International Journal of Computer Science Engineering Techniques – Volume 6 Issue 3, June - 2022

Real-Time Stream Processing with Apache Flink vs Spark Structured Streaming: An Enterprise Comparison

Kuladeep Sandra
Independent Researcher
kuladeepsandra90@gmail.com

Abstract*—Enterprises increasingly depend on real-time stream processing for fraud detection, operational analytics, recommendation, and anomaly detection. Apache Flink and Apache Spark Structured Streaming are the two dominant open-source options and represent meaningfully different architectural choices: Flink is a true streaming engine with single-event semantics and a continuous-execution model, while Spark Structured Streaming is a micro-batch system that unifies streaming with the broader Spark batch ecosystem. This paper presents a systematic comparison and grounds the discussion in five years of production operating experience with Spark Structured Streaming and Kafka in a banking and insurance environment, during which the Kafka topic count grew from 2 to approximately 340. The paper addresses three research questions: how the architectures differ and what those differences imply for latency, throughput, and state; what the practical operational challenges of running Spark Structured Streaming at enterprise scale are and where Flink would offer advantages; and how practitioners should choose between the two based on workload characteristics. The conclusion is that the choice is not purely technical: it depends on latency requirements, state complexity, organizational ecosystem, and team expertise. For the majority of enterprise use cases including the ones in our environment Spark Structured Streaming is the more pragmatic choice. For the minority that require sub-second latency or genuinely complex stateful processing, Flink is worth the steeper learning curve.

***Index Terms*—Apache Flink, Spark Structured Streaming, Kafka, latency, throughput, stateful processing, event time, windowing**

I. INTRODUCTION

Real-time data processing has moved from niche capability into mainstream enterprise requirement. Fraud detection needs to flag suspicious transactions within hundreds of milliseconds. Recommendation engines need to update on user activity within seconds. Operational analytics need to reflect business state as it changes. Anomaly detection on machine telemetry needs to catch problems before they propagate.

Apache Flink and Apache Spark Structured Streaming are the two dominant open-source options for building these systems. Both are mature, both have substantial production deployments, both integrate natively with Apache Kafka. The two are not interchangeable. They make different architectural choices that produce different latency profiles, different throughput characteristics, different operational behaviors, and different demands on the engineering teams that operate them.

The author has spent five years running production stream processing in a banking and insurance environment, during which the Kafka topic count grew from 2 to roughly 340. The streaming engine has been Spark Structured Streaming throughout, supporting real-time analytics, ETL, and operational alerting workloads across 6 business units. The lessons in this paper come from that operational experience, supplemented by literature on Flink's architecture and what-if analysis of cases where Flink would have been the better choice. We have not run Flink in production. We respect its technical architecture and recognize that some of our use cases would benefit from its design; we also recognize that switching engines is not a small undertaking and the case for doing so has to clear a high bar.

This paper addresses three research questions:

RQ1. How do Flink and Spark Structured Streaming architectures differ, and what are the implications for latency, throughput, and state management?

RQ2. What are the practical operational challenges of maintaining Spark Structured Streaming at enterprise scale, and where would Flink architecture offer advantages?

RQ3. How should practitioners choose between Flink and Spark Structured Streaming based on workload characteristics?

The paper is organized as follows. Section 2 covers background and related work. Section 3 presents the architectural comparison. Section 4 is the production case study. Section 5 examines Flink's advantages and when to choose it. Section 6 presents a tool selection framework. Section 7 concludes. The boundary conditions: focus on engines that unify batch and streaming; Kafka as the primary message broker; deployment context is on-premises plus Azure cloud.

II. BACKGROUND AND RELATED WORK

A. Evolution of Stream Processing

Stream processing has gone through three architectural generations. The first, exemplified by Apache Storm and S4, processed individual events through topologies of operators. Latency was low but exactly-once semantics were difficult. The second generation, represented by the original Spark Streaming DStream API and the early Samza, treated streams as sequences of small batches. The micro-batch model gave up some latency for stronger consistency and access to batch-engine optimizations. The third generation, beginning with Flink 1.0 and Spark Structured Streaming, has converged on a hybrid: a continuous logical model implemented either as true streaming (Flink) or as micro-batching (Spark).

The Dataflow model articulated by Akidau and colleagues [1] provided the conceptual framework both engines now share: streams are unbounded data, time has two distinct meanings (event time and processing time), and windowing is first-class. Apache Beam attempted to unify the API across engines.

B. Kafka as a Message Broker

Apache Kafka has become the de facto enterprise message broker. Its design as a distributed write-ahead log fits the requirements stream processing places on its message infrastructure: durable storage, multiple independent consumers, replay from arbitrary offsets, horizontal scalability through topic partitioning. Topic partitioning is the unit of parallelism: each partition is consumed by exactly one consumer within a group at a time.

In our environment, the topic count grew from 2 to approximately 340 over five years. The growth was not linear; it reflected expansion of streaming use cases across business units, decomposition of larger pipelines, and the addition of intermediate topics for stages benefiting from durability and replay. Managing 340 topics is qualitatively different from managing 2: naming conventions, schema evolution, consumer group coordination, and broker capacity planning all become first-class operational concerns.

C. Event Time vs. Processing Time

Event time is when an event occurred in the source system. Processing time is when the streaming system observes it. They differ by an amount that varies with network latency, broker buffering, consumer lag, and the inevitable delays of distributed systems. For most analytical workloads, event time is the correct basis for windowing.

Watermarks are the mechanism both engines use to reason about event time under out-of-order arrival. A watermark is an assertion that no events with earlier event times will subsequently arrive. Allowed lateness is a configurable grace period after the watermark passes a window's end. In our experience, watermark logic for a real business workload is almost always more subtle than documentation examples suggest. Most of the streaming bugs we have shipped to production have involved watermark logic in some form.

D. Stateful Stream Processing

Stateful processing is where architectural differences become most consequential. Flink's state backends include in-memory and RocksDB (embedded key-value store on local TaskManager disk). Flink uses Chandy-Lamport distributed snapshots for checkpoints [5]. Spark Structured Streaming's state stores are built on the broader Spark distributed cache infrastructure and were not originally designed for state much larger than fits in memory across executors. Recent versions improved this, but the architecture still favors Flink for very large keyed state.

E. Comparative Literature

Direct vendor-neutral comparisons of Flink and Spark Structured Streaming are scarcer than expected. Most published comparisons come from vendors of one system, emphasizing its strengths. The academic literature on Flink (the Stratosphere project, Flink checkpointing papers) is rigorous but does not generally compare directly to Spark. The Spark Structured Streaming paper is similarly self-focused. The gap this paper addresses is the operational experience of running one engine at enterprise scale over multiple years, including failure modes that do not appear in benchmarks and organizational considerations that determine viability.

III. ARCHITECTURAL COMPARISON

A. Flink Architecture

Flink executes a directed acyclic graph of operators continuously. Each operator instance processes events as they arrive, maintaining state on the TaskManager where it runs. Distributed snapshots taken via Chandy-Lamport provide exactly-once fault tolerance. State backends are local to TaskManagers in-memory for low latency or RocksDB for larger state. Single-event end-to-end latency can be under 100 milliseconds. Throughput is high but depends on state backend choice and checkpoint frequency.

B. Spark Structured Streaming Architecture

Spark Structured Streaming groups events into micro-batches, typically 100 to 500 milliseconds wide, and processes each micro-batch through the same engine that handles batch jobs. The Catalyst optimizer and Tungsten execution engine apply, which is a significant advantage because they have been heavily optimized over years of batch workload investment. Checkpoints are written to distributed storage (HDFS, S3, ADLS) after each micro-batch. State is managed via state stores backed by the distributed cache. End-to-end latency is typically 500 milliseconds to 5 seconds the micro-batch duration plus scheduler overhead plus sink write time.

C. Latency Trade-Off

Flink's true streaming model produces lower latency: 100 milliseconds to 1 second is achievable. Spark's micro-batch model produces 500 milliseconds to 5 seconds, more consistent but slower. The use case implications: Flink for fraud detection with sub-second SLAs, Spark for analytics dashboards and minute-level operational metrics. In our experience, Spark latency is acceptable for roughly 90 percent of use cases and the remaining 10 percent would benefit from Flink but we have not yet had a use case that justified the migration cost.

D. Throughput and Scalability

Both engines scale linearly with additional compute. Flink scales by adding TaskManagers; Spark by adding executors. Spark frequently has the throughput edge for very high-volume workloads above 100,000 events per second because it leverages batch optimization. The actual bottleneck in production is rarely the streaming engine itself; it is downstream database write capacity, slow sinks, network egress to the warehouse. Both engines handle 100,000-plus events per second comfortably. Beyond that, the differentiator is sink design, not engine choice.

E. State Management

Flink's RocksDB backend handles state into the tens or hundreds of gigabytes per TaskManager. Spark's state stores were not designed for state of that scale; performance degrades as state grows beyond what fits in cluster memory. For workloads with large keyed state long-running sessionization, complex deduplication windows, large materialized join state Flink is the better choice.

F. Fault Tolerance

Flink's distributed snapshots run periodically and recovery time depends on snapshot frequency and state size. Spark's per-micro-batch checkpoints write after each batch. Both provide exactly-once semantics within the engine. The end-to-end exactly-once guarantee depends on idempotent or transactional sinks in both cases. In our experience, checkpointing to S3 is about four to five times slower than checkpointing to local HDFS, which directly affects the latency-vs-durability trade-off and is a constraint on how aggressively the micro-batch window can be tuned downward.

IV. SPARK STRUCTURED STREAMING: PRODUCTION CASE STUDY

A. Context and Scale

The team is 30 data engineers across 6 business units in three time zones. The Kafka cluster has close to 340 topics, grown from 2 over five years. Roughly 50 active production streaming pipelines run continuously. Use cases span real-time fraud detection in banking (under-5-second SLA), customer event aggregation in insurance (hourly and daily windows), operational metrics across all units (1-minute granularity), and IoT sensor aggregation in manufacturing (30-second granularity). Infrastructure includes Kubernetes (32 nodes), an on-premises Hadoop cluster (42 nodes, historical), and roughly 4.8 million in cumulative platform investment.

B. Kafka Topic Growth and Management

Year 1 had 2 topics user events and transactions. Year 3 had 20, fragmented by business unit. Year 5 had 340, the explosion driven by data mesh adoption and per-domain topic ownership. Managing 340 topics introduced challenges that did not exist at 20: discovery (which topics exist, what their schema is, who owns them), governance (retention, access control, schema versioning), and monitoring (broker disk, consumer lag, rebalancing events). The lessons that worked: a topic naming convention enforced from day one (domain_entity_event), a mandatory schema registry (Confluent's), automated retention with topics older than 90 days archived to S3, and consumer lag monitoring as the leading indicator of downstream slowness.

C. Checkpointing Gotchas

Checkpoint storage durability. Initial deployment checkpointed to on-premises HDFS. When the Hadoop cluster had an outage, the streaming pipelines stuck because checkpoint recovery required reading from offline storage. We migrated to S3 as the primary checkpoint store, accepting the four-to-five-times slower write speed in exchange for independence from the Hadoop cluster. We mitigated the throughput cost by widening the micro-batch window from 100 milliseconds to 300 milliseconds, which amortized the checkpoint cost across more events. A separate two-hour outage occurred when HDFS filled because there was no automatic checkpoint cleanup; we now have a retention policy that deletes checkpoints older than 14 days.

Checkpoint frequency vs. latency. Checkpointing happens after each micro-batch. Too frequent, and checkpoint writes dominate latency. Too infrequent, and failure recovery loses recent events. We settled on a 300-millisecond micro-batch window with checkpoints every three batches (900 milliseconds), accepting roughly one second of potential data loss on failure. The right answer for any specific pipeline is determined by the business rules about acceptable data loss, which engineering should not decide unilaterally.

Idempotent sink requirement. Spark Structured Streaming provides at-least-once delivery to the sink, not exactly-once. A retried micro-batch can produce duplicate writes. The fix is idempotent sinks upserts or deduplication keyed on a deterministic primary key. Not all sinks support idempotence cleanly; some databases and Kafka topic writes do not. Our pattern is to add a deduplication micro-batch before the sink write, which costs roughly 3 to 5 percent throughput but eliminates the duplicate problem.

D. Micro-Batch Sizing and Output Format Impact

A subtler discovery: the micro-batch window is not just a latency knob. It affects the file output. We had an hourly aggregation pipeline writing to Parquet. With a 100-millisecond micro-batch window, the pipeline produced nearly 36,000 files per hour because Spark wrote one file per micro-batch output. The downstream consumers could not handle that file count.

The fixes we tried: widening the batch window to 30 minutes produced large files but lost granularity; calling repartition() before write added a shuffle and latency overhead; calling coalesce() was dangerous because it bottlenecked on a single executor. The solution we landed on was bucketing combined with scheduled compaction. The streaming pipeline writes to a temporary Parquet location with 10 partitions (1,000 files per hour). A nightly compaction job consolidates 24 hours of files into 10 final files. The trade-off is added complexity in the compaction pipeline, but it achieves a workable balance: reasonable file count at the analytics layer, and the streaming pipeline does not have to compromise its latency.

The lesson generalizes: micro-batch window tuning affects the entire downstream system, not just latency. File counts, storage layout, and downstream processing all interact with it.

E. Watermark and Event-Time Window Complexity

A business requirement: daily windows for per-user aggregations, cutoff at 23:59 UTC. The subtlety: what does end of day mean? Event time (when the event occurred) or processing time (when the event arrived)? The intuitive answer is processing time, and it is wrong because out-of-order arrivals would be miscounted. The right answer is event-time windowing with watermarks.

Our initial watermark was max(event_timestamp) - 2 hours, meaning events arriving up to two hours late were included in their event-time window and later events were dropped. The bug surfaced when the business team noticed missing events in the daily aggregations: some events were arriving more than two hours late and being silently dropped. We extended the watermark grace period to six hours, and the business accepted a six-hour delay before daily totals were finalized in exchange for completeness. The lesson is that event-time semantics are subtle and the watermark is fundamentally a business decision about the latency-vs-completeness trade-off, not an engineering decision about a configuration parameter.

F. Monitoring and Health Metrics

We use a three-pillar monitoring framework. Throughput: events per second per pipeline and per Kafka topic, plus events per micro-batch (which should be roughly constant). An alert fires if throughput falls below the threshold, signaling either a hung pipeline or a slow upstream consumer. Latency: end-to-end event-to-sink latency calculated as the difference between the Kafka timestamp and the sink write timestamp, plus per-micro-batch processing time. Alerts at the SLA boundary (under 5 seconds for fraud detection, under 5 minutes for analytics dashboards). Correctness: consumer lag (the gap between the offset being consumed and the latest offset published), failed micro-batch counts and error messages, and a periodic sample query against the sink to detect idempotence violations.

The dashboard is custom Grafana showing all three pillars per pipeline. The on-call escalation goes alert to Slack to on-call engineer. The framework is simple but it has caught essentially every production issue we have seen.

G. Results and Metrics

Throughput sustained at 50,000 to 100,000 events per second per pipeline. Latency in the 500-millisecond to 5-second range, meeting both the fraud detection SLA and the analytics SLA. Availability at 99.8 percent against a 99.9 percent target. Two data loss incidents in five years, both attributable to downstream sink failures rather than to Spark itself. The operational burden settled at nearly 0.5 engineer FTE for monitoring, tuning, and incident response across all 50 pipelines.

H. Failures and Lessons

Three notable failures shaped the operational practice. First, the HDFS checkpoint quota incident described in Section 4.3, which produced our retention policy. Second, the two-hour watermark bug from Section 4.5, which produced our practice of co-designing watermarks with the business team. Third, an idempotent-sink failure: the manual deduplication logic had an off-by-one error that produced duplicates after a recovery, and it survived unit testing because the test data did not include the boundary case. The fix was deterministic primary key check rather than position-based deduplication. The general lesson: testing production Spark Structured Streaming, especially recovery scenarios, requires more than unit tests. Replay-from-snapshot integration tests are essential.

V. FLINK ARCHITECTURE AND WHEN TO CHOOSE IT

A. Flink Advantages

Four use cases would have us reach for Flink. Sub-second latency for fraud detection or real-time alerting where the 1-second floor of Spark Structured Streaming is not acceptable. Flink achieves 100 to 500 milliseconds end-to-end. Complex stateful processing with state larger than 10 GB. Flink's RocksDB backend handles this gracefully; Spark's state store performance degrades. Advanced windowing with custom window triggers, late-firing semantics, or session windows with non-trivial gap definitions. Flink's window API supports these directly; Spark's is simpler and less flexible. Queryable state for use cases where an external system needs to query the streaming

state mid-pipeline. Flink supports this natively; Spark does not.

B. Flink Disadvantages in Enterprise Setting

Three things have kept us on Spark. First, the Spark ecosystem is broader: it integrates natively with Hadoop, Hive, Delta Lake, Iceberg, MLlib, and the wider Spark batch tooling. Flink has a smaller ecosystem and requires more custom integration code. Second, the mental model: Spark Structured Streaming is approachable for any engineer who has done batch Spark, because the API is the same. Flink requires learning a new mental model stateless operators on unbounded streams that is foreign to engineers transitioning from batch. Third, operations and tooling: Spark has battle-tested Kubernetes operators, mature monitoring integrations, and a much deeper operational community to learn from. Flink's tooling is growing but is not as rich as of 2022. For a team of 30 engineers, the operational burden difference is real.

C. Complementary Patterns

Three patterns combine the two engines. Spark for high-throughput aggregations, Flink for low-latency detection on the same Kafka topics, with different downstream pipelines for different SLAs. Spark Structured Streaming for the bulk of the workload plus Flink SQL for ad-hoc streaming queries that do not justify a permanent Spark pipeline. Migration path: start with Spark, evaluate Flink only when a specific use case demonstrates a need that Spark cannot meet after optimization. In practice, many teams that consider migrating discover that their Spark pipeline can meet the SLA after tuning.

D. When to Evaluate Flink

The triggers that should prompt a Flink evaluation: latency SLA below 1 second; state size growing beyond 10 GB per pipeline; windowing requirements that the Spark API cannot express cleanly; queryable state requirements; or a green-field deployment without an existing Spark investment. Absent these triggers, the path of least resistance is Spark, and the path of least resistance is usually the right path.

VI. TOOL SELECTION FRAMEWORK

For practitioners choosing between the two engines, we recommend a four-question framework. What is the latency SLA? If under 1 second, Flink is the better default. If 1 to 5 seconds or higher, Spark is acceptable and probably preferable. How large is the state? If under a few gigabytes, either works. If over 10 GB, Flink. What does the surrounding ecosystem look like? If the team already has Spark batch infrastructure, Hadoop or lakehouse integrations, and Spark expertise, the path of least resistance is Spark Structured Streaming. If the team is starting fresh or has Flink expertise, Flink is reasonable. What is the team's operational maturity? Spark has lower operational burden because of its broader tooling and community. Flink demands more operational investment, which a smaller team may not be able to sustain.

The framework is not a flowchart that produces a single answer; it is a set of considerations that interact. In our environment all four point to Spark, which is why we have stayed on it despite the latency cases that would benefit from Flink. A different organization with different starting conditions would reasonably reach a different conclusion.

VII. CONCLUSION

Returning to the three research questions:

RQ1. Flink and Spark Structured Streaming differ fundamentally in their execution models true streaming versus micro-batching and the difference manifests in latency (Flink lower), throughput (comparable), state management (Flink stronger for very large state), and operational characteristics (Flink more demanding, Spark more familiar).

RQ2. Spark Structured Streaming at enterprise scale has predictable operational challenges that we documented in Section 4: checkpointing, micro-batch sizing interactions with file output, watermark business logic, monitoring, and sink idempotence. None is fatal. All require deliberate engineering. Flink would help with the sub-second-latency cases that micro-batching cannot reach, and with the complex stateful workloads that Spark's state stores were not designed for.

RQ3. The choice between the two is not purely technical. Latency requirements, state complexity, ecosystem fit, and team expertise all matter, and the right answer for an organization is the one that balances those factors against its starting conditions. For most enterprise teams running at our scale and with our use case mix, Spark Structured Streaming is the pragmatic choice. For teams with sub-second latency requirements or unusually large state, Flink is worth the steeper curve.

The closing observation is that streaming architecture decisions are more durable than they first appear. A team that picks one engine builds operational expertise, monitoring tooling, library code, and team intuition around that engine, and the cost of switching is not the technical work of migration but the loss of all that accumulated context. We have chosen to invest in Spark Structured Streaming and to evaluate Flink case by case for individual workloads where it would be measurably better. After five years and 340 topics of operational experience, we have not yet found a case where the migration cost was justified. We expect that to change for some specific workload eventually, and when it does, the answer will not be to rebuild everything on Flink it will be to run Flink alongside Spark for the workload that needs it.

REFERENCES

[1] T. Akidau, R. Bradshaw, C. Chambers, 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 Endowment, vol. 8, no. 12, 2015.

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," IEEE Data Eng. Bull., vol. 38, no. 4, 2015.

[3] M. Armbrust, T. Das, J. Torres, et al., "Structured Streaming: A declarative API for real-time applications in Apache Spark," in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2018.

[4] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in Proc. ACM Symp. Operating Syst. Principles (SOSP), 2013.

[5] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," ACM Trans. Comput. Syst., vol. 3, no. 1, 1985.

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

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

[8] Apache Software Foundation, "Apache Kafka documentation." [Online]. Available: kafka.apache.org

[9] G. Wang, J. Koshy, S. Subramanian, et al., "Building a replicated logging system with Apache Kafka," Proc. VLDB Endowment, vol. 8, no. 12, 2015.

[10] Apache Software Foundation, "Apache Flink documentation." [Online]. Available: flink.apache.org

[11] Apache Software Foundation, "Apache Spark documentation." [Online]. Available: spark.apache.org

[12] Apache Software Foundation, "Apache Beam documentation." [Online]. Available: beam.apache.org

[13] Confluent, "Confluent Schema Registry documentation." [Online]. Available: docs.confluent.io

[14] Apache Software Foundation, "Apache Iceberg documentation." [Online]. Available: iceberg.apache.org

[15] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: A new generation of open platforms," in Proc. Conf. Innovative Data Syst. Res. (CIDR), 2021.

[16] M. Kleppmann, Designing Data-Intensive Applications. Sebastopol, CA: O'Reilly Media, 2017.

[17] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., Site Reliability Engineering. Sebastopol, CA: O'Reilly Media, 2016.