Journal of Computer Science and Technology Studies
ISSN: 2709-104X
DOI: 10.32996/jcsts
Journal Homepage: www.al-kindipublisher.com/index.php/jcsts

JCSTS
AL-KINDI CENTER FOR RESEARCH AND DEVELOPMENT

| RESEARCH ARTICLE

Ensuring Exactly-Once Semantics in Kafka Streaming Systems

Pallavi Desai
Independent Researcher, USA
Corresponding Author: Pallavi Desai, **E-mail**: pallavidesaib@gmail.com

| ABSTRACT
Kafka's exactly-once semantics mark a major advancement in distributed streaming systems, solving one of the most persistent challenges in ensuring reliable data pipelines. This article provides a detailed examination of how Apache Kafka achieves end-to-end exactly-once guarantees through multiple integrated mechanisms. Beginning with producer-side idempotence, which prevents duplicate writes during retries or network failures, it then explores Kafka's transactional API that enables atomic operations across topics and partitions. It further evaluates Kafka Connect's extensions, which carry these guarantees into external systems by embedding transaction metadata, thereby addressing the challenges of integrating heterogeneous platforms. Additionally, the article analyzes Kafka's robustness in handling broker crashes, network partitions, and consumer group rebalances—showing how its transaction state management, timeouts, and offset coordination preserve data integrity even under failure. Finally, it highlights the business value of these capabilities across industries such as finance, IoT, cybersecurity, and manufacturing, while acknowledging the modest performance trade-offs involved.

| KEYWORDS

Transactional API, Idempotent Producer, Distributed Systems, Exactly-Once Semantics, Stream Processing

| ARTICLE INFORMATION

ACCEPTED: 01 August 2025 **PUBLISHED:** 8 September 2025 **DOI:** 10.32996/jcsts.2025.7.9.49

1. Introduction

Exactly-once semantics in Apache Kafka resolve a fundamental issue in distributed streaming: The reliable delivery of messages without duplication and/or loss. This analysis looks into the technical foundation that allows Kafka to provide these assurances through layers.

The evolution of exactly-once processing in distributed systems took a significant leap when Apache Kafka introduced this capability in version 0.11.0.0. Neha Narkhede, co-creator of Apache Kafka, details how the introduction of idempotent producers and the transactional API fundamentally shifted stream processing data integrity approaches. Before these advances, developers faced an unpleasant choice: implement complex application-level deduplication or accept the limitations of weaker guarantees like at-least-once or at-most-once processing [1]. The transactional producer API in Kafka 0.11 enables atomic writes across multiple partitions, treating message batches as unified transactions that either completely succeed or fail as a unit.

The practical impact extends far beyond technical elegance. Kai Waehner, Technology Evangelist and Field CTO, notes that exactly-once processing has become essential infrastructure for organizations building real-time data systems. The fusion of exactly-once semantics with emerging architectural patterns like data meshes, composable architectures, and AI pipelines has sparked innovation across sectors [2]. Financial services organizations implementing these capabilities report dramatic reductions in reconciliation overhead, directly affecting cost structures and regulatory compliance postures.

Manufacturing operations leveraging IoT networks particularly benefit from these guarantees. Production facilities using exactly-once semantics for sensor data pipelines document marked improvements in anomaly detection precision. Eliminating duplicate events that previously triggered redundant alerts reduces operational noise, allowing focus on genuine production issues. This capability proves especially valuable as manufacturing systems increasingly adopt edge computing architectures, with Kafka deployments spanning network boundaries where connectivity remains unpredictable [2].

Security monitoring represents another domain where exactly-once guarantees prove transformative. Event processing in security contexts demands absolute reliability to maintain accurate threat assessments. Security operations centers utilizing exactly-once semantics report enhanced threat detection precision by preventing artificial risk score inflation caused by duplicate event processing. This capability grows increasingly vital as security architectures fragment across cloud-native platforms and multi-cloud environments, trends Waehner identifies as accelerating through 2025 [2].

2. The Challenge of Exactly-Once Processing

Understanding why exactly-once semantics pose such difficulty requires examining the fundamental constraints of distributed systems. Networks fail, processes crash, and messages face potential duplication during retry operations. The traditional messaging systems normally provided an unsatisfactory option of at-most-once (meaning risking data), or at-least-once (meaning accepting duplicates).

The inherently limited problem is embodied in the CAP theorem. Soulaimane Yahyaoui further writes that this principle has put down hard limits to what can be promised by a distributed system in path failures [3]. When the network partitions, and this is an inevitability of open environments in production, the architect must decide in favor of consistency or availability. Most messaging systems traditionally prioritise availability over rigid guarantees of consistency, and therefore, strict exactly-once guarantees are extremely hard to achieve.

Kafka has a distributed architecture that adds to these challenges. A typical deploy is a fault-tolerant and throughput deploy, which partitions and replicates data across the cluster in brokers. The design includes several possible points of failure: network partitions, broker failure, and leader-election situations. The research highlights how these failure modes create numerous edge cases where messages risk duplication or loss [1]. Consider a producer that receives no acknowledgment for a sent message due to network issues—retrying the operation potentially creates duplicates if the original succeeded silently.

Traditional solutions involved the use of distributed transactions involving heavyweight two-phase commit protocols, which had a great cost in performance and availability issues. These strategies necessitated synchronising all involved nodes, leading to both increased delay and new points of failure. The fact that the performance impact made exactly-once semantics impractical in high-volume systems until Kafka developed its innovative model of idempotent producers and lightweight transaction coordination.

Kafka ensures true exactly-once properties by means of complementary mechanisms at the producer, broker, and consumer levels. Rather than implementing traditional distributed transactions with their associated overhead, Kafka employs what Narkhede describes as "a lightweight transaction protocol specifically designed for streaming data workflows" [1]. This protocol enables atomic message delivery across multiple partitions, ensuring all-or-nothing outcomes even during failures.

The architectural breakthrough stems from decomposing exactly-once semantics into distinct components: idempotent production, atomic multi-partition writes, and consistent offset management. Narkhede details how the idempotent producer mechanism uses sequence numbers, enabling brokers to detect and reject duplicates, addressing fundamental producer-broker reliability without requiring distributed locks [1]. This approach delivers exactly-once semantics for message production even when producers must retry operations due to network failures or broker unavailability.

This technical innovation has transformed data pipeline implementation. Rather than building custom deduplication logic or accepting weaker processing guarantees, developers now leverage Kafka's native exactly-once capabilities. Yahyaoui observes that this advancement represents a significant milestone in distributed systems technology, enabling reliable streaming applications without sacrificing the performance characteristics essential for high-throughput operations [3].

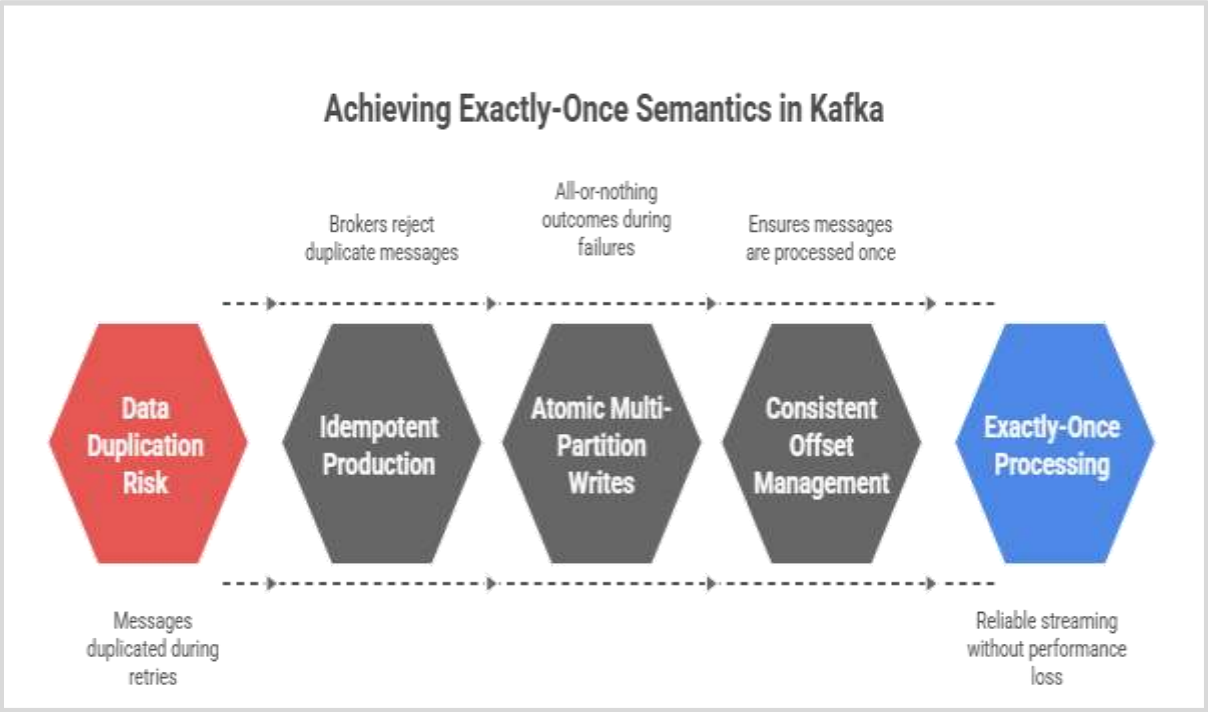

Fig 1: Achieving Exactly-Once Semantics in Kafka [1, 3]

3. Producer-Side Idempotence

Kafka is precisely once, to start with, the producer must be idempotent. This capability ensures that duplicate messages are not written to the log in case a particular producer is retried in order to transmit a message.

To enable this capability, developers must set:

```
·Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("enable.idempotence", "true");
```

·

With idempotence enabled, Kafka producers assign each message a sequence number. Brokers track these sequence numbers per producer session and reject duplicates, ensuring that even under retry conditions, messages are written exactly once.

According to Andy Bryant's detailed analysis of Kafka processing guarantees, producer idempotence addresses a key challenge in distributed messaging: the ambiguity that occurs when a producer doesn't receive an acknowledgment for a sent message. Bryant explains that without idempotence, producers face a dilemma when acknowledgments time out – they must either retry the send operation (potentially creating duplicates) or abandon it (potentially losing data) [4]. The idempotent producer resolves this dilemma by making retries safe, allowing producers to aggressively retry operations without the risk of creating duplicates.

The idempotent producer works through a mechanism that Bryant describes as "effectively a de-duplication cache on the broker side" [4]. When a producer initializes, it is assigned a unique producer ID (PID) and keeps a sequence number per partition where it writes. The broker keeps the greatest sequence number with each PID-partition combination it has processed and declines any message whose sequence number it has processed. This approach allows brokers to identify duplicate messages efficiently without maintaining an unbounded history of all messages ever processed.

Bryant notes that this mechanism differs from traditional distributed transaction protocols in a crucial way: it doesn't require coordination across multiple brokers or two-phase commits for basic idempotence [4]. This architectural decision is key to

maintaining Kafka's performance characteristics while adding reliability guarantees. The sequence number tracking occurs independently on each partition leader, allowing the system to scale horizontally without introducing cross-partition coordination overhead for basic idempotent production.

For full transactional capabilities, producers must also be assigned a transactional ID:

- `props.put("transactional.id", "prod-1");`
-

The transactional ID maintains producer state across sessions, allowing the system to resolve any in-progress transactions if a producer fails and restarts. As explained in Hevo Data's comprehensive guide to Kafka exactly-once semantics, the transactional ID serves a critical role in fencing off zombie producers – instances of a producer that were considered dead but later rejoined the cluster [5]. Without this fencing mechanism, a producer that was temporarily partitioned from the network might come back online and continue sending messages, potentially creating duplicates or an inconsistent state.

Hevo Data's analysis highlights that the transactional ID enables the cluster to track producer epochs, which increase monotonically each time a producer with a given transactional ID initializes [5]. If a broker receives messages from a producer with an older epoch than what's currently recorded for that transactional ID, it rejects those messages, effectively preventing zombie producers from causing inconsistencies. This mechanism ensures that at any given time, only one instance of a producer with a particular transactional ID can successfully write to the cluster.

When a producer with a transactional ID initializes, it performs what Hevo Data describes as a "fence" operation with the transaction coordinator [5]. This process also checks that there are no pending transactions in previous instances of the producer left to complete and completes those transactions in case any are left before permitting the new instance to start dispatching messages. A transaction coordinator has the power to abort halfway through transactions or transactions of past instances having the same transactional ID so as to have a fresh start-up of the producer instance.

Together with the producer idempotence and transactional IDs, this forms a strong basis for the exactly-once semantics of Kafka. These mechanisms overcome the producer-side challenges of reliable message delivery by assuring that every message is written into the log once and only once in the face of retries and producer shutdowns, and restarts. They are a great improvement in terms of reliability guarantees offered by Kafka and allow developers to build systems that support data integrity even through a variety of failure scenarios [4][5].

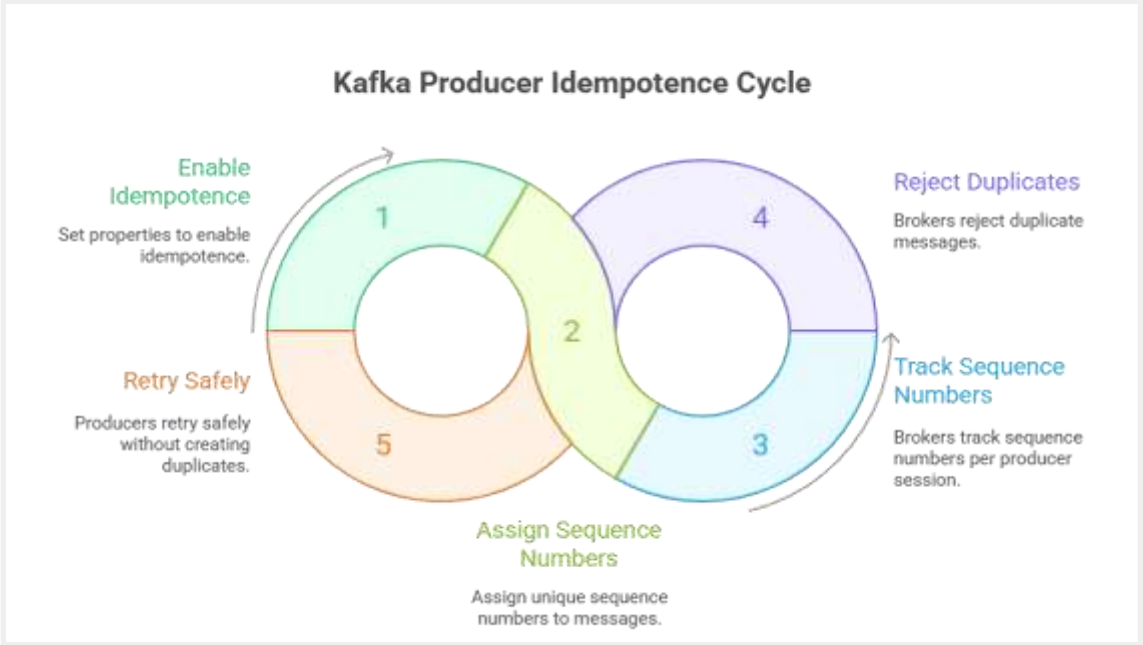

Fig 2: Kafka Producer Idempotence Cycle [4, 5]

4. Kafka's Transactional API

The transactions provided by Kafka enable sets of write operations to be performed atomically, i.e., they all succeed or all fail. This is made possible by the phase commit protocol that is coordinated by a transaction coordinator.

A typical transactional workflow looks like this:

```
·KafkaProducer<String, String> producer = new KafkaProducer<>(props);
producer.initTransactions();
try {
    producer.beginTransaction();
    // Send messages to multiple topics
    producer.send(new ProducerRecord<>("topic1", "key1", "value1"));
    producer.send(new ProducerRecord<>("topic2", "key2", "value2"));
    // Commit the transaction
    producer.commitTransaction();
} catch (KafkaException e) {
    // Abort the transaction on error
    producer.abortTransaction();
    throw e;
}
```

·

The coordinator of transactions oversees the lifecycle of any transaction and maintains the atomicity of the transactions in an environment where failure occurs. When a transaction is committed, the coordinator issues a two-phase commit of all the messages, which gives them visibility to consumers.

According to Jason Gustafson's detailed explanation in the Confluent blog, Kafka's transactional capabilities extend far beyond simple atomic writes to include the critical "read-process-write" cycle needed for stream processing applications [6]. Gustafson details that Kafka transactions solve the inherent problem with stream processing: how to ensure that consumer offsets are successfully committed at the same time as the output records producing them, thus providing end-to-end exactly-once processing. This is a fundamental requirement in the case of an application that consumes information in input topics and transforms to produce results in output topics.

Gustafson discusses that the implementation of the transaction is reliant upon one specialized component, the transaction coordinator, the role of which is to manage the state of all of the active transactions. Each transaction is assigned to a specific coordinator based on the transactional ID, with the coordinator maintaining transaction state in a dedicated internal topic for durability. This design ensures that transaction state survives broker failures, allowing the system to recover consistently even after crashes [6].

When examining the transaction protocol itself, Gwen Shapira, Neha Narkhede, and Todd Palino provide a comprehensive analysis in "Kafka: The Definitive Guide" [7]. They explain that Kafka's transaction protocol is specifically optimized for streaming workloads, differing from traditional database transactions in several key aspects. The protocol uses a two-phase approach where the transaction coordinator first writes a "prepare commit" record to its transaction log and then writes markers to each partition involved in the transaction.

These transaction markers serve a crucial role in the visibility of transactional messages to consumers. As explained in "Kafka: The Definitive Guide," the markers indicate to consumers whether the messages from a transaction should be delivered or filtered out, depending on the consumer's isolation level configuration [7]. Consumers configured with "read_committed" isolation level will only see messages from committed transactions, effectively implementing the "read committed" isolation level familiar from database systems.

What makes Kafka's transaction implementation particularly powerful is its ability to span multiple topics and partitions without sacrificing performance. The authors of "Kafka: The Definitive Guide" note that the design avoids coordination between partition leaders during normal operation, with coordination occurring only through the transaction coordinator [7]. This architecture allows the system to maintain high throughput even with transactions enabled.

For stream processing applications, Kafka transactions solve the "dual-write problem" where outputs and consumer offsets must be committed together. The `sendOffsetsToTransaction()` method, as Gustafson explains in his article, enables an application to commit consumer offsets in the same transaction as its output records [6]. Such an atomic commitment means that, in the event a processing application crashes and then is restarted, it will resume with the correct offset, avoiding losing data as well as re-processing it.

Kafka transactional API has performance implications, but these can be accommodated by most applications. Both sources admit that transactions introduce certain overheads into the operations, mostly in the form of transaction initialization and commit processes. They stress, however, that the throughput overhead is usually not very big with the properly set-up systems and so transactions prove viable in production environments where precisely-once semantics is imperative [6][7].

With these transactional capabilities, Kafka helps developers to create reliable streaming applications, which guarantee the consistency of data when different degrees of failure occur. The capacity to atomically coordinate writes across multiple topics and partitions, along with the incorporation of consumer offset management, is a key improvement in stream processing technology.

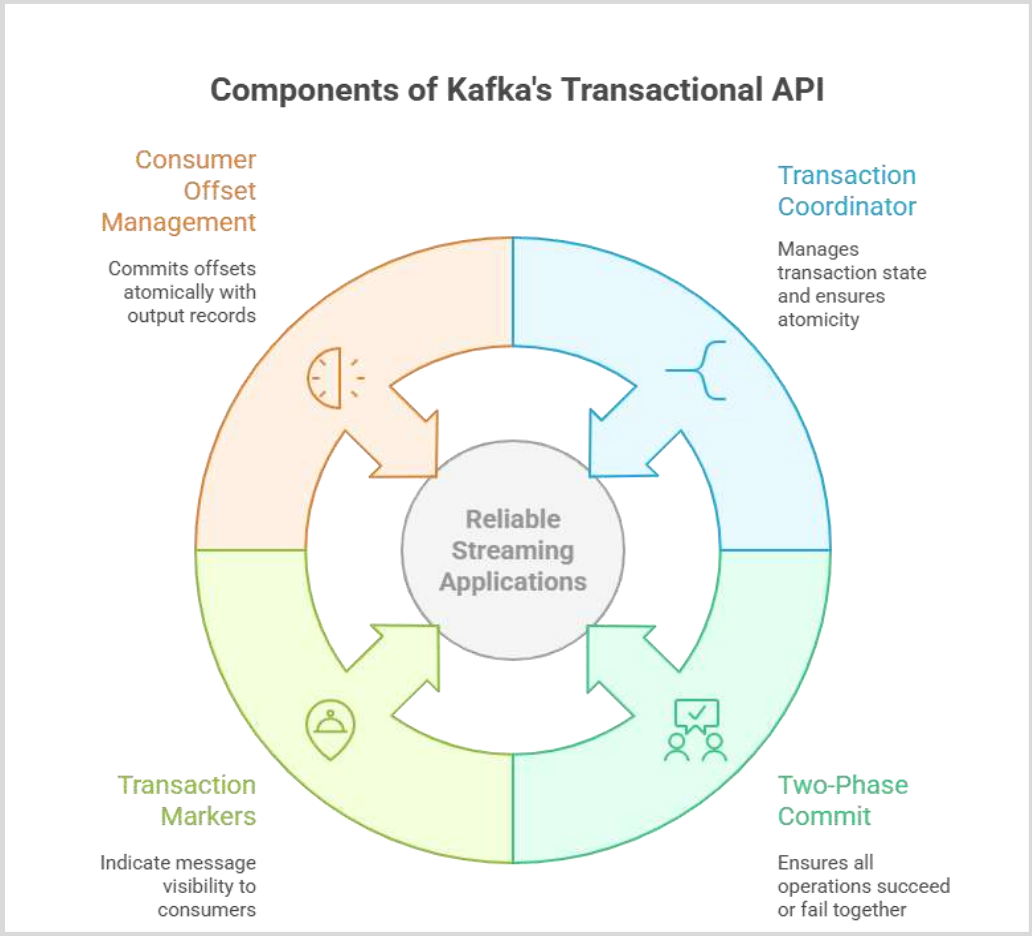

Fig 3: Components of Kafka's Transactional API [6, 7]

5. Kafka Connect for Exactly-Once Delivery

For Kafka Connect sinks, exactly-once delivery relies on the `transforms.InsertTransactionMetadata` configuration:

·transforms=insertTx

transforms.insertTx.type=org.apache.kafka.connect.transforms.InsertTransactionMetadata$Value

·

This transformation embeds transaction metadata into records, providing downstream connectors the essential context needed for exactly-once delivery to external systems.

Jay Kreps identifies extending exactly-once guarantees beyond Kafka's boundaries as perhaps the most formidable challenge in constructing truly reliable end-to-end data pipelines [8]. While Kafka's internal transaction mechanisms deliver robust guarantees within its ecosystem, integration with external systems introduces complex boundary challenges that demand specialized techniques.

The `InsertTransactionMetadata` transformation tackles this boundary problem by augmenting records with critical transaction identifiers and sequence data that enables connectors to precisely track transaction boundaries and identify potential duplicates [8]. This approach delivers what Kreps terms "effectively exactly-once delivery" even to systems lacking native transactional capabilities.

Implementation approaches vary dramatically based on target system characteristics. When working with systems offering native transaction support, such as relational databases, connectors leverage the embedded metadata to bundle records into atomic units that mirror Kafka's transaction boundaries. This approach, as Kreps observes, enables genuine end-to-end exactly-once semantics where record batches from Kafka transactions commit to external systems as indivisible units [8].

Systems lacking native transaction support require alternative strategies. Ravi Teja Bandaru details how connectors must implement bespoke idempotence mechanisms leveraging the embedded transaction metadata [9]. These implementations typically generate distinctive identifiers from transaction metadata to detect and filter duplicate records. Bandaru emphasizes how the transformation supplies what he calls "transaction coordinates" for each record—transaction ID, sequence number, and boundary markers—serving as the foundation for implementing exactly-once semantics across diverse target systems [9].

Performance implications merit careful consideration but remain manageable. Kreps acknowledges that enabling these guarantees introduces processing overhead compared to simpler delivery models [8]. However, Bandaru points out that well-designed connectors implement batching strategies that distribute the cost of transaction metadata tracking across multiple records, minimizing coordination overhead while preserving exactly-once guarantees [9].

The `InsertTransactionMetadata` transformation serves as the critical bridge spanning Kafka's internal guarantees and external systems, enabling architects to construct end-to-end pipelines with consistent delivery guarantees throughout the entire data path.

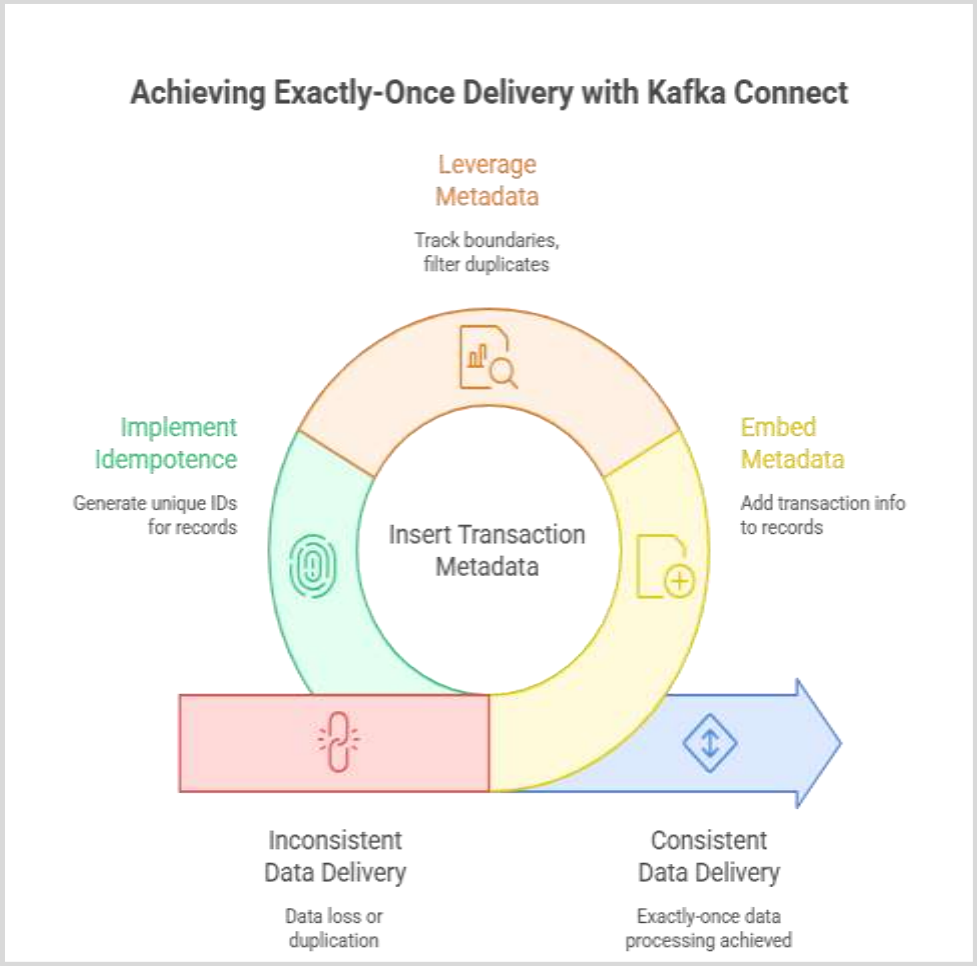

Fig 4: Achieving Exactly Once Delivery with Kafka Connect [8, 9]

6. Handling Failure Scenarios

Kafka's exactly-once semantics demonstrate remarkable resilience even under challenging failure conditions:

6.1 Broker Crashes

When a broker crashes during transaction processing, the transaction coordinator detects the failure through ZooKeeper heartbeats. Upon broker restart, the transaction coordinator definitively resolves pending transactions based on their pre-crash state.

Shivani Sudan's analysis reveals how Kafka's resilience during broker failures stems from its transaction state management architecture [10]. Sudan details how Kafka maintains transaction state in a dedicated internal topic named "__transaction_state," replicated across multiple brokers like any standard Kafka topic. This design ensures transaction state persistence even when the broker hosting the transaction coordinator fails, enabling consistent system recovery.

When a broker hosting a transaction coordinator crashes, Sudan describes a meticulous recovery sequence that preserves exactly-once guarantees [10]. A newly elected coordinator assumes responsibility for affected transactions, examines the transaction log to determine their status, and definitively resolves each transaction. Transactions that reached the "prepare commit" phase complete successfully, while earlier-stage transactions abort. This approach eliminates ambiguous transaction states during recovery, maintaining exactly-once guarantees through coordinator failures.

6.2 Network Partitions

During network partitions, Kafka's transaction protocol safeguards consistency by having the transaction coordinator abort transactions that lose communication with relevant brokers. Once the partition heals, producers can safely retry transactions.

Network partitions create particularly challenging scenarios for distributed systems, often forcing difficult tradeoffs between availability and consistency. Hevo Data's guide to Kafka exactly-once semantics explains how the transaction protocol prioritizes consistency in these situations [11]. When network partitions prevent communication between transaction components, the system takes a conservative stance, aborting transactions that cannot proceed safely.

This approach reflects the principle that exactly-once semantics must prioritize correctness over availability when facing ambiguity. Hevo Data's analysis describes how the transaction coordinator employs timeouts to detect potential network issues, aborting transactions when acknowledgments fail to arrive within the configured transaction Timeout.ms period [11]. This timeout mechanism ensures the system fails safely rather than allowing transactions to linger indefinitely in indeterminate states.

6.3 Consumer Rebalances

During consumer group rebalancing, Kafka's offset management prevents duplicate processing. By incorporating offset commits within the same transactions that process messages, consumers ensure offsets update only when processing transactions complete successfully.

Consumer group rebalancing introduces unique challenges for maintaining exactly-once semantics. Sudan highlights that without proper coordination between processing logic and offset management, rebalances potentially cause either duplicate processing or data loss [10]. The integration of consumer offset commits with the transactional API directly addresses this challenge.

This integration creates what Sudan describes as "read-process-write atomicity," treating input consumption, processing, and output production as a single atomic operation [10]. The sendOffsetsToTransaction() method allows applications to include consumer offset commits within the same transaction that produces output records, ensuring offsets commit only when the entire processing transaction succeeds.

Hevo Data further explains how this mechanism establishes a crucial link between input consumption and output production [11]. When rebalancing occurs and partitions are redistributed to different consumers, each consumer begins processing from the last committed offset for newly assigned partitions. Since these offsets commit only as part of successful transactions, the system guarantees exactly-once processing regardless of rebalance frequency.

The resilience of Kafka in the face of such failure cases is, however, an important advancement in stream processing technology. By using a combination of persistence, safety mechanisms (based on timeouts), and explicit offsets, described in detail below, Kafka can engineer applications around such resilient algorithms to support mission-critical streaming applications that guarantee integrity even in unstable conditions.

Conclusion

Kafka's exactly-once semantics represent a transformative capability that has revolutionized how organizations build reliable data streaming pipelines. By addressing the fundamental challenges of distributed systems through a carefully designed combination of idempotent producers, lightweight transaction coordination, and integrated consumer offset management, Kafka provides robust processing guarantees without sacrificing the performance characteristics that make it suitable for high-throughput applications. The extension of these guarantees to external systems through Kafka Connect bridges a critical gap in end-to-end data pipelines, enabling consistency across heterogeneous environments. As organizations increasingly depend on real-time data for critical operations, Kafka's ability to maintain data integrity even through failures and network partitions has become essential infrastructure across industries. Looking forward, exact-once semantics will continue to play a vital role in emerging architectures like data meshes, event-driven microservices, and AI/ML pipelines, further cementing Kafka's position as a cornerstone technology for reliable real-time data processing. The architectural innovations demonstrated in Kafka's exactly-once implementation represent not just a technical achievement but a significant advancement in distributed systems design that will influence data technologies for years to come.

Funding: This research received no external funding.
Conflicts of Interest: The authors declare no conflict of interest.

References

[1] Confluent, "Exactly-Once Semantics Are Possible: Here's How Kafka Does It," 2017. (https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/)

[2] Kai Waehner, "Top Trends for Data Streaming with Apache Kafka and Flink in 2025," 2024. (https://www.kai-waehner.de/blog/2024/12/02/top-trends-for-data-streaming-with-apache-kafka-and-flink-in-2025/)

[3] Soulaimaneyh, "Explaining the Fundamental Principles of Distributed Systems," Medium, 2023. (https://medium.com/@soulaimaneyh/exploring-the-fundamental-principles-of-distributed-systems-970c285a77b5)

[4] Andy Bryant, "Processing guarantees in Kafka," Medium, 2019. (https://medium.com/@andy.bryant/processing-guarantees-in-kafka-12dd2e30be0e)

[5] Vivek Sinha, "What is Kafka Exactly Once Semantics? How to Handle It?," Hevo Data Blog, 2024. (https://hevodata.com/blog/kafka-exactly-once-semantics/)

[6] Confluent, "Transactions in Apache Kafka," 2017. (https://www.confluent.io/blog/transactions-apache-kafka/)

[7] Gwen Shapira, Todd Palino, Rajini Sivaram, and Krit Petty, "Kafka: The Definitive Guide, 2nd Edition," O'Reilly Media, 2021. (https://www.oreilly.com/library/view/kafka-the-definitive/9781492043072/)

[8] Jay Kreps, "Exactly-once Support in Apache Kafka," Medium, 2017. (https://medium.com/@jaykreps/exactly-once-support-in-apache-kafka-55e1fdd0a35f)

[9] Ravi Sharma, "Extending Kafka's Exactly-Once Semantics to External Systems," Medium, 2025. (https://medium.com/@raviatadobe/extending-kafkas-exactly-once-semantics-to-external-systems-c395267935bd)

[10] Sudan, "Exactly-Once Processing in Kafka explained," Medium, 2021. (https://ssudan16.medium.com/exactly-once-processing-in-kafka-explained-66ecc41a8548)

[11] Vivek Sinha, "What is Kafka Exactly Once Semantics? How to Handle It?," Hevo Data Blog, 2024. (https://hevodata.com/blog/kafka-exactly-once-semantics/)