The American Journal of Engineering and Technology
ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 127-134
DOI 10.37547/tajet/Volume07Issue06-14

Check for updates

Data Consistency in Distributed Multi-Stage Event Processing Pipelines

Khrystyna Terletska

Senior Software Engineer at Datadog New York, USA

OPEN ACCESS

SUBMITED 18 April 2025
ACCEPTED 25 May 2025
PUBLISHED 18 June 2025
VOLUME Vol.07 Issue 06 2025

CITATION

Khrystyna Terletska. (2025). Data Consistency in Distributed Multi-Stage Event Processing Pipelines. The American Journal of Engineering and Technology, 7(06), 127–134. https://doi.org/10.37547/tajet/Volume07Issue06-14

Abstract: The article examines the problem of ensuring end-to-end data consistency in distributed multi-stage event processing pipelines, which are actively used in modern real-time systems. The relevance of the study is determined by the rapid growth of streaming analytics needs and the widespread use of Apache Kafka, making message latency, duplication, and disorder critical factors for industries ranging from fintech to IoT. The goal of this work is to propose a formal model that unifies an extended event representation and a set of invariants that guarantee correct processing even in the presence of component failures. The novelty of the approach lies in the formalization of an event as a tuple ⟨id, $ts_{sr}c$, p, v, σ⟩, where id is responsible for deduplication, $ts_{sr}c$ records the time of occurrence, p specifies the partition, v is the payload, and σ is the schema version, which enables ordering recovery and supports format evolution. The pipeline is modeled as a directed acyclic graph (DAG) of operators having the properties of determinism, idempotence, and monotonicity. CRDT aggregates are used for convergence in duplication; SLA alerts from watermark mechanisms are used to minimize data loss. The main findings indicate that, under specified conditions, the system can tolerate delays, failures, and redeliveries without compromising consistency. Extended events and formal operators enable state recovery; stream semantics are ensured by four invariants. This research is particularly relevant for professionals designing and operating real-time event-driven systems, stream processing applications, microservices architectures, and high-integrity data integration pipelines.

KEYWORDS

streaming event processing, distributed systems, data consistency, logical clocks, vector clocks, CRDT, schema evolution, checkpoints, Apache Kafka, Exactly-Once Semantics.

INTRODUCTION

Over the past five years, the volume of data requiring sub-second processing latency has increased by orders of magnitude. The streaming analytics market, according to Markets & Markets, is projected to grow from USD 29.53 billion in 2024 to USD 125.85 billion by 2029, corresponding to a compound annual growth rate of 33.6 % [1]. The infrastructural foundation of this growth has become event platforms, primarily Apache Kafka, which is used by more than 80% of Fortune 100 companies [2]. Such a scale of adoption means that for many industries—from fintech to IoT networks—continuous stream processing has become not an auxiliary but a critically important function.

The industry's pragmatic response has manifested in the increasing complexity of pipelines themselves: the same event now traverses the chain ingestion → enrichment → filtering → aggregation → storage, with each stage served by isolated microservices or dedicated frameworks. The practical cost of an error grows exponentially: a failure at any stage immediately impacts e-commerce recommendations, risk calculations in banking, or vehicle telemetry monitoring.

However, the asynchronous nature of distributed systems raises questions about the concept of end-to-end integrity. Between nodes, delays, duplications, and reordering of messages are possible. Operators aggregate data with only partial knowledge of the global time, and individual services may temporarily drop out of the network. In such conditions, the pipeline developer is forced to balance speed, availability, and correctness, with the compromise often becoming implicit: the loss of a single event can lead to an incorrect dependency graph, while redelivery can lead to inflated metrics or inaccurate accounting.

Existing theoretical frameworks do not close this gap. ACID transactions guarantee atomicity only within a single store and do not describe streams where data changes on the fly. The CAP theorem formulates the boundaries of replication but operates on static objects rather than transformational stages. Lambda and Kappa architectures provide organizational schemes for combining batch and streaming processing; however, they delegate the consistency problem to the level of individual operators and do not offer formal invariants that cover the entire multi-stage event journey. As a result, engineers are forced to invent local solutions—from custom idempotent keys to complex replay protocols—without a single overarching model capable of guaranteeing predictable pipeline behavior even in the presence of failures and schema evolution.

MATERIALS AND METHODOLOGY

The study of data consistency in distributed multi-stage event processing pipelines relies on the analysis of 13 sources: industry reports from Markets & Markets [1], Apache Kafka documentation [2, 4], scientific publications on logical and vector clocks [6, 7], Confluent materials on the Avro/Protobuf registry [9, 10], research on stream watermarks in Flink [8], CRDT approaches to aggregation [12], and Saga pattern rollback patterns in microservices [13].

The theoretical basis was formed by formalizing an event as a tuple ⟨id, ts_{src}, p, v, σ⟩, where id provides deduplication, ts_{src} specifies the time of occurrence, p defines the partition, v contains the payload, and σ represents the schema version. This representation enables the recovery of the original order and accounts for format evolution [4, 9].

Methodologically, the work combines: (1) a comparative analysis of ordering within partitions (idempotent producer and sequence-id) and causal ordering across partitions (Lamport logical clocks and vector clocks) [5, 6, 7]; (2) a systematic review of schema compatibility verification practices (BACKWARD, FORWARD, FULL modes) with automatic blocking of incompatible changes in CI/CD [9, 10]; (3) the study of the atomicity mechanism of checkpoints via barrier messages, ensuring Exactly-Once Semantics and a hybrid 2PC + Saga scheme for global commit or rollback [4, 11, 13]; (4) analysis of CRDT-based aggregate processing, guaranteeing state convergence during replay and event duplication [12]; (5) evaluation of the impact of watermarks on data loss (up to 33%) and the implementation of SLA alerts for timely watermark advancement [8].

RESULTS AND DISCUSSION

The formal foundation of the proposed model begins with the definition of a unified representation of an event as the tuple ⟨id, ts_{src}, p, v, σ⟩. The unique identifier, id, ensures deduplication; the source timestamp, ts_{src}, records the moment of occurrence; the partition key, p, determines the partition in the message log; the payload, v, contains business data; and σ indicates the schema version. Such an extended event carries sufficient context to reconstruct both the original order and the transformations applied to it at any stage during pipeline replay.

The stages themselves form a directed acyclic graph G = (S, E), where S is a finite set of operators and E is the set of delivery channels. Each vertex is described by a function f_s: E* → E*, which accepts a multiset of input events and produces a multiset of outputs. For reliability, f_s must remain deterministic: identical inputs always produce identical outputs upon any repeated execution. The second required characteristic is idempotence, meaning that repeated application of the operator to the same event does not change the result: $f_s(f_s(e)) = f_s(e)$. Finally, monotonicity is formulated as the inclusion $f_s(A) \subseteq f_s(B)$ for any $A \subseteq B$; this property guarantees that partial results can be safely extended without a global rollback.

At the system level, consistency is enforced by four invariants. The first, I_1, is the preservation of order within each partition. This is achieved by the log itself: Kafka writes and delivers events strictly in the order in which they arrived with a given key p, regardless of reader parallelism. The second, I_2, involves causal ordering between partitions; this is implemented via Lamport logical clocks or, when necessary, complete vector clocks, which enable the reconstruction of a correct directed acyclic graph (DAG) of causality in the event of delayed or conflicting events [3]. The third, I_3, is schema compatibility at stage boundaries. Each event carries a schema version σ, and upon reading, a stage performs validation: a transition σ_in → σ_out is declared permissible only if the operation belongs to the class of backward- or forward-compatible changes, for example add a field with a default value; otherwise, the stream is blocked until migration occurs. Finally, the fourth, I_4, is the atomicity of checkpoint commit upon passing a control barrier. A stage acknowledges the upstream offset only after all its local states have been saved and downstream channels have accepted the barrier, which makes the global commit equivalent to a single transaction and eliminates divergence between data replicas [4].

When these conditions are jointly satisfied, an event does not lose ordering, is correctly interpreted despite any schema evolution, and is either fully persisted at all stages or rolled back entirely. The properties of associativity, commutativity, and idempotence, inherent in CRDT-like aggregate processing, further guarantee that the result converges to a single state even in the presence of delays and packet duplication. Thus, the formal model establishes a verifiable framework on which to rely when designing distributed pipelines that require strict end-to-end consistency.

Figure 1 illustrates how partitioning a topic into multiple partitions simultaneously achieves both preserved message order (I_1) and horizontal write scaling. Each producer selects a specific partition (depending on the key or routing logic), and its events are appended strictly to the end of the chosen logical segment without overlapping with other partitions. As a result, concurrent work by Producer client one and Producer client two on different partitions (P1, P3, and P4) does not violate ordering within any of them, and it also simplifies processing and load balancing, yielding high throughput while maintaining a deterministic delivery order for each key.

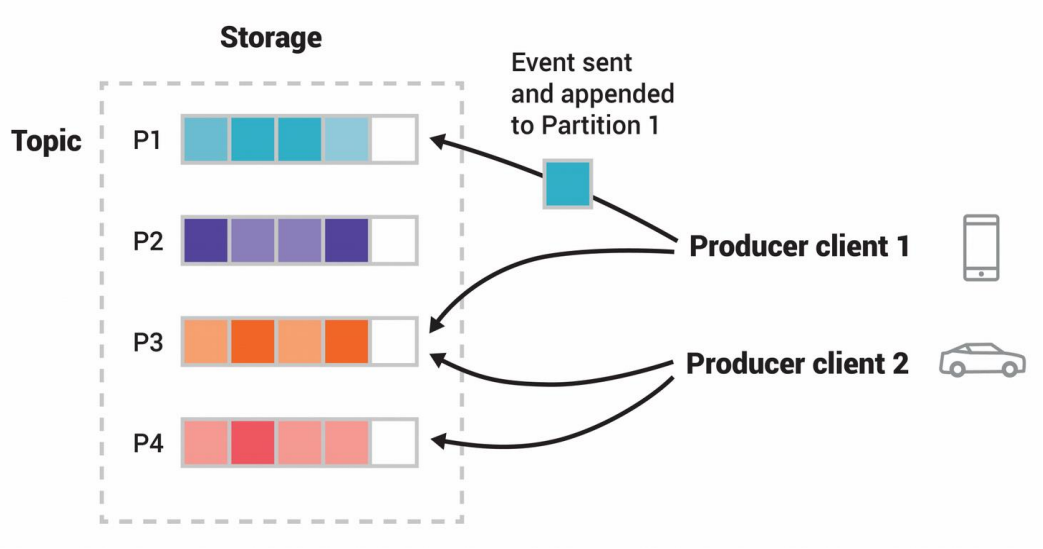

Fig. 1. Producer Clients Writing Events to Kafka Topic Partitions [4]

Event ordering begins within each partition: a Kafka log segment is an immutable list, so all records from a single producer with the same key p arrive and are read strictly in the sequence in which they were sent, provided replication quorum is maintained [5]. In failure scenarios, duplicates may occur; however, an idempotent producer assigns each record a monotonically increasing sequence ID, thereby restoring relative order after a restart and allowing the consumer to rely on a shifting rather than skipping offset. This local invariant remains inexpensive: costs are linear in the number of events, and order verification reduces to comparing neighboring offsets, which is O(1) per message.

Inter-partition order is maintained not by absolute global time but by causality. Lamport logical clocks append a counter t to each message, guaranteeing that the recipient never observes an effect with a timestamp less than its local clock if that effect occurred later [6]. For streams where dozens of services compete, this is insufficient: a unique scalar cannot distinguish competing paths. Vector clocks extend t to an N-element array (one element per active node), and partial order is then determined by component-wise comparison, providing precise happened-before relationships even with parallel processing branches [7]. An example of vector clock operation is shown in Figure 2.

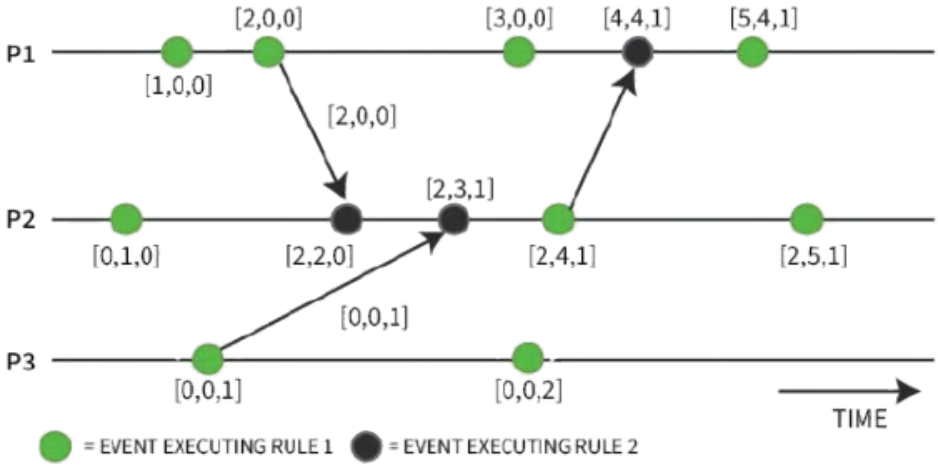

Fig. 2. Vector Clock Algorithm [7]

The price of precision is O(N) memory in each message and O(N) merge complexity. Therefore, practical pipelines usually limit the vector size to a set of essential sources or reduce it to a hash of the causal graph, which leaves a risk of false equality but avoids unbounded metadata growth.

For stages based on windowed aggregation to complete a window at all, they need a detector for the logical end of the stream. This task can be solved using watermarks: the source regularly publishes a watermark w, promising no events with timestamps ts < w. Each subsequent stage advances w forward when it has processed all prior events and its lag does not exceed the configured tolerance. If the source is stuck, an idle detector advances w after a timeout so as not to block the entire graph. Observations indicate that overly conservative strategies result in significant data loss. A study [8] found that up to 33% of records were not processed when half of the keys were delayed at the median of the delay distribution. Therefore, the platform accompanies water markers with SLA alerts. Suppose the difference between the actual wall-clock time and the last w for any key exceeds X seconds. In that case, the orchestrator raises the priority of the corresponding streams or activates a fallback replay branch.

A separate barrier mechanism is needed to capture a consistent snapshot of the state across multiple stages. The producer-coordinator inserts a special barrier message B into the log, and each stage forwards it only after it has flushed its checkpoint. When B returns to the coordinator from all branches, the commit has atomically covered the entire pipeline. A timeout turns the operation into an abort and initiates a rollback to the previous stabilized barrier; such a scheme requires only O(E) messages per round, where E is the number of channels, and scales robustly to hundreds of parallel streams.

Finally, the model's complexity has clear boundaries. Per-partition ordering scales horizontally because adding partitions does not complicate the algorithm. Complete global sorting, as shown in practice, degrades throughput fourfold and negates the benefits of sharding; therefore, most authors agree that causal (rather than total) ordering and carefully chosen keys are sufficient. Vector clocks become unacceptable when N approaches hundreds, and barriers cease to be effective if the event's path exceeds the graph's diameter—then checkpoint latency grows linearly and conflicts with the watermark generation interval. These limitations emphasize that ordering guarantees must be designed based on real usage patterns rather than as a universal total sort of everything passing through a distributed pipeline.

Schema consistency begins with each event carrying a version σ registered in a centralized Avro / Protobuf registry. The registry stores the complete history and assigns a monotonic number. The producer serializes the payload, prefixes it with the schema identifier, and the consumer extracts the ID, decodes the version, and applies the local deserializer. By default, compatibility checking is performed before publishing: the new schema is compared to the last saved one, and the operation is blocked if a violation of the selected mode—BACKWARD, FORWARD, or their transitive variants—is detected [9].

Adding a field with a default value is considered non-destructive: consumers not yet aware of the field ignore it, while producers immediately begin populating the new field. Dropping an optional field is a safe drop as long as downstream code does not rely on its presence. Renaming requires a two-step process—first add + deprecate, then physical removal; a single-step rename is deemed incompatible. Finally, changing a type (for example, int → string) is allowed only via an additional alias or migration. Otherwise, the deserialization invariant is broken.

To prevent such changes from being introduced spontaneously, teams establish a strict compatibility contract. In the registry, a domain, topic, or event type is designated as a subject, and for each subject, a BACKWARD, FORWARD, or FULL policy is assigned. The FULL policy (both backward and forward simultaneously) is used rarely due to high testing costs. Confluent, by default, enables BACKWARD, as it allows consumers to rewind to earlier offsets and reread history without additional migration [10].

Control is passed to the CI/CD pipeline: a pull request with any new schema triggers a task that registers it in a test registry and checks for restrictions. If the transition $\sigma_n \rightarrow \sigma_{n+1}$ is not compatible, then the build fails before deployment, with a diff report sent back to the developer. This check is supplemented by an RBAC policy and tags that prohibit any ad hoc schema updates in production without review; this has become a mandatory data quality practice in large installations for some time.

Correct schema evolution helps only if the pipeline operators themselves perform transformations

correctly. The basic requirement is determinism: a stateless map (key, value) function always produces the same result, and a stateful aggregate reduce (k, state, v) produces the same output when replaying the log. The second guarantee is idempotence; here, the upsert pattern is applied, where the output key matches the original ID, and the record overwrites the previous version instead of creating a new one. With Exactly-Once Transactions enabled, Kafka maintains a dual producer-id/sequence-id counter and commits only when all records of a batch have been successfully written to the log and are ready for consumption, thereby eliminating duplicates even in the event of network failures. The cost of such precision has been measured as an additional 30–40% p99 latency overhead and a minimum end-to-end latency equal to the sum of the commit intervals of all subtopologies of the stream [11].

The third property is monotonicity. Aggregates built as CRDTs or their commutative analogues possess merge operations that are simultaneously associative, commutative, and idempotent. Upon replaying the log, the storage layer simply adds new state fragments, and the result converges to a single, exact state, regardless of the delivery order, as formally proven in works on type-checking CRDT convergence and synthesizing state-based structures [12]. A comparison of typical and CRDT methods is shown in Figure 3.

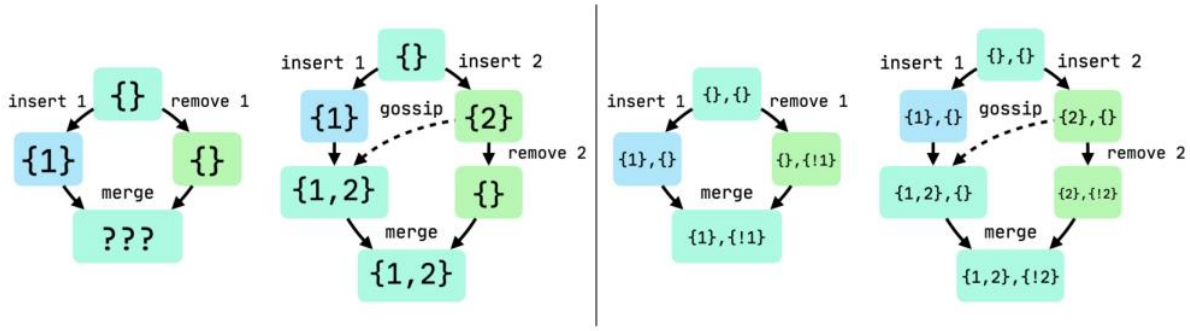

Fig. 3. Situations where a sequential type (left) has consistency issues that are resolved by a CRDT (right) [12]

The combination of determinism, idempotence, and monotonicity greatly simplifies recovery after a failure. When a node fails, the executor rereads events starting from the last barrier, reapplies all operations to a clean snapshot, and is guaranteed to reach the same final state; recovery time is bounded by the number of events between barriers and the full commit interval, which was already measured in the previous complexity evaluation section. Transient errors during writes to an external system also become reversible: retry either does not change the result or yields a non-idempotent error at the protocol level, without breaking the integrity of the entire multi-stage event path.

At the transport level, pipeline resilience is determined by which delivery semantics are chosen by producers and consumers. At least once, each message is replicated until acknowledged, but in the event of a failure, the producer may resend an already recorded frame, causing duplication and requiring idempotent operators. Exactly-once extends the protocol with two mechanisms: the idempotent-producer adds several bytes of a sequence number to the header, and the transactional coordinator commits a batch only after all affected partitions have acknowledged the write.

To prevent a node failure from nullifying computation, each stage is equipped with regular state snapshots and a write-ahead log protocol. Flink's algorithm launches an asynchronous checkpoint by marking the stream with a control barrier, then snapshots the operator state and the input log offsets; as soon as all partitions acknowledge the same barrier, the coordinator saves the global meta-descriptor and allows the producer to advance the upstream topic offset.

The coordinator inserts a special barrier message into the stream, and each stage, upon receiving it, completes its local transaction and passes the barrier onward. When the same identifier is returned to the coordinator

from all branches, a commit point is recorded. If any stage misses the timeout, a compensating chain is triggered, analogous to rollbacks in the Saga pattern described in modern microservices guides [13]. This approach avoids the blocking of a long-lived 2PC with external APIs while preserving atomicity I_4, ensuring that the pipeline either swallows the barrier in its entirety or rolls back all operations.

Partial rollback and roll-forward are based on the fact that each operator is deterministic and idempotent. Upon detecting a discrepancy, the coordinator computes the minimal prefix of stages where ordering or schema version was violated, rolls them back to the last checkpoint, and replays events up to the current barrier. Since operators are monotonic, repeated applications do not alter the already agreed-upon state, and the reconciliation time T_reconcile is bounded by the sum of the maximum checkpoint interval, the network delay to the furthest stage, and the local roll-forward time.

Thus, the presented formal model demonstrates that end-to-end consistency in multi-component distributed pipelines is achievable under several key principles: preserving order within and between partitions (invariants I_1 and I_2) ensures correct stream semantics, strict schema compatibility checks (I_3) eliminate risks of inconsistent deserialization, and checkpoint atomicity (I_4) guarantees a global all-or-nothing state commit. Detailed descriptions of deterministic, idempotent, and monotonic operators, as well as barrier and reverse-log mechanisms, confirm that with proper design and orchestration, the system can withstand failures, duplications, and delays without losing consistency. At the same time, practical limitations (metadata growth in vector clocks, the cost of total sorting, and checkpoint latencies) underscore the necessity of adapting the model to real-world scenarios.

CONCLUSION

The work demonstrates that strict end-to-end consistency in distributed multi-stage event processing pipelines is achieved under three primary conditions. First, the extended event representation ⟨id, $ts_{sr}c$, p, v, σ⟩ enables the restoration of order, deduplication of messages, and tracking of schema evolution. Second, modeling the pipeline as a directed acyclic graph (DAG) with deterministic, idempotent, and monotonic operators ensures predictability when replaying the log: identical input produces identical output, and repeated application does not alter the result. Finally, four invariants— preservation of order within partitions (I_1), causal order between partitions (I_2), schema compatibility checks (I_3), and checkpoint atomicity (I_4) — combine into a unified system that ensures correct stream semantics, protection against inconsistent deserialization, and a global all-or-nothing state commit.

The mechanisms for preserving order are based on Kafka's immutable logs (I_1) and logical or vector clocks (I_2). Lamport logical clocks provide causality when the number of nodes is small, while vector timestamps, although requiring O(N) memory, accurately reconstruct the DAG in the presence of competing branches. For schema consistency (I_3), each event carries a version σ, and a centralized Avro/Protobuf registry, together with CI/CD checks, prevents incompatible changes. Checkpoint atomicity (I_4) is achieved through barrier messages: stages persist local states and forward the barrier, and the global commit is recorded only after all branches acknowledge; on timeout, a rollback to the previous stable state is initiated.

The limitations of the model stem from the growth of metadata in vector clocks with large numbers of nodes, resulting in diminished throughput under total global sorting. Checkpoint latency also increases in deeply branched pipelines. However, the model remains applicable to real systems through the choice of causal ordering, accompanied by CRDT-like aggregates, and the dynamic tuning of watermark and barrier parameters.

REFERENCES

"Streaming Analytics Market Size, Share, Growth Drivers & Opportunities," Markets and Markets, Sep. 2024. https://www.marketsandmarkets.com/Market-Reports/streaming-analytics-market-64196229.html (accessed Apr. 30, 2025).

"Apache Kafka," Apache. https://kafka.apache.org/ (accessed May 01, 2025).

G. Manepalli, "Clocks and Causality - Ordering Events in Distributed Systems," Exhypothesi, Nov. 16, 2022. https://www.exhypothesi.com/clocks-and-causality/ (accessed May 02, 2025).

"Kafka 4.0 Documentation," Apache.

https://kafka.apache.org/documentation/ (accessed May 03, 2025).

J. Rao, “Configuring Durability, Availability, and Ordering Guarantees,” Confluent. https://developer.confluent.io/courses/architecture/guarantees/ (accessed May 05, 2025).

“Lamport’s logical clock,” Geeks for Geeks, Oct. 02, 2023. https://www.geeksforgeeks.org/lamports-logical-clock/ (accessed May 07, 2025).

“Vector Clocks in Distributed Systems,” Geeks for Geeks, Oct. 14, 2024. https://www.geeksforgeeks.org/vector-clocks-in-distributed-systems/ (accessed May 08, 2025).

T. Yasser, T. Arafa, M. ElHelw, and A. Awad, “Keyed watermarks: A fine-grained watermark generation for Apache Flink,” Future Generation Computer Systems, vol. 169, p. 107796, Aug. 2025, doi: https://doi.org/10.1016/j.future.2025.107796.

“Schema Evolution and Compatibility for Schema Registry on Confluent Platform | Confluent Documentation,” Confluent. https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html (accessed May 10, 2025).

“Schema Registry API Reference,” Confluent. https://docs.confluent.io/platform/current/schema-registry/develop/api.html (accessed May 13, 2025).

Valeriu Crudu, “Exactly-Once Semantics in Kafka - Advanced Topics Explained,” MoldStud. https://moldstud.com/articles/p-exactly-once-semantics-in-kafka-advanced-topics-explained (accessed May 15, 2025).

S. Laddad, C. Power, M. Milano, A. Cheung, and J. M. Hellerstein, “Katara: synthesizing CRDTs with verified lifting,” Proceedings of the ACM on Programming Languages, vol. 6, no. OOPSLA2, pp. 1349–1377, Oct. 2022, doi: https://doi.org/10.1145/3563336.

“SAGA Design Pattern,” Geeks for Geeks, Nov. 08, 2024. https://www.geeksforgeeks.org/saga-design-pattern/ (accessed May 20, 2025).