DATA ENGINEERING · OPERATIONS BRIEF

RESUMING A DATA FLOW WITHOUT MULTIPLYING ITS EFFECTS

Published on 10 November 2026 · Version 1.0.0 · Updated 10 November 2026
Online article:
https://datanalysis.re/ressources/data-engineering/resuming-data-flow-without-duplicate-effects

A job stops. The database has recorded the batch of orders, but the confirmation never reached the job. On restart, should the write be replayed? Without a suitable mechanism, that retry can create a second copy of every order.

The problem is not only getting the job running again. You have to determine what was actually done, what remains uncertain and what can be repeated without changing the expected result. The scenario presented here is fictional; the checks proposed must be adapted to the architecture of the flow.

Tell a failed response from a failed operation

When a request times out, the client knows it has not received a usable response. It does not necessarily know whether the receiving system carried out the operation. Retrying systematically is therefore not always free of consequences.

AWS engineering guidance describes this ambiguity and the value of idempotent APIs: the same call can be retried without multiplying the intended effects. It also sets out the limits related to the identity of the request, how long it remains valid and changes of intent. [1]

For our batch of orders, we need to be able to recognise that a new attempt corresponds to the same operation, and not to a new, legitimate order.

Give the business intent an identity

A stable key can identify the batch, the order or the operation to perform, depending on the chosen scope. It must stay the same across attempts of the same request. Conversely, a new business request must be able to have a different identity, even if some values are equal.

Comparing only the content of a request is not always enough: two genuinely different orders can contain the same items and the same quantities. The choice of key should therefore be discussed with the business owner and the owner of the source application.

You also need to define what happens when a known key comes back with a different amount or recipient. The process must not confuse a correction, a calling error and an identical repetition. The idempotency documentation of an API, Stripe for example, should be consulted for its exact rules rather than assuming universal behaviour. [2]

Protect the write, not just check beforehand

A check “does this order already exist?” followed by an insert can let a duplicate through if two attempts run at the same time. Protection must cover the moment the effect is recorded.

Depending on the system, this can rely on a uniqueness constraint, a transaction or an idempotency mechanism provided by the destination API. The choice depends on the engine, the data model and the effects to protect. An application-level check does not automatically replace an atomic write guarantee. Uniqueness constraints are among PostgreSQL’s documented mechanisms; their scope must match the chosen business identity. [3]

The recovery response must also be defined. If the request had already succeeded, the client should be able to retrieve a usable reference rather than remain in an undetermined state.

Examine every external effect

No duplicate in the orders table does not guarantee no duplicate notification, invoice or write in another system. Each effect must be identified and tested within its own scope.

An architecture that coordinates several systems may have to handle intermediate states, delayed attempts and corrections. A local protection should not be presented as a global “exactly once” processing guarantee. The recovery method must state what it guarantees and which cases remain for a person to examine.

Test uncertainty before the first incident

Useful acceptance testing is not limited to the happy path. The accompanying checklist suggests testing the following situations with fictional data and expected results defined in advance.

Test situations and expected checks

Test situation	What to check
Confirmation lost after the write	Replaying the same request creates no additional effect; the initial result can be retrieved.
Two concurrent attempts	The protection withstands concurrency, not just two successive calls.
Same key, modified content	The agreed behaviour is explicit: rejection, conflict or correction workflow.
Late replay	The limits on key retention and on replaying history are known.
One effect succeeds, another fails	The partial state is visible and recovery does not blindly re-execute effects already carried out.
Uncertain destination result	An owner and a reconciliation method are identified before any new attempt.

A passing test establishes the behaviour observed under its conditions. It does not remove the need to monitor drift, document versions and review the scope when the flow changes.

Prepare recovery as part of the service

For a specific flow, the team must be able to name the business request, its key, the system that is authoritative on its state, how long a replay is supported and who owns exceptions. It must also know which outputs to suspend when the result is uncertain.

At DATANALYSIS, integration work means connecting these rules to the writes, the checks and the uses that depend on them. The goal is a verifiable recovery, suited to the flow, rather than a "retry" button whose consequences remain unknown.

Let's scope the recovery of one specific flow. Bring its source, its destinations and an interruption scenario. We can define the effects to protect, the useful tests and the recovery responsibilities.

→ Scope the recovery of my data flow: https://datanalysis.re/contact?lang=en

→ Fillable checklist (PDF): https://datanalysis.re/downloads/data-reprise-checklist.en.pdf

→ HTML checklist: https://datanalysis.re/ressources/data-engineering/resuming-data-flow-without-duplicate-effects/checklist

In the same series

When data changes meaning: preparing changes across applications:
https://datanalysis.re/ressources/data-engineering/data-changes-meaning-across-applications

Modernising a data platform while the business keeps working:
https://datanalysis.re/ressources/data-engineering/modernising-data-platform-while-business-runs

Related expertise

Data Engineering (available in French): https://datanalysis.re/expertises/data-engineering?lang=fr

Sources and scope

Primary references consulted on 10 November 2026. The example and the checklist are our own teaching application: they are neither a client case nor a guarantee independent of the system you choose. No Stripe or PostgreSQL connector is claimed here.

[1] AWS Builders' Library | Making retries safe with idempotent APIs (ambiguity of a timed-out request, idempotent APIs, limits related to request identity, validity period and changes of intent)
https://aws.amazon.com/builders-library/making-retries-safe-with-idempotent-APIs/

[2] Stripe | Idempotent requests (idempotency rules specific to that API, to be consulted rather than assuming universal behaviour)
https://docs.stripe.com/api/idempotent_requests

[3] PostgreSQL 18 | Constraints (uniqueness constraints, whose scope must match the chosen business identity)
https://www.postgresql.org/docs/18/ddl-constraints.html

Fictional teaching scenario; no client results.