Spatial Temporal Analysis: Medallion Architecture

Submitted to:

Colorado State University – Department of Computer Science CS555

Report Prepared By:

Christopher Cowart

Introduction

Medallion lakehouse architecture has become the de-facto standard for modern data platforms. This is now embraced by the major cloud providers Databricks [1], Microsoft [2], AWS [3] and Snowflake [4]. While multi-layer refinement patterns have been used for decades, traditional ETL architectures employed Staging, Integration and Presentation layers [5]. The medallion pattern formalizes this standard pathway with Bronze (raw ingestion), Silver (cleaned and validated) and Gold (analytics ready) layers, with a systematic approach for managing data quality, supporting incremental refinement, and being auditable throughout the data lifecycle. These architectural patterns are implemented in production use in various industries [6], including scoring and credit decisioning systems, IoT telemetry, clickstream analytics, and spatio-temporal event processing. The convergence of major cloud providers on this architectural pattern validates its efficacy at enterprise scale, but comprehensive reference implementations demonstrating end to end integration are still scarce in academic literature.

While the high-level medallion pattern is well documented in industry blogs and vendor documentation, there are limited comprehensive open-source reference implementations, especially with streaming ingestion and real-time quality validation. To fill this gap, I present a comprehensive open-source implementation of the medallion lakehouse pattern realized using Apache Spark Structured Streaming, Apache Kafka, Apache Iceberg, and Amazon Deequ [7]. The architecture is validated using NYC Taxi trip records, NOAA weather observations, and NYC special events. This was necessitated not for domain specific insights but representative workload typical of production characteristics:

• Heterogeneous event streams with varying frequencies (high, medium, low)

• Quality variability necessitating in-stream validation

• Temporal correlation patterns

This publicly available dataset allows reproducible research to be realized also demonstrating patterns with applicability to proprietary domains where data cannot be shared for example healthcare, financial services, or telecommunication. This implementation demonstrates how industry standard architectural

patterns can be realized using open-source technologies to achieve ACID transactions, sub-second latency, and integrated quality validation.

Problem Characterization

Streaming workload deployment of Medallion Lake House Architectures exposes a set of interconnected technical challenges that need to be resolved simultaneously to ensure reliable, high-performance applications at-scale.

The first significant challenge for achieving exactly-once processing (idempotence) is ensuring that each event is processed only one time regardless of failure, network partition, and/or restarts of various components in the system [9].

Achieving idempotence requires coordinated checkpointing among the stream processing framework, message broker(s), and storage layer. As a result of scale, achieving these guarantees while managing offsets, transactionally committing events, and recovering from failures on the order of tens of thousands of events per second from hundreds of heterogeneous sources requires precise coordination of all aspects of processing.

Ensuring data quality without impacting performance/latency is a second significant challenge for in-stream quality validation. Unlike traditional batch systems which apply validation to separate validation jobs hours or days after data is ingested, allowing corrupt data to propagate through all levels of the pipeline until detected, inline quality validation within the primary processing path of a streaming system applies constraint based checks (completeness, range validation, distribution analysis, etc.) [7], while meeting end-to-end latency of <1 second.

Temporal correlations across multiple event streams with different ingestion rates, schema, and semantics create challenges when attempting to process correlated events from different streams in causal order. Temporal relationships across streams are maintained by watermarking strategies accounting for varying delays and out-of-order event arrival times. The ability to join two streams with disparate ingest rates requires minimizing buffering while maintaining the temporal relationship necessary for producing accurate contextual analytics.

Transactional consistency for ACID guarantees on continuous arrival of streaming data, requires providing database level guarantees to the continuously arriving events in the system. The ability of the system to concurrently support readers accessing historical data, while supporting writer continuously appending new data to the system, requires maintaining snapshot isolation and serializability of the data. Providing these guarantees on top of object storage without central coordination requires sophisticated versioning, optimistic concurrency control, and metadata management strategies.

Optimization for throughput conflicts with the requirement for low-latency, validation of quality introduces processing overhead, and transactional guarantees introduce delays due to coordination. These challenges motivate the integrated approach presented.

Dominant Approaches to the problem

Addressing the challenges of data processing has led to three primary architectural approaches: batch processing systems, pure streaming engines, and lakehouse architectures. Each of these systems has their distinct trade-offs in latency, consistency, and operational complexity.

Batch processing systems such as Apache Hadoop MapReduce and Apache Spark provide strong consistency guarantees and support rich analytical queries through distributed data-parallel processing. These systems excel at processing large historical datasets with complex transformations, offering mature ecosystems and proven scalability. However, these systems introduce inherent latency measured in hours to days, as data must accumulate before processing begins [10]. This delay renders them unsuitable for applications requiring near or near real-time decisioning or operational dashboards despite their robustness.

Pure streaming systems such as Apache Flink [11] or Apache Kafka Streams [12] address latency limitations by processing events continuously as they arrive. These systems achieve sub-second latency through continuous event processing. However, these systems typically lack built-in support for ACID transactions on stored data, making them unsuitable as a system of record [13]. Historical queries require separate storage layers and data quality validation frameworks like Deequ [7] must be integrated separately, complicating pipeline architectures and operational management.

Lakehouse architectures such as Delta Lake [1], Apache Iceberg [15], and Apache Hudi [17] emerged to bridge this gap by combining streaming ingestion capabilities with transactional consistency. Delta Lake provides ACID guarantees on object storage through optimistic concurrency control and versioned snapshots, while Iceberg offers hidden partitioning and time-travel capabilities. Comparative benchmarks demonstrate performance trade-offs among formats depending on workload characteristics [16], with Delta Lake showing advantages in query latency for append-heavy workloads while Hudi optimizes for keyed upserts, though all three support similar transaction isolation levels. Existing lakehouse implementations treat data quality validation as a downstream batch process rather than an integrated streaming component [1, 13, 15] requiring separate validation pipelines and delaying quality issue detection.

None of these approaches provide a comprehensive solution integrating streaming ingestion, real-time quality validation, transactional consistency, and end-to-end observability within a unified architectural framework; the gap this work addresses.

Methodology

The medallion architecture pattern was implemented as a three-layer medallion architecture pattern (Bronze, Silver, Gold) using an integrated streaming pipeline. Rather than finding domain-specific insights from the taxi dataset, this implementation is focused on validating the medallion architecture pattern itself. In addition to schema management for registering Iceberg tables, supporting infrastructure included cross-cutting observability components for experimental benchmarks and insights presented previously.

Architectural pattern validation was prioritized over operational tooling complexity in this implementation. To provide benchmarking without external infrastructure dependencies, metrics were embedded directly within the Iceberg tables. Schema management required execution of DDL statements for registration of Iceberg tables; a simple SQL-based framework reading DDL files from resource directories satisfies this requirement. Both choices minimized setup complexity and maintained focus on validating the medallion architecture.

Streaming first design was employed in the architecture; each layer consumes data from Apache Iceberg tables rather than message queues, enabling ACID transactions and time-travel capabilities throughout the pipeline. This design addresses the transactional consistency challenge through Iceberg's snapshot isolation, allowing concurrent readers to query historical data while writers continuously append records without coordination overhead. Apache Spark Structured Streaming was used as the processing engine due to its mature streaming abstraction.

Apache Kafka provides event ingestion with exactly-once guarantees. Apache Iceberg was selected over Delta Lake and Hudi based on its superior metadata management capabilities: hidden partitioning eliminates user-facing partition predicates, evolution capabilities support schema changes without data rewrites, and partition-level metadata enables efficient query planning at scale. The JDBC catalog ensures atomic metadata updates essential for multi-writers streaming scenarios.

Amazon Deequ integrates quality validation directly into the streaming pipeline, addressing the in-stream quality validation challenges. Deequ's constraint-based checks execute within Spark's execution plan, adding validation overhead inline rather than deferring checks to downstream batch jobs. This design trades modest processing latency for early error detection.

The Bronze layer implements exactly-once Kafka-to-Iceberg ingestion through three streaming jobs (trips, weather, events) following a common abstraction template pattern. This abstract base class encapsulates Kafka source configuration, Avro deserialization via Confluent Schema Registry, and

Iceberg sink coordination, ensuring consistency across ingestion pipelines. Each job reads Avro-serialized events from dedicated Kafka topics and appends records to partitioned Iceberg tables.

The Silver layer applies constraint-based quality validation, business rule transformations and deduplication through three Scala-based jobs extending the silver abstraction template. This abstract class integrates Deequ validation, SQL-based transformations and watermark-based deduplication into a unified processing pattern. Transformation queries reside in resource files with parameterized business rules separating business logic from processing code and providing non-developer review without code changes.

Deequ validation executes, applying completeness checks, range constraints, and distribution validations. Quality check results write to a monitoring table (silver_quality_metrics) capturing per-batch pass/fail counts, constraint violations and quarantine decisions. Failed batches quarantine to separate tables for offline analysis rather than blocking pipeline progress.

Watermarking addresses the temporal correlation challenge listed above by allowing 5-minute out-of-order arrival windows based on ingestion timestamps from Bronze. This duration balances event completeness against latency: shorter watermarks reduce end-to-end latency but risk dropping late-arriving events, while longer windows increase buffering overhead. Deduplication operates on configurable key within watermark windows, removing duplicates without maintaining unbounded state.

The Gold layer implementation serves dual purposes: demonstrating typical analytics patterns and providing instrumentation for experimental validation. The codebase includes trip_metrics_live, a streaming aggregation implementing star schema patterns common in data mart architectures, showing how Gold layers transform silver data into analytics ready dimensional models. However, for this validation study, gold analytics were disabled to dedicate processing resources to latency instrumentation. The PipelineLatencyJob replaced typical analytics: a 2-minute tumbling window aggregates events from silver.trips_cleaned, calculating throughput metrics (events per second) and latency percentiles (P50, P95, P99) through approximate percentile functions. Latency computation measures end-to-end delay as the difference between current processing time and event timestamps, capturing total pipeline latency from event generation through silver processing.

Rather than implementing observability as an architectural layer, this work treats it as a cross-cutting concern addressed through dedicated Iceberg tables. Two instrumentation mechanisms capture pipeline health: Deequ quality metrics inline within the silver layer processing, and latency instrumentation replaces the gold analytics pipeline. This observability-as-data pattern addresses the end-to-end observability challenge by enabling SQL-based pipeline analysis without external monitoring infrastructure. Latency trends, quality degradation and bottleneck identification emerges from standard

Iceberg queries, simplifying operational analysis and providing the data foundation for experimental benchmarks presented below.

The system deploys via Docker across 15+ containers: Kafka broker with 12 partitions, Schema Registry, PostgreSQL (Iceberg catalog), MinIO (object storage), Spark master, two Spark workers (2 cores, 2GB RAM), and per-layer streaming jobs. Dynamic executor allocation enables elastic scaling: job requests 0-2 executors based on backlog, within 60-second idle timeouts releasing unused resources. This deployment validates that the medallion architecture functions correctly on constrained infrastructure without requiring production-scale resources, demonstrating the pattern's viability for resource limited environments.

A synthetic data producer generates three event streams at configurable rates, enabling controlled testing of various scenarios without searching for suitable datasets. This design choice prioritizes testability and architecture validation over data authenticity: configurable event rates enable stress testing of architecture under varying load conditions, while error injection mechanisms enable systematic quality validation testing.

Experimental Benchmarks

The evaluation of the architecture as previously described, were accomplished using experimental methods. This was completed on a single laptop having limited compute resources. The results of these experiments provide evidence that the medallion architecture can operate effectively with no requirement for production-level compute infrastructure.

Benchmarking was first performed against a base configuration consisting of two Spark workers (each having 2 CPU cores and 2 GB of RAM). Resource contention occurred during processing of 500 synthetic producer generated events per second. The base configuration produced an average event rate of 316.7 events per second; this falls short of the required event rate. In addition to this, end-to-end latency measurements indicated that median (P50) latency was 711.5 seconds and that 95th percentile (P95) latency was 745.8 seconds, which indicates significant processing backlog accumulation due to resource contention. As such, the necessity for scalable elastic computing to support the processing of increased volumes of data is demonstrated.

A third Spark worker was added to the previous two Spark worker base configuration, thus doubling the amount of available computing power (a 50% increase). The resulting three worker configuration provided evidence of the architecture’s elasticity and scalability. Specifically, the three worker configuration resulted in a 35.7% increase in event rate (from 316.7 to 429.8 events per second), a 47.1% decrease in median (P50) latency (from 711.5 to 376.1 seconds), and a 44.2% decrease in 95th percentile (P95) latency (from 745.8 to 416.2 seconds), which represents a greater than proportionate decrease in latency compared to the increase in computing power. Thus, the dynamic executor allocation utilized in the medallion architecture allowed for effective use of available computing resources, enabling the processing workload to be dynamically scaled based upon the size of the

processing backlog without the need for manual intervention, thereby providing a solution to the performance bottlenecks experienced in the two-worker baseline configuration.

To test the ability of inline quality validation to isolate constraint violations from clean data, error injection testing was conducted to evaluate the inline quality validation through Amazon Deequ constraint checks implemented in the silver layer processing, as previously described. Events were created synthetically and were injected into the pipeline containing values that violate the defined constraints: trip distance is outside the 0.1-to-200-mile range, fare is less than $2.50 or greater than $1,000, and number of passengers is outside the 1 to 6 range. The inline quality validation demonstrated perfect isolation of all events violating the specified constraints to separate quarantine tables without impeding pipeline progress, producing a 100% quarantine rate. This provides evidence that inline quality validation can be used to resolve the previously stated data quality problem, demonstrating that comprehensive constraint-based checks can operate within the streaming pipeline without impacting throughput and preventing corrupt data from being propagated through downstream layers.

Testing of fault tolerance mechanisms of the medallion architecture was accomplished to verify that the architecture is resilient to failures in the system. Testing included verification of the reliability of the medallion architecture to recover from failures caused by Docker restart policies, Spark's checkpointing mechanisms implementing the exactly-once semantics described previously, and out-of-memory (OOM) failures. The Spark executor failure recovery mechanism automatically replaced failed executors and recovered the job state from checkpoints, and the jobs continued running without data loss. Docker restart policies provided further resiliency to failures at the container level, ensuring that services remained available even during infrastructure-related disruptions. Testing of manual termination of nodes verified that the pipeline continued operating after a node terminated without experiencing a catastrophic failure, verifying that the system tolerated worker failures while maintaining the guarantee of exactly-once processing. These tests demonstrated the mechanisms of idempotence described above, providing evidence that the coordinated checkpointing between Kafka offsets and Iceberg table commits enable reliable recovery of the system from failures without losing data or duplicating data across failure scenarios.

Insights Gleaned

Implementation and showed many new insights beyond what were evident in the theoretical designs of medallion or in the current literature, providing empirical support for the architectural choices made.

The architecture has shown nearly linear elasticity in its ability to handle increases in the number of tasks it processes and adding one additional Spark Worker (a 50% increase in available computing resources) resulted in a 47.1% decrease in latency. Thus, the pattern supports predictable scaling without diminishing returns, validating that Iceberg's metadata management system and Spark's dynamic executor allocation mechanism coupled with Kafka's use of partitions to allocate work among multiple machines can provide scalable solutions.

Although inline quality validation using Deequ was shown to be possible; it also provided some important insights regarding the resources needed for such a validation: to prevent the validation itself from being a processing. The initial configuration with two Spark Workers was able to validate 316.7 events per second while attempting to validate 500 events per second, which clearly shows that systems

that do not have sufficient computing power to perform inline validation at scale will degrade their throughput. When a third worker was added, the system was able to restore its event rate to 429.8 events per second while still achieving 100% quarantine effectiveness; thus, the feasibility of performing inline validation is entirely dependent upon having sufficient computing resources to absorb the overhead of validation and maintain system throughput. Although Deequ was designed as a Scala-first tool, the need to implement the validation logic in Scala required that the Silver Layer and Gold Layer had to be implemented in Scala, rather than Java; therefore, the experience with Deequ provided an important insight to how the primary tools and libraries in the Ecosystem view Java as secondary to Scala. This insight refutes the idea that the only way to achieve quality validation is by executing batch jobs; therefore, the benefits of performing quality validation are available when you have sufficient computing power to absorb the overhead of validation without creating backpressure, if your system does not have sufficient computing resources then performing inline validation may create the same performance problems as performing quality validation using batch jobs.

The coordinated checkpointing between Kafka Offsets and Iceberg commits was sufficient to ensure exactly once semantics for all operations performed without the need for distributed coordination protocols or deduplication windows. Simple checkpointing coordination is sufficient to achieve idempotence guarantees. This is due to Iceberg's atomic snapshot commits: By linking the advance of Kafka Offset to the success of snapshot commit, you get an implicit coordination mechanism without an explicit distributed transaction; thereby eliminating memory pressure and processing delays associated with deduplication window approaches.

The hidden partitioning in Iceberg removed the complexity of managing partitions through metadata-driven query optimization. In traditional Hive-style Partitioning, users must include an explicit predicate in their query to specify which physical partition they wish to access, which exposes the physical structure of the data and provides a means to create performance degradation in queries when the predicate is omitted. With Iceberg, users write standard WHERE clauses against logical columns, and the partition pruning occurs

How will the problem space transform in the future

The shape of the medallion lakehouse paradigm will evolve as industry acceptance shifts from batch-oriented data warehouses to streaming-first platforms. The need for sub-second latencies in operational analytics and real-time decisioning and the corresponding effects of pressure on architectures to collapse the former barriers between transactional and analytical systems necessitates this evolution. The future positioning of SQL-first transformation orchestration tools such as dbt [18] and SQLMesh [19] suffers doubt and uncertainty. These tools orchestrate batch transformations on top of data warehouses at present. As streaming frameworks such as Spark Structured Streaming take on the role of transformation underpinnings demonstrated with SQL processing in this work, the line between orchestration tools and processing engines tends to become blurred leading potentially to pressure on dbt/SQLMesh to adapt transformation semantics to streaming or risk destruction by lakehouse capabilities.

Quality validation will cease to be ascribed to independent frameworks and begin to extend into table format features of first-class citizens. Constraints based check schemes will become native capabilities rather than requiring solutions such as Deequ. The observability-as-data paradigm espoused in this work mirrors this trend towards historical governance and the realization of platforms for enterprise into

which lineage tracking, access control and compliance checking may be done within the lakehouse itself rather than within a separate metadata catalog environment.

Multi-cloud pressures of portability may extend to the general advantages of open table formats such as Iceberg which will allow cross-platform utilization without loss of value due to vendor bonnet-locking. The industry trajectory suggests the emergence of lakehouse platforms integrating former specialized functions of complexity leading to a unification of the modern data stack and in unified patterns of architectural form, as well as increasing the pressure on existing paradigms of the orchestration of transformation systems.

Conclusions

This study provides evidence that the architecture frameworks may be able to incorporate multiple features such as streaming ingestion, real-time quality validation, and ACID transactions into an overarching framework using open-source tools. The implementations of this study demonstrate that using open-source tools (Kafka-Iceberg-Spark) will allow for exactly once semantics through checkpoint coordination. Additionally, Deequ validation of constraints will run inline throughout streaming pipelines in place of batch processing if sufficient compute is provided.

Experimental testing under constrained resource conditions demonstrated that this pattern can be viable in non-production scale environments and has characteristics of near linear elasticity which allows for incremental scalability as workload increases. Hidden partitioning through metadata driven partitioning also eliminated complexities associated with managing partitions as seen in previous data lakes (Hive era). Additionally, the ability to analyze pipelines based on SQL queries without requiring additional monitoring infrastructure enables "observability as data".

The open-source reference implementation in this study fills the existing body of research regarding comprehensive lakehouse architectures by providing reproducible validation of these industry patterns using openly available software tools. Insights derived from this study challenge assumptions concerning the practicality of validating streams in real time and illustrate how architectural simplicity results from the coordination of checkpoints and metadata first design paradigms, and not from the need for complex orchestration layering.

Bibliography

[1] What is a medallion architecture?. Databricks. (n.d.). https://www.databricks.com/glossary/medallion-architecture

[2] Eric-Urban. (n.d.). What is a lakehouse? - microsoft fabric. What is a lakehouse? - Microsoft Fabric | Microsoft Learn. https://learn.microsoft.com/en-us/fabric/data-engineering/lakehouse-overview

[3] Build a lake house architecture on AWS. (n.d.). https://aws.amazon.com/blogs/big-data/build-a-lake-house-architecture-on-aws/

[4] *Apache iceberg tables*. Apache Iceberg tables | Snowflake Documentation. (n.d.). https://docs.snowflake.com/en/user-guide/tables-iceberg

[5] Kimball, R., & Ross, M. (2013). *The Data Warehouse Toolkit: The definitive guide to dimensional modeling*. Wiley.

[6] Behm, A., Palkar, S., Agarwal, U., Armstrong, T., Cashman, D., Dave, A., Greenstein, T., Hovsepian, S., Johnson, R., Sai Krishnan, A., Leventis, P., Luszczak, A., Menon, P., Mokhtar, M., Pang, G., Paranjpye, S., Rahn, G., Samwel, B., van Bussel, T., … Zaharia, M. (2022). Photon: A fast query engine for Lakehouse Systems. *Proceedings of the 2022 International Conference on Management of Data*, 2326–2339. https://doi.org/10.1145/3514221.3526054

[7] Schelter, S., Lange, D., Schmidt, P., Celikel, M., Biessmann, F., & Grafberger, A. (2018). Automating large-scale data quality verification. *Proceedings of the VLDB Endowment*, *11*(12), 1781–1794. https://doi.org/10.14778/3229863.3229867

[8] Legal text. General Data Protection Regulation (GDPR). (2024, April 22). https://gdpr-info.eu/

[9] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., & Whittle, S. (2015). The dataflow model. *Proceedings of the VLDB Endowment*, *8*(12), 1792–1803. https://doi.org/10.14778/2824032.2824076

[10] Dean, J., & Ghemawat, S. (2008). MapReduce. *Communications of the ACM*, *51*(1), 107–113. https://doi.org/10.1145/1327452.1327492

[11] Apache Flink: Stream and batch processing in a single engine Paris Carbone†. (n.d.-a). https://asterios.katsifodimos.com/assets/publications/flink-deb.pdf

[12] Shapira, G., Palino, T., Sivaram, R., & Petty, K. (2021). *Kafka - the definitive guide: Real-time data and stream processing at scale*. O'Reilly.

[13] Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly.

[15] Summit, D. (n.d.). *Iceberg: A fast table format for S3*. Slideshare. https://www.slideshare.net/Hadoop_Summit/iceberg-a-fast-table-format-for-s3-103201179

[16] Analyzing and comparing Lakehouse Storage Systems. (n.d.-a). https://www.cidrdb.org/cidr2023/papers/p92-jain.pdf

[17] Hudi: Uber Engineering's incremental processing framework on Apache Hadoop | Uber Blog. (n.d.-d). https://www.uber.com/blog/hoodie/