TECHNICAL NOTE 02 / VERSION 1.0

Idempotency Is an Invariant

Why a retry key must be committed with the financial effect it protects

Kylian de Groot
5 September 2026

ABSTRACT

An idempotency key is useful only when the stored key and the protected effect evolve together. This note defines a small ledger state machine and proves, by induction, that one committed effect per key survives repeated and reordered deliveries under explicit atomicity assumptions. It gives crash and concurrency counterexamples for weaker designs, distinguishes duplicate requests from conflicting reuse of a key, and separates at-most-once safety from eventual completion. The model is intended for financial imports and retryable commands; it is not a claim that a transport delivers exactly once.

Expository note: elementary results with explicit proofs. No claim of mathematical novelty or measured product performance.

1. A timeout does not identify the server state

Suppose a caller submits a 1,250-cent booking and receives no response. The booking may have failed before commitment, or it may have committed before the response was lost. A retry is justified by uncertainty at the caller. That uncertainty is not evidence that the financial effect is absent.

A request is a pair (k,p), where k is a caller-scoped operation key and p is a canonical, immutable payload. The payload determines an effect vector v(p) in Z^d and a deterministic result r(p). The components of the vector may represent account balances. This model contains no separate irreversible side effect, such as sending a bank transfer outside the transaction.

Let D be a finite partial map from keys to stored payloads and results. Let J be a finite partial map from keys to committed effect vectors. A valid state satisfies the following invariant.

$$\mathrm{dom}(D) = \mathrm{dom}(J),\ J(k) = v(p_k) \text{ for every } k \text{ in } \mathrm{dom}(D).$$

The aggregate ledger vector is the sum of J(k) over its domain. The empty state is valid. Retaining keys is part of the model: deleting a key while old requests can still return changes the guarantee.

2. The atomic transition

Define T_(k,p) on a valid state by three cases. If k is absent, atomically add D(k) = (p,r(p)) and J(k) = v(p), then return r(p). If k is present with the same payload, change nothing and return the stored result. If k is present with a different payload, change nothing and return a conflict. Payload equality here means exact equality of the canonical fields whose differences matter, not merely equal amounts.

Proposition 1. After every finite sequence of these transitions, the state invariant holds and each accepted key contributes exactly one stored effect.

> *Proof.* Induct on the length of the sequence. The empty state satisfies the invariant. For the inductive step, an absent key is added to both domains in one transition with the required effect; previous entries do not change. An identical retry changes neither map. A conflicting retry also changes neither map. Thus every case preserves the invariant, and the partial-map representation supplies at most one effect for each key.

Proposition 2. Repeating one request is idempotent on state: T_(k,p)(T_(k,p)(s)) = T_(k,p)(s) for every valid state s.

> *Proof.* If k was absent, the first transition stores the payload and effect, and the second takes the identical-retry case. If k already held that payload, both transitions leave state unchanged. If k held a different payload, both reject without changing state. These cases exhaust the possible states of k.

A successfully accepted request also has the same returned result on repetition. A conflict has no financial effect. This distinction prevents two different business operations that accidentally share a key from being treated as interchangeable successes.

3. Reordering independent accepted keys

Proposition 3. Suppose k and l are distinct and their effects are fixed independently of the current state. On valid states, their request transitions commute on the stored maps. Their aggregate effects are added in either order.

> *Proof.* Each transition reads and potentially writes only its own entry in D and J. Updating the k entry cannot change which case applies to the l entry, and conversely. The final maps therefore contain the same entries in both orders. Their aggregate is unchanged because integer-vector addition is commutative.

The independence premise excludes a balance-dependent command such as “withdraw if sufficient funds remain”. Two such commands can compete for the same funds even when they use distinct keys. Nor does this proposition promise the same log ordering, response time or external notification order. It concerns the protected maps and their additive total.

Delivery	Key	Amount	State after delivery
1	order-17	1,250 cents	1 key; total 1,250
2	order-17	1,250 cents	unchanged; stored result
3	refund-4	−300 cents	2 keys; total 950
4	order-17	1,400 cents	conflict; total remains 950

4. Two writes are not the atomic transition

Proposition 4. Separating the key write from the effect write permits a crash schedule that violates either at-most-once safety or completion, even if each individual write is durable.

> *Proof.* Effect first: write the 1,250-cent effect, crash before recording k, then retry. The retry sees no key and writes another effect, giving 2,500 cents. Key first: record k, crash before the effect, then retry. A policy that treats the stored key as completed suppresses the retry, leaving zero cents. Neither state equals the specified atomic transition. These two schedules establish the claimed failure possibilities.

Concurrency presents the same boundary without a crash. Two workers can both observe “key absent” before either writes. If both effects are committed, a unique-key check performed afterwards is too late. The unique claim and the effect must participate in a transaction with an isolation rule that admits at most one successful insertion, or in another mechanism with equivalent atomic state semantics.

The proof assumes such a mechanism. It does not infer atomicity from a function name, a cache lookup or a diagram. If D and J live in separate services, additional protocol assumptions are needed. Merely placing both calls in one application function does not supply them.

5. Safety does not imply that work will finish

Proposition 5. At-most-one committed effect per key does not imply that every submitted request eventually commits.

> *Proof.* Consider an execution in which no request transition is ever committed. The invariant holds forever and every key has zero effects, so at-most-one safety is satisfied. No submitted request completes. This execution is a counterexample to the proposed implication.

For eventual completion, add a liveness premise: a valid request reaches a functioning committer and a complete atomic transition eventually succeeds. Once that occurs, Proposition 1 protects subsequent retries. Key retention must cover the retry horizon, and key scope must distinguish tenants and genuinely different operations. This separates a property of committed state from assumptions about delivery, scheduling and recovery.

6. Engineering interpretation

The useful review question is not “does this endpoint accept an idempotency key?” It is “what invariant relates that key to the durable effect, and which failure schedules preserve it?” A financial command can then be checked against a short model with explicit boundaries. Remote side effects require their own idempotency contract or a carefully specified coordination protocol; they are outside the single-transaction proof above.

These are elementary state-machine arguments, not new distributed-systems results or a verification of the deployed Winstwaker code. The reference below gives practical context for caller-provided identifiers, atomic commitment and the distinction between a retry and a change of intent.

Reference

Malcolm Featonby. Making retries safe with idempotent APIs. Amazon Builders’ Library.

Companion verification. Download the finite checks (Python 3). Exact integer and rational checks supplement the proofs; they do not establish universal claims beyond the checked domains.