Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Reliable Event-Driven Processing in Distributed Systems: Stage-Aware Retries, Idempotency, and Exactly-Once Semantics

Jay Bankimchandra Desai

San Jose State University, USA

ARTICLE INFO

Received: 10 March 2026

Accepted: 22 March 2026

ABSTRACT

Reliable event processing and the achievement of exactly-once semantics for business-critical events in distributed systems require more than merely the right transactional semantics. Although event-driven architectures provide scalability and loosely coupled components, they introduce non-obvious failure scenarios that are difficult to reason about. This work is primarily a conceptual and analytical study: it synthesizes mechanisms and architectural patterns drawn from distributed systems literature, vendor documentation, and production system designs, evaluating their complementary roles within a unified reliability framework rather than reporting empirical measurements from a specific system deployment. Transactional APIs offer built-in correctness guarantees but have well-defined throughput limits and optimization targets that differ from those of production systems. Techniques such as exponential backoff help protect systems from transient failures by routing retried events out of the main processing pipeline, avoiding retry storms, and allowing unaffected events to continue processing at normal throughput. Dead letter queues extend this protection by allowing events that could not be processed to be safely reinjected once the root cause of a failure is repaired, ensuring that no event is dropped silently even under prolonged or severe failures. Stage-aware retry mechanisms — where workflows are explicitly checkpointed — ensure that, on failure, only the incomplete stages are replayed rather than the entire workflow. Idempotent API and protocol design guarantees that repeated execution of the same operation does not alter shared state beyond its initial application. Reactive batching strategies further regulate the flow of events under load, preventing throughput degradation from cascading into reliability failures. Workflow orchestration engines provide durable, centralized coordination of multi-stage event processing across microservice boundaries, enabling fault-tolerant execution that choreography-based approaches cannot achieve alone. Stateful stream processing frameworks enforce consistency through distributed checkpointing, allowing pipelines to recover from failures at the granularity of individual operators rather than entire workflows. Achieving exactly-once semantics requires all of these layers to be implemented together: infrastructure-level retries, stage-aware execution, idempotent interfaces, stateful checkpointing, and periodic reconciliation, because each addresses distinct failure modes that the others cannot handle alone.

Keywords: Event-Driven Architecture, Exactly-Once Semantics, Idempotent Processing, Dead Letter Queue, Distributed Systems Reliability, Workflow Orchestration, Stateful Stream Processing

1. Introduction to Event-Driven Architectures and Failure Challenges

Event-driven processing is one of the most common architectural patterns used by modern distributed applications to enable concurrency, decoupling, and scalability by shifting processing from synchronous paths to asynchronous, event-driven workflows. This pattern has two primary

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

applications: offloading latency-sensitive online tasks to asynchronous workflows, and processing batches of data offline for subsequent on-demand low-latency online processing [1]. For these reasons, event-driven architectures are frequently adopted in large systems involving many components, where high throughput and responsiveness must coexist.

Asynchronous event processing introduces several failure-handling and data-consistency challenges that are absent in synchronous request-response systems. In a synchronous system, the presence or absence of an immediate response provides a clear signal about whether an operation succeeded or failed. In asynchronous systems, this signal is absent. When a failure occurs, it may not be possible to determine which processing stages in a workflow completed successfully and which did not. This ambiguity can leave the system in a partially applied state — where some effects have been persisted, and others have not — which is particularly difficult to resolve when the affected state includes database records or external service integrations [2].

When a transient failure occurs, such as a network interruption, infrastructure fault, consumer crash, or broker rebalance, retrying the event without care can corrupt data by overwriting or duplicating mutations to shared mutable state. Conversely, failing to detect and retry errors causes events to be silently lost, with no feedback to the system or its operators. This is especially problematic in asynchronous systems, where errors are not readily surfaced. Building resilient, high-throughput event-driven systems, therefore, requires careful architecture and deliberate engineering across producers, brokers, and consumers, as well as both infrastructure-level and application-level techniques to achieve exactly-once semantics.

The methodology of this paper is analytical and design-oriented. Rather than presenting empirical benchmarks from a single system, this work performs a structured synthesis of established reliability patterns — drawn from peer-reviewed distributed systems research, widely deployed open-source platforms, and documented production architectures — and evaluates how each pattern addresses a distinct class of failure mode. The contribution is a coherent, layered reliability framework that can guide the design of production event-driven systems, supported throughout by quantitative data drawn from the cited primary sources.

The remainder of this paper is structured as follows. Section 2 examines the capabilities and limitations of transactional APIs as a baseline reliability mechanism. Section 3 describes retry topic patterns and exponential backoff as foundational infrastructure. Section 4 covers dead letter queues for failure isolation and event preservation. Section 5 discusses reactive batching strategies and their role in throughput-reliability trade-offs. Section 6 examines workflow orchestration engines and their advantages over choreography for fault-tolerant multi-stage processing. Section 7 addresses stateful stream processing and distributed checkpointing as a mechanism for operator-level recovery. Section 8 covers stage-aware retries and idempotent API design as application-layer correctness guarantees. Section 9 presents the integrated reliability stack through which exactly-once semantics is achieved in practice.

2. Transactional APIs and Their Limitations in Complex Workflows

Most modern event-processing platforms achieve exactly-once semantics through transactional APIs that enforce guarantees across event production, broker-side storage, offset management, and consumption. In this model, an event remains unconsumed until all processing is confirmed complete, enabling retry on error and rollback to a previously saved processing state [3]. Apache Kafka is the most widely studied platform implementing this model, with more than 70 peer-reviewed publications covering its transactional and delivery guarantees at the time of writing [3]. Kafka supports configurable replication factors that define how many copies of each partition are stored

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

across broker nodes. A replication factor of 3 is common in production environments, providing fault tolerance at modest coordination cost [3].

Despite these capabilities, transactional APIs carry meaningful limitations. The coordination overhead required to enforce transactional guarantees introduces non-negligible latency. Empirical results show that Kafka producer throughput is approximately 50,000 messages per second at a batch size of 1, rising to approximately 400,000 messages per second at a batch size of 50. Under conservative transactional settings — where producers must receive acknowledgment of earlier messages before sending subsequent ones — throughput can fall by an order of magnitude. Additionally, transactional APIs impose the requirement that all downstream systems handle operations idempotently, meaning that regardless of how many times an operation is applied, the final state must be identical. In practice, it is uncommon for idempotency to extend seamlessly across every downstream service, database, and external integration involved in a workflow.

Storage overhead is also a consideration. Kafka messages carry approximately 9 bytes of overhead per message in their headers, with no additional delivery-guarantee metadata. By contrast, enterprise JMS messaging systems such as Apache ActiveMQ, which implement transactional delivery guarantees, carry approximately 144 bytes of overhead per message — roughly 70% more storage for the same message volume [4]. This asymmetry illustrates why transactional APIs alone are insufficient to provide correctness guarantees across workflows that span multiple systems, and why additional architectural layers are required.

These limitations are further amplified in microservice architectures, where a single logical business transaction may span multiple independently deployed services, each with its own data store and failure domain. The decomposition of monolithic applications into microservices — while beneficial for scalability and independent deployability — introduces new reliability boundaries at each service interface. Kalia et al. demonstrate that service boundary placement directly determines the scope of transactional consistency: operations that were previously atomic within a monolith become distributed and potentially inconsistent once they cross service boundaries [13]. This reinforces the need for reliability mechanisms that operate at the workflow level rather than exclusively within individual service transactions.

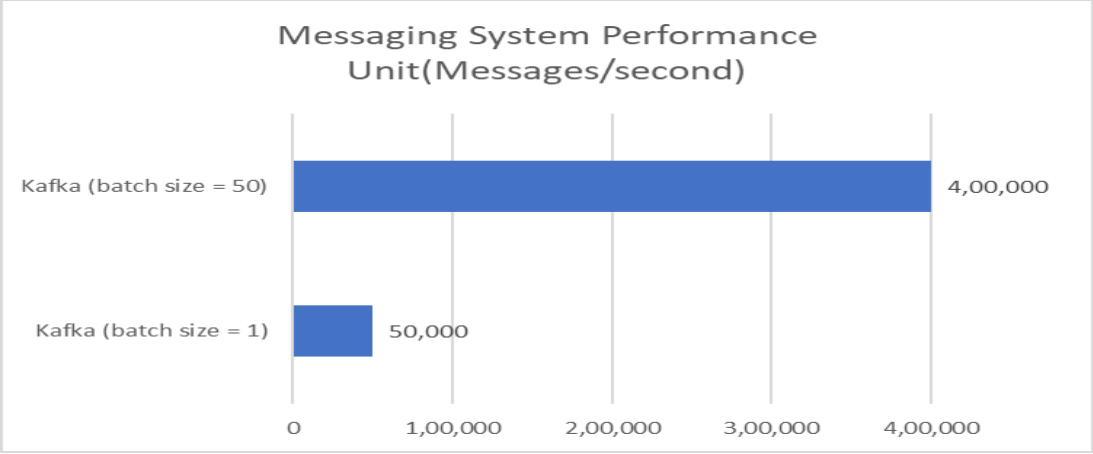

Figure 1: Transactional API Limitations in Distributed Messaging Systems [3, 4]

3. Retry Topics and Exponential Backoff as Foundation Infrastructure

Retry strategies are a foundational building block for reliable event processing in distributed systems. One of the most effective is the retry topic pattern. When event processing fails due to a transient issue with a downstream system, the affected events are routed to a dedicated staging topic rather than being replayed directly from the original partition. This isolates failures caused by downstream

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

dependencies from the healthy flow of events, ensuring that forward progress on unaffected events is not blocked or overwhelmed by repeated retry attempts in the primary processing stream.

Retry topics are typically organized into multiple stages, commonly 3, each configured with a progressively longer wait period. The wait intervals are usually governed by an exponential backoff scheme, in which the delay between successive retry attempts grows by a multiplicative factor, typically 2 or greater. This graduated delay profile gives downstream dependencies time to recover from transient failures before the next retry attempt, thereby preventing retry storms that could overwhelm an already-degraded service. Event processing pipelines commonly consist of at least 4 layers: normalization, enrichment, correlation, and presentation [5, §3]. A failure in one layer can propagate to subsequent layers unless it is properly isolated for retry [5]. When retry topics are correctly configured, automatic recovery is possible for the majority of failures, and manual intervention is reserved for catastrophic or unrecoverable events.

The practical value of this pattern is well evidenced in large-scale production systems. Wu et al. demonstrate that reliable stream processing in real-time Kafka deployments depends critically on isolating failed events from healthy processing paths, and that without such isolation, retry pressure on the primary partition degrades throughput for all consumers — not only those processing the failing events [14]. Similarly, workflow orchestration engines used in production microservice environments implement stage-level retry policies with configurable backoff multipliers, allowing different workflow stages to be assigned independent retry budgets based on the expected recovery characteristics of the services they invoke [15].

The retry topic pattern also provides meaningful operational observability. Queue depth and message age at each retry stage serve as leading indicators of downstream system health. The Dataflow Model is relevant here: it treats watermarks as lower bounds on processing progress, where a single slow-processing datum can halt an entire pipeline [6]. By separating retried events from the main topic, this bottleneck is avoided. Furthermore, watermark skew in distributed systems can span several minutes [6], meaning that retry timeout configurations must be grounded in realistic recovery time estimates rather than optimistic assumptions. This makes exponential backoff a natural and well-suited strategy for managing retry intervals in practice.

Component	Description
Retry topic pattern	Redirects failed events to staging topics instead of replaying from the partition
Exponential backoff	Increases wait time multiplicatively between retry stages
Pipeline layer isolation	Prevents errors propagating across normalization, enrichment, correlation, and presentation layers
Watermark-based progress tracking	Lower bound metric indicating pipeline progress per the Dataflow Model

Table 1: Retry Topic Infrastructure Components [5, 6]

4. Dead Letter Queues for Failure Isolation and Event Preservation

When all configured retry stages for a given event are exhausted without success, the event is routed to a dead letter queue. A dead letter queue is a persistent storage destination for events that cannot be processed through their normal retry paths and require manual analysis and remediation before reprocessing. In enterprise message brokers, dead letter queues isolate non-recoverable events from

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

the healthy message flow, ensuring that a small number of persistently failing messages cannot stall or corrupt processing for the remainder of the pipeline. According to Enterprise Integration Patterns theory, a message flow comprises 6 stages: creation, channeling, routing, transformation, consumption, and management. Errors occurring at any of these stages can propagate across the pipeline unless quarantined [7].

In many business contexts, the consequences of silently dropping events extend well beyond data incompleteness. Failing to process a small number of high-value events — such as payment confirmations, user entitlement updates, or content moderation actions — can result in revenue loss, regulatory non-compliance, and erosion of consumer trust. As the EIP pattern language expanded from the 27 patterns of the original book to the 700 pages of the published document, the Dead Letter Channel was formally identified as the architectural destination for messages that fail all processing attempts [7]. For engineers to diagnose and correct failures effectively, the metadata stored alongside dead-lettered events must include the original topic, partition, offset, timestamp, failure reason, and retry count.

Major cloud infrastructure providers have operationalized this pattern at scale. Backlund's analysis of AWS integration connector implementations documents how cloud-native messaging infrastructure exposes dead letter queue support as a first-class construct, preserving per-message diagnostic attributes alongside each dead-lettered message [16]. These attributes map directly onto the metadata schema recommended above and demonstrate that the theoretical requirements for effective failure diagnosis are both implementable and maintained in high-availability production environments. Integrating dead letter queues with cloud-native connectors further enables automated alerting and remediation workflows to be triggered as soon as messages begin accumulating, reducing mean time to recovery [16].

Controlled replay from dead letter queues is a critical operational capability. Once the root cause of a failure has been diagnosed and resolved, individual messages or batches can be replayed into the pipeline. This replay must itself be idempotent: reprocessing dead-lettered events may re-invoke downstream systems that did not complete successfully during the original attempt, and those invocations must not produce unintended side effects. When implemented correctly, dead letter queue replay enables eventual consistency and supports recovery from prolonged failures lasting multiple hours without permanent data loss.

Property	Description
Storage target	Persistent destination for events exhausting all retry stages
Failure isolation	Prevents non-recoverable events from blocking healthy message flow
Required metadata fields	Topic, partition, offset, timestamp, failure reason, retry count
Replay capability	Supports individual or batch reprocessing after failure resolution

Table 2: Dead Letter Queue Architectural Properties [7,16]

5. Reactive Batching Strategies and Throughput-Reliability Trade-offs

Retry infrastructure and dead letter queues protect event processing correctness under failure conditions, but reliability in event-driven systems must also be maintained under conditions of high load and variable throughput. A system that processes events correctly at low volume but degrades or loses events under bursts of incoming traffic provides insufficient guarantees for production use. Reactive batching strategies address this class of problem by dynamically adapting the volume of

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

events dispatched to consumers based on observed processing capacity, thereby preventing the accumulation of backpressure that can destabilize consumer groups and trigger spurious failures.

Wu et al. propose a reactive batching strategy for Apache Kafka in which batch sizes are adjusted in response to real-time measurements of consumer lag and processing latency [14]. Rather than using a fixed batch size — which either under-utilizes capacity at low load or overwhelms consumers at high load — the reactive strategy modulates batch size within a configurable range, scaling up when consumers are healthy and scaling down when lag begins to grow. This approach improves end-to-end throughput stability and reduces the frequency of consumer timeouts that would otherwise trigger unnecessary partition rebalances [14]. Partition rebalances are particularly costly from a reliability perspective because they temporarily suspend consumption across all partitions assigned to the rebalancing consumer group, introducing a window during which in-flight events may be neither committed nor retried.

The interaction between batching strategy and exactly-once semantics is non-trivial. Larger batch sizes improve throughput but increase the blast radius of a failure: when a batch fails mid-processing, all events in that batch must be retried, increasing the load on retry topics and downstream systems. Smaller batch sizes reduce the cost of individual failures but increase per-event overhead and coordination cost. The reactive batching approach of Wu et al. navigates this trade-off by treating batch size as a dynamic control variable rather than a static configuration parameter, allowing the system to balance throughput against failure isolation cost in response to observed runtime conditions [14]. This adaptability is particularly valuable in systems with heterogeneous workloads, where the optimal batch size varies across time of day, event type, and downstream service availability.

From a reliability architecture perspective, reactive batching is best understood as a stabilization layer that sits below the retry infrastructure described in Section 3. By preventing consumer overload and partition rebalances, it reduces the frequency with which events need to enter the retry pipeline in the first place, allowing the retry and dead letter mechanisms to operate on genuinely non-recoverable failures rather than artifacts of load-induced instability.

Strategy	**Description**
Fixed batch sizing	Static configuration may under-utilize capacity or overwhelm consumers under variable load
Reactive batch sizing	Dynamic adjustment based on consumer lag and processing latency
Rebalance prevention	Stable batches reduce partition rebalance frequency under load
Reliability interaction	Batch size determines the blast radius of mid-batch processing failures

Table 3: Reactive Batching Strategies and Reliability Interactions [14]

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

6. Workflow Orchestration Engines for Fault-Tolerant Multi-Stage Processing

Event-driven systems that process complex, multi-step business workflows face a structural challenge that retry topics and dead letter queues alone cannot address: the need to coordinate state and execution across multiple microservices in a way that remains correct under partial failures. Two primary coordination models exist — choreography, in which each service reacts to events produced by other services without central coordination, and orchestration, in which a dedicated workflow engine explicitly manages the sequencing, state, and retry behavior of all participating services. For fault-tolerant, exactly-once processing of complex workflows, orchestration offers significant advantages over choreography.

Nadeem and Malik provide a comparative analysis of microservices coordination approaches, demonstrating that choreography-based systems, while simpler to implement initially, become increasingly difficult to reason about as the number of participating services grows [15]. In a choreographed system, the processing state of a workflow is implicitly encoded across the local states of multiple independent services, with no single component holding a complete view. When a failure occurs, reconstructing which steps were completed successfully requires correlating logs across all participating services — a process that is both time-consuming and error-prone. Orchestration-based systems, by contrast, maintain explicit workflow state in the orchestrator, enabling precise identification of the failed step and targeted retry without re-executing completed stages [15].

Workflow orchestration engines implement this model with durable state persistence. Each workflow instance maintains a complete execution record — including which tasks have completed, which are in progress, and which have failed — stored in a persistent backend. This record survives orchestrator crashes and restarts, ensuring that in-flight workflows are not lost during infrastructure failures. Nadeem and Malik further show that orchestration engines natively support configurable retry policies at the task level, timeout management, and compensating transaction patterns such as the Saga pattern, which enables rollback of completed steps when a downstream step fails irrecoverably [15]. These capabilities are difficult to replicate in choreographed systems without significant custom engineering.

The reliability benefits of workflow orchestration are closely related to the stage-aware retry mechanisms discussed in Section 8. An orchestration engine is, in effect, a production-grade implementation of stage-aware retry: it checkpoints workflow state at the task boundary, exposes each task's completion status to the retry logic, and resumes execution from the first incomplete task on failure. The distinction is one of implementation scope — stage-aware retry as an architectural pattern describes the correctness requirement, while workflow orchestration engines provide the infrastructure to meet that requirement at scale, with operational tooling for monitoring, alerting, and manual intervention.

Coordination Model	Failure Visibility	Retry Granularity	State Location
Choreography	Distributed across service logs	Full workflow replay	Implicit, per-service
Orchestration	Centralized in the orchestrator	Per-task, selective	Explicit, durable

Table 4: Choreography vs. Orchestration for Fault-Tolerant Workflows [15]

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

7. Stateful Stream Processing and Distributed Checkpointing

For event-driven systems that process continuous data streams rather than discrete workflow instances, reliability requires a complementary mechanism: the ability to checkpoint the processing state of individual stream operators at regular intervals and resume from those checkpoints following a failure. Stateful stream processing frameworks implement this capability through distributed snapshot algorithms, providing exactly-once processing guarantees at the operator level without requiring the entire pipeline to be replayed from the beginning of the stream.

Carbone et al. describe the state management architecture of Apache Flink in detail, distinguishing between operator state — which is local to a single parallel instance of a stream operator — and keyed state, which is partitioned by event key and distributed across all parallel instances of a keyed operator [17]. Both forms of state are checkpointed periodically through an asynchronous distributed snapshot mechanism derived from the Chandy-Lamport algorithm for consistent global snapshots in distributed systems. A checkpoint captures the complete state of all operators at a consistent point in the event stream, identified by a checkpoint barrier that is injected into the event stream and propagated through the pipeline. When all operators have acknowledged the barrier, the checkpoint is considered complete and durable [17].

Upon failure, all operators are restored to their last completed checkpoint, and processing resumes from the corresponding position in the input stream. This recovery model provides exactly-once semantics for in-pipeline state transformations: each event affects operator state exactly once, regardless of the number of failures and recoveries that occur during processing. The checkpoint interval can be configured to trade recovery latency against checkpoint overhead — shorter intervals reduce the amount of reprocessing required after a failure but increase the frequency of snapshot operations [17].

The relationship between stateful checkpointing and the broader reliability stack is complementary rather than substitutive. Checkpointing addresses failures that occur within a running pipeline, ensuring that operator state is recoverable. It does not address failures in the downstream systems to which pipeline outputs are written, or failures that occur between the pipeline and its upstream event sources. These external failure modes are addressed by the retry, dead letter, and idempotency mechanisms described in other sections. Together, stateful checkpointing and external reliability mechanisms cover the full failure surface of a production stream processing system.

Mechanism	Scope	Recovery Granularity
Operator state checkpointing	In-pipeline operator state	Per-checkpoint interval
Keyed state distribution	Partitioned across parallel instances	Per-key, per-checkpoint
Barrier propagation	Global consistent snapshot	Full pipeline consistency

Table 5: Apache Flink Distributed Checkpointing Mechanisms [17]

8. Stage-Aware Retries and Idempotent API Design

Naive event retry introduces a subtle but consequential correctness problem: when an event is replayed from the beginning, any side effects already applied during the prior attempt are applied again. Real-world event processing pipelines rarely consist of a single atomic operation. They typically orchestrate a cascade of downstream effects — database writes, external API calls, cache invalidations,

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

and notification dispatches — each of which may succeed or fail independently. When a failure occurs midway through this sequence, a full replay of the event will re-execute every preceding step, producing duplicated mutations against a state that was already correctly updated. As Helland observes, duplication is not an edge case in distributed messaging systems but the expected behavior of the transport layer, given its at-least-once delivery guarantee [9]. In financial workflows, this is particularly consequential: re-executing a balance update that had already succeeded will result in double crediting or debiting, producing financial losses and costly reconciliation overhead. Ramalingam and Vaswani document precisely this class of failure, noting that financial and transactional systems are among the most sensitive to duplicate execution because the resulting state corruption — an incorrect account balance or a duplicated charge — may not be detectable until a reconciliation process runs, by which point the inconsistency may have propagated to dependent records [18].

Stage-aware retry addresses this problem directly by decomposing event-processing workflows into discrete, named, and independently checkpointed stages. Each stage represents a bounded unit of work with a well-defined entry condition, execution body, and completion record. When a failure occurs, the system persists metadata indicating which stages have already completed. On subsequent retry attempts, the processor consults this checkpoint log and resumes execution from the first incomplete stage, skipping all previously successful ones. This selective re-execution preserves correctness, eliminates redundant side effects, and substantially reduces unnecessary computation. At production scale, the efficiency gains from avoiding redundant downstream calls can be significant, both in terms of cost and system load.

Stage checkpointing must be durable enough to survive process crashes and load-balancer failovers. A checkpoint stored only in memory provides no protection against infrastructure-level failures; persistence to a durable log or distributed store is required for stage-aware retry to function correctly across the full failure surface of a distributed system. This requirement naturally aligns stage-aware retry with the workflow orchestration engines discussed in Section 6, which provide durable execution state as a first-class architectural primitive.

Idempotent API design provides a complementary and equally necessary guarantee at the interface level. Even with stage-aware retry in place, the system may invoke a downstream API more than once for a given stage — due to ambiguous acknowledgment, network partitioning, or consumer failover — without any indication of whether the prior call succeeded. An idempotent API ensures that repeated invocations of the same logical operation produce the same observable state as a single invocation. Ramalingam and Vaswani provide a formal treatment of this property, demonstrating that idempotence can be derived systematically from non-idempotent operations through the use of persistent request identifiers that enable servers to detect and suppress duplicate executions [18]. Their analysis shows that idempotent fault tolerance is achievable even in systems that were not originally designed with idempotency in mind, provided that a durable deduplication store is accessible to all participating components.

A canonical production example of this design is found in payment processing APIs, where the consequences of duplicate execution are directly financial. Stripe's payment API operationalizes idempotent fault tolerance by requiring callers to supply a unique idempotency key with each request [20]. The server stores the result of the first request associated with each key and, for any subsequent request bearing the same key, returns the stored response directly rather than re-executing the operation. This design ensures that a payment request submitted more than once — due to client retry logic, network interruption, or timeout — is charged exactly once regardless of how many times the API receives the request. The pattern is equally applicable to any downstream interface invoked within a stage-aware retry workflow: by assigning a deterministic idempotency key derived from the event

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

identifier and stage name, each stage invocation can be made safe to repeat without risk of duplicated state mutation [20].

This guarantee cannot be derived from ordered delivery alone. As Liskov and Shrira demonstrate, when a caller receives no reply, it cannot determine whether the underlying stream was broken before or after the operation was applied [10]. The promise abstraction they introduce passes through exactly two states — blocked and ready — and once a promise enters the ready state, its value is immutable. This immutability makes promises a natural model for idempotent result fetching: a repeated query for the outcome of a completed operation always returns the same result without triggering re-execution [10].

Together, stage-aware workflow orchestration and well-specified idempotent API contracts form a two-layered defense against duplicate processing. Stage-aware retry prevents redundant re-execution at the workflow level by tracking completion granularly across stages. Idempotent interfaces ensure that, even when a downstream call is issued more than once, the resulting system state remains correct and consistent. Neither layer is sufficient on its own: stage-aware retry cannot protect against duplicate calls that escape its checkpointing scope, and idempotency alone does not prevent unnecessary computation or cascading load on downstream dependencies. Their combination provides defense in depth against the full range of duplication failure modes encountered in production distributed systems.

Mechanism	Description
Stage decomposition	Breaks workflows into named, scoped units with persistent completion records
Stage checkpointing	Survives process crashes and load-balancer failovers
Idempotent API contracts	Ensures duplicate invocations produce identical state outcomes
Persistent request identifiers	Enable server-side deduplication of repeated requests [18]
Promise immutability	Value remains unchanged once promise enters ready state [10]

Table 6: Stage-Aware Retry and Idempotency Mechanisms [9, 10, 18]

9. Achieving Exactly-Once Semantics Through Layered Strategies

No single technique can guarantee exactly-once processing in distributed event-driven systems. Instead, exactly-once processing emerges from the composition of complementary approaches organized into a reliability stack, where each layer addresses failure modes that the others cannot. Infrastructure-level techniques — retry topics with exponential backoff stages and dead letter queues for persistent failures — form the foundation of this stack, ensuring that no event is permanently abandoned. Reactive batching strategies stabilize throughput under load, reducing the frequency with which events enter retry paths due to consumer overload rather than genuine downstream failure. Workflow orchestration engines provide durable, centralized coordination of multi-stage processing across service boundaries, with built-in support for selective task retry and compensating transactions. Stateful stream processing frameworks checkpoint operator state at regular intervals, enabling recovery from in-pipeline failures at fine granularity without full pipeline replay.

Journal of Information Systems Engineering and Management
2026, 11(2s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Stage-aware retry and idempotent APIs extend correctness guarantees to the application layer. At the scale of systems like Google Spanner, potentially serving millions of servers across hundreds of datacenters and managing trillions of rows, even low relative rates of duplicate processing or event loss correspond to large absolute error counts [11]. Together, these techniques address the 2 primary failure modes of exactly-once processing: under-processing, where events are dropped without the application detecting the loss, and over-processing, where duplicate execution corrupts application state. Spanner's commit wait protocol illustrates the cost of over-processing: in conjunction with producer and consumer acknowledgment tuning, it ensures that results do not reach clients before at least $2\times\varepsilon$, typically 4 ms has elapsed from the commit timestamp, eliminating the possibility of clients observing inconsistent state caused by concurrent writes [11].

Periodic reconciliation processes complete the reliability stack. By periodically comparing the state of upstream sources against downstream targets, reconciliation workflows surface rare or compounded inconsistencies that result from one or more failures occurring in combination. Bailis et al. demonstrate that eventual consistency in distributed systems does not self-correct all anomalies: while many inconsistencies resolve naturally through convergence, a meaningful class of violations — particularly those involving invariant-breaking concurrent writes — require explicit detection and repair logic to be systematically resolved [19]. This finding directly motivates the inclusion of reconciliation as a mandatory layer in the reliability stack rather than an optional observability add-on. Research on cloud-native distributed databases has further shown that systems such as Amazon Aurora replicate data 6 times across 3 availability zones specifically to guard against silent data loss [12], and that the overhead of cross-region replication can meaningfully widen consistency windows [12]. The reconciliation interval can be tuned to balance latency against compute cost, depending on the consistency requirements of the application.

Taken together, all layers — infrastructure retry mechanisms, reactive batching, workflow orchestration, stateful stream checkpointing, idempotent interfaces, and periodic reconciliation — form an integrated reliability stack that delivers practical and observable exactly-once semantics across the full range of production failure conditions.

Conclusion

Event processing at scale in distributed environments cannot be addressed by any single pattern or protocol; correctness emerges instead from the orchestration of composable, complementary mechanisms designed to cover a comprehensive set of failure modes across asynchronous, multi-system workflows. Transactional APIs establish processing boundaries and provide a foundation for delivery guarantees, but they are constrained in throughput, require downstream idempotency that is difficult to achieve uniformly, and carry storage overhead that limits their applicability across heterogeneous system landscapes. Retry topics with exponential backoff resolve most transient failures by decoupling problematic events from healthy processing paths and calibrating recovery windows to realistic dependency behavior, while dead letter queues preserve every unrecoverable event — converting silent data loss into actionable engineering signals and enabling controlled, surgical replay once failures are resolved. Reactive batching strategies prevent load-induced consumer instability from generating spurious failures, ensuring that retry infrastructure operates on genuinely non-recoverable events rather than artifacts of throughput pressure. Workflow orchestration engines address the coordination challenge of multi-stage, cross-service processing, maintaining a durable and centralized execution state that enables selective task retry, compensating transactions, and precise failure diagnosis. Stateful stream processing frameworks complement orchestration by checkpointing operator state at fine granularity within continuous pipelines, enabling recovery from in-pipeline failures without full stream replay. Stage-aware retry mechanisms eliminate the correctness risk of naive replay by checkpointing workflow state at discrete points, allowing processors to resume from

the exact point of failure without re-executing the side effects of already-completed stages; idempotent API design, grounded in persistent request identifiers and formal deduplication guarantees, ensures that any downstream interface called more than once does not produce inconsistent state. Periodic reconciliation serves as the final layer, systematically detecting and correcting residual faults arising from rare, compounded, or previously undetectable failure combinations that no other mechanism can fully address. Rather than treating each layer as an independent validation barrier, this reliability stack is most effective when its components are designed and operated as an integrated whole, enabling distributed event-driven systems to achieve exactly-once guarantees that are both practically strong and resilient to the unpredictable failure conditions inherent in production-scale environments.

References

[1] Diego García-Gil et al., "A comparison on scalability for batch big data processing on Apache Spark and Apache Flink," Big Data Analytics, 2017. [Online]. Available: https://link.springer.com/content/pdf/10.1186/s41044-016-0020-2.pdf

[2] Martin Kleppmann, "Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems." Sebastopol, CA: O'Reilly Media, 2017. [Online]. Available: https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/

[3] Theofanis P. Raptis and Andrea Passarella, "A survey on networked data streaming with Apache Kafka," IEEE Access, 2023. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=10213406

[4] Jay Kreps et al., "Kafka: A distributed messaging system for log processing," in Proc. NetDB Workshop, 2011. [Online]. Available: https://pages.cs.wisc.edu/~akella/CS744/F17/838-CloudPapers/Kafka.pdf

[5] Valerii Vladimirovich Petrov, "Current Issues and Methods of Event Processing in Systems with Event-Driven Architecture. Journal of Theoretical and Applied Information Technology, 2021. https://www.jatit.org/volumes/Vol99No9/2Vol99No9.pdf

[6] Tyler Akidau et al., "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, Vol. 8, No. 12, 2015. https://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf%20%28Google

[7] Olaf Zimmermann et al., "A Decade of Enterprise Integration Patterns: A Conversation with the Authors," IEEE Xplore, 2016. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7368007

[8] Pat Helland, "Idempotence Is Not a Medical Condition," ACM Digital Library, 2012. https://dl.acm.org/doi/pdf/10.1145/2160718.2160734

[9] Barbara Liskov, Liuba Shrira, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," ACM Digital Library, 1988. https://dl.acm.org/doi/pdf/10.1145/960116.54016

[10] James Curtis Corbett, et al., "Spanner: Google's Globally Distributed Database." ACM Digital Library, 2013. https://dl.acm.org/doi/pdf/10.1145/2491245

[11] Emmanuel Arinola, Yusuf Adebayo, "Optimizing Distributed Database Systems for Scalable Cloud Computing Environments," ResearchGate, 2025. https://www.researchgate.net/profile/Bruce-William-2/publication/396371834

[12] Anup K. Kalia, “Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices,” ACM Digital Library, https://dl.acm.org/doi/pdf/10.1145/3468264.3473915

[13] Han Wu et al., "A Reactive Batching of Apache Kafka for Reliable Stream Processing in Real Time," [Online]. Available: https://www.researchgate.net/profile/Han-Wu-28/publication/346444550_A_Reactive_Batching_Strategy_of_Apache_Kafka_for_Reliable_Stream_Processing_in_Real-time/links/620a5e3eafa8884cabe2f98c/A-Reactive-Batching-Strategy-of-Apache-Kafka-for-Reliable-Stream-Processing-in-Real-time.pdf

[14] Anas Nadeem, Muhammad Zubair Malik, "A Case for Microservices Orchestration Using Workflow Engines," ACDigital Library, 2022 [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3510455.3512777

[15] Noora Backlund, "Implementing Amazon Web Services integration connector with IBM App Connect Enterprise," Jamk.fi, 2020. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/346814/Thesis_Noora_Backlund_FINAL.pdf?sequence=2

[16] Paris Carbone et al., "State management in Apache Flink," VLDB, 2017. [Online]. Available: https://www.vldb.org/pvldb/vol10/p1718-carbone.pdf,

[17] G. Ramalingam and Kapil Vaswani, "Fault Tolerance via Idempotence," ACM Digital Library, 2013. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2429069.2429100

[18] Peter Bailis et al., "Bolt-on causal consistency," ACM Digital Library, 2013. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2463676.2465279