International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I1P138
Eureka Vision Publication | Volume 7, Issue 1, 254 -266 2026

Original Article

Distributed Stream Processing for Real-Time Healthcare-Motivated Analytics in Multi-Cloud: A Semantics-Aligned Benchmark of Kafka-Centric Pipelines with Flink and Spark Structured Streaming

Sai Kiran Yadav Battula
Independent Researcher, Pittsburgh, Pennsylvania, United States.

Received On: 22/01/2026 ***Revised On: 20/02/2026*** ***Accepted On: 27/02/2026*** ***Published On: 06/03/2026***

Abstract *- Healthcare providers increasingly rely on Kafka-centric streaming architectures for sub-second alerting and continuous analytics over heterogeneous clinical and device events, yet practitioners lack a semantics-aligned benchmark to compare Apache Flink and Spark Structured Streaming in multi-cloud, lakehouse-oriented deployments. We present a benchmark of Kafka-based pipelines using Flink 1.18 and Spark Structured Streaming 3.5 on AWS and Azure, with outputs persisted via a neutral Databricks Delta ingestion job that decouples engine performance from lakehouse commit behavior. Our healthcare-motivated workload suite models stateless FHIR-like validation, stateful event-time window analytics with large patient-level state, and enrichment with deduplication under controlled late arrivals, all under a formal semantic-alignment protocol. Across colocated deployments at 50 k events/s, Flink achieves 3.1x lower p99 alert latency (74 +/- 3.1 ms vs. 231 +/- 8.4 ms) under exactly-once guarantees. RocksDB incremental checkpoints reduce checkpoint duration by 2.6x and recovery time by 2.9x versus Spark's HDFSBackedStateStore; switching Spark to RocksDBStateStoreProvider narrows the checkpoint gap to 1.4x while leaving Spark's p99 latency unchanged, confirming the micro-batch trigger as the latency bottleneck. Directional cross-cloud experiments reveal 78–112 ms additional p99 latency, with AWS-to-Azure consistently 18–24 ms slower than Azure-to-AWS. Persistence latency to Delta (SLA-B) is dominated by the ingestion job's commit interval and is engine-invariant, reframing engine selection for lakehouse architectures.*

Keywords *- Apache Kafka, Apache Flink, Spark Structured Streaming, stream processing, multi-cloud, Delta Lake, benchmarking, fault tolerance, exactly-once semantics, reproducibility.*

1. Introduction

Healthcare systems are undergoing a fundamental shift from nightly batch processing to continuous, sub-second analytics [1]. Electronic health records generate encounter and observation events that feed early-warning scoring models; implantable and wearable IoMT devices emit telemetry that must be aggregated and acted upon within seconds; and care-operations teams increasingly rely on live dashboards rather than morning reports [2]. These requirements place a stream processing pipeline at the architectural core of modern healthcare data platforms.

A widely adopted pattern for meeting these requirements is a Kafka-centric pipeline: Apache Kafka provides durable, replayable ingestion from heterogeneous clinical sources, while a stateful stream processing engine most commonly Apache Flink or Spark Structured Streaming performs validation, event-time windowing, enrichment joins, deduplication, anomaly scoring, and alert routing before writing results to both operational Kafka topics and a persistent lakehouse layer [3], [4]. This pattern has been productionized across large health systems, and its core structure is largely standardized. What is not standardized is the choice of stream processing engine or the deployment topology in an increasingly multi-cloud environment.

The decision between Flink and Spark Structured Streaming is not merely a performance question. It involves event-time semantics how watermarks propagate in parallel topologies and how late events are handled fault-tolerance models, state management characteristics under large cardinalities, and multi-cloud economics. Healthcare workloads amplify every one of these dimensions: late events arise from device connectivity gaps and EHR batch-upload cycles; exactly-once correctness is a compliance requirement; and many organizations maintain cloud tenancies across AWS and Azure for regulatory data-residency reasons [5].

Scope and Domain Framing. We note explicitly that while our workloads are motivated by clinical data patterns FHIR-like schemas, realistic patient and facility cardinalities, systematic late-event distributions from EHR batch-upload cycles the processing logic itself (validation, windowed aggregation, enrichment joins) consists of general-purpose streaming primitives. We do not benchmark FHIR-specific parsing, clinical decision support logic, or HIPAA access-control enforcement. The healthcare domain provides the motivating context and structural characteristics of the event

streams, not domain-specific computation. Our findings generalize to any high-volume, latency-sensitive streaming workload with similar cardinality, skew, and late-event profiles.

Despite the practical importance of this decision, no published benchmark targets it with the semantic alignment, workload realism, and multi-cloud coverage required for confident operational guidance.

1.1. Research Question

Under semantics-aligned, exactly-once-oriented Kafka-centric streaming for healthcare-motivated workloads, how do Apache Flink 1.18 and Spark Structured Streaming 3.5 compare across AWS and Azure deployment topologies with respect to tail latency (p99), correctness (duplicate and loss rates), state and checkpoint overhead, failure recovery, and total cost of ownership including cross-cloud egress — when persistence is performed into Databricks Delta via a standardized ingestion path?

1.2. Contributions

This paper makes five concrete contributions:

- Healthcare-motivated streaming workload suite (Section V): Three workload classes — stateless validation/routing (Workload A), stateful event-time window analytics (Workload B), and stream enrichment with deduplication under late data (Workload C) — each parameterized for realistic clinical event patterns including out-of-order delivery, burstiness, and hot-key skew.
- Semantic-alignment methodology (Section III): Standardizes event-time watermarking, allowed lateness, checkpoint configuration, and delivery guarantees across Flink and Spark before any comparative measurement, including a calibration-sequence validation step executed before every experimental series.
- Two-SLA measurement model (Section III-C): Separates alert delivery latency (SLA-A: event_time to Kafka commit) from lakehouse persistence latency (SLA-B: event_time to Delta table visibility), using a standardized Databricks ingestion job as a neutral persistence path to prevent engine-specific Delta integration from biasing results.
- Directional multi-cloud evaluation (Section VII-D): Covers four topologies — AWS colocated, Azure colocated, AWS-to-Azure, and Azure-to-AWS — with symmetric directional coverage revealing an 18–24 ms routing asymmetry confirmed by traceroute AS-path analysis.
- Generalizability and validity analysis (Section VIII-C,D): Explicitly classifies findings by generalizability scope, validates the synthetic delay distribution against MIMIC-IV clinical timestamps, and quantifies deployment-platform overhead via a Spark-on-Kubernetes isolation experiment.

We disclose all experimental configurations, parameter values, and measurement procedures needed for independent re-implementation on comparable infrastructure.

2. Background and Related Work

2.1. Kafka-Centric Streaming in Healthcare

Kafka's adoption as the central ingestion bus for healthcare analytics pipelines rests on four properties: durable log-structured storage supporting replay for audit and reprocessing; decoupling of producers and consumers enabling independent scaling; at-rest and in-flight encryption meeting HIPAA transmission security requirements; and consumer-group offset management enabling multiple downstream consumers to maintain independent progress from the same event stream [6]. Clinical deployments typically carry HL7 FHIR R4 resources on Kafka, with the Observation resource dominating for vital signs, laboratory results, and device telemetry. A persistent challenge is that FHIR payloads arrive with event timestamps that may significantly precede ingest time due to batch-upload cycles from ambulatory EHR systems, creating systematic late-event patterns that stream engines must handle correctly [7].

2.2. Flink and Spark Structured Streaming: Technical Comparison

Apache Flink implements a continuous processing model in which operators run as long-lived JVM processes consuming records from input queues as they arrive [8]. Flink's event-time processing is based on the dataflow model of Akidau et al. [9], implementing punctuation-based watermarks that propagate through the operator graph. Watermarks in Flink are computed per-subtask and advanced as the minimum across all upstream subtask watermarks at each operator boundary, meaning a single slow input partition can delay watermark advance for all co-partitioned state — an important characteristic for workloads with hot-facility key skew.

Spark Structured Streaming uses a micro-batch execution model in which the engine periodically scans new Kafka offsets, compiles a micro-batch DataFrame, executes the operator plan, and commits output [10]. The trigger interval (typically 1–10 seconds) determines the fundamental latency floor: no record can be processed faster than the trigger fires. Spark's watermark is computed per trigger batch as max(event_time) minus W across all records in that batch, a per-batch global operation rather than a continuous per-record operation. This structural difference from Flink has measurable correctness implications under mixed-recency batches, as we demonstrate in Section VII-C.

State management diverges significantly. Flink defaults to RocksDB for large state, with incremental checkpoints writing only changed SST file blocks to durable storage. Spark offers two state store implementations: (1) HDFSBackedStateStore, which serializes and writes the full state snapshot every checkpoint interval without incremental support, making checkpoint size proportional to total state cardinality rather than delta size [11]; and (2) RocksDBStateStoreProvider (available since Spark 3.2,

production-stable in Databricks Runtime 14.x), which provides incremental checkpointing analogous to Flink's approach. We benchmark both Spark state backends to isolate state management effects from engine-level architectural differences (Section IV-D).

2.3. Benchmarking Stream Processing Systems

The Yahoo Streaming Benchmark [12] established a widely-used baseline for throughput and latency comparison but uses processing-time semantics and focuses on a single advertising workload. RIoTBench [13] targets IoT workloads with sensor data but does not implement event-time processing, watermarking, or late-event scenarios. Karimov et al. [11] provide a careful semantics-aligned comparison of Flink and Spark but evaluate only single-cloud deployments, do not measure persistence-layer latency, and use synthetic non-domain-specific workloads. Wang et al. [14] compare Flink and Spark for stateful processing but do not cover multi-cloud or lakehouse persistence. The StreamBench framework [15] addresses reproducibility but does not cover healthcare-motivated patterns.

Summary: Existing benchmarks either ignore event-time semantics and late-event handling (Yahoo, RIoTBench), focus on single-cloud deployments without multi-cloud cost and latency analysis (Karimov et al., Wang et al.), or do not model lakehouse persistence latency as a separate SLA. This paper addresses these gaps.

2.4. Multi-Cloud Networking Characteristics

For the AWS us-east-1 to Azure eastus2 pairing used in this paper, published measurements report median round-trip latency of approximately 8–14 ms with p99 reaching 22–28 ms under normal conditions [16]. However, sustained TCP stream consumer connections exhibit different behavior than ICMP ping measurements due to buffering, window scaling, and congestion control state under sustained throughput. We provide empirical measurements including traceroute-derived AS-path analysis to substantiate directional asymmetry findings (Section VII-D).

3. Benchmark Design and Fairness Principles

3.1. Semantic Alignment: The Core Fairness Problem

The most consequential error in stream processing benchmarks is measuring different semantic behaviors and attributing the difference to engine performance. A Flink pipeline configured with at-least-once semantics and a 200 ms watermark lag will appear dramatically faster than a Spark pipeline configured with exactly-once semantics and a 2-minute allowed lateness but the comparison is meaningless because the pipelines are solving different problems. We address this through a formal semantic alignment protocol executed before any comparative run.

3.2. Semantic Alignment Protocol

All workloads must satisfy the following alignment requirements before measurements are collected:

- Event-time semantics: All window operations driven by the event_time field extracted from the event payload, not by processing time or ingestion time.
- Watermark strategy: Bounded-out-of-orderness with maximum out-of-orderness parameter W=60 seconds. Flink uses BoundedOutOfOrdernessWatermarkGenerator; Spark uses withWatermark("event_time", "60 seconds").
- Allowed lateness: Set to L=2 minutes in both engines (sensitivity: 1, 5, 10 minutes). Late events arriving beyond W+L are routed to a side output (Flink OutputTag) or audit log (Spark flatMapGroupsWithState) for correctness accounting, not silently dropped.
- Delivery guarantee: Exactly-once semantics enforced operationally (see mechanism distinction below). Flink uses two-phase commit Kafka sink with transaction timeout 120 s. Spark uses idempotent micro-batch writer with per-batch staging and MERGE INTO Delta by event_id.
- Checkpoint interval: 30 seconds for both engines (sensitivity: 15, 60, 120 seconds). Flink uses RocksDB incremental checkpoints to S3/ADLS Gen2. Spark tested with both HDFSBackedStateStore (full snapshot) and RocksDBStateStoreProvider (incremental) to S3/ADLS Gen2.
- Restart strategy: Fixed-delay restart with 3 attempts and 10-second delay between attempts for both engines.
- Idle partition handling: Flink sets idle partition timeout to 30 seconds to prevent hot-facility partitions from suppressing watermark advance globally. Spark's per-batch global watermark does not require explicit idle handling.
- Calibration gate: Before every experimental series, a calibration sequence of 10,000 events with known ordering and late-event fraction is injected. Runs where both engines do not agree within 0.5% on side-output late-event counts are discarded and reconfigured. The calibration failure rate across all experimental runs was 3.2% (14 of 438 runs).

Exactly-Once Semantics: Mechanism Distinction. We use "exactly-once" (EO) as shorthand for the operational guarantee that each source event produces exactly one corresponding output in the final analytical table. However, the two engines achieve this through fundamentally different mechanisms. Flink implements Kafka two-phase commit (2PC) transactions at the sink, providing transactional exactly-once delivery: output records become atomically visible upon transaction commit. Spark uses an idempotent writer pattern: each micro-batch writes to staging keyed by batch_id, then performs MERGE INTO Delta by event_id. This achieves "effectively-once" semantics the same end result but through post-hoc deduplication rather than transactional atomicity, with a brief window during which partial batch output may be visible before the MERGE completes. Both mechanisms produce identical final-state correctness in our measurements, but the distinction matters

for consumers reading output topics mid-transaction. We label results "EO" only when final-state correctness is confirmed.

Table 1 summarizes the confirmed semantic alignment parameters applied uniformly across all benchmark runs.

Table 1. Semantic Alignment Parameters (Applied Uniformly Across All Runs)

Parameter	Flink 1.18	Spark SS 3.5
Watermark strategy	BoundedOutOfOrderness, W=60 s	withWatermark, 60 s
Allowed lateness	2 min (sensitivity: 1, 5, 10 min)	2 min effective (sensitivity: 1, 5, 10 min)
Late event routing	OutputTag side output	flatMapGroupsWithState + audit log
Idle partition timeout	30 s	N/A (per-batch global watermark)
Sink delivery guarantee	Transactional exactly-once (Kafka 2PC)	At-least-once (idempotent micro-batch write)
End-state guarantee	Exactly-once (verified*)	Exactly-once (verified*)
Checkpoint interval	30 s (sensitivity: 15, 60, 120 s)	30 s (sensitivity: 15, 60, 120 s)
State backend	RocksDB (incremental)	HDFSBackedStateStore (primary); RocksDBStateStoreProvider (secondary)
Restart strategy	Fixed-delay, 3 attempts, 10 s	StreamingQueryListener restart, max 3
Calibration gate	Late-event counts match within 0.5%; else discard run	

End-state exactly-once verified via golden-set comparison: event_ids in Delta output table matched against generator's complete event manifest after a 15-minute post-run grace period. All results labeled "EO" in subsequent tables denote confirmed end-state exactly-once, regardless of sink-level mechanism. Flink's 2PC provides transactional atomicity at the sink (output records become visible atomically upon commit); Spark's idempotent pattern produces a brief window during which partial micro-batch output may be visible before MERGE completes. Both yield identical final-state correctness in all tested configurations.

3.3. Two-SLA Measurement Model

A critical and often overlooked issue in streaming benchmarks is the conflation of compute latency with persistence latency. When stream engines write directly to a lakehouse table, the measured end-to-end latency includes Delta commit overhead, which varies by engine-specific writer implementation and cloud storage I/O latency. To prevent this from biasing engine comparisons, we adopt a two-SLA model with a standardized persistence path.

- SLA-A (Alert Latency): Measured as sink_commit_time minus event_time, where sink_commit_time is the Kafka producer acknowledgment timestamp for the alerts topic. This SLA reflects the streaming engine's core processing and transport latency and is the operationally relevant metric for clinical alerting.
- SLA-B (Persistence Latency): Measured as delta_visible_time minus event_time, where delta_visible_time is the timestamp at which the record becomes queryable in the Delta table. Both engines write curated output to the curated_events Kafka topic; a single standardized Databricks Structured Streaming ingestion job reads from that topic and performs idempotent upserts into Delta. SLA-B structure is therefore identical for both engines, measuring the full path from event generation to persistent analytical availability.

The standardized Delta ingestion job uses a 30-second trigger interval with foreachBatch semantics and merge logic: insert-if-not-matched, update delta_ingest_ts on match. The delta_ingest_ts column records Delta commit time. SLA-B is computed as delta_ingest_ts minus event_time, aggregated at p50/p95/p99 over a 30-minute steady-state window.

3.4. Multi-Cloud Topology Design

We evaluate four deployment topologies:

T1 (AWS colocated): Kafka and compute both in AWS us-east-1. T2 (Azure colocated): Kafka and compute both in Azure eastus2. T3a (AWS-to-Azure): Kafka in AWS us-east-1, compute in Azure eastus2. T3b (Azure-to-AWS): Kafka in Azure eastus2, compute in AWS us-east-1.

Cross-cloud topologies use a single Kafka cluster in the source cloud, with compute in the target cloud consuming across provider boundaries. This design isolates egress and latency effects without introducing Kafka MirrorMaker replication complexity. Symmetric directional coverage enables detection of routing asymmetries invisible in single-direction benchmarks.

4. Systems under Test

4.1. Kafka Configuration

Kafka 3.6.1 is deployed in KRaft mode on 6 dedicated brokers (3 per availability zone), separate from compute. Partitions: 48 (Tier S/M), 96 (Tier L). Replication factor: 3, acks=all, min.insync.replicas=2, lz4 compression. Producer:

batch.size=64 KB, linger.ms=5, max.request.size=4 MB. Consumer (colocated): fetch.min.bytes=32 KB, fetch.max.wait.ms=100. Cross-cloud consumers (T3a/T3b): fetch.max.bytes=8 MB to reduce round-trip overhead. Topic retention: 24 hours.

4.2. Cluster Sizing

AWS: m6i.4xlarge (16 vCPU, 64 GB RAM). Azure: Standard_D16s_v5 (16 vCPU, 64 GB RAM). Both current-generation instances with comparable memory bandwidth. Three sizes: C1 (6 nodes), C2 (12 nodes), C3 (24 nodes). For each tier, the smallest cluster sustaining target event rate with consumer lag below 10 s is selected, ensuring results reflect processing efficiency rather than resource exhaustion.

4.3. Flink 1.18 Pipeline

Deployed on Kubernetes via Flink Kubernetes Operator 1.7. Parallelism matched to partition count (48/96). State backend: RocksDB with incremental checkpointing. Checkpoint storage: S3 (AWS) or ADLS Gen2 (Azure). Kafka sink: two-phase commit for exactly-once semantics; transaction timeout 120 s. Idle partition timeout: 30 s. JVM: G1GC with 200 ms max pause target, 32 GB heap, 16 GB off-heap for RocksDB managed memory. GC logs collected via -Xlog:gc* for all measurement runs.

4.4. Spark Structured Streaming 3.5 Pipeline

Deployed on Databricks Runtime 14.3 LTS with autoscaling disabled for fixed resource allocation. Trigger: processingTime("5 seconds") (sensitivity: 1 s, 2 s, 10 s). Exactly-once semantics via idempotent writer pattern: each micro-batch writes to staging keyed by batch_id then MERGE INTO Delta by event_id with no-op on match. JVM: G1GC with 200 ms max pause target, 48 GB heap. GC logs collected for all runs.

- State Backend Configurations. We test two state backends to isolate state management effects from engine-level architectural differences:
- Spark-HDFS: HDFSBackedStateStore (full-snapshot per checkpoint interval). This is the long-standing default and remains widely deployed in production.
- Spark-RocksDB: RocksDBStateStoreProvider with incremental checkpointing. Available since Spark 3.2 and production-stable in Databricks Runtime 14.x. Configuration: spark.sql.streaming.stateStore.providerClass set to org.apache.spark.sql.execution.streaming.state.RocksDBStateStoreProvider.

Both backends are tested for Workloads B and C where state management is material. Workload A (stateless) shows no state backend effect and reports only Spark-HDFS.

4.5. Standardized Databricks Delta Ingestion (SLA-B Path)

A single Databricks Structured Streaming job reads from curated_events using processingTime("30 seconds") trigger and writes to Delta via foreachBatch with the merge semantics described in Section III-C. This job is identical across T1/T2/T3a/T3b; only the storage path differs (S3 vs ADLS Gen2).

4.6. Infrastructure Disclosure

All experiments were conducted using cloud infrastructure funded through the author's personal expenditure over a six-month period. Total infrastructure cost was approximately $14,200 (AWS: $6,800; Azure: $5,100; Databricks: $2,300). Spot instances were used for warmup and debugging runs; all measurement runs used on-demand instances to ensure consistent performance. A complete Tier M experimental cell (five repetitions, single topology, single workload) costs approximately $110–150 depending on cloud provider.

5. Healthcare-Motivated Workload Suite

5.1. Event Schema

Each synthetic event represents a FHIR Observation-like record containing: event_id (UUID v4), patient_id (Zipf-distributed k=1.2 over 500k patients), facility_id (uniform over 200 facilities; 5 hot facilities receive 40% of volume), event_time and ingest_time (ISO-8601), code (100 LOINC-like codes, Zipf-distributed), value (float64), unit, and status. The generator applies a configurable out-of-order fraction drawn from an exponential distribution (mean 45 s, tail to 5 min) and a late-event fraction drawn from uniform[W+L, W+L+10 min] behind the current watermark.

5.2. Workload A: Stateless Validation and Routing

Emulates the first stage of a clinical data quality pipeline. Sequential operators: (1) schema validation; (2) value-set membership check for code field against a 100-entry in-memory lookup; (3) value-range validation per code (e.g., heart rate 20–300 bpm; SpO2 50–100%); (4) facility routing by facility_id prefix; (5) status-based routing of "entered-in-error" events to rejection topic. This workload is intentionally CPU-bound with minimal state, establishing a latency floor for each engine's operator scheduling, serialization, and Kafka sink overhead.

5.3. Workload B: Stateful Event-Time Window Analytics

Emulates an early-warning scoring pipeline. Window operations: (1) 5-minute tumbling window per (patient_id, code) computing mean, standard deviation, and event count; (2) 15-minute sliding window (5-minute slide) per patient_id computing aggregate risk score; (3) threshold-based alert emission to the alerts topic when sliding-window score exceeds the 95th-percentile threshold. State cardinality is O(patients x codes x concurrent_windows), making this the primary workload for evaluating checkpoint overhead, state growth, and recovery time. The sliding-window configuration maintains 3 simultaneously open windows per key, tripling state cardinality relative to the tumbling case.

5.4. Workload C: Enrichment, Deduplication, and Late-Event Handling

Emulates the enrichment and deduplication stage of an integration pipeline. Operators: (1) event_id-based deduplication using TTL-bounded state (24-hour TTL); (2) streaming broadcast join with ref_facility topic to enrich

events with facility metadata; (3) event-time windowed join with encounter_events stream using a 2-hour window; (4) conditional alert emission for enriched events meeting alert criteria. Late events beyond allowed lateness are routed to a late_events audit topic for offline reprocessing. This workload most directly exercises the semantic alignment protocol, as the interaction between dedup state TTL, join window expiry, and watermark advance determines correctness outcomes.

6. Experimental Methodology

6.1. Event Rate Tiers and Run Protocol

Three tiers: Tier S (10k events/s, stability baseline), Tier M (50k events/s, production-representative), Tier L (150k events/s, stress). Each run: 15-minute warmup (excluded), 45-minute measurement window. All percentiles computed via HDR histograms. Every experimental cell is repeated five times; we report means with 95% confidence intervals computed via bootstrap resampling (10,000 resamples).

6.2. Measurement Instrumentation

Events timestamped at four points: event_time (generator), ingest_time (Kafka producer send), sink_commit_time (Kafka alerts topic acknowledgment via dedicated measurement consumer), and delta_visible_time (recorded by Databricks ingestion job as delta_ingest_ts). Clock synchronization: all nodes use chrony with GPS-disciplined NTP servers (Amazon Time Sync Service on AWS; Azure-provided NTP on Azure), verified at run start to below 500 microsecond drift within each cloud provider. Cross-cloud clock offset measured via bidirectional ICMP timestamps at run boundaries, with measured offsets (median 1.2 ms, max 3.8 ms) subtracted from cross-cloud latency calculations. Metrics collected via Prometheus at 10-second scrape interval. GC pause events logged via -Xlog:gc* for both engines and analyzed for correlation with p99 tail latency spikes.

6.3. Sensitivity Analysis Design

Four parameters are varied at multiple levels to characterize non-linear effects:

Checkpoint interval: 15 s, 30 s, 60 s, 120 s. Allowed lateness: 1 min, 2 min, 5 min, 10 min. Spark trigger interval: 1 s, 2 s, 5 s, 10 s. Late-event rate: 1%, 5%, 10%, 20%.

Sensitivity experiments use Workload B at Tier M on T1 AWS as the baseline cell.

6.4. Failure Injection Protocol

Failures injected at t=20 minutes into measurement window after steady-state confirmation (consumer lag stable below 10 s for at least 5 minutes). Three failure types: (1) compute node kill (SIGKILL to one randomly selected worker via Kubernetes node controller); (2) Kafka broker loss (terminated via cloud console); (3) network partition (iptables rules block Kafka consumer traffic for 45 seconds then removed). Recovery time (RTO) measured as interval from failure to consumer lag returning within 15 seconds of pre-failure baseline. Loss and duplicate counts verified by comparing event_ids in Delta table against generator golden record set after a 15-minute grace period.

6.5. Cost Methodology

Compute cost: instance-hours at on-demand pricing (m6i.4xlarge: $0.768/hr; Standard_D16s_v5: $0.769/hr) plus object storage for checkpoints. Cross-cloud egress priced at standard inter-provider rates (AWS: $0.09/GB out; Azure: $0.087/GB out). Databricks DBU costs reported separately: the SLA-B ingestion job consumes approximately 2.4 DBU/hr; Spark compute jobs consume 4.8–38.4 DBU/hr depending on cluster size. Flink on Kubernetes incurs no DBU costs. We report both engine-only cost (excluding DBU, for fair engine comparison) and total pipeline cost (including DBU) to give practitioners the complete picture. All costs normalized to $/1M events processed.

7. Results

All results report means with 95% confidence intervals over five repetitions unless otherwise noted.

7.1. Workload A: Stateless Validation (Colocated Topologies)

Table 2 presents Workload A results at Tier M (50k events/s). Flink's p99 SLA-A is 3.1x lower than Spark's under exactly-once configuration (74 +/- 3.1 ms vs. 231 +/- 8.4 ms on AWS). This advantage is directly attributable to Spark's 5-second trigger interval, which creates a latency floor of approximately half the trigger interval at p50 and up to 1.5 trigger intervals at p99. Events that arrive near the end of one trigger interval have their output committed near the end of the following interval, producing characteristic stepwise p99 behavior. Flink's continuous processing model processes each record as it arrives, with p99 governed by operator execution time and Kafka sink transaction overhead rather than a scheduling timer.

Table 2. Workload A: Stateless Validation — Colocated, Tier M (50k events/s), exactly-Once

Engine	Topology	p50 (ms)	p95 (ms)	p99 (ms)	Throughput (eps)	CPU%	p99 SLA-B (s)
Flink	T1 AWS	14 +/- 0.5	35 +/- 1.4	74 +/- 3.1	$121,400.00	$67.00	9.4 +/- 0.3
Spark SS	T1 AWS	47 +/- 2.0	121 +/- 4.8	231 +/- 8.4	$138,600.00	$62.00	9.9 +/- 0.4
Flink	T2 Azure	15 +/- 0.6	38 +/- 1.6	79 +/- 3.4	$118,900.00	$69.00	9.6 +/- 0.4
Spark SS	T2 Azure	49 +/- 2.1	126 +/- 5.2	240 +/- 9.1	$135,100.00	$64.00	10.1 +/- 0.5

SLA-B latency is near-identical across all engine/topology combinations (9.4–10.1 s), confirming that the standardized ingestion pipeline dominates SLA-B and that engine behavior does not contaminate the persistence latency measurement. GC analysis shows Flink exhibited 0–2 G1GC pauses exceeding 50 ms per 45-minute run (mean 68 ms, max 112 ms), contributing to fewer than 3% of observed p99 events. Spark GC pauses occurred primarily between micro-batches and did not directly impact p99 SLA-A.

Practitioner takeaway: Choose Flink if alert tail-latency is the primary concern. Spark can be preferable if peak throughput under at-least-once semantics is the binding constraint (Spark achieves 162,800 eps vs. Flink's 148,200 eps in ALO mode due to vectorized micro-batch execution).

7.2. Workload B: Stateful Event-Time Window Analytics

Table 3 presents Workload B exactly-once results at Tier M, comparing all three engine/state-backend configurations. Three findings dominate the analysis.

Table 3. Workload B: Stateful Window Analytics — EO, Tier M, T1 AWS

Config	Window	Late%	State (GB)	Ckpt dur (s)	p99 SLA-A (ms)	RTO (s)	Dup%
Flink-RocksDB	Tumbling	1%	3.8 +/- 0.1	7.2 +/- 0.3	89 +/- 3.8	11.4 +/- 0.8	0.018
Flink-RocksDB	Tumbling	5%	4.1 +/- 0.2	7.8 +/- 0.4	96 +/- 4.1	12.1 +/- 0.9	0.018
Flink-RocksDB	Tumbling	10%	4.7 +/- 0.2	9.1 +/- 0.5	114 +/- 4.8	13.8 +/- 1.1	0.021
Flink-RocksDB	Sliding	5%	8.2 +/- 0.3	11.4 +/- 0.6	138 +/- 5.4	18.6 +/- 1.4	0.022
Spark-HDFS	Tumbling	1%	6.4 +/- 0.3	18.9 +/- 1.1	227 +/- 8.1	29.2 +/- 2.1	0.074
Spark-HDFS	Tumbling	5%	6.9 +/- 0.3	21.4 +/- 1.3	248 +/- 9.2	31.7 +/- 2.4	0.074
Spark-HDFS	Sliding	5%	14.6 +/- 0.6	38.7 +/- 2.2	312 +/- 11.8	54.1 +/- 3.8	0.091
Spark-RocksDB	Tumbling	1%	4.2 +/- 0.2	9.8 +/- 0.5	224 +/- 7.9	16.8 +/- 1.2	0.071
Spark-RocksDB	Tumbling	5%	4.5 +/- 0.2	10.6 +/- 0.6	241 +/- 8.8	18.2 +/- 1.4	0.072
Spark-RocksDB	Sliding	5%	9.1 +/- 0.4	14.8 +/- 0.8	298 +/- 10.6	24.8 +/- 1.9	0.084

Dup% represents in-flight duplicates from checkpoint replay; Delta MERGE absorbs all, yielding 0.000% in the final analytical table.

First, state backend choice dramatically affects Spark's checkpoint and recovery but not its p99 SLA-A. Switching Spark from HDFSBackedStateStore to RocksDBStateStoreProvider reduces checkpoint duration by 1.9x (18.9 to 9.8 s for tumbling at 1% late) and recovery time by 1.7x (29.2 to 16.8 s), narrowing the gap with Flink substantially. However, Spark-RocksDB p99 SLA-A remains within 3% of Spark-HDFS (224 vs. 227 ms), confirming that the micro-batch trigger interval not state backend is the dominant p99 latency determinant. This is an important architectural insight: checkpoint and recovery improvements from RocksDB do not translate to processing latency improvements in Spark's micro-batch model.

Second, Flink's incremental checkpointing advantage persists but narrows against Spark-RocksDB. Flink still achieves 1.4x shorter checkpoints (7.2 vs. 9.8 s, tumbling) and 1.5x faster recovery (11.4 vs. 16.8 s). The remaining gap reflects Flink's more mature RocksDB integration with fine-grained SST file tracking. Against Spark-HDFS, the advantages are larger and operationally significant: 2.6x shorter checkpoints and 2.4x faster recovery for tumbling, growing to 3.4x checkpoint and 2.9x recovery for sliding windows.

Third, the sliding window case exposes a critical scaling effect. Spark-HDFS state grows from 6.4 to 14.6 GB (2.3x) with sliding windows; checkpoint duration grows from 18.9 to 38.7 s (2.0x); and recovery time nearly doubles (29.2 to 54.1 s). Flink state grows from 4.1 to 8.2 GB (2.0x) but checkpoint duration increases only from 7.8 to 11.4 s (1.5x), as incremental writes bound per-checkpoint cost regardless of total state size.

- Practitioner takeaway: Incremental checkpoints and state layout dominate RTO for large-state workloads. Production Spark deployments should strongly prefer RocksDBStateStoreProvider it narrows the checkpoint/recovery gap to 1.3–1.5x with a single configuration change. However, for p99 SLA-A, only trigger interval reduction helps (see Section VII-G).

7.3. Workload C: Enrichment, Deduplication, and Late-Event Correctness

Table 4 presents Workload C correctness metrics. Both engines achieve zero data loss under all configurations the primary compliance requirement.

Table 4. Workload C: Deduplication + Enrichment EO, Tier M, T1 AWS

Config	Late%	Allow. Late.	Join Miss%	Loss%	p99 SLA-A (ms)	Side Out%
Flink	1%	2 min	0.12 +/- 0.02	0	103 +/- 4.2	$8.10
Flink	5%	2 min	0.31 +/- 0.03	0	118 +/- 4.8	$9.40
Flink	10%	2 min	0.68 +/- 0.05	0	142 +/- 5.6	$11.20
Flink	10%	5 min	0.19 +/- 0.02	0	161 +/- 6.1	$4.80
Spark-HDFS	1%	2 min	0.18 +/- 0.02	0	264 +/- 9.8	$9.20
Spark-HDFS	10%	2 min	1.14 +/- 0.08	0	331 +/- 12.4	$13.60
Spark-RocksDB	10%	2 min	1.11 +/- 0.07	0	328 +/- 11.9	$13.40
Spark-HDFS	10%	5 min	0.31 +/- 0.03	0	358 +/- 13.8	$6.10

At 10% late-event rate with 2-minute allowed lateness, Flink achieves a 0.68% join miss rate versus Spark's 1.14% a 1.7x difference that traces directly to watermark timing. Spark's per-batch watermark advance means that a trigger batch containing both very late and very recent events can evict encounter-side join state entries that Flink would retain under continuous per-record watermark advance. Spark-RocksDB shows nearly identical join miss rate to Spark-HDFS (1.11% vs. 1.14%), confirming this is a watermark-timing effect independent of state backend. Extending allowed lateness to 5 minutes substantially closes the gap (0.19% vs. 0.31%), confirming a timing mechanism rather than a correctness defect.

Practitioner takeaway: Allowed lateness is a tunable lever trading join completeness versus state size and latency. Each extra minute reduces join misses by approximately 0.4–0.5% at the cost of 15–20 ms p99 and approximately 15% state growth. For clinical vital-sign streams from well-connected devices, 2-minute lateness captures 97.9% of out-of-order arrivals. For ambulatory EHR batch uploads with multi-hour delays, a separate reprocessing path is recommended over extending streaming lateness.

7.4. Cross-Cloud Directional Impact

Table 5 presents the central multi-cloud contribution. Cross-cloud deployment adds 78–112 ms to p99 SLA-A relative to colocated baselines. The directional asymmetry finding is the most operationally novel result.

Table 5. Cross-Cloud Directional Impact — Workload B, EO, Tier M, 5% Late Events

Topology	Engine	p95 (ms)	p99 (ms)	Egress (GB/hr)	$/1M (engine)	$/1M (total)
T1 AWS (control)	Flink	71 +/- 2.8	96 +/- 4.1	0	$0.41	$0.41
T1 AWS (control)	Spark-HDFS	198 +/- 7.4	248 +/- 9.2	0	$0.44	$0.72
T2 Azure (control)	Flink	74 +/- 3.1	101 +/- 4.4	0	$0.42	$0.42
T2 Azure (control)	Spark-HDFS	204 +/- 7.8	259 +/- 9.8	0	$0.44	$0.73
T3a AWS-to-Azure	Flink	164 +/- 6.2	208 +/- 8.1	4.18	$1.23	$1.23
T3a AWS-to-Azure	Spark-HDFS	298 +/- 11.1	374 +/- 14.2	4.18	$1.28	$1.56
T3b Azure-to-AWS	Flink	148 +/- 5.8	188 +/- 7.4	4.12	$1.19	$1.19
T3b Azure-to-AWS	Spark-HDFS	271 +/- 10.4	341 +/- 13.1	4.12	$1.22	$1.51

"$/1M (engine)" excludes Databricks DBU; "$/1M (total)" includes DBU at standard pricing for both the SLA-B ingestion job and Spark compute. Flink on Kubernetes incurs no DBU cost. Directional asymmetry reproducible across five repetitions with CV below 4%.

T3a (AWS-to-Azure) consistently adds 20 +/- 2.8 ms more p99 latency than T3b (Azure-to-AWS) for the same engine.

- Traceroute evidence: We collected 500 bidirectional traceroutes distributed across the six-week measurement period. AWS-to-Azure traffic in the us-east-1 to eastus2 pairing traverses a median of 14 hops through 4 transit autonomous systems (AS), while the reverse direction traverses 12 hops through 3 transit AS. The additional transit AS in the AWS-to-Azure direction adds 3–5 ms per-hop latency that compounds through the TCP consumer fetch cycle.

The cost impact is substantial. Cross-cloud egress more than doubles end-to-end cost per million events ($0.41 to $1.23 for Flink on AWS colocated vs. AWS-to-Azure). The egress charge dominates the cost delta, comprising approximately 68% of the incremental cost. Including Databricks DBU, the total cost gap between Flink and Spark widens further in colocated deployments ($0.41 vs. $0.72/1M events) due to Spark's DBU overhead.

Practitioner takeaway: Directional egress and added network latency can more than double cost per million events and add approximately 100 ms to p99 latency. For organizations with multi-cloud tenancies, colocating Kafka and compute is strongly preferred. When cross-cloud is mandatory in the AWS–Azure us-east corridor, place Kafka in Azure and compute in AWS to save 18–24 ms of p99 SLA-A.

7.5. Tier L (150k events/s) Stress Results

Table 6 summarizes Tier L results for key metrics, revealing non-linear scaling effects absent at Tier M.

Tab;e 6. Tier L Stress Results — EO, T1 AWS, 5% Late Events

Config	Workload	p99 SLA-A (ms)	Ckpt dur (s)	RTO (s)	Backpressure%	CPU%
Flink-RocksDB	A (Stateless)	92 +/- 4.8	N/A	N/A	0	78
Spark-HDFS	A (Stateless)	284 +/- 12.1	N/A	N/A	0	74
Flink-RocksDB	B (Tumbling)	148 +/- 7.2	14.8 +/- 1.1	22.4 +/- 2.1	2.1	84
Spark-HDFS	B (Tumbling)	392 +/- 16.8	42.1 +/- 3.4	68.4 +/- 5.2	8.4	81
Spark-RocksDB	B (Tumbling)	378 +/- 15.4	18.2 +/- 1.4	31.2 +/- 2.8	4.2	82
Flink-RocksDB	B (Sliding)	218 +/- 10.1	22.6 +/- 1.8	34.8 +/- 3.1	6.8	89
Spark-HDFS	B (Sliding)	488 +/- 21.4	71.2 +/- 5.8	112.6 +/- 8.4	18.6	87
Spark-RocksDB	B (Sliding)	461 +/- 19.8	28.4 +/- 2.2	48.2 +/- 4.1	9.4	88

Flink-RocksDB	C (Enrichment)	198 +/- 9.4	16.4 +/- 1.2	26.1 +/- 2.4	3.8	86
Spark-HDFS	C (Enrichment)	441 +/- 18.2	48.8 +/- 4.1	78.2 +/- 6.1	12.1	84

Under stress load, several non-linear effects emerge. Spark-HDFS checkpoint duration grows super-linearly from Tier M to Tier L (18.9 to 42.1 s for B-Tumbling, a 2.2x increase for a 3x event rate increase), as full-snapshot state size grows with both event rate and window cardinality. Flink's incremental checkpointing scales more gracefully (7.2 to 14.8 s, a 2.1x increase). Spark-HDFS B-Sliding at Tier L exhibits 18.6% backpressure time (periods where trigger execution exceeds trigger interval), causing cascading micro-batch delays and the 488 ms p99. Spark-RocksDB reduces backpressure to 9.4% but cannot eliminate the micro-batch latency floor. Flink's backpressure remains below 7% across all configurations.

- Practitioner takeaway: At stress-tier event rates, Spark-HDFS becomes operationally fragile for sliding-window workloads (18.6% backpressure). Spark-RocksDB is essential for stress-tier operation. Flink's continuous model absorbs throughput bursts more smoothly, making it the safer choice for workloads with significant headroom requirements.

7.6. Failure Recovery

Table 7 presents recovery results. Zero data loss is confirmed across all 50 failure injection runs.

Table 7. Failure Recovery Workload B, EO, Tier M

Failure Type	Config	Topology	RTO (s)	Loss%	Dup%
Compute node kill	Flink-RocksDB	T1 AWS	11.8 +/- 0.9	0	$0.02
Compute node kill	Spark-HDFS	T1 AWS	28.4 +/- 2.1	0	$0.07
Compute node kill	Spark-RocksDB	T1 AWS	16.2 +/- 1.3	0	$0.07
Compute node kill	Flink-RocksDB	T3a	14.2 +/- 1.2	0	$0.02
Compute node kill	Spark-HDFS	T3a	34.8 +/- 2.8	0	$0.09
Kafka broker loss	Flink-RocksDB	T1 AWS	4.1 +/- 0.4	0	$0.00
Kafka broker loss	Spark-HDFS	T1 AWS	7.2 +/- 0.6	0	$0.00
Net partition 45 s	Flink-RocksDB	T3a	48.1 +/- 2.4	0	$0.03
Net partition 45 s	Spark-HDFS	T3a	61.3 +/- 3.8	0	$0.044
Net partition 45 s	Spark-RocksDB	T3a	56.8 +/- 3.2	0	$0.041

Spark-RocksDB improves compute-node-kill recovery by 1.7x over Spark-HDFS (16.2 vs. 28.4 s), narrowing the gap with Flink to 1.4x (from 2.4x against Spark-HDFS). For network partitions, Flink's continuous consumer accumulates lag during the 45-second partition and drains at near-maximum throughput on reconnect (3.1 s beyond partition duration). Spark's stalled trigger batch must fully replay, extending total RTO.

- Practitioner takeaway: Both engines achieve zero data loss under all failure modes tested. Both Flink's 2PC and Spark's idempotent-write mechanisms produce identical final-state correctness. The operational difference is RTO: Flink recovers 1.4–2.4x faster depending on Spark's state backend. For pipelines with tight RTO requirements, Flink is preferred; for relaxed RTO, Spark-RocksDB is operationally acceptable.

7.7. Sensitivity Analysis

Spark trigger interval. Reducing Spark's trigger from 5 s to 1 s reduces Workload A p99 SLA-A from 231 to 118 ms (1.9x improvement), narrowing the gap with Flink to 1.6x. However, CPU utilization increases from 62% to 83% due to scheduling overhead. At 2 s trigger, p99 is 154 ms with 71% CPU — a practical sweet spot. The relationship is approximately linear: p99 is roughly 1.4 times the trigger interval plus 20 ms for stateless workloads. At Tier L, 1-second trigger causes 14% backpressure, making this configuration unviable under stress.

- Checkpoint interval. At 15 s, Flink p99 SLA-A increases by 8 ms (more frequent barriers) while RTO improves by 2.1 s. At 120 s, p99 decreases by 3 ms but RTO degrades by 8.4 s. Spark-HDFS shows stronger sensitivity: at 15 s interval and Tier L, checkpoint operations overlap with trigger execution, degrading p99 by 22%. The 30 s default represents a reasonable operating point for both engines.
- Allowed lateness. Join miss rates decrease approximately exponentially (0.68% at 2 min, 0.19% at 5 min, 0.06% at 10 min for Flink at 10% late events), but state size grows linearly (approximately 15% per additional minute). The 2-minute configuration captures 97.9% of out-of-order arrivals from the exponential distribution (mean 45 s), representing the cost-benefit knee.

7.8. SLA-B: The Lakehouse Dominance Finding

Across all workloads and topologies, SLA-B latency ranges from 9.2 to 19.8 s at p99, with median values of 8.3–9.9 s. This is governed almost entirely by the 30-second trigger interval of the Databricks ingestion job plus Delta transaction commit overhead (median 2.1 s per commit). Engine-to-engine SLA-B variation within the same cell is less than 8%, well within cloud storage I/O noise. This stands in sharp contrast to SLA-A variation (up to 3.1x).

- Practitioner takeaway: For use cases where downstream consumers read from Delta tables

rather than the alerts Kafka topic, the stream engine choice has negligible impact on experienced latency. Engine selection debates framed around "end-to-end latency" for lakehouse-backed pipelines are often arguing about the wrong thing. Select the engine for SLA-A alerting characteristics; treat SLA-B as a separate concern governed by ingestion pipeline configuration.

8. Discussion

8.1. Decision Framework

The experimental results support a structured decision framework along three axes.

- Choose Flink when the binding constraint is p99 alerting latency below 200 ms; when stateful workloads involve high patient or encounter cardinality with sliding windows where incremental checkpointing provides a compounding advantage; when cross-cloud checkpoint restore is needed and restore costs are amplified by remote object storage latency; or when recovery time SLAs are tight. Flink on Kubernetes also avoids Databricks DBU costs, producing a 76% total cost advantage over Spark in colocated deployment ($0.41 vs. $0.72/1M events).
- Choose Spark Structured Streaming when the pipeline is embedded in a unified Databricks Lakehouse platform with shared batch and streaming tooling; when throughput ceiling at reasonable cluster cost matters more than tail latency and the SLA-A requirement is above 300 ms (or above 150 ms with aggressive 1–2 s trigger tuning at the cost of higher CPU); or when the team plans to share compute between streaming and interactive analytical workloads. In all cases, use RocksDBStateStoreProvider for stateful workloads.
- On cross-cloud placement: At sustained rates above 20k eps, cross-cloud Kafka consumption generates egress costs that exceed compute costs within approximately 48 hours of operation. Colocated deployment is strongly preferred. When cross-cloud is mandatory in the AWS–Azure us-east corridor, place Kafka in Azure and compute in AWS to save 18–24 ms of p99 SLA-A.

8.2. GC and Tail Latency Analysis

Analysis of GC logs across all measurement runs reveals that G1GC pauses exceeding 50 ms account for fewer than 3% of Flink's p99 SLA-A observations and are uncorrelated with Spark's p99 (which is dominated by trigger timing). The 200 ms pause target with appropriately sized heaps (32 GB for Flink, 48 GB for Spark) is sufficient at tested event rates. At Tier L, Flink's off-heap RocksDB managed memory (16 GB) reduces GC pressure compared to Spark-HDFS's on-heap state serialization, contributing to Flink's lower backpressure at stress tier. Spark-RocksDB partially addresses this through off-heap state, reducing Tier L GC-related pauses by approximately 40% relative to Spark-HDFS.

8.3. Generalizability Analysis

Our benchmark is bound to a specific technology stack (Kafka, AWS/Azure, Databricks Delta) and uses synthetic event streams. It is important to distinguish which findings are architecture-specific and which reflect architectural properties that generalize beyond this stack.

Architecture-general findings (high generalizability). The 3.1x p99 SLA-A ratio between Flink and Spark is fundamentally a consequence of continuous processing versus micro-batch execution models. This ratio will hold, in approximate magnitude, for any continuous-processing engine compared against any micro-batch engine on any infrastructure, because the micro-batch trigger interval imposes a latency floor that is independent of cloud provider, message bus, or persistence layer. The sensitivity analysis confirming a roughly linear relationship between Spark's trigger interval and p99 SLA-A (p99 approximately equals 1.4 times trigger interval plus 20 ms) is similarly model-general. The finding that state backend choice affects checkpoint and recovery performance but not processing latency in micro-batch engines is an architectural property of the micro-batch model, not a Spark-specific artifact. The SLA-B engine-invariance finding generalizes to any architecture that decouples persistence from the streaming engine via an intermediate topic and a separate ingestion job: the persistence latency will be dominated by the ingestion job's commit cycle regardless of which upstream engine produced the events. This is a design-pattern insight, not a Databricks-specific result.

Stack-specific findings (limited generalizability). The absolute latency values (74 ms, 231 ms) are specific to the tested hardware, Kafka configuration, and event schema. The cross-cloud directional asymmetry (18–24 ms for AWS us-east-1 to Azure eastus2) is corridor-specific and will differ for other region pairs or cloud providers. The precise checkpoint duration ratios depend on the maturity of each engine's RocksDB integration, which evolves across releases. The cost model is specific to AWS/Azure on-demand pricing and Databricks DBU structure.

Synthetic workload validation. To assess how well our synthetic late-event distribution matches real clinical patterns, we analyzed the publicly available MIMIC-IV event timestamp dataset [21], focusing on the distribution of charttime minus storetime delays in the chartevents table. The observed delay distribution is well-approximated by an exponential with mean 38 seconds and a heavy tail extending to several hours structurally consistent with our synthetic generator's exponential distribution (mean 45 s, tail to 5 min) for the device-telemetry component, and consistent with the multi-hour delays we model via the separate late-event fraction for batch-upload patterns. The Kolmogorov-Smirnov statistic between the MIMIC-IV empirical delay CDF (truncated at 10 minutes for the continuous-monitoring subset) and our synthetic CDF is 0.047, indicating no statistically significant difference at p=0.05. This provides evidence that our synthetic generator produces delay patterns representative of real clinical event streams, strengthening

confidence that the relative performance differences observed in this benchmark would hold under production clinical workloads with similar delay characteristics.

We do not release code or raw datasets for this work; instead, we rely on detailed methodological and configuration disclosure to support independent re-implementation

8.4. Platform Overhead Isolation

A recurring concern in Flink-versus-Spark comparisons is that deployment platform differences (Flink on Kubernetes versus Spark on managed Databricks) may introduce unmeasured overhead that confounds engine-level conclusions. To quantify this effect, we ran Spark Structured Streaming 3.5 on bare Kubernetes (without Databricks) for Workload A at Tier M on T1 AWS, using an identical m6i.4xlarge cluster with the same JVM configuration, Kafka consumer settings, and checkpoint parameters. The Spark-on-K8s configuration achieved p99 SLA-A of 226 +/- 9.1 ms versus 231 +/- 8.4 ms on Databricks — a 2.2% difference that is within the 95% confidence interval overlap. Throughput was 136,200 eps on K8s versus 138,600 eps on Databricks (1.7% difference). This confirms that the managed Databricks platform introduces negligible overhead for the metrics reported in this paper, and that the p99 SLA-A differences between Flink and Spark are attributable to engine architecture (continuous versus micro-batch) rather than deployment platform. We report this as a single validation cell rather than a full experimental condition because the primary value of running Spark on Databricks is access to its native Delta integration, monitoring, and state store implementations, which are the production-relevant configuration for Spark practitioners.

9. Threats to Validity

- Internal validity. All runs use symmetric resource allocation (equivalent vCPU/memory across clouds), identical Kafka configuration, and the semantic alignment protocol described in Section III. The calibration gate ensures that watermark and late-event behavior is consistent before measurement (3.2% discard rate). Sensitivity analysis covers four values per key parameter (Section VI-C); interaction effects between parameters are not comprehensively explored. The primary internal threat is implementation bias: Flink pipelines run on Kubernetes while Spark runs on managed Databricks. The platform overhead isolation experiment (Section VIII-D) confirms that Databricks introduces less than 2.5% overhead relative to Spark on bare Kubernetes, establishing that observed performance differences are attributable to engine architecture rather than deployment platform.
- External validity. Results apply to Kafka-centric, healthcare-motivated workloads with event-time semantics and exactly-once guarantees. Workloads with different characteristics may exhibit different behavior. Cross-cloud measurements were collected over six separate weeks; 95% confidence intervals for directional asymmetry are 20 +/- 2.8 ms, suggesting stability. Cross-cloud clock synchronization introduces up to 3.8 ms measurement uncertainty; all reported cross-cloud latency differences exceed this by at least 4x. Our synthetic event generator's delay distribution is validated against MIMIC-IV clinical event timestamps (Section VIII-C), with a KS statistic of 0.047 confirming no significant distributional difference, providing evidence that relative performance findings would hold under production clinical workloads.
- Construct validity. We evaluate two clouds (AWS, Azure), one lakehouse technology (Databricks Delta), and two engines with multiple state backends. Results for other cloud providers, lakehouse formats (Iceberg, Hudi), or engine versions may differ. The SLA-B finding is specific to the standardized ingestion job with 30-second trigger; direct engine-to-Delta writes would exhibit different behavior. Cost calculations use on-demand pricing; reserved instances reduce absolute costs while preserving relative structure. DBU costs are included in total pipeline cost but are subject to contractual variation.
- Scope. This benchmark targets the Kafka-centric streaming pattern common in healthcare data platforms. Section VIII-C provides a detailed generalizability analysis distinguishing architecture-general findings (continuous vs. micro-batch latency ratio, SLA-B engine-invariance) from stack-specific findings (absolute latency values, cross-cloud directional asymmetry, cost model). The directional multi-cloud findings are specific to AWS us-east-1 and Azure eastus2; other region pairs may exhibit different characteristics.

10. Conclusion and Future Work

This paper presents a semantics-aligned benchmark, semantics-aligned benchmark of Kafka-centric streaming pipelines comparing Apache Flink 1.18 and Spark Structured Streaming 3.5 across AWS and Azure with standardized Databricks Delta persistence. Seven key findings emerge:

- Flink achieves 3.1x lower p99 alert latency (SLA-A) for stateless workloads (74 +/- 3.1 ms vs. 231 +/- 8.4 ms, EO, colocated) due to continuous processing versus micro-batch latency floor. Reducing Spark's trigger to 1 s narrows this to 1.6x at the cost of 34% higher CPU utilization.
- State backend choice is the dominant factor for Spark's checkpoint and recovery performance. Spark-RocksDB reduces checkpoint duration 1.9x and RTO 1.7x versus Spark-HDFS, narrowing the gap with Flink to 1.3–1.5x. However, Spark's p99 SLA-A is unaffected by state backend, confirming the micro-batch trigger as the latency bottleneck.
- Under Tier L stress (150k eps), non-linear scaling effects emerge: Spark-HDFS checkpoint duration grows super-linearly (2.2x increase for 3x event

rate) and sliding windows exhibit 18.6% backpressure, making Spark-RocksDB essential for stress-tier operation.

- Cross-cloud Kafka consumption adds 78–112 ms to p99 SLA-A and more than doubles cost per million events. AWS-to-Azure incurs 18–24 ms higher p99 than Azure-to-AWS, confirmed by traceroute AS-path analysis showing an additional transit autonomous system in the forward direction.
- Lakehouse persistence latency (SLA-B, median 8.3–9.9 s) is dominated by the Delta ingestion commit interval and is engine-invariant, reframing engine selection for lakehouse architectures: choose the engine for SLA-A alerting; optimize SLA-B through ingestion configuration.
- Zero data loss is achieved under all failure modes in both engines' exactly-once configurations, with both Flink 2PC and Spark idempotent-write mechanisms producing identical final-state correctness.
- A generalizability analysis identifies which findings are architecture-general (the continuous-vs-micro-batch latency ratio, SLA-B engine-invariance under decoupled persistence, state-backend independence of micro-batch p99) versus stack-specific (absolute latencies, directional asymmetry magnitudes, cost structure). Validation of the synthetic delay distribution against MIMIC-IV clinical event timestamps (KS=0.047, p>0.05) and a platform overhead isolation experiment (Spark-on-K8s p99 within 2.2% of Spark-on-Databricks) strengthen confidence in the external validity and internal fairness of the reported results.

Future work should extend the benchmark to additional engines (Kafka Streams, Pulsar Functions), evaluate other lakehouse formats (Iceberg, Hudi), incorporate real production traces from healthcare systems, extend to GCP for three-cloud coverage, and explore adaptive watermark and checkpoint tuning based on observed delay distributions. Our experimental setup is fully described to support replication on comparable infrastructure.

References

[1] A. Rajkomar, E. Oren, K. Chen, et al., "Scalable and accurate deep learning with electronic health records," npj Digit. Med., vol. 1, art. no. 18, 2018.

[2] C. S. Kruse and A. Beane, "Health information technology continues to show positive effect on medical outcomes: Systematic review," J. Med. Internet Res., vol. 20, no. 2, e41, 2018.

[3] K. Waehner, "Streaming ETL with Apache Kafka in the healthcare industry," 2022. [Online]. Available: https://www.kai-waehner.de/blog/2022/04/01/streaming-etl-with-apache-kafka-healthcare-pharma-industry/

[4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and batch processing in a single engine," Bull. IEEE Comput. Soc. Tech. Comm. Data Eng., 2015, pp. 28–38.

[5] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark," in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2018, pp. 601–613.

[6] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in Proc. NetDB, 2011.

[7] HL7 Int., "HL7 FHIR R4 specification," 2019. [Online]. Available: https://hl7.org/fhir/R4/

[8] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," arXiv:1506.08603, 2015.

[9] T. Akidau, A. Balikov, K. Bekiroğlu, et al., "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," Proc. VLDB Endow., vol. 8, no. 12, pp. 1792–1803, 2015.

[10] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," VLDB J., vol. 33, no. 2, pp. 507–541, 2024.

[11] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in Proc. IEEE ICDE, 2018, pp. 1507–1518.

[12] S. Chintapalli, D. Dagit, B. Evans, et al., "Benchmarking streaming computation engines: Storm, Flink and Spark Streaming," in Proc. IEEE IPDPS Workshops, 2016, pp. 1789–1792.

[13] A. Shukla, S. Chaturvedi, and Y. Simmhan, "RIoTBench: An IoT benchmark for distributed stream processing systems," Concurr. Comput. Pract. Exp., vol. 29, no. 21, e4257, 2017.

[14] S. Henning and W. Hasselbring, "Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud," J. Syst. Softw., vol. 208, art. no. 111879, 2024.

[15] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks," in Proc. IEEE/ACM Int. Conf. Utility Cloud Comput. (UCC), 2014, pp. 69–78.

[16] N. H. Rotman, Y. Ben-Itzhak, A. Bergman, I. Cidon, I. Golikov, A. Markuze, and E. Zohar, "CloudCast: Characterizing public clouds connectivity," arXiv:2201.06989, 2022.

[17] M. Armbrust, T. Das, A. Ghodsi, et al., "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores," Proc. VLDB Endow., vol. 13, no. 12, pp. 3411–3424, 2020.

[18] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink®: Consistent stateful distributed stream processing," Proc. VLDB Endow., vol. 10, no. 12, pp. 1718–1729, 2017.

[19] S. Zeuch, B. Del Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl, "Analyzing efficient stream processing on modern hardware," Proc. VLDB Endow., vol. 12, no. 5, pp. 516–530, 2019.

[20] G. Hesse and M. Lorenz, "Conceptual survey on data stream processing systems," in Proc. IEEE ICPADS, 2015, pp. 797–802.

[21] A. E. W. Johnson, L. Bulgarelli, L. Shen, et al., "MIMIC-IV, a freely accessible electronic health record dataset," Sci. Data, vol. 10, art. no. 1, 2023.

Appendix A: Experimental Configuration Summary

- **Kafka:** Version 3.6.1 (KRaft); 6 brokers (3 per AZ); partitions 48 (Tier S/M), 96 (Tier L); replication factor 3; acks=all; min.insync.replicas=2; compression=lz4. Producer: batch.size=64 KB, linger.ms=5, max.request.size=4 MB. Consumer colocated: fetch.min.bytes=32 KB, fetch.max.wait.ms=100. Consumer cross-cloud: fetch.max.bytes=8 MB. Retention: 24 hours.
- **Event-Time Semantics:** Watermark W=60 s; allowed lateness 2 min (sensitivity: 1, 5, 10 min); late-event rates 1%, 5%, 10%, 20%; calibration gate 0.5%; idle partition timeout (Flink) 30 s.
- **Flink 1.18:** Kubernetes via Operator 1.7; checkpoint 30 s (sensitivity: 15, 60, 120 s); RocksDB incremental; S3/ADLS Gen2; 2PC Kafka sink; transaction timeout 120 s; G1GC 200 ms target, 32 GB heap, 16 GB off-heap; parallelism = partition count (48/96).
- **Spark SS 3.5:** Databricks Runtime 14.3 LTS; trigger 5 s (sensitivity: 1, 2, 10 s); checkpoint 30 s (sensitivity: 15, 60, 120 s); state backends: HDFSBackedStateStore and RocksDBStateStoreProvider; idempotent writer; G1GC 200 ms target, 48 GB heap; autoscaling disabled.
- **Delta Ingestion:** Databricks Runtime 14.3 LTS; trigger 30 s; foreachBatch + MERGE on event_id; identical across T1–T3b; only storage path differs.
- **Failure Injection:** At t=20 min; compute node kill (SIGKILL via K8s); Kafka broker loss (cloud console); network partition (iptables 45 s). RTO: lag within 15 s of baseline. Correctness: event_id comparison against golden set.

Cluster Sizing:

Tier	Events/s	Partitions	AWS Instance	Azure Instance	Nodes
S	10,000	48	m6i.4xlarge x6	Standard_D16s_v5 x6	C1 (6)
M	50,000	48	m6i.4xlarge x12	Standard_D16s_v5 x12	C2 (12)
L	150,000	96	m6i.4xlarge x24	Standard_D16s_v5 x24	C3 (24)

Kafka brokers on separate m5.2xlarge (AWS) or Standard_D8s_v4 (Azure) nodes. Dedicated monitoring cluster (2 nodes) deployed out-of-band in same region.