    "function_cost": function_cost,
    "start_time": start_time,
    "end_time": end_time
}
```

Figure 9: JSON payload structure.

Following that, these devices transmit this payload via MQTT, and CSV files are uploaded to an Amazon S3 bucket, named `device-metric-raw-ingest-bucket`, using the AWS *boto3* client. This communication protocol was selected due to its extensive usage and outstanding lightweight messaging capabilities for Internet of Things (IoT) ecosystems. However, since MQTT does not persist messages by default (unless a persistent session is configured), any unpublished data is lost if a device goes offline. Despite this limitation, MQTT remains a highly scalable solution, allowing numerous devices to publish data simultaneously without significant latency overhead.

For this experiment, Tthe event-driven processing pipeline within AWS IoT Core dynamically routes incoming data to Amazon S3, reducing infrastructure complexity and enabling scalable IoT deployments while maintaining data integrity and system reliability.

The simulator generates data every second, handling an estimated 19,500 messages daily, requiring a high-performance ingestion framework for efficient data processing. To achieve this, IoT Core uses an endpoint that receives structured AWS objects. The data is then published under the topic `devicetocloud/function`. Since a single device can publish across multiple topics without data interference, and multiple devices can subscribe to the same topic, each object remains logically isolated within its respective topic, ensuring atomicity and flexible data exchange between devices and the cloud. Subsequently, we can subscribe to the stated topic. Figure 10 confirms that the payload is successfully published on IoT Core using the MQTT test client server.

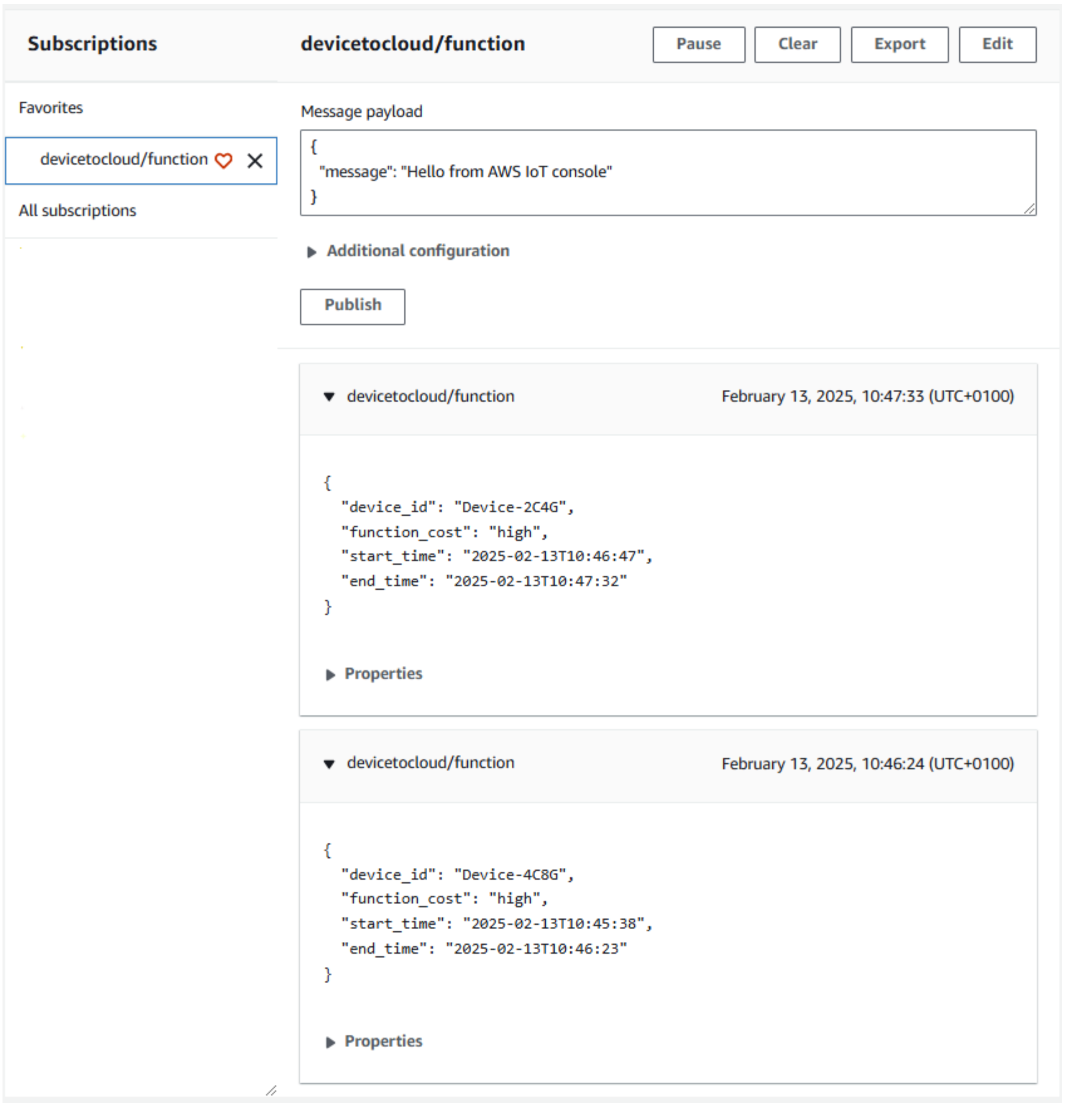
3.2.1.1 Data Storage

Amazon Simple Storage Service (S3) has been selected as the storage solution for our raw data, such as JSON and CSV files, due to its cost-effectiveness, scalability, and flexibility. S3 operates on a pay-as-you-go pricing model, making it a very affordable solution for data storage compared to traditional databases. It offers virtually unlimited storage capacity, allowing us to handle large volumes of data without worrying about capacity constraints. Its schemaless design also allows for storing and retrieving data in various formats, including JSON and CSV, making it particularly advantageous for managing heterogeneous datasets without requiring predefined schemas.

This repository remains intact for auditability and reproducibility of system performance. Furthermore, S3 integrates seamlessly with other AWS services, such as AWS Glue and Amazon Athena, facilitating effective data processing and analysis workflows.

In contrast, services like Amazon Redshift and Amazon RDS are not well-suited for our storage and processing needs due to their structural constraints and cost considerations. Amazon Redshift requires a predefined schema, making it less flexible for evolving JSON data typically found in semi-structured logs. Additionally, real-time ingestion costs are significantly higher compared to other storage solutions. Similarly, Amazon RDS is designed for structured, transactional data rather than large-scale log ingestion. While it supports some JSON processing, it lacks the efficiency and scalability needed for handling high-frequency telemetry data from edge devices.

Therefore, Amazon S3 was chosen as the optimal storage solution due to the combination of affordability, and seamless integration with other AWS-based



```
{
  "message": "Hello from AWS IoT console"
}
```

devicetocloud/function February 13, 2025, 10:47:33 (UTC+0100)

```
{
  "device_id": "Device-2C4G",
  "function_cost": "high",
  "start_time": "2025-02-13T10:46:47",
  "end_time": "2025-02-13T10:47:32"
}
```

devicetocloud/function February 13, 2025, 10:46:24 (UTC+0100)

```
{
  "device_id": "Device-4C8G",
  "function_cost": "high",
  "start_time": "2025-02-13T10:45:38",
  "end_time": "2025-02-13T10:46:23"
}
```

Figure 10: MQTT test client verifying payload publication.

services. Its ability to store both raw and processed data without schema constraints ensures efficient management of monitoring data collected from edge devices while enabling future advanced analytics and machine learning applications.

3.2.2 ETL Workflow: Data Transformation

In this step, an AWS Glue ETL pipeline was developed to process telemetry data collected from edge devices and store it in an Apache Iceberg table. This dataset, which includes metrics such as CPU usage, memory consumption, disk I/O, and network activity, is initially stored in Amazon S3 in both CSV and JSON formats. The ETL job, found in *Appendix A*, extracts, processes, and loads this data into Iceberg tables, ensuring schema evolution, data completeness, and efficient querying. The pipeline is built using a **Glue context**, which extends the underlying Spark context to provide Glue-specific capabilities such as reading from the AWS Glue Data Catalog, working with DynamicFrames, and managing job execution within the distributed Spark environment.

The process begins by retrieving the raw CSV files from Amazon S3, organized by date. The pipeline loads this data into a Spark DataFrame, where timestamps are converted into a standardized format. Given that telemetry data may contain gaps due to network latency or irregular reporting intervals, the ETL job detects and generates missing timestamps using Spark window functions. The missing data points are reconstructed as sequences of timestamps, keeping the dataset continuous for time-series analysis. Subsequently, missing values for the given metrics are forward-filled, propagating historical values through gaps.

Once the CSV data is processed, the pipeline retrieves JSON files containing function execution metadata, including `start_time` and `end_time` for each recorded function. These JSON records are dynamically loaded from S3 and transformed into a structured format. The CSV telemetry data is then joined with the JSON function metadata based on device ID and timestamp range, enriching the telemetry dataset with additional execution details. Any duplicate records that may have resulted from overlapping data ingestion are removed to maintain a clean and accurate dataset.

After completing the transformation, the telemetry data is structured into an Apache Iceberg table, which internally stores data in a columnar format such as Parquet. The job assigns partitioning keys based on `year`, `month`, `day`, and `device_id` optimizing query performance and minimizing unnecessary scans during retrieval. The processed dataset is then inserted into two separate Iceberg tables: one following a *Copy-On-Write (CoW)* model, which maintains strong consistency by rewriting entire files during updates, and another using *Merge-On-Read (MoR)*, which supports more efficient incremental updates by maintaining separate delta logs.

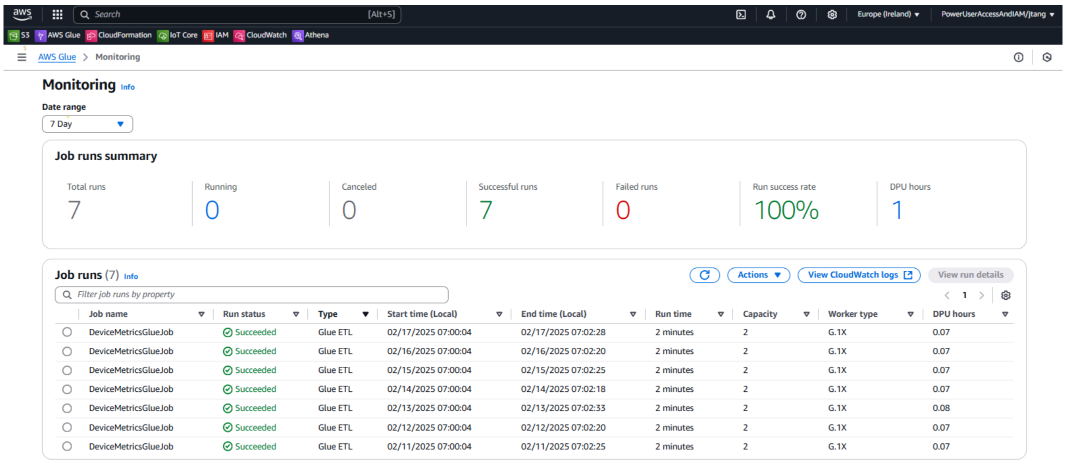
At the final stage, the Glue job successfully commits the processed data to the Iceberg tables and stops the Spark session. The telemetry data, now structured and stored efficiently, are accessible for further analysis in monitoring dashboards, performance evaluation, and predictive analytics. The combination of Iceberg schema evolution, automated data enrichment, and partitioning strategies ensures that telemetry data can be queried with minimal overhead while maintaining a high level of accuracy and completeness.

Moreover, Figure 11 the execution details of the *DeviceMetricsGlueJob*, which runs daily with an average DPU (Data Processing Unit) usage of 0.07 DPU hours per execution. According to AWS Glue pricing, the cost per DPU hour is $0.44(AWS, 2025)$.

The cost per job run is estimated as:

$$0.07 \textbf{ DPU hours} \times 0.44 \textbf{ USD/DPU hour} = 0.0308 \textbf{ USD} \tag{1}$$

Thus, the estimated cost for running this job once per day is approximately



Date range: 7 Day

Job runs summary

Total runs	Running	Canceled	Successful runs	Failed runs	Run success rate	DPU hours
7	0	0	7	0	100%	1

Job runs (7)

Job name	Run status	Type	Start time (Local)	End time (Local)	Run time	Capacity	Worker type	DPU hours
DeviceMetricsGlueJob	Succeeded	Glue ETL	02/17/2025 07:00:04	02/17/2025 07:02:28	2 minutes	2	G.1X	0.07
DeviceMetricsGlueJob	Succeeded	Glue ETL	02/16/2025 07:00:04	02/16/2025 07:02:20	2 minutes	2	G.1X	0.07
DeviceMetricsGlueJob	Succeeded	Glue ETL	02/15/2025 07:00:04	02/15/2025 07:02:25	2 minutes	2	G.1X	0.07
DeviceMetricsGlueJob	Succeeded	Glue ETL	02/14/2025 07:00:04	02/14/2025 07:02:18	2 minutes	2	G.1X	0.07
DeviceMetricsGlueJob	Succeeded	Glue ETL	02/13/2025 07:00:04	02/13/2025 07:02:33	2 minutes	2	G.1X	0.08
DeviceMetricsGlueJob	Succeeded	Glue ETL	02/12/2025 07:00:04	02/12/2025 07:02:20	2 minutes	2	G.1X	0.07
DeviceMetricsGlueJob	Succeeded	Glue ETL	02/11/2025 07:00:04	02/11/2025 07:02:25	2 minutes	2	G.1X	0.07

Figure 11: Job run monitoring.

$0.0308 (3.08 cents) per execution. For a full month (30 days), assuming the job runs daily, the total estimated cost would be:

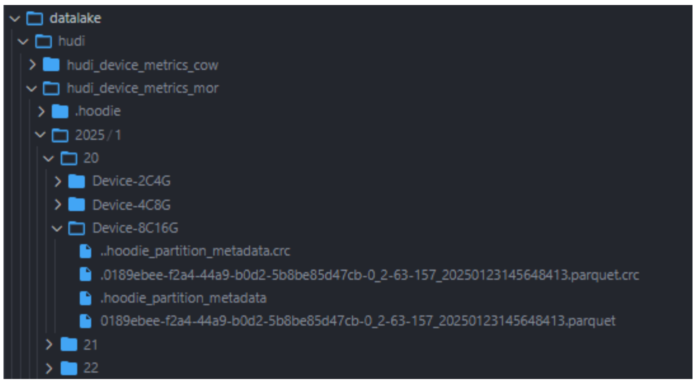
$$0.0308 \times 30 = 0.924 \ \textbf{USD} \tag{2}$$

indicating that the job incurs less than **$1 per month** in execution costs under the current configuration. This cost analysis provides insight into the efficiency of resource allocation in AWS Glue, demonstrating the feasibility of running automated ETL processes in a cost-effective manner.

Due to its low computational overhead, we deployed the ETL process in the cloud following the guidelines outlined in *Section 3.3.1.1*, which establish that Iceberg tables are preferable for our industrial simulator. Without delving into extensive details, we briefly examined Hudi's behavior and found it to be more complex to configure, as its setup heavily relies on fine-tuning various parameters. Furthermore, schema evolution in Hudi presents greater challenges compared to Iceberg, making it less suitable for our use case.

Following the same cloud-based deployment approach, we evaluated Apache Hudi as an alternative storage framework, the process was virtually identical to the Spark job designed for Apache Iceberg, with the main distinction being Hudi's additional configuration options for data ingestion. The specific settings used for Hudi are detailed in *Appendix B*.

Once the data was cleaned, we conducted a series of experiments, which are discussed in detail in the following chapter. This is one of the major steps in in data engineering, as it streamlines subsequent stages in the Big Data pipeline while facilitating seamless collaboration and agile development. Figure 12 illustrates the repository structure generated when using Apache Hudi as the storage framework.



```
datalake
  hudi
    hudi_device_metrics_cow
    hudi_device_metrics_mor
      .hoodie
      2025 / 1
        20
          Device-2C4G
          Device-4C8G
          Device-8C16G
            ..hoodie_partition_metadata.crc
            .0189ebee-f2a4-44a9-b0d2-5b8be85d47cb-0_2-63-157_20250123145648413.parquet.crc
            .hoodie_partition_metadata
            0189ebee-f2a4-44a9-b0d2-5b8be85d47cb-0_2-63-157_20250123145648413.parquet
        21
        22
```

Figure 12: Hudi repository.

3.2.3 Visualization and Monitoring of the Lakehouse

In the realm of industrial simulations, the data consumption layer serves as the critical endpoint where raw data is transformed into actionable insights. As the final module of our architecture, described in *Section 3.1*, it plays a pivotal role for assessing the operational status and resource utilization of edge devices within our simulation environment. Given the substantial complexity and volume of data processed, it is imperative to employ a visualization tool capable of real-time system observability and time-series analysis. Furthermore, the insights generated in this layer are essential for downstream users, including data analysts, system operators, and decision-makers, who rely on this information to efficiently interpret and act upon it, facilitating informed decision-making and continuous system improvement.

After a thorough evaluation of various visualization tools, *Grafana* was selected for its exceptional capabilities in real-time monitoring, time-series analytics, and system observability. Grafana's event-driven architecture adeptly captures immediate system fluctuations, rendering it highly suitable for overseeing critical industrial processes and assessing resource performance in dynamic settings. This differs from tools like *Amazon QuickSight*, which is primarily designed for static reporting and business intelligence insights.

Grafana facilitates the tracking of real-time sensor telemetry, resource availability, and system logs. By utilizing query-driven dashboards that dynamically update based on live data streams, users can monitor performance metrics, detect anomalies, and analyze historical trends in edge device activity. To enhance system oversight, a comprehensive monitoring dashboard was developed to track:

- **Edge device health and operational states over time** (Figure 13, left side). This feature provides continuous monitoring of each device's status, enabling prompt identification of issues.
- **Current available system resources** (Figure 13, right side). This offers the latest overview of CPU and memory utilization, assisting in the evaluation of resource allocation and load balancing.

- **Aggregated function costs over the last 24 hours** (Figure 14). Tracking workload trends helps optimize resource usage and improve overall system efficiency by identifying areas of high consumption.

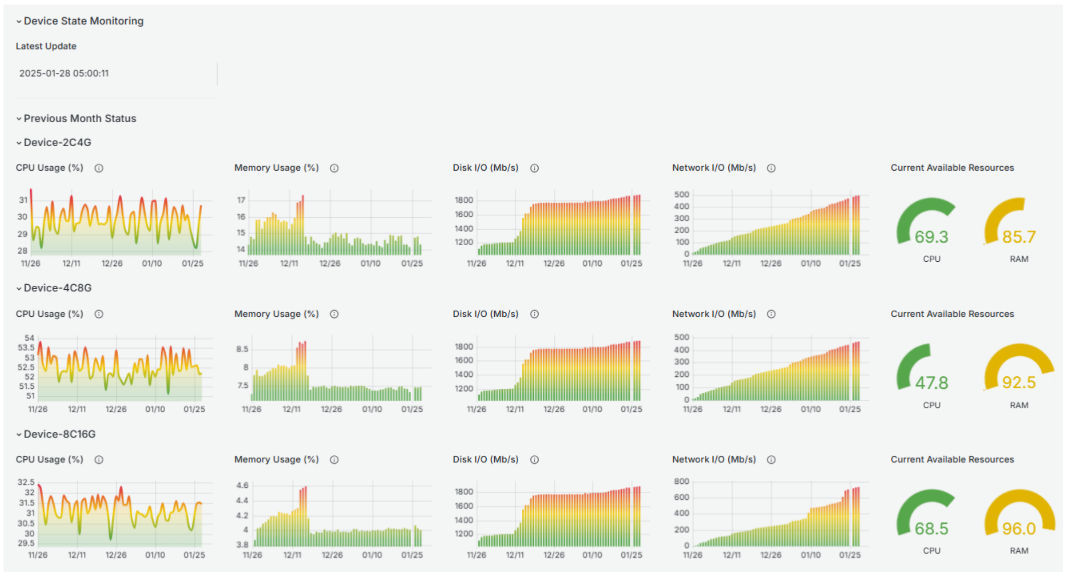


Figure 13: Grafana dashboard for resource states.

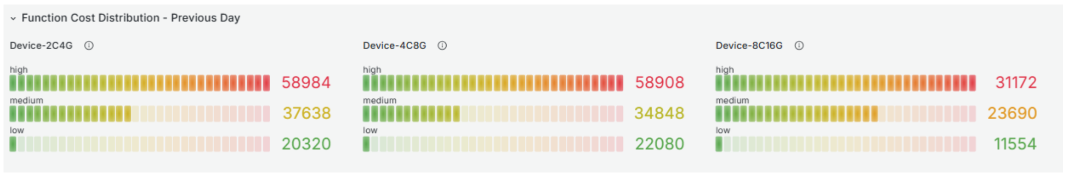


Figure 14: Grafana dashboard for workloads.

The construction of this dashboard leveraged *Amazon Managed Grafana*, with Amazon Athena serving as the query engine for data retrieval. This configuration enables direct interaction with both historical and real-time datasets, allowing for the analysis of device activity patterns and system resource fluctuations without the need for extensive pre-processing. Other than real-time monitoring, this framework promotes federated data collaboration, eliminating data silos. It enables cross-functional teams, including data engineers, analysts, and machine learning specialists, to access unified datasets, without unnecessary duplication or transformation across fragmented systems.

For instance, Figure 15 explains how data engineers can utilize *Apache Spark* for ETL processes, properly cleaning and storing data within the lakehouse. Simultaneously, analytics teams can query structured Iceberg tables via *Grafana*, generating dashboards and reports directly from the same dataset. Looking ahead, machine learning teams can employ *AWS SageMaker* to streamline the entire machine learning lifecycle, from data preprocessing and feature engineering to model training and deployment.

This study demonstrates how data lakehouses can integrate workflows across teams, breaking traditional data silos and fostering collaboration across departments. By adopting Apache Iceberg, our architecture provides scalability, consistency, and interoperability, making it a robust foundation for real-time observability, analytical insights, and future AI-driven applications.

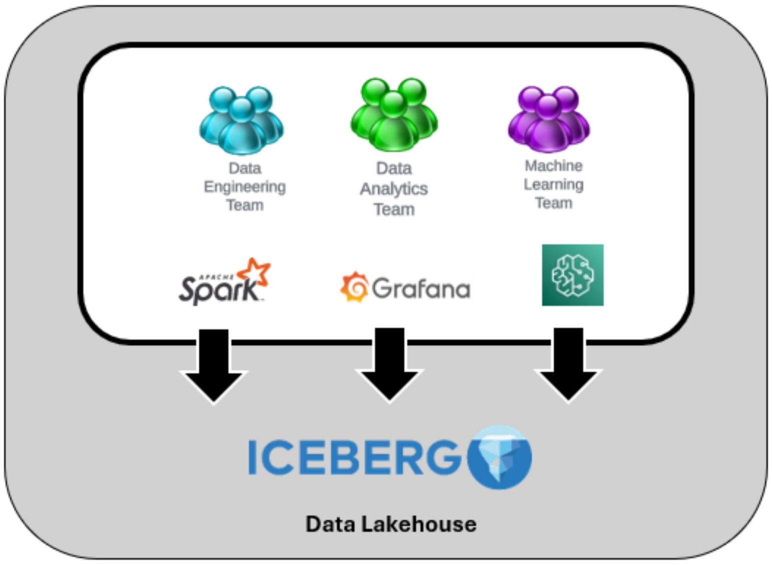


Figure 15: Example of lakehouse usage.

3.2.4 Implementing Infrastructure as Code (IaC) with AWS CDK

The adoption of Infrastructure as Code (IaC) represents a paradigm shift in cloud infrastructure management. With a declarative and programmatic provisioning of resources to enhance scalability, consistency, and automation, it defines infrastructure through code rather than manual configurations. IaC mitigates configuration drift, reduces human errors, and fosters repeatability across environments. This approach is particularly valuable in dynamic, data-intensive architectures, where resources must be efficiently orchestrated to support ingestion, processing, and analytics workflows.

In this research, AWS Cloud Development Kit (CDK) serves as the core IaC framework, allowing the seamless deployment and management of cloud components. This stack defines a structured access control mechanism for IoT devices, data processing pipelines, and visualization components, ensuring that security, automation, and operational efficiency are embedded into the system's architecture. Through AWS CDK, key infrastructure elements, including IAM roles and policies, S3 storage layers, AWS Glue jobs, Athena query permissions, and Grafana access controls, are provisioned in a modular, reusable, and maintainable manner. This implementation not only streamlines infrastructure deployment but also reinforces best practices in cloud automation, facilitating adaptability for future expansions and optimizations.

Infrastructure Components

a) IoT Role and Role Alias for Device Authentication

To enable secure authentication and interaction with AWS services, IoT devices leverage an IoT Role Alias (`DeviceMetricsRoleAlias`) as an intermediary. Devices authenticated via X.509 certificates assume the IAM role `RoleAliasRole`, which grants two key permissions: (1) write access to the S3 path `device-metric-raw-ingest-bucket/daily_raw_data/` for data ingestion, and (2) publish privileges to the MQTT topic `devicetocloud/function` for real-time messaging.

Device behavior is governed by the IoT policy `DeviceMetricsPolicy`, attached to each certificate. This policy enforces three core requirements: (1) devices must authenticate using their X.509 certificate to assume the role alias, (2) publish function cost metrics to the designated topic, and (3) maintain MQTT connectivity with AWS IoT Core.

Since all devices authenticate with a shared certificate, they assume the same IAM Role, `RoleAliasRole`, while eliminating the need for individual IAM credential management.

b) IoT Thing and MQTT Data Flow

An IoT Thing, `DeviceMetricsThing`, is provisioned to represent the logical entity for monitoring devices. An IoT Rule, `DeviceToS3`, processes MQTT messages received on `devicetocloud/function`, in raw form and storing them in Amazon S3 using a timestamp-based key structure as in *Appendix C*:

```
daily_raw_data/yyyy/MM/dd/device_id/device_id_timestamp.json
```

c) S3 Buckets for Data Storage

The stack utilizes two pre-existing S3 buckets with distinct purposes. The **Raw Data Storage** bucket maintains unprocessed device metrics, including both simulator-generated data and real-time IoT messages. In parallel, the **Processed Data Storage** bucket contains transformed and structured data after AWS Glue processing, optimized for analytical queries and visualization. Both buckets implement strict access policies, permitting only authorized IAM roles to perform read or write operations on their respective objects.

d) AWS Glue Role and Scheduled Processing

A dedicated IAM role manages AWS Glue job execution for structured processing of device metrics. This role provides two critical permissions: read access to raw data stored in S3 and write access to the processed data bucket for structured transformation.

The workflow is automated through a scheduled **Glue trigger** that executes at predefined intervals (e.g., *6:00 AM UTC*), systematically processing incoming telemetry, optimizing data formats, and ensuring reliable periodic ingestion of raw device metrics for downstream analysis.

e) Grafana IAM Role for Data Visualization

A dedicated Grafana IAM Role facilitates visualization of processed device metrics. The role grants three key permissions: (1) querying AWS Athena to retrieve structured insights, (2) reading AWS Glue metadata for schema definitions, and (3) accessing S3 storage to retrieve Athena query results. This configuration enables Grafana to efficiently retrieve and visualize device performance metrics while maintaining proper access control boundaries.

3.2.5 Experiments: Performance and Feature Evaluation for Data Lake Management

To determine the optimal solution for our data lake architecture, we conducted a comprehensive evaluation of Apache Iceberg and Hudi, focusing on critical operational and performance metrics. Our experimental methodology centered on measuring elapsed time, e.g. the total duration required to process ingested datasets from initial unpack operations through final reporter operations, providing complete end-to-end processing visibility.

Our experiments included three key evaluation dimensions:

- **Performance benchmarking**. Comparing daily ingestion batch processing across both frameworks, with particular focus on write throughput and efficiency across different time intervals
- **Concurrency validation**. Stress-testing control mechanisms under high parallelism through simulated concurrent insertions, deletions, and reads to evaluate transactional robustness and isolation guarantees
- **Feature analysis**. Examination of data rollback capabilities, versioning, and schema evolution patterns

The performance evaluation will reveal processing advantages, while feature testing will assess the system's flexibility in accomodating changes without incurring the overhead of costly table rewrites. Notably, the versioning analysis will demonstrate the platform's ability to maintain data integrity and support recovery in dynamic data environments. Additionally, schema evolution testing will further provide critical insights into structural compatibility as analytical workloads evolve.

Based on these multidimensional findings, encompassing both quantitative performance metrics and qualitative feature assessments, these results not only underscore its maturity for production workloads but also offer practical guidance for designing maintainable, high-performance data lakes capable of adapting to shifting business needs.

Chapter 4

Results of Local Experiments

As industrial simulation systems grow in complexity, managing dependencies, configurations, and execution environments presents several challenges. Containerization has emerged as a critical solution, providing portable, self-contained units that encapsulate applications with all their dependencies. Unlike traditional monolithic deployment environments, containers share the host operating system's kernel while maintaining isolation, achieving superior resource efficiency without sacrificing portability across diverse environments.

This chapter presents experimental results from our containerized industrial simulation framework, which establishes a standardized execution environment independent of underlying hardware or cloud infrastructure. This approach strengthens scalability, reproducibility, and deployment flexibility, supporting different configurations and data processing workflows.

4.1 Containerization of the Industrial Simulator

To streamline this process, we implemented ***Docker***, a leading containerization platform, packages, distributes, and executes applications within lightweight, isolated environments known as containers. We encapsulated all core services of the industrial simulator, including the simulation engine, Jupyter notebooks, Apache Iceberg, Apache Spark, REST services for Iceberg catalog and metadata management, and Grafana for visualization. This encapsulation established a controlled, reproducible testing environment, mitigating the risks associated with executing ETL jobs directly in production, which could otherwise inccur excessive costs.

By deploying our experimental workflows within this containerized setup, we ensured robust unit testing, a fundamental best practice when handling Big Data workloads. This approach allowed for iterative testing and refinement without compromising production stability, reinforcing the reliability and efficiency of our data processing pipeline.

4.2 Experiments

4.2.1 Daily Ingestion Batch

Once we established the lakehouse architecture, our study focused on identifying efficient strategies for daily batch ingestion. We opted for an append-only strategy due to its simplicity, performance benefits, and compatibility with advanced features such as time travel and scalable metadata. Unlike overwrite-heavy ingestion, which complicates historical tracking, append-only workflows preserve data lineage by continuously adding records into partitions (e.g., based on ingestion time) while allowing retention policies to manage snapshot deletions without compromising historical integrity.

Compared to overwrite-based approaches, append-only workflows are particularly effective for intermediate or staging datasets, while overwrites are best suited for final dimensions or fact tables. Even in cases requiring frequent updates, some engineers prefer periodically wiping and re-ingesting smaller tables rather than maintaining complex update workflows (LocationOld2728, 2024). This approach simplifies data management, improves ingestion efficiency, and reduces operational complexity.

Both Apache Iceberg and Apache Hudi support time travel, enabling datasets to be reconstructed at any given point through historical snapshot queries. Overwrite operations, on the other hand, erase prior states unless explicitly versioned. While they are useful for full table refreshes, they are less effective for preserving incremental changes or intermediate states. These operations are useful for full table refreshes; however, they are less effective when the goal is to preserve incremental changes or intermediate states.

To fully grasp the differences between **Copy-on-Write (CoW)** and **Merge-on-Read (MoR)** tables, ingestion workflows must go beyond simple appends. Hybrid strategies may emerge, requiring specific configurations in table properties such as `write.merge.mode`, `write.delete.mode`, and `write.update.mode`, each of which can be set to *CoW* or *MoR*, with **CoW** as the default. Proper configuration of these parameters is essential for ensuring consistency in table behavior. Although hybrid workflows offer flexibility, our experiments were conducted exclusively on pure tables of each type, to derive clear insights for optimizing our architecture.

To evaluate the performance of append-based ingestion, we analyzed different strategies in Hudi to determine the most suitable approach. As shown in Table 5, *bulk insert* significantly improves ingestion speed compared to *upserts*, particularly in *Merge-On-Read* mode, where it achieves the fastest ingestion times. To further assess pure insertion performance, we then compared Apache Iceberg's *append* mode with Hudi's *bulk insert* operation.

Strategy	**Upsert Time (s)**	**Bulk Insert Time (s)**
Hudi CoW	26.21	14.47
Hudi MoR	15.05	5.83

Table 5: Comparison between upsert and bulk insert in Apache Hudi.

In Hudi, *bulk insert* optimizes batch ingestion by appending new data without triggering record-level indexing or compaction, significantly improving ingestion speed while preserving metadata efficiency. Unlike *upserts*, which introduce additional overhead due to record matching and file rewriting, bulk insert operates similarly to Iceberg's *append* mode. Iceberg achieves this through lightweight updates, utilizing snapshots and manifests to separate metadata from actual data. This fundamental similarity makes *bulk insert* a suitable benchmark for evaluating pure insertion performance.

4.2.1.1 Performance Evaluation: Iceberg vs. Hudi

During the evaluation of various systems over the course of a day, we measured and averaged ingestion and query times, and obtained the results presented in Table 6. Our findings indicate that Iceberg consistently outperforms Hudi in both ingestion and read efficiency, particularly in *CoW* mode. To establish a fair comparison, we used `bulk_insert` in Hudi to match Iceberg's append-based ingestion. While Hudi demonstrated faster ingestion times in *MoR* mode, which aligns with its known strengths in efficient bulk inserts and upsert operations, Iceberg still provided superior overall performance across ingestion, query execution, and metadata scalability.

This makes Iceberg the preferred choice for batch data workloads, where rapid updates or real-time capabilities are not a priority, as it minimizes ingestion latency while maintaining consistent read performance and efficient metadata management. In contrast, Apache Hudi, while well-suited for real-time use cases, introduces additional overhead in batch ingestion workflows. Given these advantages, Iceberg was selected for this study to ensure optimal performance, scalability, and long-term maintainability in our data lake architecture.

Strategy	**Ingestion Time (s)**	**Read Time (s)**
Iceberg CoW	11.25	0.16
Hudi CoW	14.47	1.07
Iceberg MoR	9.60	0.20
Hudi MoR	5.83	1.22

Table 6: Storage Strategy Comparison: Apache Iceberg vs. Hudi.

4.2.2 Optimistic Concurrency Control (OCC) in Iceberg

Following the ingestion paradigm, the next research question focused on evaluating how **Optimistic Concurrency Control (OCC)** handles concurrent write operations in AWS-based Apache Iceberg lakehouses. OCC is essential for maintaining data consistency and transactional integrity in multi-writer environments, particularly when handling *Copy-on-Write* (*CoW*) and *Merge-on-Read* (*MoR*) strategies. Unlike traditional distributed locking mechanisms, Iceberg relies on snapshot-based validation to detect and resolve conflicts at commit time.

For this implementation, *AWS Glue Data Catalog* serves as the metadata repository, tracking table snapshots and schema versions. The OCC mechanism follows three key steps:

- **Commit Validation**. When a write operation initiates, Iceberg assesses whether the table metadata has changed since the transaction started.
- **Conflict Detection**. If conflicting modifications are detected (e.g., another process has committed a newer snapshot), the transaction is rejected to prevent metadata inconsistencies.
- **Retry with Latest State**. The write operation restarts using the most recent snapshot state to align with updated table metadata.

This study evaluates concurrent insertions and deletions in *CoW* and *MoR* tables, simulating multiple Apache Spark contexts, each representing independent writers and readers operating simultaneously. *CoW* prioritizes structured, pre-processed data but introduces higher ingestion overhead, whereas *MoR* defers deletions to query time, reducing initial write latency. These trade-offs influence system behavior in environments that require transactional integrity under concurrent workloads.

Concurrency Control Mechanism

Apache Iceberg handles concurrent writes through **Optimistic Concurrency Control (OCC)**, replacing traditional locking mechanisms with snapshot-based validation. Commit integrity depends on verifying whether metadata has changed since the transaction began. If conflicting modifications are detected, e.g., an outdated snapshot being referenced, the transaction is rejected, and the process must retry with the latest snapshot state.

Instead of native distributed locks, Iceberg integrates snapshot versioning and optimistic concurrency tracking to uphold consistency during concurrent operations. Table 7 illustrates OCC behavior as follows:

1. *Snapshot B* and *D* are successfully committed and included in the table.
2. *Snapshot C* fails validation at commit time due to outdated metadata and is discarded as an **orphaned snapshot**.
3. *Snapshot D* commits after retrying with the most recent table state and is added to the table's history.

Orphaned snapshots represent commits no longer referenced in table metadata, often resulting from failed transactions, aborted processes, or manual cleanup. These unreferenced snapshots accumulate in storage until explicitly removed through metadata compaction or retention policies, potentially increasing storage costs and query overhead.

Steps	**Writer 1**	**Writer 2**
Read Snapshot	Read *Snapshot A*	Reads *Snapshot A*
Write Changes	Generate *Snapshot B*	Generate *Snapshot C*
Commit Attempts	Commits *Snapshot B* (update metadata catalog)	Fails because the catalog now points to *Snapshot B*
Retry	–	Re-reads the latest state (*Snapshot B*), applies changes, generates *Snapshot D*, and commits.

Table 7: Optimistic Concurrency Control (OCC) behavior.

In Iceberg, OCC combined with snapshot versioning and partition tracking, reinforces transactional stability in AWS-based lakehouses. This framework adjusts write operations under CoW and MoR conditions, maintaining controlled ingestion workflows and preserving metadata consistency in industrial data pipelines.

4.2.2.1 Parallel Insertions and Reads

The configuration of three writers and two readers was selected as it offers an optimal balance for evaluating performance. It introduces sufficient write concurrency to expose rewrite overhead without inducing excessive contention.

At the same time, it enables concurrent read evaluation under realistic workload conditions. Increasing writer concurrency degraded performance due to higher I/O and lock contention; reducing the number of readers, on the other hand, limited the assessment of read latency under load.

a) Performance of Copy-on-Write Tables

In the *CoW* model, new insertions require full file rewrites, leading to higher insertion latency due to frequent modifications. With three writers inserting two rows each, the average total insertion time for six rows reached **21.42 seconds**, exceeding the average read time of **26.85 seconds**. This outcome highlights the performance trade-off in *CoW*, where each modification rewrites the entire file, increasing computational overhead. The insertion of two rows per writer was chosen to simulate light, yet frequent, data updates typical of near real-time ingestion scenarios.

- Each Spark writer process experienced execution times between **20.17 seconds** and **22.12 seconds**, indicating the cost of full file rewrites.
- Spark reader processes completed queries in **26.82 seconds** to **26.87 seconds**, demonstrating how pre-optimized storage structures in *CoW* improve read performance.

In a multi-writer scenario, as shown in Table 8, the overhead from repeated file rewrites increases contention and reduces insertion throughput, negatively affecting real-time ingestion performance.

Ingestion Times			
Process ID	**Row 1 Time (s)**	**Row 2 Time (s)**	**Total Time (s)**
0	**17.87**	**2.31**	**20.17**
1	**18.42**	**3.54**	**21.96**
2	**19.04**	**3.08**	**22.12**

Read Times				
Reader ID	**1st Read (s)**	**2nd Read (s)**	**3rd Read (s)**	**Total Time (s)**
0	**20.05**	**3.68**	**3.09**	**26.82**
1	**20.17**	**3.60**	**3.10**	**26.87**

Table 8: Concurrent ingestion and read times in Apache Iceberg CoW tables.

b) Performance of Merge-on-Read Tables

For *MoR*, writes are optimized by storing modifications in delta logs, with merges deferred to later stages. Testing across multiple Spark contexts revealed an average total insertion time of **18.57 seconds** for six rows, being faster than

CoW for our data volume. However, the deferred merges increased the average read time to **31.95 seconds**, introducing additional overhead.

- Individual Spark writer processes completed inserts within **17.50 to 19.25 seconds**, showing that incremental updates improve write efficiency.
- Reader queries required **31.86 to 32.04 seconds**, as they involved merging base data with delta logs in real-time.

As evidenced in Table 9, *MoR* improves ingestion latency in concurrent workloads, making it more suitable for high-frequency updates. Yet, without scheduled compaction, accumulating delta logs can increase query complexity and degrade performance over time.

Ingestion Times			
Process ID	**Row 1 Time (s)**	**Row 2 Time (s)**	**Total Time (s)**
0	**16.00**	**2.95**	**18.95**
1	**16.43**	**2.81**	**19.25**
2	**15.75**	**1.76**	**17.50**

Read Times				
Reader ID	**1st Read (s)**	**2nd Read (s)**	**3rd Read (s)**	**Total Time (s)**
0	**23.94**	**4.65**	**3.45**	**32.04**
1	**23.83**	**4.60**	**3.43**	**31.86**

Table 9: Concurrent ingestion and read times in Apache Iceberg MoR tables.

4.2.2.1 Parallel Deletions and Reads

In this evaluation, three concurrent processes each attempted to delete three rows simultaneously. This setup provided a controlled yet realistic level of parallelism. Higher degrees of concurrency led to increased I/O pressure and locking overhead, whereas fewer processes limited its visibility. Thus, this configuration was considered optimal for assessing deletion performance under constrained system resources.

a) Performance of Copy-on-Write Tables

CoW tables handle deletions by rewriting entire data files without the removed rows. Rather than maintaining separate delete files, the system replaces affected data files entirely, ensuring that only clean, updated versions remain. This can be found in the metadata summary where `deleted-data-files` and `added-data-files` match, indicating that an entire file was rewritten. While this approach provides faster query performance by eliminating the need to merge delete markers at read time, it comes at the cost of higher write latency and increased I/O operations.

In scenarios where multiple processes attempt to perform deletions concurrently in *CoW* tables with Apache Iceberg, one process typically succeeds in committing its changes first. This occurs due to the race condition inherent in concurrent writes. To mitigate such race conditions in concurrent environments, proper locking mechanisms, snapshot isolation, or conflict resolution strategies should be implemented. *Optimistic concurrency control (OCC)* and strategic partitioning can also help minimize contention when handling high-frequency delete operations in Iceberg *CoW* tables.

Once a process successfully commits its deletion, the Iceberg table metadata is updated, making the intended row inaccessible to other concurrent processes. As a result, when the remaining processes attempt to delete the same row, they fail because the row no longer exists within the metadata scope. Consequently, their transactions are aborted due to an inability to locate the row. This behavior is particularly relevant in *CoW* tables, where deletions involve rewriting entire data files instead of marking records for deletion via delete files (as in *Merge-on-Read* tables).

When multiple delete operations overlap, the first committed transaction determines the valid snapshot, causing subsequent transactions operating on outdated metadata to fail at commit time. Table 10 illustrates this behavior, showing that only three rows were successfully deleted while the rest were aborted. This highlights the importance of transactional consistency in *CoW* table operations, ensuring that competing writes or deletions do not interfere with each other in an unpredictable manner.

Impact of Window Operations and Snapshot Management in Iceberg

In a distributed environment like *Apache Spark*, executing window operations without explicitly defining a partition leads to suboptimal data redistribution. Without a partition column specified in the window function, Spark groups all records into a single logical partition, resulting in sequential processing on a single node and, consequently, significant performance degradation. However, this behavior does not alter the underlying partitioning structure of the *DataFrame*; it merely bypasses its utilization for the specific window operation.

We noticed a slight deterioration in performance at this stage. Unexpectedly, performance was poorer when indexing strategies like partitioning were used than when they were not. The non-partitioned table outperformed the partitioned one in *CoW* tables. Given this, integrating automatic snapshot expiration and metadata cleanup into data lifecycle management is essential. These processes help minimize storage overhead, enhance query performance, and streamline rollback operations when needed. While Iceberg provides built-in snapshot retention policies to automate expiration based on workload patterns, we recommend the following strategies to further optimize data man-

Non-Partitioned Table		
Process ID	**Deletion Time (s)**	**Deletion Status**
1	**12.40**	Deleted
2	–	Aborted
0	–	Aborted
1	**2.70**	Deleted
0	–	Aborted
2	–	Aborted
1	**2.34**	Deleted
0	–	Aborted
2	–	Aborted

Partitioned Table		
Process ID	**Deletion Time (s)**	**Deletion Status**
1	**13.83**	Deleted
2	–	Aborted
0	–	Aborted
1	**1.67**	Deleted
0	–	Aborted
2	–	Aborted
1	**1.63**	Deleted
0	–	Aborted
2	–	Aborted

Table 10: Concurrent deletion times in non-partitioned vs. partitioned Iceberg CoW tables.

agement:

- **Time-based expiration:** Retaining snapshots for a fixed period (e.g., 7–30 days in high-ingestion environments) prevents excessive metadata accumulation.
- **Count-based expiration:** Limiting the number of retained snapshots (e.g., keeping only the last 10–20 snapshots) balances rollback flexibility and storage efficiency.

Even after snapshot expiration, metadata files that are no longer referenced (e.g., orphaned manifest files) can persist in storage. To fully remove these unnecessary files and maintain a lean metadata structure, Iceberg provides an *orphan file cleanup* mechanism.

Regularly performing these tasks keeps Iceberg-managed tables optimized, scalable, and cost-effective in dynamic data lakehouse environments. Nevertheless, after implementing these measures, the impact on deletion was minimal. The average execution time decreased by only **1.72%**, from **5.81 seconds**, in non-partitioned tables, to **5.71 seconds**, in partitioned tables (as shown in Table 10).

Optimizing Data Distribution with Hash-Based Writes

To consolidate the correct selection of a unique row, the configuration `write.distributed.mode = hash` was set to distribute write operations in Iceberg using a hash-based strategy with a primary composite key of `timestamp` and `device_id`. Instead of relying on *Spark's* default shuffle mechanism, this approach assigns records to writers based on a hash function applied to specific columns, such as primary keys or partition keys. This balances data distribution across writers, preventing partition overload and minimizing skew.

Optimized Partitioned CoW Table		
Process ID	**Deletion Time (s)**	**Status**
1	**11.82**	Deleted
2	**11.35**	Deleted
0	–	Aborted
1	**3.17**	Deleted
0	**3.06**	Deleted
2	–	Aborted
1	**3.06**	Deleted
0	**3.20**	Deleted
2	–	Aborted

Table 11: Concurrent deletion times in non-partitioned CoW tables.

In multi-writer environments, concurrent processes may attempt to modify the same partition, leading to conflicts and commit failures. *Hash-based distribution* mitigates this issue by evenly spreading the workload, reducing contention, and lowering the probability of multiple writers modifying the same files simultaneously. This strategy improves consistency and optimizes file layout for future query performance. Table 11 also demonstrated better execution efficiency, successfully allowing another writer to perform concurrent deletions without conflicts.

Impact of Compaction on Read Performance

Applying the compaction strategies previously discussed resulted in significant improvements in query performance. By reducing metadata fragmentation and optimizing data layout, query execution times were notably reduced. The results in Table 12 demonstrate that after compaction, total read times improved from an average of **25.18 seconds** to **13.59 seconds**, representing a **46.02%** reduction in query latency and enhancing overall efficiency.

Read Performance Before Compaction				
Reader ID	**1st Read (s)**	**2nd Read (s)**	**3rd Read (s)**	**Total Time (s)**
0	**18.68**	**3.63**	**2.89**	**25.20**
1	**18.64**	**3.58**	**2.93**	**25.15**

Read Performance After Compaction				
Reader ID	**1st Read (s)**	**2nd Read (s)**	**3rd Read (s)**	**Total Time (s)**
0	**12.07**	**0.65**	**0.45**	**13.17**
1	**12.34**	**1.05**	**0.62**	**14.01**

Table 12: Read performance before vs. after compaction in CoW Tables.

b) Performance of Merge-on-Read Tables

Non-Partitioned MoR Table		
Process ID	**Deletion Time (s)**	**Status**
0	9.61	Deleted
2	9.68	Deleted
1	8.86	Deleted
0	4.67	Deleted
1	4.42	Deleted
2	4.28	Deleted
2	2.32	Deleted
1	1.51	Deleted
0	1.46	Deleted

Partitioned MoR Table		
Process ID	**Deletion Time (s)**	**Status**
2	7.72	Deleted
1	7.06	Deleted
0	7.26	Deleted
1	1.77	Deleted
2	2.36	Deleted
0	2.62	Deleted
1	1.72	Deleted
2	1.69	Deleted
0	1.43	Deleted

Table 13: Concurrent deletion times in non-partitioned vs. partitioned Iceberg MoR tables.

In *MoR* tables, deleted rows are not immediately removed from the dataset. Instead, delete markers are stored separately in delete files, which track the rows to be ignored at query time. These delete markers, known as *position deletes*, reference the exact file path and row index, ensuring that deleted records are dynamically excluded when data is read. The metadata summary reflects this process with fields like `added-position-delete-files`, `total-delete-files`, and `total-position-deletes`, indicating that deletions accumulate until a compaction process merges them with the base data files. This method improves write performance by avoiding immediate file rewrites but requires periodic compaction to maintain query efficiency.

Table 13 collects the average deletion times for *Merge-on-Read* tables in Apache Iceberg. Without using partitions, the deletion operations took an average of **5.21 seconds**, whereas partitioned tables achieved a lower average of **3.74 seconds**. This reduction highlights the efficiency gained by leveraging partitioning, as it minimizes file scanning and improves deletion execution times.

Impact of Compaction on Read Performance

During this evaluation, we also observed a notable decline in the first iteration in query performance in *MoR* tables, which increased from **31.95 seconds** on average to **121.98 seconds**. However, performance improved significantly when a compaction job was conducted, rewriting data and removing orphaned files, thereby reducing the average execution time to **9.50 seconds**. This optimization not only mitigated query overhead but also allowed *Merge-on-Read* tables to surpass *Copy-on-Write* tables in query efficiency.

Read Performance Before Compaction				
Reader ID	**1st Read (s)**	**2nd Read (s)**	**3rd Read (s)**	**Total Time (s)**
0	**56.50**	**33.02**	**32.07**	**121.59**
1	**56.29**	**33.40**	**32.68**	**122.37**

Read Performance After Compaction				
Reader ID	**1st Read (s)**	**2nd Read (s)**	**3rd Read (s)**	**Total Time (s)**
0	**8.85**	**0.31**	**0.29**	**9.45**
1	**8.87**	**0.32**	**0.36**	**9.55**

Table 14: Concurrent read times in Apache Iceberg in MoR tables before and after compaction.

The read performance before and after compaction in Apache Iceberg's *MoR* tables is presented in Table 14. Prior to compaction, the total read time averaged **121.98 seconds**, with each query incurring substantial overhead due to the on-the-fly merging of delta logs. After compaction, the total read time significantly improved to **9.50 seconds**, demonstrating a **12.8x** speedup. This result highlights the critical role of compaction in reducing query overhead by consolidating incremental updates into optimized storage structures.

4.2.3 Data rollback and Versioning

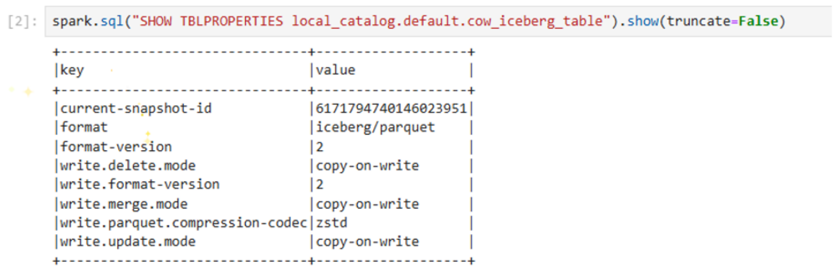
Another experiment conducted focused on Apache Iceberg's built-in rollback and snapshot versioning capabilities. These features eliminate the need for manual recovery efforts or expensive third-party backup solutions, ensuring data integrity with minimal operational overhead.

Data pipelines are inherently vulnerable to errors or failures that can compromise data integrity. A common scenario occurs when incorrect transformations are applied, leading to unintended data loss. For instance, consider a situation where an analyst mistakenly executed a transformation that deleted data for *December 11, 2024*. This error disrupted downstream analytics, making immediate recovery essential to restore data integrity. Instead of resorting to complex migrations or manually re-ingesting historical data, Apache Iceberg's rollback functionality enabled a swift restoration.

Resolution Process

To resolve the issue, the following steps were taken:

1. Before initiating the rollback, we verified the snapshot ID corresponding to the last consistent table state we aimed to restore, as illustrated in Figure 16.

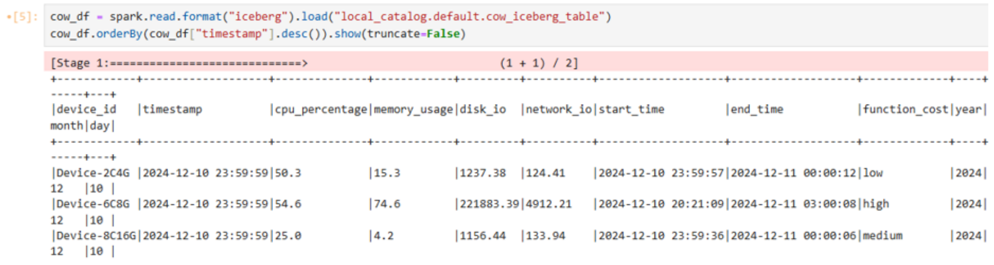


```
[2]: spark.sql("SHOW TBLPROPERTIES local_catalog.default.cow_iceberg_table").show(truncate=False)

+-------------------------------+-------------------+
|key                            |value              |
+-------------------------------+-------------------+
|current-snapshot-id            |6171794740146023951|
|format                         |iceberg/parquet    |
|format-version                 |2                  |
|write.delete.mode              |copy-on-write      |
|write.format-version           |2                  |
|write.merge.mode               |copy-on-write      |
|write.parquet.compression-codec|zstd               |
|write.update.mode              |copy-on-write      |
+-------------------------------+-------------------+
```

Figure 16: Snapshot ID verification before rollback.

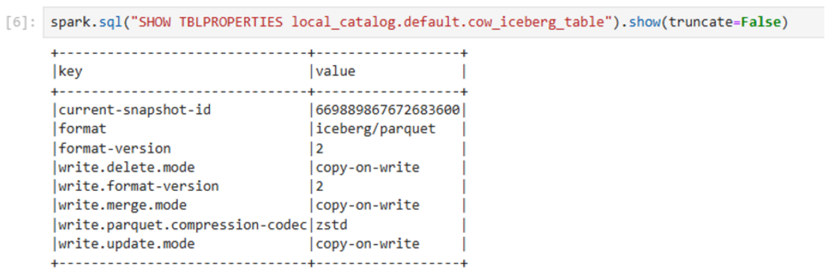
To simulate data loss, we deleted the affected day's records and examined the latest table state, as shown in Figure 17. Following the deletion, the current snapshot ID changed, capturing the table's latest state after the error, as illustrated in Figure 18.



```
•[5]: cow_df = spark.read.format("iceberg").load("local_catalog.default.cow_iceberg_table")
      cow_df.orderBy(cow_df["timestamp"].desc()).show(truncate=False)

[Stage 1:============================>                              (1 + 1) / 2]
+------------+-------------------+--------------+------------+---------+----------+-------------------+-------------------+-------------+----+
-----+---+
|device_id   |timestamp          |cpu_percentage|memory_usage|disk_io  |network_io|start_time         |end_time           |function_cost|year|
month|day|
+------------+-------------------+--------------+------------+---------+----------+-------------------+-------------------+-------------+----+
-----+---+
|Device-2C4G |2024-12-10 23:59:59|50.3          |15.3        |1237.38  |124.41    |2024-12-10 23:59:57|2024-12-11 00:00:12|low          |2024|
12   |10 |
|Device-6C8G |2024-12-10 23:59:59|54.6          |74.6        |221883.39|4912.21   |2024-12-10 20:21:09|2024-12-11 03:00:08|high         |2024|
12   |10 |
|Device-8C16G|2024-12-10 23:59:59|25.0          |4.2         |1156.44  |133.94    |2024-12-10 23:59:36|2024-12-11 00:00:06|medium       |2024|
12   |10 |
```

Figure 17: Table state after data deletion.



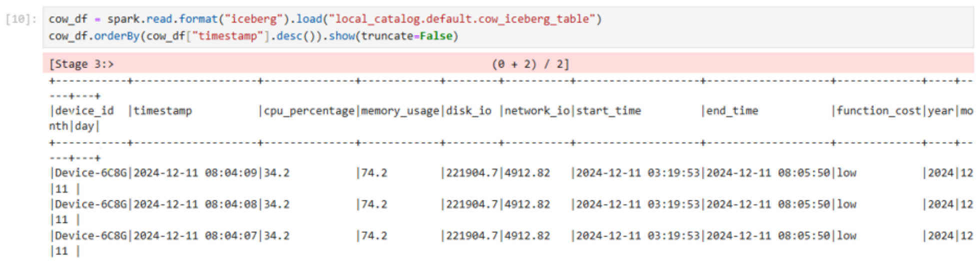
```
[6]: spark.sql("SHOW TBLPROPERTIES local_catalog.default.cow_iceberg_table").show(truncate=False)

+-------------------------------+------------------+
|key                            |value             |
+-------------------------------+------------------+
|current-snapshot-id            |669889867672683600|
|format                         |iceberg/parquet   |
|format-version                 |2                 |
|write.delete.mode              |copy-on-write     |
|write.format-version           |2                 |
|write.merge.mode               |copy-on-write     |
|write.parquet.compression-codec|zstd              |
|write.update.mode              |copy-on-write     |
+-------------------------------+------------------+
```

Figure 18: Updated snapshot ID after data deletion.

2. To restore the table, we executed Iceberg's rollback command, reverting the table to the state of a previous snapshot without relying on external backups. A `<snapshot_id>` parameter must be added, corresponding to the identifier of the snapshot captured prior to the erroneous transformation.

3. After executing the rollback, we queried the table again to confirm that the deleted data was successfully restored. Figure 19 verifies that the table was reverted.



```
[10]: cow_df = spark.read.format("iceberg").load("local_catalog.default.cow_iceberg_table")
      cow_df.orderBy(cow_df["timestamp"].desc()).show(truncate=False)

[Stage 3:>                                                          (0 + 2) / 2]
+-----------+-------------------+--------------+------------+----------+----------+-------------------+-------------------+-------------+----+--
---+---+
|device_id  |timestamp          |cpu_percentage|memory_usage|disk_io   |network_io|start_time         |end_time           |function_cost|year|mo
nth|day|
+-----------+-------------------+--------------+------------+----------+----------+-------------------+-------------------+-------------+----+--
---+---+
|Device-6C8G|2024-12-11 08:04:09|34.2          |74.2        |221904.7|4912.82   |2024-12-11 03:19:53|2024-12-11 08:05:50|low          |2024|12
|11 |
|Device-6C8G|2024-12-11 08:04:08|34.2          |74.2        |221904.7|4912.82   |2024-12-11 03:19:53|2024-12-11 08:05:50|low          |2024|12
|11 |
|Device-6C8G|2024-12-11 08:04:07|34.2          |74.2        |221904.7|4912.82   |2024-12-11 03:19:53|2024-12-11 08:05:50|low          |2024|12
|11 |
```

Figure 19: Verification of restored data after rollback.

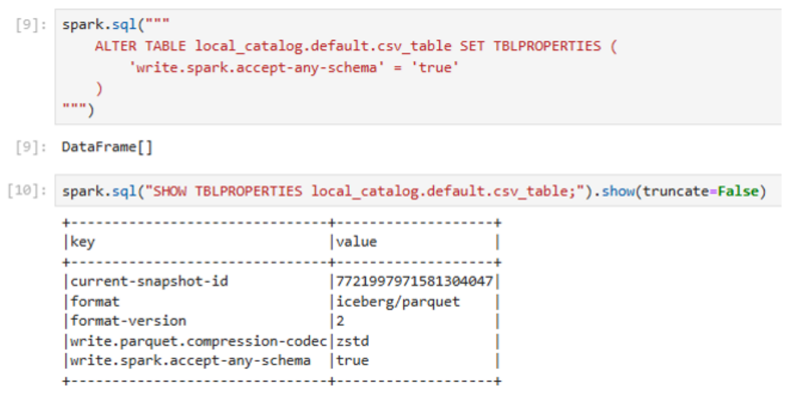
In general, Apache Iceberg reduces downtime and preserves data consistency with minimal disruption to business operations. The restored data matched its original state without relying on external backups, ensuring integrity. The ability to pinpoint and restore a specific snapshot significantly decreased recovery time and the need for manual intervention.

4.2.4 Schema Evolution

To enable flexible schema evolution when writing data with different schemas into a target table, Apache Iceberg provides configurable table properties that facilitate schema modifications while maintaining data integrity.

1. **Enabling Schema Evolution in the Target Table.** In order to prevent failures due to schema changes, the property `write.spark.accept-any-schema` grants schema evolution at runtime, as depicted in Figure 20.

- This property is only required on the target table because the source table is read-only and does not undergo schema modifications.
- When querying old data after a schema change, new columns appear as NULL in pre-existing records, maintaining backward compatibility.



```
[9]: spark.sql("""
         ALTER TABLE local_catalog.default.csv_table SET TBLPROPERTIES (
             'write.spark.accept-any-schema' = 'true'
         )
     """)

[9]: DataFrame[]

[10]: spark.sql("SHOW TBLPROPERTIES local_catalog.default.csv_table;").show(truncate=False)

+-------------------------------+------------------+
|key                            |value             |
+-------------------------------+------------------+
|current-snapshot-id            |7721997971581304047|
|format                         |iceberg/parquet   |
|format-version                 |2                 |
|write.parquet.compression-codec|zstd              |
|write.spark.accept-any-schema  |true              |
+-------------------------------+------------------+
```

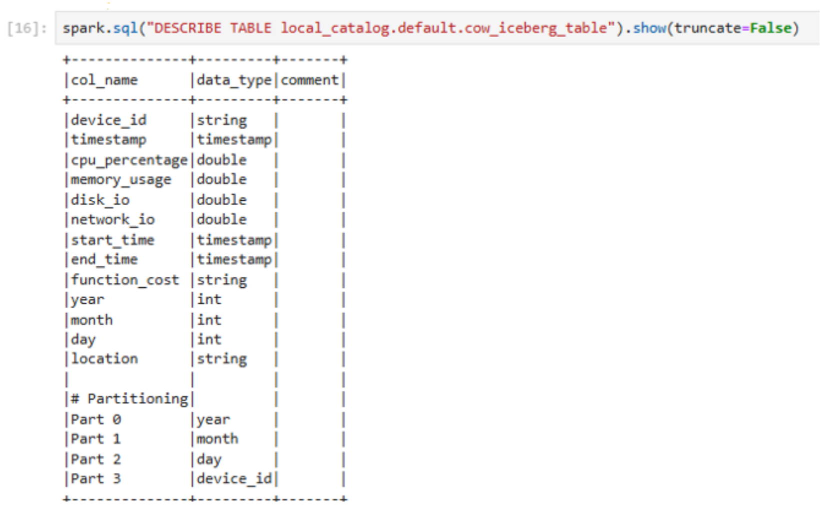
Figure 20: Enabling schema evolution.

2. **Changing Partitioning Strategies.** Partitioning modifications adhere to specific rules:

   - Using `ALTER TABLE ... SET PARTITION SPEC` only applies the new partitioning to newly written data.
   - Queries on old data remain functional, but performance may decrease due to mixed partitioning schemes.
   - To fully apply the new partitioning across all data, historical files must be rewritten using `REWRITE DATA`.

3. **Other Schema Modifications in Iceberg.** Iceberg provides schema evolution capabilities without requiring full table rewrites:

   - **Renaming Columns** is supported without modifying physical data, as Iceberg tracks column IDs instead of column positions.
   - **Dropping Columns** removes the column from metadata but does not modify historical files unless explicitly rewritten.



```
[16]: spark.sql("DESCRIBE TABLE local_catalog.default.cow_iceberg_table").show(truncate=False)
```

```
+--------------+---------+-------+
|col_name      |data_type|comment|
+--------------+---------+-------+
|device_id     |string   |       |
|timestamp     |timestamp|       |
|cpu_percentage|double   |       |
|memory_usage  |double   |       |
|disk_io       |double   |       |
|network_io    |double   |       |
|start_time    |timestamp|       |
|end_time      |timestamp|       |
|function_cost |string   |       |
|year          |int      |       |
|month         |int      |       |
|day           |int      |       |
|location      |string   |       |
|              |         |       |
|# Partitioning|         |       |
|Part 0        |year     |       |
|Part 1        |month    |       |
|Part 2        |day      |       |
|Part 3        |device_id|       |
+--------------+---------+-------+
```

Figure 21: Successful addition of the new column `location`.

Finally, we attempted to add a new column, `location`, which was successfully added, as shown in Figure 21.

Chapter 6

Discussion

This project contributes to the design and optimization of a cloud-based transactional data lake for evolving data models by implementing a monitoring simulator built on an Iceberg architecture. The primary goal was to evaluate edge device performance while enabling scalable, high-frequency metric tracking. The setup successfully demonstrated ingestion strategies, system responsiveness, and query behavior under streaming-like workloads. However, the use of a synthetic dataset, designed for controlled benchmarking, limited the complexity and variability typically present in real-world telemetry data. While an ETL pipeline was introduced to ensure clean, schema-consistent data by forward-filling missing values and addressing partial records, and schema drift. As a result, while the simulator's findings are meaningful within their test scope, caution is needed when extrapolating to production-scale environments where data heterogeneity and system volatility are more prominent.

A key challenge during development was balancing simulation control with production realism. Although the data structure was partially synthetic, the metrics were generated by actual virtual machines emulating edge devices. This added a layer of realism, particularly in capturing real system metrics such as CPU and memory usage under varying conditions. However, the scope of metadata remained limited. Only device IDs were available as categorical variables, while other useful contexts, such as whether a device was idle, heavy load, or belonged to a certain class, were absent. This restricted the ability to test metadata filtering, more complex schema evolution scenarios, and workload segmentation. Future versions of the simulator would benefit from integrating enriched telemetry with labeled metadata or open-source IoT datasets to reflect real deployment conditions more accurately.

Another limitation was the relatively modest data volume used during testing. Although the ingestion rate was deliberately elevated to evaluate the impact of compaction and query latency in both Copy-on-Write (CoW) and Merge-on-Read (MoR) modes, each test cycle processed less than 10 GB of data. In contrast, real-world telemetry pipelines often handle hundreds of gigabytes or terabytes daily, depending on device density and frequency. Consequently, it remains uncertain whether the observed performance patterns, such as metadata accumulation in Iceberg or small file handling in Hudi, would generalize at production scale. Addressing this gap would require incorporating streaming data generators or other distributed workload emulation tools, to better assess long-term storage costs, metadata scaling, and snapshot lifecycle behavior under sustained ingestion pressure.

Additionally, CoW and MoR configurations were compared using manual configuration of `write.merge.mode`, `write.update.mode`, and `write.delete.mode`; however, the simulator lacked support for automatic mode switching or hybrid ingestion strategies. In practical deployments, pipelines could benefit from dynamic strategies, leveraging MoR during high-frequency update periods or peak hours, and CoW during batch consolidation or archival phases. The absence of such adaptability limited the scope of write amplification and compaction responsiveness for optimizing data lakehouse operations at scale.

Deployment Implications

From a deployment perspective, one key takeaway is the architectural portability of the solution. Despite being tested locally, the simulator was designed with cloud compatibility in mind, particularly for AWS-based deployments.

Due to restricted access to AWS credits and infrastructure, full-scale testing on the cloud platform was not feasible during the project timeline. This limitation prevented the evaluation of critical deployment aspects such as provisioning time, scaling behavior, DevOps complexity, and cost-efficiency, factors that are essential to assessing production readiness in enterprise environments. Nonetheless, during my internship, I was granted access to a corporate cloud environment that allowed me to validate parts of the simulator architecture and test compatibility with services like AWS Glue and S3. While this access was limited in scope and duration, it provided valuable insight into real-world deployment workflows and reinforced the importance of cloud-native design for production-ready systems.

The simulator's integration with services like S3, Glue Data Catalog, and CloudWatch shows strong potential for practical application. However, although Iceberg outperformed Hudi in this controlled setup in terms of read performance and schema flexibility, such outcomes may differ depending on actual schema volatility, ingestion frequency, and long-term retention policies.

A further limitation is that the simulator did not address streaming ingestion scenarios directly. The architecture and testing focused on batch and micro-batch workflows, which aligns well with Iceberg's current strengths but do not explore Hudi's more mature capabilities for continuous data processing. In the context of edge applications, especially in logistics, manufacturing, or smart grid operations, streaming pipelines are not only common but foundational, making this an open area for future experimentation.

Overall, the project has revealed both the strengths and limitations of the proposed approach. While ingestion and query optimizations were demonstrated using modern lakehouse tools, the experimental scope was constrained in terms of dataset diversity, ingestion mode flexibility, and operational realism. This experience highlighted the challenge of balancing reproducibility with realism in data engineering research. Future iterations should aim to incorporate a wider range of data types, support for semi-structured and streaming ingestion, dynamic workload simulation, and broader deployment benchmarking, including cost modeling and fault resilience under device failure conditions or anomalous input.

To conclude, this project validated core design decisions related to Iceberg-based monitoring simulators and also exposed critical challenges in data generation, workload modeling, and system adaptability. These reflections clarify not only what worked and what did not, but also why a deeper understanding of real data and deployment conditions is essential when designing scalable monitoring systems.

Chapter 7

Future Work

To evolve the prototype into a fully scalable enterprise solution, several enhancements are needed across ingestion tuning, edge resource scheduling, system resilience, and autonomous monitoring. These improvements are critical to sustain high-throughput, real-time operations in industrial settings. For instance, in high-variability ingestion scenarios, adaptive compaction intervals could reduce write amplification, while dynamic commit thresholds, adjusted based on workload patterns, would control metadata growth and improve long-term query performance. Additionally, enhanced schema evolution mechanisms are needed to support flexible schema changes and optional fields without interrupting ingestion workflows.

Edge computing performance can be significantly improved through intelligent workload distribution. By incorporating runtime-aware task scheduling based on device CPU and memory load, latency sensitivity, and power availability, the system can dynamically allocate queries and compute tasks to the most appropriate edge nodes. Implementing this approach would require a lightweight telemetry layer that continuously exposes device-level performance metrics, such as current resource usage and availability, to the orchestration logic. Coupled with predictive hardware configuration, which leverages historical load patterns and failure trends, this strategy would enable elastic scaling, minimize overprovisioning and underutilization, and reduce overall infrastructure costs.

Another promising direction is the integration of anomaly detection. Time-series anomaly detection (e.g. moving z-score, or LSTM-based detectors) could monitor ingestion lag, queue buildup, or I/O saturation to trigger autoscaling or device isolation. For instance, if an edge node consistently exceeds CPU thresholds during ingestion spikes, the system could offload or pause low-priority jobs in real time.

Ultimately, these capabilities should converge into a unified, self-optimizing framework that integrates Kubernetes-based orchestration, real-time monitoring agents, and cost-aware autoscaling policies. Such a system would support the resilient and adaptive workload distribution by reacting to real-time resource usage, performance anomalies, and operational costs. This approach would minimize manual intervention while ensuring that the platform remains scalable, fault-tolerant, and production-ready across diverse and dynamic edge environments.

Although this study was conducted in a controlled simulation environment, validation under real-world conditions is essential. Testing in industrial contexts, such as smart factories ingesting data from industrial control systems, fleets of autonomous vehicles under intermittent connectivity, or smart grid networks with continuous telemetry, would assess the system's capacity for handling schema drift, sustained high-volume ingestion, and fault recovery at scale. These proposed optimization areas are summarized in Table 15.

4.1 Conclusion

This study delivered a robust and deployable prototype for scalable data ingestion and query performance optimization within an Iceberg-based data lakehouse architecture. By leveraging effective partitioning strategies and fine-tuned writing configurations, the system maintains analytical efficiency while accommodating dynamic edge workloads. Key techniques such as small file management, query pruning, and compaction tuning were applied to support long-term performance as data volumes grow.

Beyond technical implementation, the architecture offers practical and commercial value. It provides a production-oriented foundation for organizations managing distributed edge infrastructures, enabling scalable monitoring alongside real-time visibility into device health, resource usage, and ingestion behavior.

Looking ahead, future enhancements should focus on workload-aware optimization. Dynamically allocating computational tasks based on real-time edge device telemetry and historical performance patterns would support predictive scaling and intelligent task distribution. These capabilities are essential for building a self-optimizing platform that autonomously adapts to changing operational conditions, resource constraints, and workload fluctuations.

By combining technical robustness with operational adaptability, the system addresses key requirements for modern industrial environments and positions itself as a scalable solution ready for enterprise deployment.

Technique	**Already Implemented?**	**Further Optimization**
Schema evolution	Yes	Extend to partitioned schema handling and support for optional fields.
CoW & MoR ingestion optimizations	Yes	Evaluate compaction strategy per workload and automate mode selection based on update frequency.
Compaction frequency tuning	Partially	Dynamically adjust batch size and commit interval in response to ingestion rates.
Runtime-aware workload orchestration	No	Implement Kubernetes-based scheduling with telemetry-driven load balancing across edge nodes.
Anomaly detection for system stability	No	Apply time-series models to detect ingestion spikes, queue buildup, or I/O saturation and trigger corrective actions.
Predictive hardware configuration	No	Allocate jobs based on historical performance trends and current availability to improve elasticity.

Table 15: Optimization areas in the Data Lakehouse.

Bibliography

References

Alshammari, H. H. (2023). The internet of things healthcare monitoring system based on mqtt protocol. *Alexandria Engineering Journal*, *69*, 275–287. https://doi.org/10.1016/j.aej.2023.01.065

Apache Software Foundation. (2024). Apache kafka documentation [Retrieved December 18, 2024]. https://kafka.apache.org/documentation/

AWS. (2025). Aws glue pricing [Retrieved February 17, 2025]. https://aws.amazon.com/glue/pricing/

Berardi, D., Callegati, F., Giovine, A., Melis, A., Prandini, M., & Rinieri, L. (2023a). When operation technology meets information technology: Challenges and opportunities. *Future Internet*, *15*(3), 95. https://doi.org/10.3390/fi15030095

Berardi, D., Callegati, F., Giovine, A., Melis, A., Prandini, M., & Rinieri, L. (2023b). When operation technology meets information technology: Challenges and opportunities. *Future Internet*, *15*(3), Article 95. https://doi.org/10.3390/fi15030095

Blinowski, G., Ojdowska, A., & Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, *10*, 20357–20374. https://doi.org/10.1109/ACCESS.2022.3151124

Chandar, V. (2017, March). *Hudi: Uber engineering's incremental processing framework on apache hadoop*. Uber Engineering. Retrieved October 15, 2024, from https://www.uber.com/en-ES/blog/hoodie/

Chaudhary, A., Sharma, V., & Alkhayyat, A. (Eds.). (2024). *Intelligent manufacturing and industry 4.0: Impact, trends, and opportunities*. CRC Press.

Chen, M. (2024). What is big data? [Retrieved September 23, 2024]. https://www.oracle.com/big-data/what-is-big-data/

Cheok, A. D., Edirisinghe, C., & Shrestha, M. L. (2024). *The rise of machines: Future of work in the age of ai*. CRC Press.

Databricks. (2024). *Delta lake on databricks*. Retrieved January 20, 2025, from https://docs.databricks.com/aws/en/delta/

Delta-IO. (2025). Delta lake flink connector [Retrieved February 24, 2025]. https://github.com/delta-io/delta/tree/master/connectors/flink

Dremio. (2022). *Table format comparison: Iceberg, hudi, delta lake*. Retrieved January 20, 2025, from https://www.dremio.com/wp-content/uploads/2022/07/Table-Format-Comparison-Iceberg-Hudi-Delta-Lake.pdf

GeeksforGeeks. (2023). 6v's of big data [Retrieved February 21, 2023]. https://www.geeksforgeeks.org/5-vs-of-big-data/

Gopalan, R. (2022). *The cloud data lake: A guide to building robust cloud data architecture*. O'Reilly Media, Inc.

Hai, R., Koutras, C., Quix, C., & Jarke, M. (2023). Data lakes: A survey of functions and systems. *IEEE Transactions on Knowledge and Data Engineering*, *35*(12), 12571–12590. https://doi.org/10.1109/TKDE.2023.3234567

Ho, S. (2022, October). Iceberg's best secret: Exploring metadata tables [Retrieved November 8, 2025]. https://youtu.be/s5eKriX6_EU

Hudi, A. (2021). *Use cases*. Retrieved December 15, 2024, from https://hudi.apache.org/docs/use_cases

IBM. (2024). Data warehouses vs. data lakes vs. data lakehouses. https://www.ibm.com/think/topics/data-warehouse-vs-data-lake-vs-data-lakehouse

Iceberg, A. (2024a). *Apache iceberg nightly documentation*. Retrieved September 10, 2024, from https://iceberg.apache.org/docs/nightly/

Iceberg, A. (2024b). Iceberg table format specification [Retrieved September 14, 2024]. https://iceberg.apache.org/spec/#overview

Kalogerakis, S., & Magoutis, K. (2023). Discovery of breakout patterns in financial tick data via parallel stream processing with in-order guarantees. *Proceedings of the 17th ACM International Conference on Distributed and Event-based Systems*, 115–126. https://doi.org/10.1145/3583678.3596897

Kalsoom, T., Ahmed, S., Rafi-ul-Shan, P. M., Azmat, M., Akhtar, P., Pervez, Z., & Ur-Rehman, M. (2021). Impact of iot on manufacturing industry 4.0: A new triangular systematic review. *Sustainability*, *13*(22), 12506. https://doi.org/10.3390/su132212506

Khan, N., Ahmad, K., Tamimi, A. A., Alani, M. M., Bermak, A., & Khalil, I. (2024). Explainable ai-based intrusion detection system for industry 5.0: An overview of the literature, associated challenges, the existing solutions, and potential research directions. *arXiv preprint*, *arXiv:2408.03335*. https://arxiv.org/abs/2408.03335

Kujawski, M. (2023). Data warehouse, data lake, and data lakehouse: Comparison of data platforms. *Medium*. https://medium.com/@mariusz_ kujawski/data-warehouse-data-lake-and-data-lakehouse-comparison-of-data-platforms-842f0288b71

Lake, D. (2023, July). *Deletion vectors in delta lake.* Retrieved December 23, 2024, from https://delta.io/blog/2023-07-05-deletion-vectors/

Lorica, B., Armbrust, M., Xin, R., Zaharia, M., & Ghodsi, A. (2025, January). What is a lakehouse? [Retrieved from Databricks blog]. https://www.databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html

Mohamed, O. A. M. (2023). *How generative ai transforming supply chain operations and efficiency?* [Master's thesis]. Politecnico di Milano. https://hdl.handle.net/10589/218912

Pech, M., Vrchota, J., & Bednář, J. (2021). Predictive maintenance and intelligent sensors in smart factory. *Sensors*, *21*(4), 1470. https://doi.org/10.3390/s21041470

Rajpurohit, A. M., Kumar, P., Kumar, R. R., & Kumar, R. (2023). A review on apache spark. *Kilby*, *100*(7th).

Reis, J., & Housley, M. (2022a). *Fundamentals of data engineering: Plan and build robust data systems.* O'Reilly Media.

Reis, J., & Housley, M. (2022b). *Fundamentals of data engineering: Plan and build robust data systems.* O'Reilly Media.

Sá, J. O. e., Gonçalves, R., & Kaldeich, C. (2024). Benchmark of market cloud data warehouse technologies. *Procedia Computer Science*, *239*, 1212–1219.

Saleh, A., Tarkoma, S., Pirttikangas, S., & Lovén, L. (2024). Publish/subscribe for edge intelligence: Systematic review and future prospects [Available at SSRN 4872730]. *SSRN*. https://ssrn.com/abstract=4872730

Serra, J. (2024). *Deciphering data architectures.* O'Reilly Media, Inc.

Šestak, M., & Vovk, T. (2023). Using apache spark for ensuring data quality in modern data lake pipeline architectures [ISSN: 1613-0073]. *Proceedings of the CEUR Workshop Proceedings*, *1613*(0073). http://ceur-ws.org/Vol-1613/

Toshev, M. (2015). *Learning rabbitmq.* Packt Publishing.

Zhou, X., Ge, S., Chi, J., & Qiu, T. (2024). *Industrial edge computing: Architecture, optimization and applications.* Springer Nature.