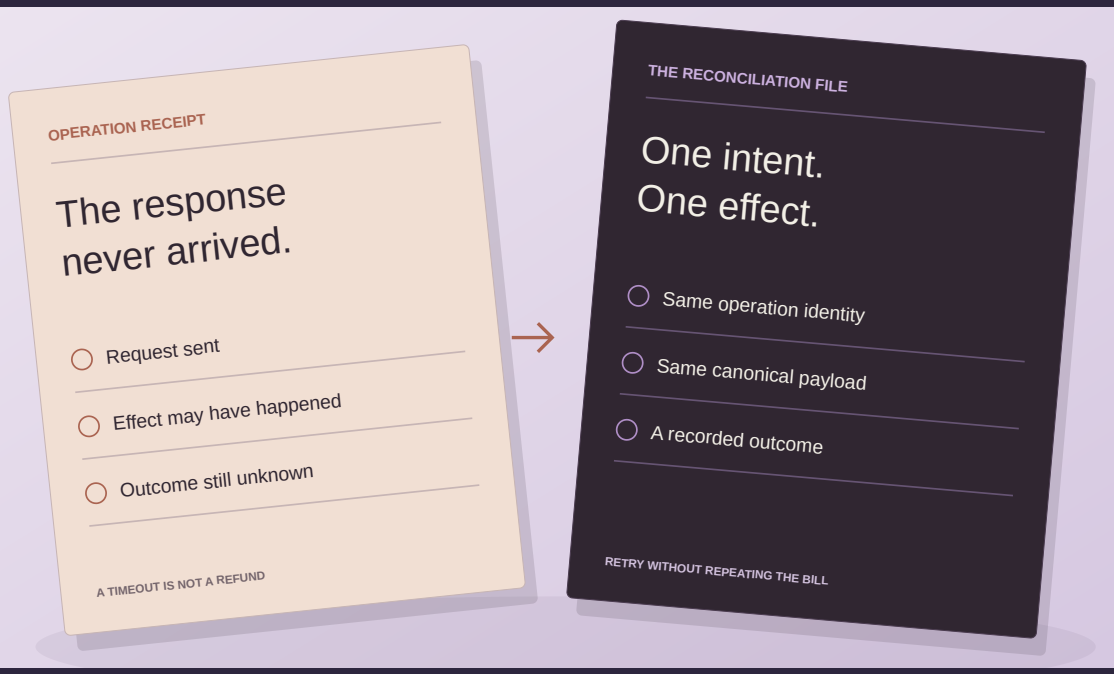
kymata labs

RESEARCH PAPER 09 / AGENT SYSTEMS & RELIABILITY

The Retry Button Can Spend Twice

When the answer disappears, repeating the action is a new decision.



Kymata Labs Research
2026-11-16 · Edition 1.0

Independent analysis
kymatalabs.com

Small studio. Open worlds.

THE LONGER VIEW

Executive summary.

A failed response does not establish a failed action. This paper examines the gap between a caller's knowledge and a remote service's effects, then develops a recovery contract around durable logical identity, scoped idempotency keys, payload validation and reconciliation. It distinguishes HTTP semantics, a working Idempotency-Key draft, inspected Powertools behavior and a bounded Stripe API contract without inventing provider retention guarantees. A local Python fixture deliberately loses a response after an effect commits, tests repeated and concurrent submissions, and contrasts durable outcome recovery with an expiring response cache. The result is a practical acceptance exercise for consequential agent tools, with safety and eventual progress treated as separate obligations.

> Inspected exact official repository snapshots for RFC 9110, the HTTPAPI Idempotency-Key working draft, Powertools for AWS Lambda Python documentation and implementation, and Stripe OpenAPI version 2026-09-30.endive. The draft is not represented as a published RFC. A Python standard-library fixture uses two local SQLite files and an invented service contract to test response loss, scope, payload conflict, concurrent submission and cache expiry. No network, provider, model or payment calls were made by the fixture. Measured results do not establish any real provider's retention, exactly-once guarantees or incident rate.

01 / The response disappeared. The action may not have.

02 / Idempotency describes an effect, not a successful conversation

03 / One intention, several attempts

04 / The same key must mean the same commitment

05 / Three clocks, three different promises

06 / A wrapper cannot atomically commit two systems

07 / Reconciliation is work, not an error message

08 / An experiment that deliberately loses the answer

09 / What the example does not prove

10 / The strongest case for retrying automatically

11 / Design the second attempt before celebrating the first

Q&A / Questions worth asking

Sources / References

© 2026 Kymata Labs LLC. All rights reserved.
Read the web edition: kymatalabs.com/research/the-retry-button-can-spend-twice

The response disappeared. The action may not have.

The most dangerous word on a failed purchase screen can be *again*.

Consider a hypothetical agent placing an order. The service accepts the request, creates the order, and sends a response. The connection disappears before the agent receives it. From the agent's perspective, the tool failed. From the supplier's perspective, the order exists. A helpful assistant that starts a fresh purchase has turned a communication failure into a second commitment.

Nothing in that sequence requires a dishonest model or a malicious instruction. A conventional worker can make the same mistake. Agents make the distinction worth emphasizing because their apparent ability to recover can conceal a change in what they are recovering. Recovering knowledge of an existing action is different from creating another action that resembles it.

This paper's claim is narrow: **a timeout must not become permission to repeat a consequential action under a new identity.** Reliable recovery requires a durable identity for the user's intended operation, a defined idempotency contract at the side-effect boundary, and a reconciliation path for outcomes that remain uncertain. An automatically generated key is only one part of that design.

Idempotency describes an effect, not a successful conversation

HTTP semantics defines an idempotent method by the intended effect of repeated identical requests: it is the same as the effect of one request. That does not require identical response messages, and it does not prohibit separate logs for each request. The inspected RFC 9110 source says a client should not automatically retry a non-idempotent method unless it knows the semantics are actually idempotent or can detect that the original request was never applied. [1]

The distinction matters to builders who treat every exception as a transient inconvenience. Retrying a read and retrying a request that initiates a payment can require different evidence. A connection timeout says the caller did not obtain a timely answer. It does not establish which server-side steps occurred before communication stopped.

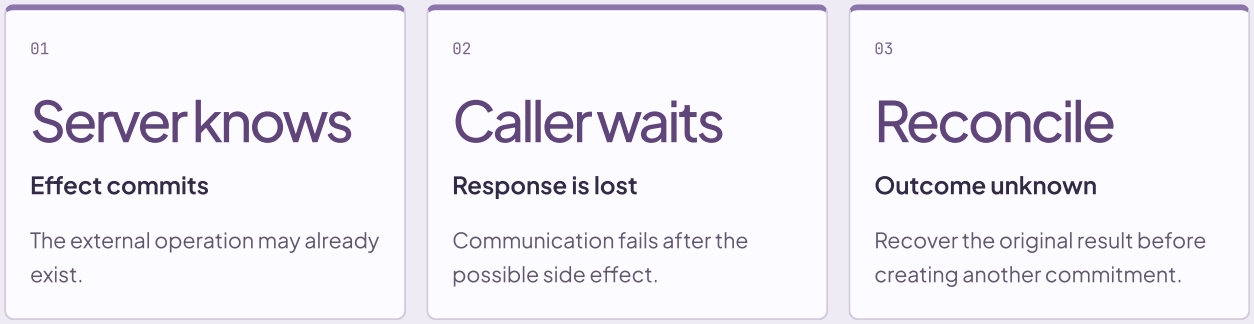
Nor does attaching a header magically settle the matter. The inspected IETF HTTPAPI Idempotency-Key document is a working draft, not a published RFC. It explicitly says a general client cannot assume a server will honor a supplied key without prior knowledge. The resource should document its behavior, including expiration policy. These statements describe a proposed protocol contract, not a guarantee that a particular API implements it. [2]

The practical question is therefore not “does the request have a UUID?” It is “what does this server promise to recognize as the same operation, for how long, under which scope, and with what handling of concurrent or changed requests?” The answer belongs in an integration contract and its tests. An agent's fluent explanation cannot substitute for it.

FIGURE 01 FAILURE MECHANISM

The action and the answer can part company.

A hypothetical lost-response sequence explains why a caller's timeout does not establish that nothing happened.



Illustrative mechanism, not a reported customer incident. HTTP retry semantics depend on the intended effect and the operation's actual contract.

Kymata Labs explanation grounded in HTTP semantics and the HTTPAPI draft. [1] [2]

One intention, several attempts

A useful design separates the logical action from its delivery attempts. The logical action is the user's intended commitment: this order, for this account, with these terms. An attempt is a particular effort to deliver it. Attempts may have different timestamps, transport identifiers or worker owners while still referring to the same action.

If the retry layer creates a new logical identity every time it catches an exception, the downstream service may reasonably treat each request as new. If it uses the entire request body as the identity, a changed timestamp or trace identifier can have the same effect. Conversely, hashing only the amount can wrongly collapse two independent purchases of equal value. Neither "always make it unique" nor "always hash the payload" is a sufficient policy.

The application should create and persist the logical identity before dispatch. It should record the key used downstream, the operation and account scope, the meaningful payload, and the current outcome state. A recovering worker then resumes the existing record. It does not ask the model to reconstruct identity from the conversation or generate another apparently appropriate purchase.

Record	Purpose	Failure when omitted
Logical action ID	Names the user's intended operation	Retries become fresh business requests
Service and account scope	Places the key in the correct namespace	Different principals collide or share results
Semantic payload fingerprint	Detects changed terms under the same key	A revised purchase inherits an old result
Attempt history	Records deliveries without renaming the action	Transport events obscure business identity
Outcome and receipt	Preserves what is known about the effect	A timeout is silently converted to failure

The identity should survive the application process, a queue redelivery and a user's repeated click. The last case matters. If a browser generates a new action ID for each click, the server can perfectly deduplicate each request while still creating two purchases. The product needs to decide whether the second click resumes an unfinished action or intentionally begins another. That is a user-interface and business rule, not merely a networking detail.

The same key must mean the same commitment

A key reused with a changed payload creates another ambiguity. Suppose the first request names a 2,500-unit purchase and a later request names 2,600 under the same key. Returning the original response without checking the change could make the caller believe the new terms were applied. Executing both defeats the purpose of the key.

The inspected HTTPAPI draft says a key must not be reused with a different request payload. It describes optional fingerprints for detecting such reuse, recommends a 422 response for a detected mismatch, and recommends 409 when the original request is still processing. These are draft recommendations. A provider may expose different status codes or behavior; the application must follow its documented contract. [2]

A fingerprint also requires a definition of meaning. Include fields that can alter the effect: amount, currency, destination, quantity, selected product or operation type as appropriate. Exclude fields that are explicitly irrelevant to business identity, such as a fresh trace identifier, only after making that choice deliberate. The draft notes that fields outside a partial fingerprint are treated as irrelevant for duplicate detection. [2]

Powertools for AWS Lambda makes this distinction visible in its configuration. Its documentation separates the expression selecting an idempotency key from `payload_validation_jmespath`, which validates additional selected fields. The example involving an amount shows why the distinction exists: the same key should not silently accept changed consequential input. Its persistence implementation enables that additional validation only when configured. [3][5]

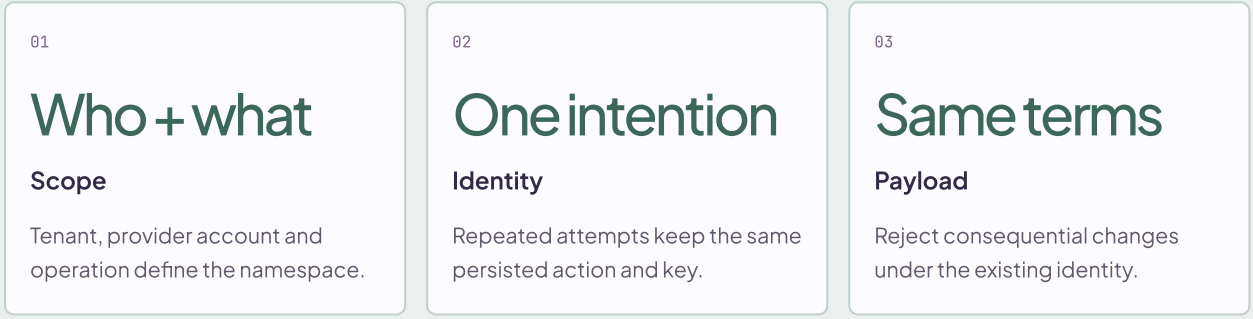
Canonicalization is part of the contract. Field ordering should not turn equivalent JSON objects into different business actions. At the same time, a permissive normalizer should not erase meaningful distinctions. The local example uses sorted JSON over a small, typed payload and integer fictional units. It does not solve every canonicalization problem, Unicode equivalence, decimal currency representation or schema migration.

Scope is equally consequential. A tenant's key should not retrieve another tenant's result. A creation request and a cancellation request should not share a result simply because their caller supplied the same string. The HTTPAPI draft discusses a composite lookup key combining the client's key with other client-specific attributes. The local fixture uses tenant and operation scope explicitly. [2][7]

FIGURE 02 IDENTITY CONTRACT

A key needs an envelope.

Duplicate detection requires a stable logical action, the right namespace and validation of the terms that change its effect.



Original application design. Fingerprint coverage and key scope must match the chosen service's documented behavior.

HTTPAPI working draft, Powertools configuration and Kymata Labs fixture. [2] [3] [5] [7]

Three clocks, three different promises

Many retry designs collapse three time limits into one. The caller's timeout determines how long it waits for a response. A worker lease or in-progress deadline determines when another worker may take responsibility. The idempotency retention window determines how long a service remembers a request for duplicate suppression. None of those numbers, by itself, says when the external effect becomes impossible.

Powertools' inspected documentation gives a concrete example of why the distinction matters. It describes a default idempotency expiration of 3,600 seconds, configurable through `expires_after_seconds`. After expiration, the same payload can be processed as a new transaction. It also describes an in-progress expiry used for Lambda timeouts, after which another invocation may execute. These are behaviors of the captured package documentation, not universal AWS guarantees or claims about every deployment. [3]

Those mechanisms can be exactly right for retrying a computation. The difficult case is a function that has already caused a remote effect before it times out. An expired local lock does not undo that effect. A worker taking over needs either the same valid downstream idempotency identity, authoritative evidence of the existing result, or a procedure that preserves uncertainty rather than creating another commitment.

Retention therefore belongs in the application's recovery design. If the maximum plausible retry horizon is longer than the provider's duplicate-suppression window, the application must not assume an old key remains protective. It may need a durable business record, a provider object that can be retrieved, or a reconciliation process that refuses blind resubmission. Increasing a cache TTL can reduce a gap; it does not establish an indefinite exactly-once guarantee.

The clocks can also cross during an incident. An operator may return to an unresolved action hours or days after an automated worker stopped retrying. A manual “try again” control should show what is known about the original request and whether its idempotency window is still valid. Human intervention is not inherently safe if the interface hides the relevant history.

A wrapper cannot atomically commit two systems

An idempotency library can coordinate repeated function calls in its own persistence layer. It cannot automatically make an arbitrary external side effect and that local record one atomic transaction. This is a boundary of the design, not an indictment of the library.

The inspected Powertools handler attempts to save an in-progress record before calling the wrapped function. Existing in-progress work produces an appropriate in-progress error; completed work can return stored output. In the captured implementation, an exception raised by the wrapped function leads to an attempt to delete its idempotency record, enabling a later execution. A successful function is followed by serialization and persistence of its response. [4]

Now consider a hypothetical wrapped function that creates an order remotely and then raises because the response was lost. Locally, the function failed. Remotely, the effect may exist. Retrying the function is safe only if the function's own relationship with the remote service makes it safe. The decorator cannot infer the supplier's state from the Python exception.

The captured code also distinguishes failure while saving a successful response from an ordinary handler exception. That is another reminder to read the actual implementation and its documented semantics rather than compress every error into one retry category. This paper does not claim to have executed Powertools against AWS or measured an incident rate. It uses inspected code to identify the boundary an integration must account for. [3][4]

The remedy may involve a durable outbox, a downstream idempotent operation, a business uniqueness constraint, or a saga with explicit compensation. None is a magic word. An outbox can reliably dispatch a message while still delivering it more than once. Compensation can be a new consequential action with

its own failure modes. The design must identify the component that prevents repeated delivery from creating a repeated business effect.

Reconciliation is work, not an error message

An outcome ledger should distinguish at least a pending action, a confirmed result and an unresolved outcome. A confirmed rejection that guarantees no effect may permit another attempt under the established contract. An ambiguous timeout should remain unresolved until evidence changes that classification.

Reconciliation means obtaining evidence about the original action. A stable provider object ID can be useful. The inspected Stripe OpenAPI contract, version `2026-09-30.endive`, separately describes creating a PaymentIntent, retrieving an existing PaymentIntent and confirming it. Its confirmation description includes several possible states, including requirements for additional action or capture. The contract supports a distinction between an object and its progress through a workflow. It does not, by itself, establish a universal retention policy for every Stripe idempotency key. [6]

That qualification is important. This paper does not claim that a caller can always recover a lost creation response by searching for its own key, or that a particular Stripe retry is safe for a fixed number of hours. Such claims would require additional inspected provider documentation and integration tests. The engineering recommendation is to establish a retrieval or reconciliation contract for the chosen operation before depending on it.

A missing search result may be weaker evidence than it appears. The search might lag, require a different account scope, or exclude an operation whose response is still being processed. If the provider does not promise that absence means the operation never happened, absence should not automatically authorize a second commitment. Positive matching evidence is easier to use: a receipt for the right account, action and payload can resolve an unknown into a confirmed result.

A reconciliation workflow also needs an owner. Unresolved actions should have an age, a next check, and an escalation path. The user should see that a result is being checked rather than being encouraged to submit again. For financial workflows, any associated reservation should remain committed until the spending outcome is understood. That accounting rule is developed in the companion paper, *Give Your Agent a Budget Before You Give It Authority*.

An experiment that deliberately loses the answer

The accompanying Python standard-library example uses two local SQLite files: one for an application ledger and one for a simulated external service. Download reference 7 and run `python3 example.py`. The program performs assertions and writes `result.json` beside itself. It makes no network, provider, model or payment calls. Its service contract is invented for demonstration and is not presented as Stripe behavior. [7]

The fake service applies an effect and commits its receipt, then raises a timeout to simulate losing the answer. The application's action becomes `unknown`. The application object is recreated using the same database, demonstrating that the action identity survives that restart. Ten subsequent submissions of the same action return its unresolved state without dispatching another effect.

A changed amount under the same action identity is rejected. A reconciliation lookup finds the matching receipt and resolves the action to `succeeded`. A much later submission returns the durable result even though the fake service's response cache would have expired. The action has one effect throughout this sequence. The system is recovering the outcome of an existing operation, not guessing that another execution will be harmless.

A separate test creates an unknown action before transmission and leaves no remote receipt. Reconciliation leaves it unknown. The fixture deliberately declines to treat absence as permission to execute. That is conservative: the fixture's author knows nothing was sent, but the recovery component does not possess an authoritative no-effect proof. Real implementations can make progress when their contracts supply such proof.

The fixture also submits one new logical action from thirty threads. Its local transactional action record permits one dispatch, and the service records one effect. Another tenant can use the same textual service key without inheriting the first tenant's receipt because the key is scoped. These are finite, local observations, not a distributed fault-tolerance certification.

Local fixture case	Observed effects	Result
Lost response, then ten repeated submissions	1	Unknown until reconciliation
Same action after reconciliation and cache expiry	1	Durable result returned
Thirty concurrent submissions of one action	1	One local admission to dispatch
Same action identity with changed amount	No additional effect	Payload conflict rejected
Cache-only duplicate before expiry	1	Cached response returned
Cache-only duplicate after expiry	2	Same key treated as new work

The final two rows are a separate counterexample. The fake service remembers a key for sixty arbitrary time units. A duplicate at time one reuses the result. A duplicate at time sixty-one creates a second effect. The key string did not change. The promise behind it expired. No real provider is implicated by this deliberately chosen behavior. [7]

FIGURE 03

MEASURED LOCAL COUNTEREXAMPLE

The key stayed. The promise expired.

A durable application result and a deliberately expiring cache produce different outcomes in separate fixture cases.

01

1 effect

Durable action record

Lost response, ten repeats, reconciliation and late replay.

02

1 effect

Cache before expiry

Same key at time 1 reuses a response stored at time 0.

03

2 effects

Cache after expiry

Same key at time 61 becomes new work after a 60-unit window.

Fictional service contract and arbitrary time units. Not Stripe or AWS behavior; no real payment, model or provider calls.

Kymata Labs example.py and recorded result.json; November 16, 2026. [7]

What the example does not prove

The fixture's service retains authoritative, searchable effect receipts indefinitely. A real integration must establish its own retention, consistency, authorization and lookup behavior. The fixture assumes trusted tenant identity; it is not an authentication system. It uses one local database per component, not regional replicas or a distributed transaction protocol. It does not test payment settlement, webhooks, compensation or financial dispute handling.

Its client also chooses safety over automatic progress when the outcome is unknown. A crash after local admission but before dispatch could leave work stuck. A production implementation needs a recovery strategy that can distinguish recoverable preparation from uncertain dispatch, or safely repeat delivery using an established downstream contract. The example intentionally avoids claiming it has solved that larger liveness problem.

That limitation should be visible because “exactly once” is often used too loosely. This experiment demonstrates one effect for the tested logical actions and failure paths under its stated assumptions. It does not prove that an arbitrary remote API will execute exactly once, or that all actions will eventually complete. Preventing duplicates and guaranteeing eventual progress are different obligations.

The same caution applies to the word *payment*. The fictional amounts make the consequence legible. They are not money, provider charges or measured customer losses. The result is a reproducible failure demonstration, not evidence that a particular provider commonly duplicates transactions.

The strongest case for retrying automatically

A system that never retries can be reliably useless. Networks fail, workers restart and users expect recovery. An idempotent operation with a clear contract should benefit from bounded automated retries. Read-only requests, deterministic local calculations and downstream operations that honor durable identities can often recover without interrupting a person.

The argument is therefore for informed automation. Assign retry responsibility to a defined layer. Keep attempts attached to the same logical operation. Respect the actual provider's concurrency, validation and retention rules. Bound retry count and elapsed time, and avoid having several independent layers multiply each other's attempts. Use delay and backoff to reduce load when appropriate, while remembering that spacing requests does not make a non-idempotent action safe.

Human escalation should be proportionate. A confirmed no-effect rejection may have an automated remedy. An unresolved consequential action may need investigation. A completed operation should show its receipt and prevent an accidental second submission. The interface should describe the state the system actually knows, not the state that makes the next button easiest to implement.

There is a commercial benefit to this discipline as well as a correctness benefit. A product that can explain what happened, what remains uncertain and what it is checking gives support staff a usable incident record. A product that reports only “tool failed” transfers the diagnosis to whoever receives the complaint. The latter may look simpler until the first ambiguous commitment needs to be untangled.

Design the second attempt before celebrating the first

Before connecting an agent to a consequential API, run a failure exercise in which the effect succeeds and the answer is lost. Restart the application. Repeat the user's action. Change one consequential field under the old identity. Try a concurrent delivery. Advance beyond the duplicate-suppression window. Then inspect effects and receipts, not just the text returned by the assistant.

The acceptance standard should name both safety and progress. Can the workflow avoid a second effect for the same intended action? Can it recover a confirmed result or route unresolved work to a responsible owner? Which failures stop progress, and what evidence allows progress to resume? Those questions expose the real integration contract much faster than a demonstration in which every response arrives on time.

The first successful call proves that the service can do the work. The second attempt reveals whether the application understands what work it already asked for. A reliable agent needs that understanding in durable state, outside the conversational guesswork. When the answer disappears, the system's first job is to find out what happened to the original action. Pressing again is a decision that must wait for that knowledge—or for a contract that makes repeating it safe.

Questions worth asking.

Does a timeout mean the payment or order failed?

No. It establishes that the caller did not obtain a timely answer. The remote action might have succeeded, failed before taking effect or remained in progress. Recovery needs evidence about the original operation or a verified contract that makes repeating it safe. [1][7]

Should I generate a new idempotency key for every retry?

Not for another delivery attempt of the same logical action. Preserve the original action identity and downstream key under the actual service contract. A genuinely new operation needs a new identity; changed consequential terms under an old identity should be handled as a conflict, not silently accepted. [2][3]

Can a cached result prevent duplicates forever?

Only within the guarantees of the relevant system. The inspected Powertools documentation describes configurable expiration, after which the same payload can be processed again. The local cache-only counterexample deliberately produces a second effect after its invented window expires. It is not a claim about Stripe retention. [3][7]

Does an idempotency decorator make every external API call safe?

No. It can coordinate repeated function calls within its persistence boundary. If a function causes a remote effect and then raises, local exception handling does not establish the remote outcome. The integration still needs an appropriate downstream contract and recovery strategy. [4]

What does the downloadable example actually demonstrate?

Under an invented local service contract, one lost-response action stays at one effect through ten repeated submissions, restart, reconciliation and late replay. Thirty concurrent submissions of another action produce one effect; payload changes are rejected. It uses Python and SQLite, no real payments, and does not prove distributed exactly-once delivery or automatic progress for every unknown outcome. [7]

SOURCES, NOT SCENERY

References.

1. **RFC 9110: HTTP Semantics, idempotent methods**
   IETF HTTP Working Group · 2026–11–16 · Primary standard source

   Inspected section idempotent.methods in the official RFC source. Supports intended-effect definition and cautions on automatic retries of non-idempotent methods. RFC publication is June 2022; capture date shown separately. Exact inspected path: rfc9110.xml. Commit bde37f634681bd5462568b73aa722af9c9a3ef5e. Reference date is capture date, not original publication date. Original bytes and SHA-256 preserved in the source receipt.

   https://github.com/httpwg/http-core/blob/bde37f634681bd5462568b73aa722af9c9a3ef5e/rfc9110.xml
   Accessed copy: https://raw.githubusercontent.com/httpwg/http-core/bde37f634681bd5462568b73aa722af9c9a3ef5e/rfc9110.xml

2. **The Idempotency-Key HTTP Header Field, working draft**
   IETF HTTPAPI Working Group · 2026–11–16 · Primary Internet-Draft source

   Inspected key uniqueness, optional fingerprint, client/server responsibilities, expiry documentation, concurrent requests and error handling. This is a mutable working draft pinned to one commit, not a published RFC or evidence that every server honors the header. Exact inspected path: draft-ietf-httpapi-idempotency-key-header.md. Commit 1ba3ab05879772d3492cb07febcd70b01a9be1df. Reference date is capture date, not original publication date. Original bytes and SHA-256 preserved in the source receipt.

   https://github.com/ietf-wg-httpapi/idempotency/blob/1ba3ab05879772d3492cb07febcd70b01a9be1df/draft-ietf-httpapi-idempotency-key-header.md
   Accessed copy: https://raw.githubusercontent.com/ietf-wg-httpapi/idempotency/1ba3ab05879772d3492cb07febcd70b01a9be1df/draft-ietf-httpapi-idempotency-key-header.md

3. **Powertools for AWS Lambda Python: Idempotency**
   AWS Powertools · 2026–11–16 · Primary library documentation

   Inspected event key and payload validation configuration, concurrency, default 3600–second expiration, in-progress timeout behavior and exception handling. No deployment against AWS was performed. Exact inspected path: docs/utilities/idempotency.md. Commit 379929b3c8a79852ab4c482e9cfc53020107ebe7. Reference date is capture date, not original publication date. Original bytes and SHA-256 preserved in the source receipt.

   https://github.com/aws-powertools/powertools-lambda-python/blob/379929b3c8a79852ab4c482e9cfc53020107ebe7/docs/utilities/idempotency.md
   Accessed copy: https://raw.githubusercontent.com/aws-powertools/powertools-lambda-python/379929b3c8a79852ab4c482e9cfc53020107ebe7/docs/utilities/idempotency.md

4. **Powertools idempotency handler implementation**
   AWS Powertools · 2026–11–16 · Primary implementation source

   Inspected save_inprogress before function execution, in-progress detection, stored-response return, handler-exception record deletion and success-response persistence. Claims refer to this captured implementation; it cannot prove a remote side effect did not occur. Exact inspected path: aws_lambda_powertools/utilities/idempotency/base.py. Commit 379929b3c8a79852ab4c482e9cfc53020107ebe7. Reference date is capture date, not original publication date. Original bytes and SHA-256 preserved in the source receipt.

   https://github.com/aws-powertools/powertools-lambda-python/blob/379929b3c8a79852ab4c482e9cfc53020107ebe7/aws_lambda_powertools/utilities/idempotency/base.py
   Accessed copy: https://raw.githubusercontent.com/aws-powertools/powertools-lambda-python/379929b3c8a79852ab4c482e9cfc53020107ebe7/aws_lambda_powertools/utilities/idempotency/base.py

5. **Powertools idempotency persistence base implementation**
   AWS Powertools · 2026–11–16 · Primary implementation source

   Inspected configuration of event_key_jmespath and conditional enabling of payload_validation_jmespath. Used to distinguish identity selection from additional consequential-input validation. Exact inspected path: aws_lambda_powertools/utilities/idempotency/persistence/base.py. Commit 379929b3c8a79852ab4c482e9cfc53020107ebe7. Reference date is capture date, not original publication date. Original bytes and SHA-256 preserved in the source receipt.

   https://github.com/aws-powertools/powertools-lambda-python/blob/379929b3c8a79852ab4c482e9cfc53020107ebe7/aws_lambda_powertools/utilities/idempotency/persistence/base.py
   Accessed copy: https://raw.githubusercontent.com/aws-powertools/powertools-lambda-python/379929b3c8a79852ab4c482e9cfc53020107ebe7/aws_lambda_powertools/utilities/idempotency/persistence/base.py

6. **Stripe OpenAPI contract, version 2026-09-30.endive**
   Stripe · 2026-11-16 · Primary API specification

   Inspected /v1/payment_intents creation, /v1/payment_intents/{intent} retrieval and /confirm descriptions. Used only for operation/object/state distinctions. No general Stripe idempotency retention, guaranteed key lookup, customer loss or live endpoint behavior is asserted. Exact inspected path: openapi/spec3.json. Commit a045100cf4af0acb548cb9c6f236bb117f83458f. Reference date is capture date, not original publication date. Original bytes and SHA-256 preserved in the source receipt.

   https://github.com/stripe/openapi/blob/a045100cf4af0acb548cb9c6f236bb117f83458f/openapi/spec3.json
   Accessed copy: https://raw.githubusercontent.com/stripe/openapi/a045100cf4af0acb548cb9c6f236bb117f83458f/openapi/spec3.json

7. **Reproducible local fixture: example.py**
   Kymata Labs Research · 2026-11-16 · Original executable illustration

   Python standard library only. Run python3 example.py to perform the assertions and write result.json. Local fictional scenario; no network, provider, model or payment calls. The publication reports the measured fixture result, not a production benchmark.

   https://www.kymatalabs.com/research/the-retry-button-can-spend-twice/assets/example.py