TUTORIAL

Implementing Apache Kafka Exactly-Once Processing in Big Data Pipelines

A practical tutorial for implementing Kafka exactly-once processing in big data pipelines, from prerequisites and transactional setup to validation and production safeguards.

Programming - July 09, 2026

Why exactly-once matters in a Kafka pipeline

The practical problem is simple: a data pipeline that reprocesses records after failures can create duplicate events, double-counted metrics, corrupted sinks, or inconsistent state in downstream systems. In Kafka-based big data architectures, that problem usually appears when producers retry, consumers restart mid-batch, or a processing job writes results before it knows whether offset commits succeeded.

This tutorial shows how to implement Kafka exactly-once processing in a way that is useful in real operations. You will build a transactional processing flow that reads from Kafka, processes records, writes results, and commits offsets atomically so that a failure does not create duplicate side effects in the target topic or downstream sink. You will also verify the behavior, identify the boundaries of the guarantee, and check what must be true before production use.

What exactly-once means in practice

Exactly-once processing is not a promise that a pipeline never retries. It means the pipeline is designed so that a retry does not cause the same logical record to be applied twice to the output of the transaction-aware part of the system.

In Kafka, this depends on three pieces working together:

- Idempotent production so retries do not create duplicate records at the broker level.
- Transactions so related writes and offset commits are committed or aborted together.
- Consumer isolation so readers only see committed transactional data.

That combination solves the common failure case where a consumer processes a batch, crashes before committing offsets, then replays the same input and writes duplicate output. It does not automatically make every downstream database or object store exactly once. If your sink is not transaction-aware, you must add deduplication or an idempotent write strategy there as well.

Stop here if your sink cannot be made idempotent

If the final destination is a legacy database table, object store path, or external API that cannot deduplicate writes by key, stop and design that sink first. Kafka transactions protect Kafka-side reads and writes, but they do not magically make a non-transactional sink safe. A reliable pipeline needs an explicit output identity strategy, such as upserts by event key, a transactional sink connector, or a deduplication table.

Prerequisites and decision rules

Before implementation, confirm that the workflow actually fits your use case.

Use this approach when

- The pipeline consumes from Kafka and produces back to Kafka, or to a sink that can participate in transactional or idempotent writes.
- Duplicate prevention is more important than raw throughput.

- You can tolerate the operational overhead of transactions and careful offset management.
- You need deterministic replay behavior after failures.

Do not rely on this alone when

- The processing step calls external systems that cannot roll back.
- The pipeline fans out to multiple unrelated sinks with no coordinated commit model.
- You need exactly-once semantics end to end across a non-transactional storage layer without an idempotent design.

Verify these prerequisites first

- Broker and client versions support idempotent and transactional producers.
- Consumers can be configured to read only committed data when necessary.
- Topic replication and broker durability settings are aligned with your availability target.
- The application can store a stable transactional identifier per instance or per processing job.
- Monitoring exists for producer errors, aborted transactions, consumer lag, and offset commit failures.

For a broader pre-production review of data platform controls, pair this work with the Big Data Production Readiness Checklist and confirm that security, rollback and monitoring requirements are documented before rollout.

Target architecture

The finished state should look like this:

- A consumer reads records from an input topic using a controlled group.
- The application processes a batch or record set in memory.

- A transactional producer writes the output to a destination topic.
- The same transaction includes the offset commit for the input records.
- Downstream readers use the proper isolation level so they do not observe uncommitted writes.

This design gives you a single commit point for the Kafka-side part of the pipeline. If the application fails before commit, the transaction is aborted and the output is invisible. If it fails after commit, the committed output and the offset advancement remain consistent.

Prepare the Kafka side

Goal

Configure the cluster and topics so the application can safely use transactional production and committed reads.

Action

Make sure the input and output topics are created with replication and retention settings suitable for your reliability target. Keep the topic design simple at first: one input topic, one output topic, and a clear consumer group for the processor.

Operationally, the most important settings are durability-related: replication factor, minimum in-sync replicas, and acknowledgement behavior. Those values determine whether the cluster can survive broker loss without losing committed data. Verify the exact settings in your environment rather than assuming defaults are acceptable.

If the consumers downstream of the processor must never see in-flight data, set them to read committed records only.

Expected output

- Input records are durable enough for the failure model you expect.
- Output records can be committed transactionally.
- Downstream consumers are configured to avoid uncommitted reads where needed.

Validation

- Confirm that the topic replication factor matches production expectations.
- Confirm that consumers which depend on strong consistency use the committed-read isolation setting available in your client library.
- Confirm that broker logs and metrics expose transaction and replication issues.

Common failure

A frequent mistake is to focus on producer configuration and ignore topic durability. If the output topic is under-replicated or the cluster is unhealthy, exactly-once logic can still fail at commit time or lose data during broker outages.

Implement the transactional producer

Goal

Create a producer that can safely retry writes without duplicating committed output.

Action

Enable idempotence and transactions in the producer configuration, and assign a stable transactional identifier to each running processor instance.

A typical configuration pattern looks like this:

The exact allowed values and retry strategy depend on the client library version, so verify your client documentation. The key point is that the producer must be able to identify itself consistently and participate in transactions across retries.

In application logic, the transaction flow is usually:

- Begin a transaction.
- Consume a set of input records.
- Process the records.
- Write transformed records to the output topic.
- Send the corresponding input offsets to the transaction.
- Commit the transaction.

If any step fails, abort the transaction and let the consumer replay the input.

Expected output

- Producer retries do not create duplicate committed output.
- Output and offset commits advance together.
- Failed attempts are rolled back by transaction abort.

Validation

- Deliberately restart the processor during a batch and confirm that no duplicate committed output appears.
- Check that committed output is visible only after transaction completion.
- Confirm that consumer offset advancement matches the final committed output position.

Common failure

Using a normal producer with offset commits in separate steps is the classic error. That pattern can still duplicate output after a crash because the write and the offset commit are not atomic.

Implement the consume-transform-produce loop

Goal

Wire the consumer, processing logic and transactional producer into a single failure-safe loop.

Action

Keep the batch size small enough that transaction duration remains manageable. Large batches increase the impact of a failure and can stretch transaction time beyond what is operationally comfortable.

A simplified processing pattern is below:

In real code, add explicit error handling around each step. If processing fails, abort the transaction and log enough metadata to identify the batch, partition and offset range.

For a practical example of building safe distributed processing skeletons with validation and defensive defaults, see the Python Script for Distributed Log Processing in Big Data. The implementation style is similar: validate inputs early, fail safely, and keep the processing path observable.

Expected output

- Each successful batch produces committed output once.
- Failed batches are replayed without duplicating committed records.

- Logging identifies the batch and offset range involved in each failure.

Validation

- Force a processing exception before commit and confirm the transaction aborts.
- Force a crash after output write but before offset commit and confirm replay does not create duplicate committed results.
- Confirm that logs show the same input record being processed again only when the prior attempt did not commit.

Common failure

A subtle failure is swallowing exceptions and continuing the loop. If the application keeps running after a partial failure without aborting the transaction, you can commit an inconsistent batch.

Validate consumer behavior and output visibility

Goal

Ensure readers observe only committed records and that replay behavior matches the design.

Action

Test the pipeline from the perspective of downstream consumers, not only from the producer side. Read the output topic using the expected isolation level and verify that uncommitted writes are invisible.

A useful validation set is:

- Produce a known input batch.
- Interrupt the processor mid-transaction.
- Restart the processor.
- Compare the final output count and keys against the input batch.
- Check that there are no duplicated committed keys.

If you use Spark or another distributed engine downstream, validate that the job does not introduce a separate duplication path during shuffle or sink writes. If a failure appears to be in the distributed compute layer rather than Kafka itself, use a focused workflow like Troubleshoot Apache Spark Shuffle Failures in Big Data Jobs to isolate whether the issue is transport, shuffle, or commit related.

Expected output

- Only committed records are visible to readers that require strong consistency.
- The pipeline reprocesses failed batches without multiplying output.
- Validation data matches the expected source-to-target mapping.

Common failure

Teams often validate only the happy path. Exactly-once behavior is only proven when you test interruption, replay and recovery. Without failure injection, you are only testing ordinary at-least-once processing.

Operational safeguards before production

Goal

Reduce the chance that a correct design fails under real operational conditions.

Action

Add guardrails in three areas:

- Transaction management: use timeouts and alert on aborted or expired transactions.
- Consumer management: track lag, rebalance frequency and offset commit failures.
- Sink safety: ensure downstream systems can accept replay or deduplicate by key.

Also plan for deployment coordination. A new processor instance using the same transactional identifier must not run concurrently with an old instance. If two instances share the same identity incorrectly, the cluster can fence one of them off or produce confusing failures.

Document rollback behavior. If a deployment introduces errors, the safe rollback path is usually to stop the new processor, let transactions complete or abort cleanly, and restart the previous version with a compatible configuration.

Expected output

- Monitoring exists for transactional failures and consumer lag.
- Ownership of transactional identifiers is clear.
- Rollback does not require ad hoc manual recovery.

Validation

- Perform a controlled deployment restart and confirm that the pipeline resumes without duplicates.
- Confirm alerting fires on repeated aborts or commit failures.
- Verify that the rollback runbook includes stop, drain, validate and restart steps.

Common failure

A common production issue is treating the pipeline as exactly-once while the deployment process is not. If the same transactional identity is reused incorrectly, the runtime may fence producers or create unstable restarts.

Minimal implementation checklist

Use this as a final readiness pass:

- The pipeline has a stable transactional identifier strategy.
- The producer is configured for idempotent, transactional writes.
- Input offsets are committed in the same transaction as output writes.
- Consumers that need strong consistency read only committed data.
- The output sink is either transactional, idempotent or otherwise deduplicated.
- Failure injection has been performed at least once.
- Monitoring covers transaction aborts, lag and commit failures.
- The rollback process has been documented and rehearsed.

What production-ready looks like

A production-ready exactly-once Kafka pipeline does not mean no failures occur. It means failures are expected, isolated and recoverable without creating duplicate committed output in the Kafka-managed path.

If you can explain where the transaction boundary starts and ends, show that replay does not duplicate committed records, and prove that your sink is safe under retry, you have implemented the pattern correctly. The final test is operational: after a restart, a rebalance or a transient broker issue, the pipeline should resume with consistent state and predictable output, not with a hidden accumulation of duplicates.

Use this guidance together with ASP.NET Core JWT authentication with refresh tokens and distributed SQL queries to connect the workflow with related operational context already available on the site.