International Journal of Computer Science Engineering Techniques – Volume 5 Issue 1, January - 2021

Exactly-Once Semantics in Distributed Stream Processing at Scale

Jeevan Krishna Paruchuri

Independent Researcher
paruchuri.g167@gmail.com

Abstract—Exactly-once semantics in distributed stream processing remain difficult to achieve in practice, particularly when pipelines span multiple subsystems with independent failure domains: a message broker, a stream processor, and a persistent sink. Although Apache Kafka and Apache Spark Structured Streaming each advertise exactly-once guarantees within their own boundaries, the end-to-end guarantee experienced by an application is determined by the weakest link in the chain, and the conditions under which the chain holds together are often poorly understood by practitioners. This paper presents a formal analysis of the conditions required for end-to-end exactly-once guarantees in Kafka-to-Spark-to-sink pipelines, supplemented by empirical failure observations from a production financial services streaming platform processing tens of thousands of payment transactions per second under a two-hundred-millisecond latency service-level agreement. The paper contributes a five-category failure taxonomy for exactly-once violations observed in production, including a previously under-documented class of failure involving staging table isolation under concurrent foreachBatch invocations; empirical data on incident frequency, detection time, and impact; and a set of architectural recommendations for separating latency-critical hot paths from analytical cold paths, designing genuinely idempotent sinks, and implementing reconciliation-based monitoring of exactly-once guarantees. The central message is that exactly-once is achievable but requires deliberate architectural design across all three pipeline layers; relying on framework defaults is insufficient for production systems where duplicates or losses carry regulatory consequences.

Index Terms—exactly-once semantics, stream processing, Apache Kafka, distributed systems, fault tolerance, message delivery guarantees

I. INTRODUCTION

Financial services workloads occupy a particularly demanding region of the stream processing design space. Payment authorization, real-time fraud detection, and regulatory transaction reporting all require pipelines that deliver each input record to downstream consumers exactly once, neither dropping records nor producing duplicates, even under failure of individual nodes, network partitions, or coordinated restarts. The cost of violating these guarantees is not measured in inconvenience: a duplicated payment debit can produce a customer-visible incident, a dropped fraud signal can result in an unrecovered loss, and inconsistencies in regulatory reporting can attract scrutiny from supervisory authorities.

A common engineering response to the difficulty of achieving exactly-once guarantees is to settle for at-least-once delivery combined with downstream deduplication. Under sufficient assumptions stable record keys, bounded out-of-order arrivals, and a deduplication store with adequate retention this approach is workable. In practice, however, the assumptions are rarely as tight as the design assumes. Record keys evolve as

upstream systems change, deduplication windows are exceeded under unusual but not impossible conditions, and the deduplication store itself becomes a new failure domain whose own exactly-once properties must be reasoned about. The result is that at-least-once-with-deduplication is often a redistribution of the exactly-once problem rather than a solution to it.

This paper studies exactly-once semantics in pipelines built on Apache Kafka and Apache Spark Structured Streaming, the dominant open-source stack for high-throughput streaming in enterprise environments [7], [8], [1]. The scope is intentionally restricted to the Kafka-to-Spark-to-sink topology, where the sink is either a transactional table format such as Delta Lake [2] or a relational database accessed through JDBC. Within this scope, the paper distinguishes three related but distinct notions of exactly-once that are often conflated in practitioner discussions and even in framework documentation.

Exactly-once delivery refers to the guarantee, at the message broker layer, that each record produced to the broker is delivered to a consumer exactly once. Exactly-once processing refers to the guarantee, at the stream processor layer, that each record drawn from the broker advances the processor's state exactly once. Exactly-once output, sometimes called end-to-end exactly-once, refers to the guarantee that each input record contributes to the externally observable output of the pipeline exactly once. The first two guarantees, taken together, do not imply the third: a processor that has correctly applied a record to its internal state may still fail to commit that state to an external sink atomically with the offset advancement, leaving open the possibility of replay and duplicate emission.

The paper is organized around three research questions:

RQ1: What are the formal conditions required for end-to-end exactly-once guarantees in Kafka and Spark Structured Streaming pipelines?

RQ2: What are the common failure modes that violate these conditions in production deployments?

RQ3: What architectural patterns effectively mitigate these failure modes?

The contributions of the paper are threefold. First, the paper formalizes the necessary conditions for end-to-end exactly-once in three-layer pipelines and identifies the precise points at which each condition can be violated, with particular attention to the foreachBatch sink integration pattern that is widely used but whose exactly-once properties are subtle. Second, the paper presents a five-category failure taxonomy derived from formal analysis and empirical observation, including a category staging table isolation failures under concurrent foreachBatch invocations that does not appear to be documented in the published literature but was responsible for a four-hour data inconsistency incident in a production payment data pipeline. Third, the paper offers a set of architectural recommendations grounded in two years of production operation of a fraud detection pipeline processing tens of thousands of transactions per second under a two-hundred-millisecond end-to-end latency service-level agreement.

The remainder of the paper is organized as follows. Section 2 reviews background on exactly-once semantics, Kafka delivery guarantees, Spark Structured Streaming, and sink-side guarantees, and surveys related work on stream processing fault tolerance. Section 3 presents the formal analysis of exactly-once conditions and identifies failure points in the three-layer pipeline. Section 4 presents the failure taxonomy. Section 5 reports empirical observations from the production fraud detection and analytical platforms. Section 6

develops architectural recommendations. Section 7 discusses limitations and the gap between theoretical and practical exactly-once. Section 8 concludes.

II. BACKGROUND AND RELATED WORK

2.1 Exactly-Once Semantics

The formal foundations of distributed exactly-once processing trace to the consistent global snapshot algorithm of Chandy and Lamport [1985], which establishes that a distributed system can record a globally consistent state without halting message exchange, provided that channels are FIFO and that markers can be inserted into the message stream. Modern stream processing systems including Apache Flink build their exactly-once guarantees on adaptations of this algorithm [3], [4]. The key insight is that exactly-once at the processor level reduces to atomic commitment of two pieces of state: the processor's internal state derived from the input records, and the position in the input stream up to which records have been consumed. If both pieces of state can be committed atomically, replay from the committed position will reproduce the same processor state and the same outputs.

Apache Kafka introduced exactly-once semantics at the broker layer in 2017 through a combination of idempotent producers and transactional producers [9]. Idempotent producers attach producer identifiers and sequence numbers to records, allowing the broker to reject duplicates that arise from producer-side retries. Transactional producers extend this with the ability to commit writes to multiple partitions atomically, supporting consume-transform-produce patterns in which the consumer offset advancement is committed as part of the same transaction as the produced output.

Spark Structured Streaming provides exactly-once guarantees through a checkpoint mechanism that records, for each completed micro-batch, the input offsets that have been processed and the output that has been written [1]. On recovery, the processor reads the most recent checkpoint and resumes from the recorded offsets. The exactly-once guarantee depends on the assumption that the sink is either idempotent reapplying a write produces the same result or that the sink commit and the checkpoint commit are atomically coupled.

2.2 Kafka Delivery Guarantees

Kafka exposes three delivery guarantees configurable at the producer level: at-most-once, at-least-once, and exactly-once. At-most-once is achieved by configuring the producer to fire and forget, accepting that records may be lost on broker or network failure. At-least-once is the default and requires producer acknowledgment configuration that waits for replication. Exactly-once requires enabling the idempotent producer and, for multi-partition transactions, the transactional producer. On the consumer side, offset management determines whether a consumer's progress is recoverable. Auto-commit offers convenience at the cost of correctness: offsets may be committed for records that have not yet been processed by the application, leading to data loss on failure. Manual offset management gives the application control to commit offsets only after processing has completed and external side effects have been durably persisted.

Consumer group rebalancing is a frequently overlooked source of exactly-once violation. When the membership of a consumer group changes due to scaling, failure, or deployment Kafka reassigns partitions across the new set of consumers. Records that were in flight at the time of rebalancing may be reprocessed by a different

consumer instance, and any side effects produced by the original consumer that were not committed before the rebalance may be duplicated.

2.3 Spark Structured Streaming

Spark Structured Streaming, introduced as the successor to the earlier DStream API, models stream processing as incremental computation over an unbounded table [1]. The system supports two execution modes: micro-batch, in which input data is processed in discrete batches of typically several seconds, and continuous processing, in which records are processed individually with lower latency at the cost of more limited operator support. Micro-batch is the dominant mode in production use. Output modes append, complete, and update determine how the result of the streaming query is written to the sink, and each output mode has different exactly-once implications. Append mode is the simplest case because each record contributes exactly one output row, which can be made idempotent through standard techniques. Update and complete modes are more complex because the same input record may contribute to multiple output rows over time.

The Structured Streaming checkpoint records, for each micro-batch, the offsets consumed from each input source and the metadata required to recover the writer state. The checkpoint is written to a configurable location, which the framework documentation recommends but does not enforce should be on durable replicated storage. In practice, checkpoint placement on local disk is a common misconfiguration that compromises recovery and, therefore, exactly-once guarantees.

2.4 Sink-Side Guarantees

The sink layer is where exactly-once guarantees are most often broken. Three patterns dominate. The first is idempotent writes, in which the sink schema and operation are designed so that reapplying the same write produces the same result. Idempotent writes are typically implemented as MERGE or UPSERT operations keyed on a stable record identifier. The second is transactional writes, in which the sink supports atomic commit of multiple records and the processor coordinates the sink commit with the offset advancement. Delta Lake supports transactional writes through its optimistic concurrency control over a transaction log [2]. The third is the side effect pattern, in which the processor invokes an external system through an arbitrary callback typically using Spark's foreachBatch sink and is responsible for ensuring that the side effect is committed before the checkpoint advances.

2.5 Related Work

The Dataflow model of Akidau et al. [2015] provides a unified abstraction for batch and streaming computation in which exactly-once is treated as a property of the result, not of the execution. The model influenced both Apache Beam and Apache Flink. Carbone et al. [2017] describe the asynchronous barrier snapshotting algorithm used by Flink, which extends Chandy-Lamport to support exactly-once guarantees with minimal coordination overhead. Studies of stream processor fault tolerance [5], [6] establish the broader design space within which exactly-once is one possible point. Empirical studies of Kafka and Spark Streaming reliability in production environments are less common in the academic literature, although industry experience reports have begun to appear in venues such as USENIX SREcon. The present paper extends this literature by contributing a failure taxonomy and empirical data from a production financial services environment.

3. Formal Analysis of Exactly-Once Conditions

3.1 System Model

A streaming pipeline is modeled as a triple P = (S, T, K), where S denotes the source subsystem (a Kafka cluster), T denotes the transformer subsystem (a Spark Structured Streaming application), and K denotes the sink subsystem (a Delta Lake table or a JDBC-accessible database). Each subsystem has its own failure domain, its own durability semantics, and its own notion of progress. The source S exposes a sequence of records, each identified by an offset within a topic partition. The transformer T consumes records from S, applies a deterministic computation, and produces output records that are submitted to the sink K. The sink K accepts writes and acknowledges them as durably committed.

Let r denote an input record drawn from the source. The pipeline P satisfies end-to-end exactly-once for r if and only if exactly one externally observable output is produced from r in any execution that includes failures. Externally observable outputs include rows written to the sink, but they exclude internal state changes that are not visible outside the pipeline.

3.2 Necessary Conditions

End-to-end exactly-once requires three conditions to hold simultaneously. First, the source must support replayable, ordered reads: for any committed offset position, the source must be able to replay all records from that position forward in the same order. Kafka satisfies this condition through its log-structured storage. Second, the transformer must atomically commit two pieces of state the output produced from a batch of records and the offset position advanced past those records such that on recovery, either both are visible or neither is. Third, the sink must either support idempotent application of writes, so that replay of a previously applied write is harmless, or it must support transactional commit coordinated with the transformer's offset advancement.

These three conditions are necessary and, when all are satisfied, sufficient. The difficulty in practice is that each condition can be violated in subtle ways that are not surfaced by the framework documentation. The remainder of this section enumerates the principal failure points.

3.3 Failure Points

Four failure points warrant detailed discussion. The first is consumer group rebalancing during processing. When a Kafka consumer group is rebalanced, partitions are reassigned, and any in-flight processing on the previous owner is abandoned. If the previous owner had produced side effects writes to a sink, updates to an external store but had not yet committed the corresponding offset, the new owner will reprocess the same records and reproduce the side effects. Mitigation requires that side effects be made idempotent or that they be committed atomically with the offset advancement, which Kafka transactional producers support only for sinks that participate in the same transaction.

The second failure point is executor failure between sink write and checkpoint commit. In Spark Structured Streaming, a micro-batch first writes its output to the sink and then commits the checkpoint. If the executor fails after the sink write has succeeded but before the checkpoint has been committed, recovery will replay the same micro-batch, and the sink will receive the same output a second time. Idempotent sinks tolerate this; non-idempotent sinks do not.

The third failure point is sink timeout causing retry of a non-idempotent write. Sink writes that exceed a timeout threshold may be retried by the framework or by the connector library. If the original write actually succeeded but the acknowledgment was lost, the retry will apply the write a second time. This is a particular hazard for JDBC sinks that do not enforce uniqueness constraints on the natural keys of the data.

The fourth failure point, treated in detail in the next subsection, is the use of foreachBatch with external side effects that are not transactionally coupled to the checkpoint.

3.4 The foreachBatch Problem

Spark Structured Streaming exposes a sink integration mechanism, foreachBatch, that hands each completed micro-batch to a user-supplied function. The function may apply arbitrary logic, including writes to systems for which no native Spark sink exists. The pattern is widely used in practice because it accommodates legacy databases, MERGE operations against transactional table formats, and complex multi-statement sink updates.

The exactly-once properties of foreachBatch are subtle. The framework guarantees that each micro-batch is delivered to the user function with a stable batch identifier and that, on recovery, the same batch identifier will be presented for any micro-batch that was not previously confirmed as fully complete. The framework does not, however, know what side effects the user function performs, and it cannot couple those side effects to the checkpoint commit. The user function is responsible for ensuring that, given the same batch identifier and the same input batch, applying the function twice produces the same external state as applying it once.

Consider a foreachBatch function that performs a MERGE operation against a Delta Lake target table. The function reads the micro-batch into a staging structure, performs a MERGE that updates rows in the target based on matching keys, and returns. If the MERGE succeeds but the checkpoint commit fails due to executor failure or transient storage error, recovery will replay the same batch. The MERGE will be applied a second time. If the MERGE is keyed on a stable source identifier and uses INSERT ON MATCH UPDATE semantics, the second application is harmless. If the MERGE is keyed on a processing-time identifier or computes derived values that depend on the current state of the target, the second application can produce incorrect results. The root cause is that the MERGE is idempotent only with respect to a fixed input, and the input itself must be ensured stable across replay.

A second, more insidious failure mode arises when the foreachBatch function uses a shared staging table to materialize the micro-batch before the MERGE. If two concurrent invocations of the function arising from independent micro-batches that overlap in time during a recovery scenario write to the same staging table, the MERGE may operate on a mixture of records from both batches, producing a result that corresponds to neither. This staging table isolation failure is the principal subject of one of the empirical incidents reported in Section 5.

IV. A FAILURE TAXONOMY FOR EXACTLY-ONCE VIOLATIONS

The formal analysis of Section 3 identifies the conditions under which exactly-once guarantees can be violated. To organize the design space and to support diagnosis in operational settings, this section presents a taxonomy of five failure

categories observed in production deployments and corroborated by formal reasoning. Each category is described in terms of its formal trigger, the conditions under which it manifests, and the observed frequency in the empirical environment described in Section 5. Table 1 summarizes the taxonomy.

4.1 Category 1: Checkpoint-Sink Desynchronization

Checkpoint-sink desynchronization arises when the sink commit and the checkpoint commit are not atomically coupled. The two principal manifestations are checkpoint-before-sink, in which the checkpoint records that a batch has been processed but the sink write has not been durably committed, leading to data loss; and sink-before-checkpoint, in which the sink write succeeds but the checkpoint fails to advance, leading to replay and duplication. Spark Structured Streaming's default ordering is sink-then-checkpoint, so the more common manifestation is duplication rather than loss.

4.2 Category 2: Rebalancing-Induced Duplication

Rebalancing-induced duplication arises when consumer group membership changes during processing and partitions are reassigned to new consumer instances. The new owner replays records from the most recently committed offset, which may be earlier than the actual processing progress on the previous owner. Any side effects produced by the previous owner that were not flushed and committed prior to the rebalance are duplicated. Frequent triggers include scaling operations, rolling deployments of consumer applications, and consumer crashes induced by garbage collection or out-of-memory conditions.

4.3 Category 3: Idempotency Violations

Idempotency violations arise when an operation that the application designer believes to be idempotent is not in fact idempotent. The most common form is a MERGE or UPSERT keyed on a processing-time identifier rather than a source-time identifier: replay produces a different processing-time identifier and therefore a different merge key, causing the replay to insert a duplicate row rather than updating the original. A related form is a derived computation in the merge clause that depends on the current state of the target, so that applying the same logical update twice produces a different result the second time.

4.4 Category 4: Staging Table Isolation Failures

Staging table isolation failures arise when concurrent invocations of a sink-side function share a staging structure a temporary table, an external scratch directory, or an in-memory cache and one invocation observes data written by another. The category is particularly insidious because it is difficult to reproduce in test environments where concurrency is low, and because the symptoms rows that appear to come from neither input batch are difficult to associate with a single root cause. The category does not appear to be widely documented in the published streaming literature, but it was responsible for a four-hour data inconsistency incident in the production environment described in Section 5.

4.5 Category 5: Clock Skew and Watermark Drift

Clock skew and watermark drift arise in event-time processing pipelines where late-arriving records can either be dropped, if they fall behind the watermark, or contribute to recomputation of windows that have already been committed, depending on configuration. Clock skew between source systems can produce records whose event

timestamps are inconsistent with their arrival order, causing watermarks to advance prematurely and late records to be silently dropped. The formal exactly-once guarantee is preserved each record is processed at most once but the externally observable output omits records that the application designer would expect to be included.

Table 1: Taxonomy of Exactly-Once Failure Modes

Category	Formal trigger
1. Checkpoint–sink desync	Non-atomic sink and offset commit
2. Rebalance duplication	Partition reassignment during in-flight processing
3. Idempotency violation	Merge key not stable across replay
4. Staging table isolation	Concurrent function calls share staging
5. Clock skew / watermark drift	Event-time records past watermark

V. EMPIRICAL OBSERVATIONS FROM PRODUCTION

This section reports empirical observations from a production financial services streaming platform operated over a period of approximately two years. The platform serves two distinct workload classes: a real-time fraud detection pipeline subject to a two-hundred-millisecond end-to-end latency service-level agreement and processing tens of thousands of transactions per second at peak, and an analytical pipeline that consumes the same source streams and feeds Delta Lake tables on Azure Data Lake Storage Gen2 for downstream reporting and machine learning. The empirical observations are framed as production experience rather than controlled experiments and are intended to ground the failure taxonomy of Section 4 in concrete operational evidence.

5.1 Hot Path: Latency-Critical Java Consumers

The fraud detection pipeline operates outside Spark Structured Streaming. Early experimentation with Spark micro-batch processing showed that the per-batch overhead, even at the smallest practical batch sizes, was incompatible with the two-hundred-millisecond latency budget. The hot path was reimplemented using Java Kafka consumers performing manual offset management, with feature lookups served from a Redis-backed feature cache and inference performed by a gradient-boosted tree model executed through ONNX Runtime in under five milliseconds per record. The choice of Java over Python was driven by the observation that Python's global interpreter lock and garbage collection pauses were incompatible with the latency requirement; Java with appropriate garbage collector tuning produced more predictable tail latencies.

End-to-end latency improved from approximately 380 milliseconds in the initial implementation to within the 200-millisecond budget after a sequence of optimizations, of which the single largest was the introduction of connection pooling for Redis. Without pooling, each transaction incurred 40 to 60 milliseconds of Redis connection establishment overhead; with pooling, the per-transaction Redis cost fell to under one millisecond. Connection pooling alone moved the system from infeasible to feasible. Consumer batch size was tuned to fifty records per poll, which was found to be optimal across a range up to one hundred.

Idempotency in the hot path was implemented through Redis-backed deduplication keys derived from source-system transaction identifiers. Two production incidents over two years of operation are attributable to the hot path. The first was a Redis memory exhaustion incident in which the deduplication store reached its memory limit and began evicting keys; tail latency spiked to approximately 600 milliseconds for a window of 18 minutes before remediation. The second was a code regression introduced during a deployment that affected idempotency key construction. Neither incident resulted in missed fraud detection or in incorrect approval decisions, because the monitoring layer detected anomalous behavior before downstream business impact accumulated. The two incidents in two years correspond to a low absolute frequency, but they illustrate that even carefully designed idempotency layers introduce new failure domains that must themselves be monitored.

5.2 Cold Path: foreachBatch MERGE to Delta Lake

The analytical pipeline uses Spark Structured Streaming to consume the same source topics and write to Delta Lake tables, with foreachBatch used to perform MERGE operations that combine inserts and updates against the target tables. The cold path is not subject to the same latency constraints as the hot path; micro-batch intervals on the order of seconds are acceptable.

The principal exactly-once incident on the cold path involved staging table isolation, the failure category described in Section 4.4. The foreachBatch function had been implemented to materialize each incoming micro-batch into a staging table whose name was derived from the target table identifier rather than from the batch identifier. Under normal operation, micro-batches were processed sequentially and the staging table was overwritten between batches without incident. During a recovery scenario in which a failed batch was replayed concurrently with an in-progress new batch, both invocations wrote to the same staging table. The subsequent MERGE operations applied a mixture of records from both batches, producing a result in the target table that corresponded to neither input.

The incident was observable downstream as inconsistencies in payment data: rows whose values did not correspond to any single source transaction. The root cause took approximately four hours to identify, during which the affected tables were treated as authoritative for downstream consumers. The remediation was to redesign the foreachBatch function to use per-batch isolated staging structures keyed on the framework-supplied batch identifier, ensuring that concurrent invocations could not interfere. The episode illustrates that the exactly-once properties of foreachBatch depend on assumptions about the user function's internal isolation that are not enforced or even surfaced by the framework.

5.3 Checkpoint Storage Durability

A separate observation, smaller in impact but worth documenting, involves the placement of Spark Structured Streaming checkpoints. The framework documentation recommends that checkpoints be stored on durable replicated storage but does not enforce this. Early in the platform's operation, a streaming application was deployed with its checkpoint directory on a node-local file system. A node failure resulted in the loss of the checkpoint and the requirement to reprocess records from the earliest available Kafka offset. The exactly-once guarantee was technically preserved reprocessing of a Kafka topic from the beginning is consistent with the framework's

recovery model but the downstream sink received a large volume of duplicate writes that the idempotent MERGE design absorbed without producing incorrect results. The episode reinforced the operational requirement that checkpoints must be on the same durability tier as the sink data.

5.4 Failure Frequency Summary

Table 2 summarizes the principal incidents observed during the two-year operation of the platform, classified by failure category.

Table 2: Observed Incidents by Failure Category (two-year window)

Category	Occurrences	Detection time
1. Checkpoint–sink desync	1 (checkpoint loss)	Minutes (alerting)
2. Rebalance duplication	0 confirmed	
3. Idempotency violation	1 (code regression, hot path)	Minutes (monitoring)
4. Staging table isolation	1 (cold path foreachBatch)	Hours
5. Clock skew / watermark drift	0 confirmed	
Aux: Idempotency store outage	1 (Redis memory)	Minutes

Two observations from the table merit emphasis. First, the absolute incident count is low six incidents across two failure categories plus the auxiliary store outage but the resolution time for the staging table isolation incident is notably longer than for the others, reflecting the difficulty of diagnosing failures that do not match well-known categories. Second, none of the incidents resulted in missed fraud detection decisions, which the authors attribute to monitoring practices that detected anomalies before business impact accumulated, rather than to the framework guarantees themselves.

VI. ARCHITECTURAL RECOMMENDATIONS

6.1 Separate Hot and Cold Paths

Pipelines that combine latency-critical and latency-tolerant workloads should not attempt to satisfy both with a single processor. The empirical experience reported in Section 5 shows that micro-batch processors such as Spark Structured Streaming are not viable for sub-second latency requirements, and that purpose-built consumer applications in languages with predictable garbage collection are better suited to the hot path. The cold path can use a micro-batch processor with confidence, since the latency budget accommodates batch overhead and the analytical workload tolerates the longer reconciliation cycles. Separating the paths also allows the exactly-once strategy to be tailored to each: the hot path uses idempotency keys backed by an external store, while the cold path uses transactional sinks with foreachBatch designed for true idempotency.

6.2 Idempotent Sink Design

Sink operations should be designed to be idempotent under replay, and the merge keys should be derived from source-system identifiers that are stable across replay. Processing-time identifiers, including batch identifiers generated by the processor at runtime, must not appear in merge keys. Derived computations within the merge clause should avoid dependence on the current state of the target table; where such dependence is unavoidable, the operation should be designed so that reapplying the merge produces a consistent result. CDC pipelines should use timestamp-based or sequence-based conflict resolution drawn from

the source system's transaction log rather than from the processor's wall clock.

6.3 Staging Table Isolation

Each invocation of a foreachBatch function that uses a staging structure must use a staging structure isolated from concurrent invocations. The framework-supplied batch identifier provides a stable basis for naming staging tables, partitions, or temporary directories. Implementations that rely on shared scratch tables under the assumption that micro-batches are processed sequentially should be regarded as fragile, because recovery scenarios can produce concurrent execution that violates the assumption. The cost of per-batch isolation creating and dropping a staging structure per micro-batch is modest compared to the cost of a four-hour incident response.

6.4 Reconciliation-Based Monitoring

Framework guarantees of exactly-once should be verified empirically through reconciliation checks that compare source record counts to sink record counts on a regular cadence. Reconciliation can be implemented as a periodic batch job that reads offsets from Kafka and counts rows in the sink, alerting on discrepancies that exceed a configurable threshold. Reconciliation does not replace exactly-once design, but it provides an independent check that surfaces violations the framework cannot detect. In the empirical environment described in Section 5, reconciliation was the principal mechanism by which the staging table isolation incident was eventually localized to the cold path.

6.5 Checkpoint Storage Durability

Spark Structured Streaming checkpoints must be stored on the same durability tier as the sink data. Local file systems, ephemeral cluster storage, and node-local volumes are not acceptable, even though the framework permits their use and does not warn against it. The recommendation is operationally simple but is under-emphasized in framework documentation, and the consequences of misconfiguration are not visible until a failure occurs. Checkpoint storage should be replicated, durable, and accessible from any node that may resume the streaming application after recovery.

VII. DISCUSSION AND LIMITATIONS

The findings of this paper highlight a persistent gap between theoretical exactly-once guarantees and the guarantees experienced by applications running in production. Apache Kafka and Apache Spark Structured Streaming each advertise exactly-once semantics, and each delivers on this promise within its own boundary. The boundary, however, does not extend to the externally observable output of the pipeline. The end-to-end guarantee requires correct composition of the source, processor, and sink layers, and the composition is the responsibility of the application designer.

The three-layer guarantee model exposes a principle that is implicit in the formal analysis of Section 3 but worth stating explicitly: the actual exactly-once guarantee experienced by an application is determined by the weakest link in the chain. A perfectly idempotent sink does not rescue a non-replayable source. A transactional source does not rescue a non-idempotent sink. A correctly checkpointed processor does not rescue a sink-side function that lacks staging isolation. The implication is that exactly-once is not a property that can be acquired by selecting the right framework; it is a property that must be designed at the level of the entire pipeline.

A second observation concerns the role of monitoring. The empirical experience reported in Section 5 shows that monitoring both reconciliation checks and downstream business metric monitoring was the mechanism by which incidents were contained before they produced material harm. This is consistent with the broader principle that exactly-once is not a property to be assumed but a property to be measured. The architectural recommendations of Section 6 include reconciliation-based monitoring for this reason.

The limitations of the work are several. The empirical observations are drawn from a single production environment in financial services on the Azure cloud platform, and the workload mix and failure patterns may not generalize to other environments. The two-year observation window is sufficient to surface incidents but is short relative to the long-tail failure events that can occur in distributed systems. The failure taxonomy is grounded in observed incidents and formal reasoning but may not be exhaustive; additional failure categories may emerge as the underlying frameworks evolve and as new sink technologies become common. The choice to use Java consumers for the hot path and Spark Structured Streaming for the cold path reflects specific latency requirements; environments with different requirements may make different choices, and the architectural recommendations should be read as conditional on the workload characteristics described.

A final caveat: the paper's central claim is not that exactly-once is unachievable in Kafka and Spark pipelines, but that it is achievable only through deliberate design across all three pipeline layers. Readers seeking exactly-once guarantees should treat framework documentation as a starting point for design, not as a substitute for it.

VIII. CONCLUSION

This paper has presented a formal analysis of the conditions required for end-to-end exactly-once guarantees in Kafka and Spark Structured Streaming pipelines, a five-category taxonomy of exactly-once failure modes observed in production, empirical data from a financial services streaming platform processing tens of thousands of transactions per second, and a set of architectural recommendations for achieving exactly-once guarantees in practice.

The principal contribution is the identification of staging table isolation failures as a distinct failure category that does not appear to be widely documented in the published streaming literature but that produced a four-hour data inconsistency incident in the empirical environment. The category arises specifically in the foreachBatch sink integration pattern when the user-supplied function uses a staging structure shared across concurrent invocations, and it can be reproduced only under recovery scenarios that produce concurrent execution. The mitigation per-batch isolated staging keyed on the framework-supplied batch identifier is straightforward, but it depends on awareness that the failure mode exists.

More broadly, the paper argues that exactly-once is not a property that can be acquired by configuration of a single framework; it is a property that emerges from correct composition of source, processor, and sink layers, each with its own failure domain and its own durability semantics. Practitioners working on production financial services streaming systems should treat framework guarantees as necessary but not sufficient, and should design idempotency,

isolation, and reconciliation as first-class concerns of the pipeline architecture. The empirical experience reported here suggests that monitoring particularly reconciliation-based monitoring that compares source and sink record counts independently of framework guarantees is the most effective mechanism for containing the incidents that inevitably arise. Future work should extend the failure taxonomy to additional sink technologies and processor frameworks, and should investigate formal verification techniques that could surface staging isolation hazards at design time rather than after a production incident.

REFERENCES

[1] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., and Whittle, S. (2015). The Dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proceedings of the VLDB Endowment, 8(12), 1792–1803.

[2] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. (2013). MillWheel: Fault-tolerant stream processing at internet scale. Proceedings of the VLDB Endowment, 6(11), 1033–1044.

[3] Armbrust, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Stoica, I., Wendell, P., Xin, R., and Zaharia, M. (2018). Structured Streaming: A declarative API for real-time applications in Apache Spark. Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data, 601–613.

[4] Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., Łuszczak, A., Switakowski, M., Szafrański, M., Li, X., Ueshin, T., Mokhtar, M., Boncz, P., Ghodsi, A., Paranjpye, S., Senster, P., Xin, R., and Zaharia, M. (2020). Delta Lake: enterprise-grade ACID table capabilities on cloud storage. Proceedings of the VLDB Endowment, 13(12), 3411–3424.

[5] Balazinska, M., Balakrishnan, H., Madden, S. R., and Stonebraker, M. (2008). Fault-tolerance in the Borealis distributed stream processing system. ACM Transactions on Database Systems, 33(1), 1–44.

[6] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). Concurrency Control and Recovery in Database Systems. Addison-Wesley.

[7] Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., and Tzoumas, K. (2017). State management in Apache Flink: Consistent stateful distributed stream processing. Proceedings of the VLDB Endowment, 10(12), 1718–1729.

[8] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache Flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36(4).

[9] Chandy, K. M., and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems, 3(1), 63–75.

[10] Chandramouli, B., Goldstein, J., Barnett, M., DeLine, R., Fisher, D., Platt, J. C., Terwilliger, J. F., and Wernsing, J. (2014). Trill: A high-performance incremental query processor for diverse analytics. Proceedings of the VLDB Endowment, 8(4), 401–412.

[11] Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., and Zdonik, S. B. (2003). Scalable distributed stream processing. CIDR.

[12] Dean, J., and Barroso, L. A. (2013). The tail at scale. Communications of the ACM, 56(2), 74–80.

[13] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. Journal of the ACM, 32(2), 374–382.

[14] Gilbert, S., and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 33(2), 51–59.

[15] Gray, J., and Reuter, A. (1992). Transaction Processing: Concepts and Techniques. Morgan Kaufmann.

[16] Hellerstein, J. M. (2010). The declarative imperative: Experiences and conjectures in distributed logic. ACM SIGMOD Record, 39(1), 5–19.

[17] Helland, P. (2007). Life beyond distributed transactions: An apostate's opinion. CIDR, 132–141.

[18] Hwang, J. H., Balazinska, M., Rasin, A., Çetintemel, U., Stonebraker, M., and Zdonik, S. B. (2005). High-availability algorithms for distributed stream processing. Proceedings of the 21st International Conference on Data Engineering (ICDE), 779–790.

[19] Kafka, F. (2017). Apache Kafka: Exactly-once semantics. Confluent Engineering Blog / Apache Kafka Documentation.

[20] Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.

[21] Kleppmann, M., and Kreps, J. (2015). Kafka, Samza and the Unix philosophy of distributed data. IEEE Data Engineering Bulletin, 38(4), 4–14.

[22] Kreps, J. (2014). Questioning the Lambda architecture. O'Reilly Radar.

[23] Kreps, J., Narkhede, N., and Rao, J. (2011). Kafka: A distributed messaging system for log processing. Proceedings of the NetDB Workshop.

[24] Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J. M., Ramasamy, K., and Taneja, S. (2015). Twitter Heron: Stream processing at scale. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 239–250.

[25] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7), 558–565.

[26] Lamport, L. (1998). The part-time parliament. ACM Transactions on Computer Systems, 16(2), 133–169.

[27] Lin, J., and Ryaboy, D. (2013). Scaling big data mining infrastructure: The Twitter experience. ACM SIGKDD Explorations Newsletter, 14(2), 6–19.

[28] Marz, N., and Warren, J. (2015). Big Data: Principles and Best Practices of Scalable Realtime Data Systems. Manning.

[29] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. (2013). Naiad: A timely dataflow system. Proceedings of the 24th ACM Symposium on Operating Systems Principles, 439–455.

[30] Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010). S4: Distributed stream computing platform. IEEE International Conference on Data Mining Workshops, 170–177.

[31] Noghabi, S. A., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I., and Campbell, R. H. (2017). Samza: Stateful scalable stream processing at LinkedIn. Proceedings of the VLDB Endowment, 10(12), 1634–1645.

[32] Ongaro, D., and Ousterhout, J. (2014). In search of an understandable consensus algorithm. USENIX Annual Technical Conference, 305–319.

[33] Pacheco, F., Rangan, K., et al. (2018). Production experience with Kafka at financial scale. Industry experience report.

[34] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. (2013). Omega: Flexible, scalable schedulers for large compute clusters. Proceedings of the 8th ACM European Conference on Computer Systems, 351–364.

[35] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011). Conflict-free replicated data types. Symposium on Self-Stabilizing Systems, 386–400.

[36] Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. ACM SIGMOD Record, 34(4), 42–47.

[37] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., and Ryaboy, D. (2014). Storm @Twitter. Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 147–156.

[38] Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., Rao, J., Kreps, J., and Stein, J. (2015). Building a replicated logging system with Apache Kafka. Proceedings of the VLDB Endowment, 8(12), 1654–1655.

[39] Wang, G., Chen, L., Dikshit, A., Gustafson, J., Chen, B., Sax, M. J., Roesler, J., Blee-Goldman, S., Cadonna, B., Mehta, A., Madan, V., and Rao, J. (2019). Consistency and completeness: Rethinking distributed stream processing in Apache Kafka. Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data.

[40] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. Proceedings of the 24th ACM Symposium on Operating Systems Principles, 423–438.

[41] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. (2016). Apache Spark: A unified engine for big data processing. Communications of the ACM, 59(11), 56–65.