**Spatial Temporal Analysis: Medallion Architecture**

**Submitted to:**

Colorado State University – Department of Computer Science CS555

**Report Prepared By:**

Christopher Cowart

**Introduction**

The medallion lakehouse architecture has emerged as the industry standard for modern data platforms. This model has been adopted by major cloud providers including Databricks [1], Microsoft [2], AWS [3], and Snowflake [4]. While multi-layer data refinement patterns have existed for decades, traditional ETL architectures used Staging, Integration and Presentation layers [5]. The medallion pattern formalizes this approach with Bronze (raw ingestion), Silver (cleaned and validated), and Gold (analytics ready) layers, providing a systematic approach to managing data quality, supporting incremental refinement, and maintaining auditability throughout the data lifecycle. These architectural patterns are deployed in production across diverse industries [6], from scoring and credit decisioning systems to IoT telemetry, clickstream analytics, and spatial-temporal event processing. The convergence of major cloud platforms on this pattern validates its effectiveness at enterprise scale, yet comprehensive reference implementations demonstrating end-to-end integration remain scarce in academic literature.

In regulated industries such as financial services, these patterns transcend performance optimization to become operational requirements. ACID transactions ensure regulatory compliance and enable repeatable, auditable queries essential for model governance. Quality validation prevents erroneous decisions that carry direct financial liability, particularly in high-stakes domains like credit decisioning and fraud detection. Comprehensive audit trails support explainability mandates increasingly required by regulatory frameworks such as GDPR, FCRA, and model risk management standards [8]. As organizations face growing pressure to demonstrate data lineage and decision traceability, the systematic layering of medallion architecture provides both technical infrastructure and a governance framework.

While the high-level medallion pattern is well-documented in industry blogs and vendor documentation, comprehensive open-source reference implementations that integrate streaming ingestion, real-time quality validation remain limited. To bridge this gap, I present a complete open-source implementation of the medallion lakehouse pattern 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 used not for domain specific insights but as a representative workload exhibiting production characteristics:

* Heterogeneous event streams with varying frequencies (high, medium, low)
* Quality variability requiring in-stream validation
* Temporal correlation patterns

This publicly available dataset enables reproducible research while demonstrating patterns directly applicable to proprietary domains where data cannot be shared such as 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**

Implementing medallion lakehouse architectures for streaming workloads introduces several interconnected technical challenges that must be addressed simultaneously to achieve production-grade reliability and performance.

Exactly-once processing (idempotency)presents the first fundamental challenge. Distributed streaming systems must guarantee that each event is processed precisely once despite failures, network partitions, and restarts [9]. Achieving idempotency requires coordinated checkpointing between stream processing frameworks, message brokers, and storage systems. At scale, maintaining these guarantees while ingesting thousands of events per second from multiple heterogeneous sources demands careful coordination of offset management, transactional commits, and failure recovery mechanisms. Simple solutions like deduplication windows introduce memory pressure and processing delays, while distributed coordination protocols add latency and operational complexity.

In-stream quality validation introduces a second critical challenge; Ensuring data quality without compromising throughput or latency. Traditional batch systems defer quality checks to separate validation jobs running hours or days after ingestion, allowing corrupted data to propagate through multiple pipeline stages before detection. Streaming architectures require inline validation that executes within the main processing path, applying constraint-based checks (completeness, range validation, distribution analysis, etc.) [7] while maintaining sub-second end-to-end latency. Balancing validation thoroughness against processing speed creates fundamental trade-offs between quality assurance and system performance.

Multi-stream temporal correlation presents challenges when processing heterogeneous event streams with different ingestion rates, schemas, and semantics. Maintaining temporal consistency across streams ensures that correlated events from different sources are processed in causal order requires watermarking strategies that account for varying delays and out-of-order arrival patterns. Stream joins must handle rate mismatches without excessive buffering while preserving the temporal relationship necessary for accurate contextual analytics.

ACID transactional consistency on streaming data requires extending database guarantees to continuously arriving events. Systems must support concurrent readers querying historical data while writers continuously append new records, maintaining snapshot isolation and serializability. Achieving these guarantees on object storage without centralized coordination demands sophisticated versioning, optimistic concurrency control, and metadata management strategies.

Observability across multi-layer pipelines poses operational challenges. Traditional monitoring focuses on individual component metrics (queues, processing rates, error counts, etc.) but fail to provide visibility into cross-layer latency attribution, data quality degradation propagation, and processing bottleneck identification. Comprehensive observability requires embedding instrumentation within the data architecture itself, tracking timestamps and quality metrics as a first-class citizen rather than external monitoring concerns.

These challenges are deeply interconnected; optimization for throughput conflicts with latency requirements, quality validation adds processing overhead, and transactional guarantees introduce coordination delays. Addressing them within a unified architectural framework motivates the integrated approach presented in this work.

**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 [11]. 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 [12] or Apache Kafka Streams [13] address latency limitations by processing events continuously as they arrive. Flink achieves sub-second processing latency through pipelined execution and efficient state management while Kafka Streams provides tight integration with Kafka’s exactly-once semantics. However, these systems typically lack built-in support for ACID transactions on stored data, making them unsuitable as a system of record [14]. 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 show Iceberg achieving superior query performance on large datasets due to advanced metadata management [16], 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, 14, 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**

This work implements a three-layer medallion lakehouse architecture (Bronze, Silver, Gold) addressing the challenges outlined above through an integrated streaming pipeline. This implementation focuses on validating the medallion architecture pattern rather than deriving domain-specific insights from taxi data. Supporting infrastructure includes schema management for Iceberg table registration and cross-cutting observability components to enable experimental benchmarks and insights present above.

The implementation prioritizes architectural pattern validation over operational tooling complexity. While production systems typically employ external observability platforms such as Prometheus/Grafana or CloudWatch, this work embeds metrics directly within the Iceberg tables to enable benchmarking without external infrastructure dependencies. Schema management requires DDL execution for Iceberg table registration; a simple SQL-based framework reading DDL files from resource directories fulfills this prerequisite, whereas a production environment would utilize dedicated migration tools such as Liquibase or Flyway for versioned database evolution. These choices minimize setup complexity and maintain focus on demonstrating the medallion architecture patterns rather than operational tooling integration, enabling reproducible validation of the core architectural approach within constrained timelines and resources.

The architecture employs a streaming-first design where each layer consumes data through Apache Iceberg tables rather than message queues, enabling ACID transactions and time-travel capabilities throughout the pipeline. This 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 serves as the processing engine due to its mature streaming abstractions, Data Frame API, and native Iceberg integration. Spark’s micro-batching model provides natural checkpointing boundaries for exactly-once semantics while maintaining sub-second latency through 10-second trigger intervals.

Apache Kafka provides event ingestion with exactly-once guarantees through idempotent producers and transactional writes, addressing the idempotency challenge. 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 backed by PostgreSQL ensures atomic metadata updates essential for multi-writer 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, preventing corrupted data from propagating through multiple pipeline stages.

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 with metadata augmentation (ingestion timestamp, Kafka offset, partition).

Exactly-once semantics derive from coordinated checkpointing: Spark commits Kafka offsets only after Iceberg successfully commits a table snapshot, ensuring idempotent recovery from failures. Checkpoint locations stored in distributed storage enable stateful recovery, allowing jobs to resume from the last committed offset after restarts. This eliminates deduplication windows that introduce memory pressure and processing delays as discussed above.

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 (e.g. trips\_cleaned.sql) with parameterized business rules (rush hour windows, distance categories, fare tiers, etc.), separating business logic from processing code and enabling non-developer review without code changes.

Deequ validation executes inline within each micro-batch, 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, maintaining throughput while ensuring quality visibility.

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 combinations (e.g., timestamp + pickup\_location\_id + dropoff\_location\_id for trips) 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, showcasing 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 (silver\_quality\_metrics table), and latency instrumentation replaces the gold analytics pipeline (pipeline\_latency\_metrics table). 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 emerge 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, 2 GB ram), and per-layer streaming jobs. Dynamic executor allocation enables elastic scaling: job request 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. By focusing on architecture validation rather than data insights, the synthetic producer eliminates dependencies on external data sources while maintaining full control over test scenarios and failure modes.

**Experimental Benchmarks**

Experimental validation of the medallion architecture was conducted under resource-constrained conditions to demonstrate architectural elasticity, data quality validation effectiveness, and fault tolerance mechanisms described above. All experiments were executed on a single laptop with limited compute resources, validating that the architecture functions correctly without requiring production-scale infrastructure.

Initial benchmarking with a baseline configuration of two Spark workers (2 core 2GB RAM each) revealed resource contention when processing 500 events per second from the synthetic producer described in the methodology. The baseline configuration achieved an average throughput of 316.7 events per second, falling short of the target rate, while end-to-end latency measurements showed median (P50) latency of 711.5 and 95 percentile (P95) latency of 745.8 seconds, indicating significant processing backlog accumulation. This resource contention validated the need for elastic scaling capabilities within the architecture to address computational bottlenecks.

The addition of a third spark worker, representing a 50% compute increase, demonstrated the architecture’s elasticity and scalability characteristics. The scaled configuration achieved 429.8 events per second throughput (35.7% improvement) while reducing P50 latency to 376.1 seconds (47.1% reduction) and P95 latency to 416.2 seconds (44.2 reduction), with latency reductions exceeding the proportional resource increase. Dynamic executor allocation enabled efficient resource utilization, allowing jobs to scale elastically based on processing backlog without manual intervention, addressing the performance bottlenecks observed in the baseline configuration.

Error injection testing was conducted to validate the effectiveness of inline quality validation through Amazon Deequ constraint checks integrated within the silver layer processing described in the methodology. Synthetic events were injected with values violating configured constraints: trip distances outside the 0.1-200-mile range, fare amounts below $2.50 or above $1,000, and passenger counts outside the 1-6 range. The quality validation mechanism demonstrated complete isolation of all constraint-violating events to separate quarantine tables without blocking pipeline progress, achieving a 100% quarantine rate. This validates the inline quality validation approach for addressing the data quality challenge outlined above, demonstrating that comprehensive constraint-based checks can execute within the streaming pipeline while maintaining throughput and preventing corrupted data propagation through downstream layers.

Fault tolerance testing validated the architecture’s resilience through Docker restart policies and Spark’s checkpointing mechanisms implementing the exactly-once semantics described previously. Out-of-memory (OOM) failure scenarios were handled by Spark’s executor failure recovery mechanisms, allowing jobs to continue running through automatic executor replacement and checkpoint-based recovery without data loss. Docker restart policies provided additional resilience for container-level failures, ensuring service availability across infrastructure disruptions. Manual node termination tests demonstrated continued pipeline operation without catastrophic failure, validating that the system tolerates worker failures while maintaining exactly-once processing guarantees. These tests validate idempotency mechanisms outlined above, demonstrating that coordinated checkpointing between Kafka offsets and Iceberg table commits enables reliable recovery without data loss or duplication across failure scenarios.

**Insights Gleaned**

**How will the problem space transform in the future**

**Conclusions**

**Bibliography**