Eliminating Cross-Database Split-Brain by Construction: Pre-Freeze Coordination with Node-Autonomous Recovery

yxpil*

October 25, 2026

Abstract

Cross-database split-brain is normally treated as a protocol problem: to stop two independently deployed databases from disagreeing about a shared outcome, one adds a consensus group, a quorum rule, or an external arbiter. We take the opposite route and eliminate the failure class instead of defending against it. The business-asset databases in our design never exchange a message with one another, each performs only single-node local transactions with its own local replication, and every cross-database decision is taken by a stateless business-layer coordinator. The coordinator reserves (freezes) each participant's local assets before any real value moves, records the global transaction in a guarded state machine, and compensates along an explicit per-operation phase model, so that a compensation can never be applied in the wrong direction. Recovery is layered: a poller and a dead-letter handler resume transactions when a participant is unreachable, verifying the real asset state before deciding whether to move forward or compensate, and a second layer lets a recovered node inspect its own local freeze markers and settle its own debt autonomously when the coordinator cannot reach it. We implement the design as an executable teaching artifact: three operating-system processes stand in for three databases, JSON files stand in for shards and replicas, and seven scripted fault cases exercise normal ordering, freeze rejection, mid-commit failure, shard partition and recovery, node crash and restart, orphan-freeze reconciliation, and retry exhaustion. All seven cases pass; we report the measured transcript, the per-case wall-clock durations, and the coordinator's protocol traffic. The artifact is not a production system. We state the six gaps that separate it from production, the trade-offs it accepts (eventual rather than serializable consistency, write amplification, a new critical component, and a freeze TTL that must be tuned to the real recovery-time distribution), and the kinds of system for which we believe a simpler mechanism should be preferred.

*Undergraduate student, School of Software Engineering, Yunnan Technology and Business University, China.

1 Introduction

1.1 The problem we want to remove, not to handle

Suppose a platform must debit a user's balance in one database and a merchant's stock in another. The two databases are deployed independently, are administered by different teams, and are not replicas of one another: they hold disjoint data and answer disjoint queries. A crash, a network partition, or a slow call on one side must not leave the pair in a state that no user-visible rule can explain.

The textbook answer is a distributed commit. A transaction manager in the middle runs two phases, participants vote, and a coordinator decides. This works, and it has been well understood since the 1980s [Gray and Reuter, 1992, Garcia-Molina et al., 2002]. It also carries three costs that a high-concurrency deployment feels immediately: participants hold row locks for the duration of the global transaction, the coordinator is a blocking single point, and the protocol itself creates the possibility of a split-brain outcome, in which two participants reach opposite conclusions about a global transaction and disagree about which is authoritative.

That last cost is the subject of this paper. A great deal of engineering goes into preventing participants from splitting, and the standard vocabulary for that effort is the vocabulary of consensus: replicated logs, quorums, majorities, leases, arbitrators [Lamport, 1998, Ongaro and Ousterhout, 2014]. A remarkable property of that vocabulary is that it presupposes communication. Consensus is a protocol among participants that exchange messages, and the hard part of consensus is what to do when those messages stop arriving.

We ask a different question. Instead of asking how to keep the databases from splitting, we ask whether the cross-database commit can be designed out of the architecture entirely.

1.2 Our approach

We design a framework in which the two business-asset databases never send each other a message. Not "rarely", not "only during recovery": never. Each database exposes only single-node local transactions, with its own local high availability, and its own idea of what a durable write is. All cross-database coordination is pushed up into a stateless business-

layer coordinator, which may be restarted, replicated, or scaled out at will because it holds no state that is not also recorded in a durable transaction log.

Three mechanisms carry the design.

Pre-freeze. Before any real value moves, the coordinator asks each participant to reserve the assets it will need, by writing a freeze marker inside the participant's own local transaction. A freeze marker does not deduct anything; it reduces the balance or the stock that the participant will promise to anybody else, and it carries an expiry time. Because reservation happens before mutation, the dangerous half of the problem – deciding to move value and then failing to move it – becomes a local, individually transactional decision at each participant.

A guarded state machine plus an explicit per-operation phase. The global transaction advances through six states, and every transition is checked by a guard that rejects illegal moves. Independently, every operation inside a transaction carries one of four phases (PENDING, FROZEN, COMMITTED, COMPENSATED), and the compensation routine branches on the phase alone: a committed operation is refunded, a merely frozen operation is released. The direction of a compensation is therefore a function of recorded state, never a guess about business semantics. This is the most common source of accidents in compensation-based designs, and the phase model is the mechanism that makes it structurally impossible.

Two-layer self-healing. The first layer is a poller that scans non-terminal transactions and a dead-letter handler that reconnects to the involved participants, verifies their real asset state, and then either moves the transaction forward or starts compensating. The second layer exists because the first one has a blind spot: it assumes the coordinator can eventually reach the participant. When a participant node is down for a long time, that assumption fails, and nobody can deliver the compensation. The second layer therefore gives the node itself the authority to settle its own debt on restart: it queries the global transaction record store, reconciles the record against its own local freeze markers, performs the compensation inside its own local transaction, and writes the outcome back.

1.3 Contributions

This paper contributes the following, all of which are backed by an executable artifact and by measurements reported in Section 6.

1. A precise formulation of *structural elimination* of cross-database split-brain: by removing all inter-database communication, the class of failure is excluded by construction rather than defended against by a quorum argument, together with the conditions under which that argument is and is not valid (Sections 3 and 4).

2. A concrete coordination protocol built on that structure: pre-freeze reservation with expiring markers, a six-state global transaction machine with a transition guard, and a four-phase per-operation model that fixes the direction of every compensation (Section 3).

3. A two-layer recovery design that covers both the case in which the coordinator can reach a participant and the case in which it cannot, with pseudocode for both layers (Sections 3.7 and 3.8).

4. An executable teaching artifact of that protocol: 21 source files and 1677 lines in the framework package, 8 case files and 279 lines in the case package, seven scripted fault cases, and a measured evaluation reporting per-case durations, the coordinator's protocol traffic, and the verbatim transcript of the driver (Sections 5 and 6).

5. An explicit accounting of the design's cost and failure envelope: the write amplification it imposes, the consistency level it provides, the residual risk it retains, and nine specific observations and defects that the artifact's own code reveals, including a window in which the recorded phase lags the real asset state (Sections 6.5, 9).

1.4 Scope and honesty statement

The artifact studied here is a *teaching simulation*, and we say so before presenting any of its numbers. Three operating-system processes stand in for three databases; the "shards" and "replicas" are JSON files; the "network partition" is a flag that makes one shard raise an error; the "node crash" is the termination of a process. There is no real database engine, no real replication, no real message queue, and no real partition. What it does provide is a faithful, executable model of the protocol's control flow, its state machine, its compensation directions, and its two recovery layers, with fault injection at the points where those mechanisms are supposed to fire. The measured durations therefore describe the protocol's control flow and the script that drives it, not the throughput of any production system. Section 6 repeats this caveat next to each number, and Section 9 lists the six gaps that separate the artifact from production, together with the author's own judgement that most business systems should not use this design at all.

2 Background

2.1 The consistency-model space

Cross-database coordination is a well-trodden path with four principal branches, and the right way to position our design is against all four. Section 8 discusses each of them in more detail; here we state only what the design must do differently.

Two-phase commit [Gray and Reuter, 1992] asks each participant to prepare and to vote, and then broadcasts a global decision. The decision is durable and participants can always ask for it, which is what makes the protocol tractable. The cost is that a participant that has voted yes must hold its locks until the decision arrives, and a coordinator that crashes between the two phases leaves the participants blocked.

The quorum family answers the blocking problem by making the decision recoverable by consensus: a replicated log with a majority quorum, as in Raft [Ongaro and Ousterhout,

2014], or a time-based variant of it [Lamport, 1998]. These protocols assume that the participants of the consensus group can reach each other, and they spend their complexity budget on the case where they cannot.

TCC (try-confirm/cancel, the service-composition descendant of sagas) splits each business operation into a *Try* that reserves resources, a *Confirm* that makes them final, and a *Cancel* that releases them. *SAGA* [Garcia-Molina and Salem, 1987] instead splits a long transaction into subtransactions, each with a compensating action, orchestrated by a central coordinator. Both are lighter than two-phase commit because no participant holds a lock across the global transaction, and both make the business developer write the compensation.

Reliable messages and the transactional outbox write the business data and an outbox record in one local transaction, publish the record, and apply it in a consumer that retries on failure. This is the dominant production pattern for eventual consistency [Richardson, 2018], and its correctness rests on a strong and usually implicit assumption: the consumer will eventually be reachable, so retrying the message is always a valid strategy.

Our design borrows from two of these families and rejects the others. It borrows TCC's reserve-in-Try idea, but reduces the reservation to a uniform local marker instead of a business-specific reserve interface, which removes the need for the business developer to write a third interface per operation. It borrows SAGA's central orchestration and its compensation model. It rejects two-phase commit because of lock holding and blocking, and it rejects the reliable-message family for the reason that matters most here: in our failure model the consumer is not a queue worker that will eventually be retried, it is a database node that may be down for a long time, and a coordinator that cannot reach it must still make progress somewhere. That gap is what motivates the second recovery layer.

2.2 What it means to eliminate a failure class

The impossibility result of Fischer, Lynch and Paterson [Fischer et al., 1985] says that in an asynchronous system with even one faulty process, no protocol can guarantee both agreement and progress. A design that claims to have avoided that dilemma is either misstating its assumptions or has changed the problem. We have changed the problem: agreement *about which participant is authoritative for a shared key* is not a question our system ever asks, because no key is shared between the databases. What remains is a question about a single logical operation that happens to touch two independent stores, and that question is answered by a single authority: the business-layer coordinator.

The distinction is worth stating precisely, because it is the difference between a design and a slogan. The framework does not claim that distributed transactions are easy, and it does not claim that eventual consistency is as good as serializability. It claims that one particular failure mode – two independent databases disagreeing about a global transaction because a partition split the participants – can be made impossible to express, and that the price of that exclusion is a set of costs which we measure and state.

3 Design

3.1 Threat model and goals

The system under discussion coordinates a logical operation over two business-asset databases, a user-asset database and a merchant-stock database, and keeps a third component, a global transaction record store, that holds one record per global transaction. The participant databases are deployed independently, are replicated locally, and are not under a common control plane. The coordinator is a business-layer service that may be restarted at any time, may run in several copies, and holds no state of its own beyond what the record store already contains.

The fault model we defend against is: a participant process is down; a participant shard is unreachable; a participant call fails transiently or permanently; the coordinator process is restarted between any two steps; the record store is temporarily unavailable; and, in the extreme, the record store's primary data is destroyed. We do not defend against a participant that loses already-committed data, against a Byzantine participant, or against a network that silently corrupts payloads.

The goals are: (G1) no cross-database split-brain; (G2) no compensation in the wrong direction; (G3) every transaction reaches a terminal state, either resolved or explicitly handed to a human; (G4) the caller observes one operation, not a sequence of steps; (G5) the design is understandable enough that an operator can debug a stuck transaction by reading one record and two local marker sets.

3.2 Architecture

Figure 1 shows the whole system. The three database processes are at the top, the coordinator and its modules are in the middle, and a business caller is at the left. The two business-asset databases are drawn with a red cross between them: they never send each other a message. The green dashed line is the second recovery layer, in which a restarted node reads the record store directly. The orange arrow is the coordinator writing transaction state and per-operation phases.

3.3 The global transaction state machine

The global transaction record carries a state drawn from six values, and a guard that defines the legal edges. The legal transitions are

The guard admits three things and nothing else: an edge that is in the table above; a self-edge, so that replaying a transition is idempotent; and an edge from *any* non-terminal state to `MANUAL_REVIEW`, which is the escape hatch used when retries are exhausted. Every attempt to move the transaction along an edge that is not in the table is rejected by an exception, and the rejection happens before any persistent write.

Two properties of this table are worth naming. First, the terminal states have no outgoing edges, so a resolved transaction

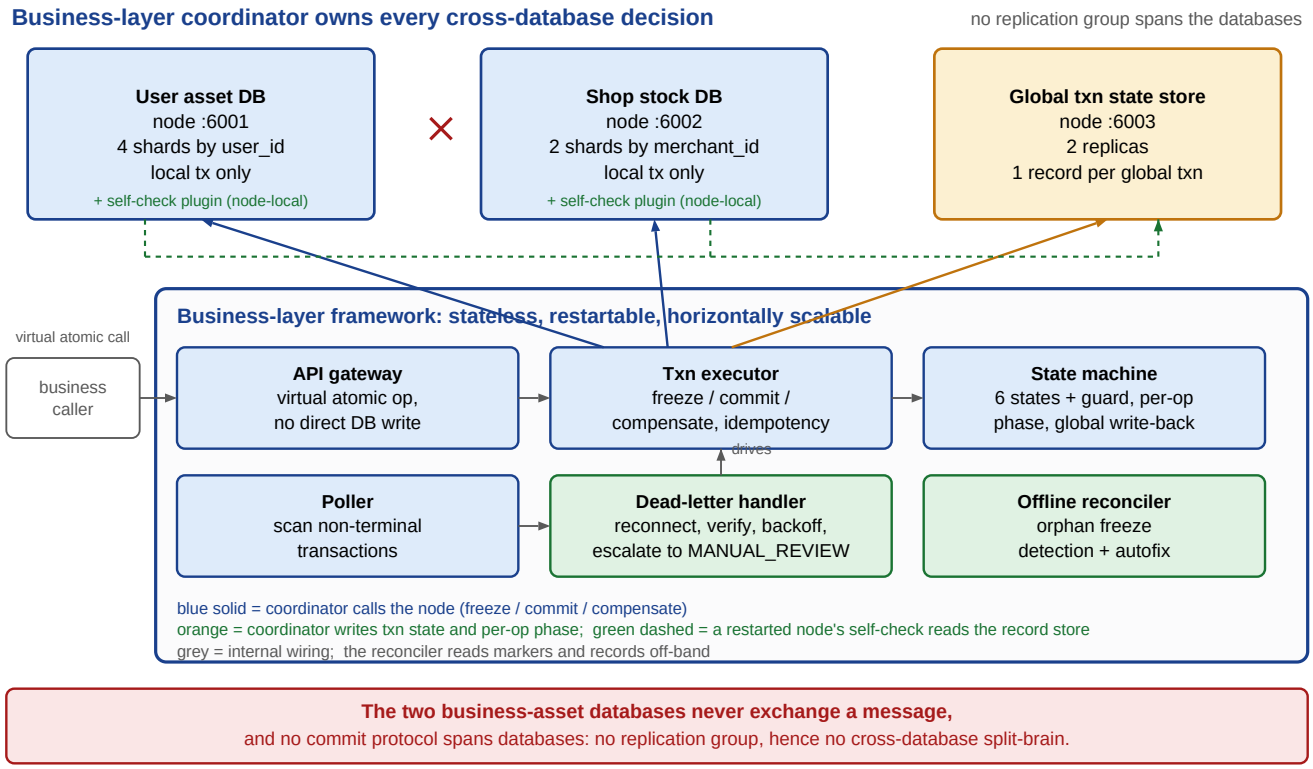


Figure 1: Architecture. Three independent database processes (a user-asset database, a merchant-stock database, and a global transaction record store) sit above a stateless business-layer framework that owns every cross-database decision. The two business-asset databases never exchange a message, and no commit protocol spans the databases, so there is no replication group in which a split could occur. The only cross-component traffic besides the coordinator's own calls is the green dashed path: when a node restarts, its self-check plugin reads the record store in order to settle a transaction it still owes.

cannot be resurrected. Second, `MANUAL_REVIEW` is reachable from everywhere but leads nowhere: it is a hand-off to a human, not a state the machine works itself out of. This is deliberate. The alternative – retrying forever – hides an unresolvable condition behind an unbounded queue, and a system that never admits defeat is a system whose backlog is not observable.

Figure 2 shows the machine and the phase model that sits inside each transaction.

3.4 The per-operation phase model

A global transaction is a sequence of operations; in the artifact an order places one operation on the user database and one on the merchant database. Each operation carries a phase, and the phase, not the business semantics, decides what a compensation must do:

phase	meaning	compensation
`PENDING`	not reserved yet	nothing to undo
`FROZEN`	reserved, no real value moved	release the reservation
`COMMITTED`	real value moved	refund the value
`COMPENSATED`	undone	nothing to undo

This table is the whole of the mis-compensation defence. A compensation routine that asks "what should I do about this operation?" and answers "refund it, probably" is one refactoring away from double-spending. A compensation routine that reads a phase and dispatches mechanically cannot be wrong in that way: the phase was set by the code that performed the corresponding real effect, so the record of what happened and the recipe for undoing it cannot drift apart.

There is a subtlety that the phase model deliberately does not hide. The phase is a record of the past, and a record can be stale. We return to this in Sections 4.5 and 6.5: the pair "participant commits" and "coordinator records the phase" is not atomic, and the artifact contains a window in which the record understates reality.

3.5 The pre-freeze protocol

Algorithm 1 shows the forward path. The structure is: freeze every participant first, then commit every participant; a failure at any point diverts to the compensation path. The order matters, and the alternative order is worse. If we committed first and froze second, a failure between the two commits would leave real value moved with no reservation to fall back on, and the compensation would have to guess. By freezing first, the compensation always has a well-defined target: the reservation

Table 1: The global transaction state machine. Every transition is checked by a guard before any persistent write; the three terminal states have no outgoing edges, so a resolved transaction cannot be resurrected.

state	successors	terminal?
`INIT`	`FROZEN`, `ROLLBACK`, `MANUAL_REVIEW`	no
`FROZEN`	`SUCCESS`, `ROLLBACK`, `MANUAL_REVIEW`	no
`ROLLBACK`	`ROLLED_BACK`, `MANUAL_REVIEW`	no
`SUCCESS`	(none)	yes
`ROLLED_BACK`	(none)	yes
`MANUAL_REVIEW`	(none)	yes: a hand-off to a human, not a state the machine works itself out of

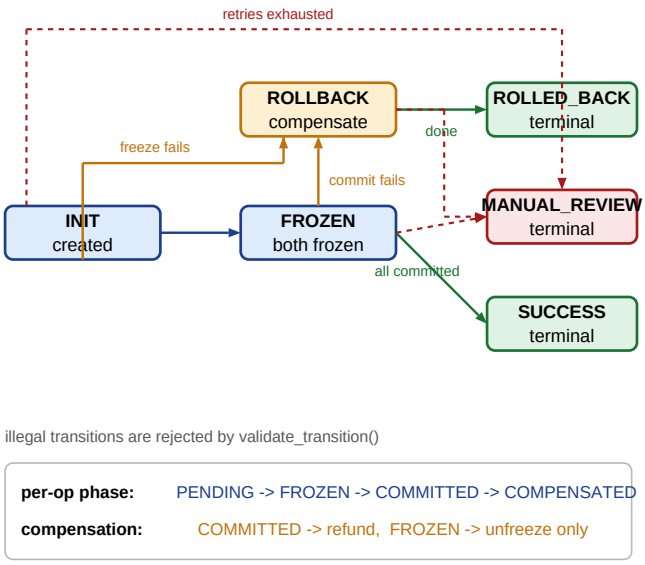


Figure 2: The global transaction state machine (top) and the per-operation phase model (bottom). Illegal transitions are rejected by the guard before any write. The phase model fixes the direction of a compensation: `COMMITTED` implies a refund, `FROZEN` implies a release and nothing else.

that the participant itself holds.

Two details are load-bearing. The freeze is *idempotent* at the participant: a participant that already holds a marker for this transaction returns success without creating a second reservation, which is what makes the retry loop safe. And the record of the phase is persisted immediately after each individual operation, so that a crash between two operations leaves an accurate prefix rather than an all-or-nothing statement.

The freeze marker itself is deliberately weak. It records an amount and an expiry time, and it reduces the participant's available balance or stock, but it does not move any value. The value moves exactly once, in the commit step. The consequence is that the dangerous direction of failure – value moved, reservation lost – is replaced by the harmless direction – value not moved, reservation present, reservation releasable.

Algorithm 1 Pre-freeze and commit (coordinator, one global transaction)

Require: operations ops, idempotency key k
$r \leftarrow$ record with key k, if any; **return** r
create r: $state = \texttt{INIT}$, phases `PENDING`
for each $op \in r.ops$ **do**
  try $phase \leftarrow \texttt{FROZEN}$ ▷ reserve
  persist $r.ops$
  catch unavailable $\Rightarrow$ BACKOFF(r)
  catch business error $\Rightarrow$ ROLLBACK(r)
end for
$state \leftarrow \texttt{FROZEN}$ ▷ guarded
for each $op \in r.ops$ **do**
  try $phase \leftarrow \texttt{COMMITTED}$ ▷ deduct
  persist $r.ops$
  catch unavailable $\Rightarrow$ BACKOFF(r)
  catch business error $\Rightarrow$ ROLLBACK(r)
end for
$state \leftarrow \texttt{SUCCESS}$; **return**

3.6 Compensation

Algorithm 2 shows the compensation path. It is short, and it is short on purpose: the branch is taken on the phase and nowhere else.

The compensations themselves are idempotent at the participant as well: a refund adds the amount back and drops the marker, and a release only drops the marker. Repeating a compensation therefore cannot double-refund.

3.7 Layer one: poller and dead-letter handler

The first recovery layer is the standard pair. A *poller* scans the record store for transactions that are not terminal and whose scheduled retry time has arrived, and hands each to a *dead-letter handler*. The handler does four things, in this order: (1) it checks reachability of the participants the transaction involves; (2) it *verifies the real asset state*, comparing the recorded phase of each operation against the reservation markers that actually exist on the participants, and raising a drift alert when they disagree; (3) it asks the executor to drive the transaction forward, which the state machine routes to the freeze, commit, or compensation path as appropriate; and (4) on failure, it schedules a retry with exponential backoff, esca-

Algorithm 2 Compensate (coordinator)

Require: global transaction r in a non-terminal state

$state \leftarrow$ ROLLBACK ▷ guarded
for each $op \in r.ops$ **do**
  if $phase =$ COMMITTED **then**
    refund the real value at the participant
  else if $phase =$ FROZEN **then**
    release the reservation at the participant
  end if
  $phase \leftarrow$ COMPENSATED; persist $r.ops$
end for
$state \leftarrow$ ROLLED_BACK

lating to MANUAL_REVIEW once the attempt budget is spent.

Step (2) is the part that distinguishes this handler from a naive retry loop. Before deciding to retry a forward step, the handler asks the participant what it actually has, rather than trusting the coordinator's own record of what it asked for. In the artifact the verification is a read of the participant's marker set; the handler notes two specific inconsistencies, a recorded freeze with no marker on the participant, and a recorded commit that still has a marker.

The backoff is the standard exponential schedule, $b_i = b_0 \cdot 2^{i-1}$, with the attempt counter persisted in the record so that the budget survives a coordinator restart.

3.8 Layer two: node startup self-check

The second layer exists because layer one has an unfillable gap. Layer one lives in the coordinator and can only act on a participant it can reach. If the participant is down for a long time, the coordinator can do nothing but count attempts, and when the budget is spent the transaction is handed to a human with the participant's reservation still in place. For a business in which the participant is a merchant shard that may be down for half an hour, that is a bad outcome, and it is the outcome that the standard compensation literature papers over by assuming that the downstream will come back.

The second layer moves the authority to the participant. Each participant node embeds a self-check plugin that runs once when the node starts. The plugin asks the record store for the transactions that are not terminal, looks at its own local markers, and decides, for each transaction, what it owes. The rules are deliberately restrictive, and they are the reason the node is allowed to act on its own:

1. It may compensate only a transaction that the record store marks as ROLLBACK, or one whose reservation has expired according to the expiry time it wrote itself. A node can never decide, on its own initiative, that a transaction should be undone.

2. It touches only the markers and values inside its own database, and it writes the outcome back to the record store. Self-healing without a write-back would leave the record store and the participants disagreeing, which is the very failure we are trying to avoid.

Figure 3 contrasts the two layers. In panel (a) the coordinator can reach the participant and drives the transaction itself. In panel (b) the participant is unreachable, so the coordinator can only schedule retries, and the settlement happens after the participant returns and its self-check plugin runs.

3.9 Offline reconciliation and backups

The two layers above are online mechanisms: they act only while the system is running and the record store is reachable. A third mechanism is off-line and unconditional. The reconciler reads the marker sets of both participants and the records of the store, and cross-checks them. It detects three situations: a marker with no corresponding record, which is an *orphan freeze*; a terminal transaction that still has markers, which is a leak; and a store whose primary data has gone missing while a backup exists. The first two are repaired automatically, in the participants' own local transactions, and each repair raises an alert. The third is reported, because restoring the store is an operator decision.

The orphan freeze deserves a note, because it is the residual risk of the whole design. A freeze marker is a local fact: the participant really did reserve the value, and that reservation reduces the balance that the participant will promise to anybody else. If the record of the transaction is destroyed, the reservation becomes invisible to the coordinator and to the reconciler's cross-check, and the value stays reserved until the marker expires or an operator intervenes. The artifact's snapshot-and-restore mechanism is the intended answer, but a snapshot that is itself lost, or one that predates the transaction, leaves the orphan behind. We regard this as the sharpest residual risk of the design and return to it in Section 9.

3.10 Idempotency and the no-bypass constraint

Two invariants complete the design. First, every operation carries a global transaction identifier and an idempotency key; a repeated submission with the same key returns the existing record instead of executing again, and the participants' own freeze, commit, refund and release operations are idempotent, so a repeated call cannot double-spend. Second, the asset mutations are reachable only through the coordinator: the business code submits to a gateway, and the gateway is the only component that holds the participant clients. This is a convention rather than a mechanism in the artifact – the Python interpreter does not enforce it – but it is the convention that makes the freeze-before-commit property meaningful, since any bypass would move value without a reservation.

4 Correctness argument

4.1 Assumptions

We state the assumptions explicitly, because the argument below is only as strong as they are.

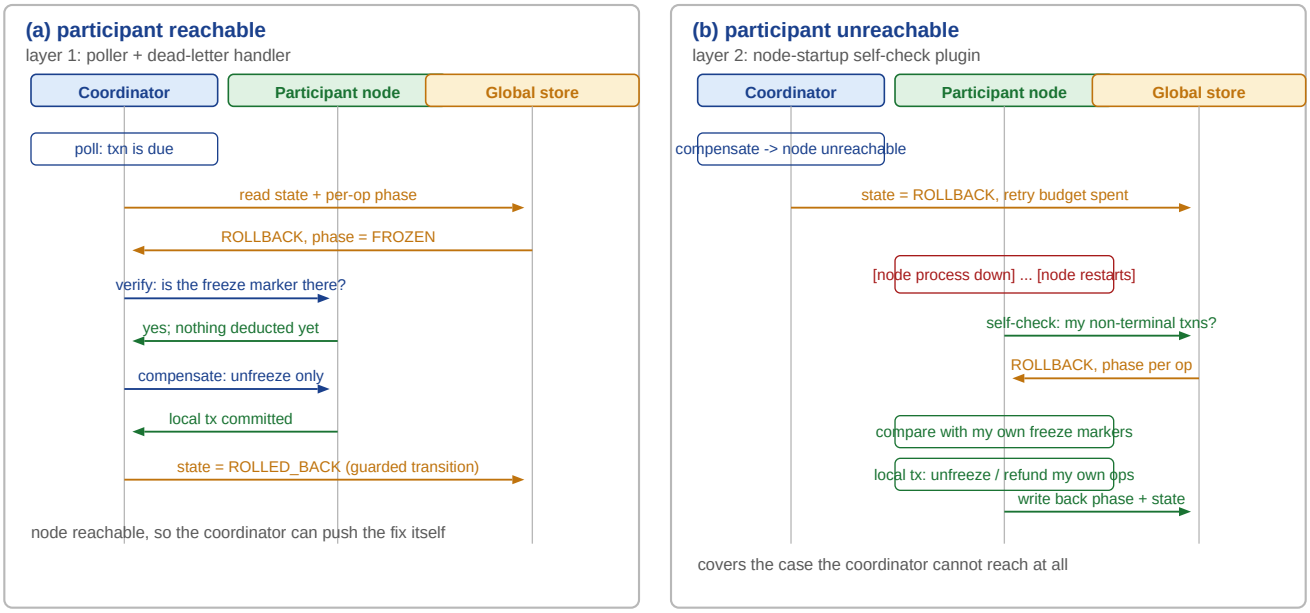


Figure 3: The two recovery layers. (a) The participant is reachable: the poller finds the transaction, the dead-letter handler verifies the real asset state, and the executor drives the transaction forward. (b) The participant is down: the coordinator's calls fail, it schedules retries, and the settlement is performed by the participant's own self-check plugin after the node restarts, which reads the record store, compensates locally, and writes the result back.

(A1) A participant's local transaction is atomic and durable: a reservation, a deduction, a refund, and a release either take effect completely or not at all, and the effect survives a crash of the participant process.

(A2) A participant's operations are idempotent under repetition, and the idempotency key is unique per logical operation.

(A3) The record store is the single authority for the state of a global transaction, and every participant's marker set is eventually reachable from the coordinator or from the participant's own self-check.

(A4) The transition guard is the only path by which a transaction state changes, so that illegal transitions cannot be recorded.

(A5) The coordinator is restartable and its in-memory state is reconstructible from the record store.

(A6) Clock skew between the coordinator and the participants is bounded by the freeze TTL, so that an expiry decision is never taken on the basis of a badly skewed clock.

4.2 The phase model fixes compensation

Proposition 1 (Direction of compensation)**.** *Under (A1) and (A2), the compensation routine refunds exactly those operations that moved real value, and releases exactly those operations that did not.*

Sketch. Each operation's phase is assigned at the point where the corresponding participant-local effect is performed: `FROZEN` is written after the reservation has been recorded, and `COMMITTED` is written after the deduction has been recorded. The compensation routine branches on the phase alone, and performs a refund only for `COMMITTED` and a release only for `FROZEN`. By (A1) the participant-local effect and the phase write are each individually atomic, and by (A2) repeating either effect changes nothing. Therefore the compensation undoes precisely the effects that were applied, and each effect is undone at most once. □

The proof is only as strong as the correspondence between the phase write and the participant effect, and that correspondence is not atomic. We examine the gap in Section 4.5.

4.3 Split-brain is excluded by construction

Proposition 2 (Structural exclusion)**.** *Under the architecture of Figure 1, no state of the system contains two business-asset databases that have taken incompatible conclusions about the same global transaction.*

Sketch. A split requires a shared decision to be taken more than once, under partition, by components that are assumed to be able to reach each other. In our architecture, the business-asset databases hold disjoint data and never exchange a message, so there is no decision about a shared key that two of them could take differently. The only component that records the state of a global transaction is the record store, and only

the coordinator and a participant's self-check plugin write to it; both write under the guard, and both are single authorities for the transaction they write. Two different components could write the same record, so the record store is not a linearisable register under concurrent writers – and that is precisely the gap that Section 4.5 identifies and that Section 9 lists as a missing mechanism. □

4.4 Progress and its limits

Proposition 3 (Bounded progress)**.** *Under (A3) and (A5), every global transaction eventually reaches a terminal state.*

Sketch. If the participants are reachable, the poller repeatedly finds the transaction and the dead-letter handler drives it; the state machine admits no cycles, so the transaction must leave `INIT` and `FROZEN` and either reach `SUCCESS`, enter `ROLLBACK`, or exhaust its attempt budget and enter `MANUAL_REVIEW`. If the participants are not reachable, the attempt budget is finite, so the transaction reaches `MANUAL_REVIEW`. In both cases the terminal state is reached, and the record store's persistence of the attempt counter makes the budget survive coordinator restarts. □

The proposition is about *termination*, not about correctness of the outcome, and it is a weak form of liveness: the terminal state may be `MANUAL_REVIEW`, which is a hand-off to a human, and a transaction may terminate in a state that still holds a reservation.

4.5 Where the safety argument does not hold

Three points break the argument, and we list them rather than hide them.

The phase write is not atomic with the participant effect. Between the participant committing the deduction and the coordinator recording `COMMITTED`, the record understates reality: it still says `FROZEN`. The artifact's dead-letter handler detects exactly this disagreement, since it compares the recorded phase against the markers that exist on the participant, and raises a drift alert. It does not repair it. If the coordinator restarts in that window, the state machine will route the transaction to the compensation path, and the compensation will read `FROZEN` and release the reservation rather than refund the value that was in fact deducted. The artifact's own code contains this window; we describe it precisely in Section 6.5 and list the missing mechanism in Section 9. The seven cases do not exercise it, because the fault injector never kills the coordinator between a participant commit and the phase write.

Two authorities can write the same record. The coordinator's transitions go through the guard; the node's self-check writes the terminal state directly. The artifact therefore has a single authority for *deciding* but not for *writing*, and the missing mechanism is a version number on the record with a compare-and-set, or a leader election that makes the coordinator the only writer.

The record store's loss is not compensated. If the record of a transaction disappears, the reservation it caused remains in place, and no online mechanism can see it. Only the offline reconciler, and only if it still has a record to cross-check against, can find it. The residual orphan freeze is therefore not a bug in the design but a consequence of choosing a single authority: it concentrates the failure of the record store into stranded reservations.

Table 2: The seven modules of the framework package.

module	responsibility
`core/state_machine`	six states, the transition table, the guard, the terminal-state test
`core/executor`	the virtual atomic operation, the freeze and commit paths, compensation, the retry budget
`scheduler/poller`	scans non-terminal records, hands due ones to the dead-letter handler
`scheduler/deadletter`	reachability probe, real-asset verification, drift alerts, retry scheduling
`node/self_check`	the second recovery layer: local reconciliation and autonomous compensation on node start
`reconcile/offline_reconcile`	cross-check of markers against records; orphan-freeze detection and repair; store-loss report
`gateway`	the only component that holds participant clients; the entry point for business code

5 Implementation

5.1 Module map

The artifact is a pure-Python package that uses only the standard library. Table 2 lists its modules.

Recounted directly from the source tree, the framework package contains 21 Python files and 1677 lines, and the case package contains 8 files and 279 lines. The three node-server processes, the driver, and the cluster harness add a further 306 lines.

5.2 Storage layer

Each node process stores its data in JSON files, one per shard, and each shard file is the unit of a local transaction. A local transaction is implemented as: acquire a lock on the shard file, read the JSON, mutate it in memory, write a temporary file and atomically replace the original, and release the lock. The lock is a directory created atomically, which makes it a mutual-exclusion primitive across processes, with a timeout and a stale lock reaper. The atomic replacement is what prevents a reader from seeing a half-written file. The effect is a genuine, if slow, per-shard transaction: a crash in the middle of a local transaction leaves the previous contents intact.

The record store keeps one main file and two replica files, and every logical write is applied to all three. It also writes periodic snapshots.

Table 3: Configuration values, read from the framework's configuration module. All are demonstration values.

parameter	value	role
BASE_BACKOFF	0.3 s	base of the exponential retry backoff
MAX_ATTEMPTS	3	attempts before MANUAL_REVIEW
FREEZE_TTL	30 s	reservation expiry
NODE_HTTP_TIMEOUT	3.0 s	per-call timeout
USER_SHARDS	4	user-asset shards
SHOP_SHARDS	2	merchant-stock shards
TXN_REPLICAS	2	record-store replicas

5.3 Process-level cluster and fault injection

The driver starts three node-server processes, each on its own TCP port, and waits for their health endpoints to answer before proceeding. Fault injection is done through an administrative endpoint on the node: a shard can be marked unreachable for a period, a node's next commit can be made to fail with a business or an availability error, and a node can be told to re-run its self-check. A process-level crash is a real process termination followed by a real restart, which is what makes case 5 an honest test of the second recovery layer.

5.4 Parameters

Table 3 lists the configuration. These are the values that are actually in the code. They are demonstration values, chosen so that the seven cases run quickly; they are not production recommendations, and Section 9 says which of them would have to be re-derived for a real deployment.

5.5 Reproducing the numbers

All reported numbers are produced by scripts in the paper's `scripts` directory. The framework's own driver is the primary artifact and is unmodified:

command	produces
`python run_all.py`	the transcript of Section 6.3
`python scripts/run_cases_timed.py -runs 3`	`case_timings.csv`
`python scripts/count_protocol_traffic.py`	`protocol_traffic.csv`
`python scripts/sanitize_transcript.py`	the English transcript
`python scripts/make_figures.py`	the four figures

The timing and traffic scripts import the framework's own harness and its own seven case modules and wrap them; they do not modify them. The traffic script replaces the client object's request method with a counting proxy and then runs the cases unchanged.

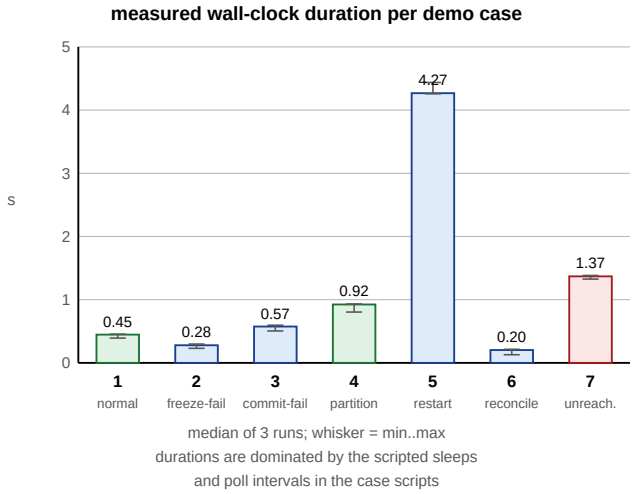


Figure 4: Per-case wall-clock duration, median of three runs of the full suite, with the observed range as whiskers. The dominant term in cases 4, 5 and 7 is the scripted sleep in the case driver, not protocol work.

6 Evaluation

6.1 Protocol and its limits

The evaluation answers three questions: does the protocol reach the intended terminal state in each of the seven scripted faults; how long does each case take and how much traffic does the coordinator generate; and what does the artifact's own code reveal that the cases do not exercise.

The protocol is: start a fresh cluster, run the seven cases in order on the same cluster, and record for each case the terminal state of every global transaction it created, the number of recorded state transitions, the wall-clock duration, the number of coordinator-originated requests, and the number of logical writes to the record store. The whole suite was run three times; the table reports the median and the observed range.

We state the limits of this protocol before the results. The three operating-system processes are real processes, but they communicate over loopback HTTP and store their data in JSON files, so the durations measure scripted sleeps and process lifecycle events rather than network or disk behaviour. The fault injector is a flag, not a partition. There is no contention, no concurrency, and no load. The numbers below characterise the control flow of the protocol; they are not performance results, and we do not present them as such.

6.2 Results

All seven cases pass, and the driver exits with status zero. The terminal states are the intended ones in every case. Figure 4 shows the same durations graphically; the whiskers are the observed range over the three runs.

Two observations are visible in the table. The first is that the happy path (case 1) is the cheapest case in requests and writes,

Table 4: Per-case results over three runs of the full suite. Durations are wall-clock, in seconds, and include the scripted sleeps of the case driver. Requests are coordinator-originated HTTP requests; the counter runs in the coordinator process, so requests issued by a node's own self-check are not included, and the request column is a lower bound. Store writes are logical writes to the record store; each such write is applied to the main file and to both replicas.

case	scenario	terminal state	att.	trans.	median s	range s	req.	writes
case1_normal	normal order, idempotent replay	SUCCESS	0	3	0.447	0.392–0.455	24	7
case2_freeze_fail	freeze rejected, insufficient funds	ROLLED_BACK	0	3	0.279	0.230–0.298	16	5
case3_commit_fail	merchant commit rejected mid-commit	ROLLED_BACK	0	4	0.575	0.505–0.596	28	9
case4_network_recover	shard partition, then recovery	SUCCESS	1	3	0.924	0.804–0.932	39	8
case5_node_selfcheck	node crash, then restart	ROLLED_BACK	2	3	4.268	4.258–4.437	29	9
case6_reconcile_orphan	orphan freeze, store loss	n/a	0	0	0.203	0.129–0.215	13	0
case7_retry_exhaust	participant permanently unreachable	MANUAL_REVIEW	3	2	1.369	1.326–1.383	46	5
sum of the medians					8.065			

and the second is that the most expensive case in requests is the one that is supposed to be expensive: case 7, where the participant never comes back, generates 46 requests across three attempts and then a hand-off to a human. The attempt budget, not the retry count, is what bounds it.

The transitions column deserves a comment, because it is smaller than a reader might expect. Case 5 records three transitions even though the transaction reaches ROLLED_BACK: the final write is performed by the restarted node's self-check plugin, which writes the record directly and therefore does not fire the coordinator's transition callback. The transition log is thus a record of the coordinator's decisions only, which is itself a small illustration of the two-authority problem discussed in Section 4.5.

6.3 Transcript

Figure 5 is the driver's own output. The framework prints in Chinese; the listing is the same run with the prose translated and every identifier, state name, number and assertion value left byte-identical, produced by a script that ships with the paper and that refuses to emit a transcript containing a CJK character. The untranslated UTF-8 output is shipped alongside it.

6.4 Protocol traffic and write amplification

The trade-off most often overlooked by a design of this shape is its write amplification. The measured traffic gives the number directly. The happy path performs four participant-local transactions (two reservations and two deductions) and issues seven logical writes to the record store: one to create the record, one after each of the two reservations, one for the transition to FROZEN, one after each of the two deductions, and one for the transition to SUCCESS. Each of those writes is applied to the main file and to both replicas, so the record store performs 21 file writes for a transaction that moves value twice. The record store is thus written roughly 1.75 times per participant-local transaction, and it is written on every step of every transaction.

The cost is worse than a constant, because the record store's write rewrites the entire record set rather than a single record: each logical write re-serialises every transaction the store knows about. The cost of one write is therefore linear in the number of transactions the store holds, and the cost of a batch of n transactions is quadratic. This is visible in the source and is not exercised by the seven cases, which hold only a handful of records; we report it as a code-level observation rather than as a measurement.

The read traffic is dominated by record-store lookups. Across the seven cases the coordinator issued 65 record reads against 43 record writes, and the reads are mostly single-record fetches made to re-check a transaction's state before acting on it. Two of these reads are full-table fetches triggered by the idempotency check: the de-duplication scans the whole record set and compares keys, instead of relying on a unique index. Its cost therefore grows linearly with the number of records, and the check happens on every submission that carries a key.

6.5 Negative results and code-level observations

The evaluation of an artifact should report what it does not do. We list nine observations, all derived from the artifact's own code and from the state left behind by the run, and we mark the ones that the seven cases do not exercise.

1. *The phase write is not atomic with the participant effect.* Between the participant committing the deduction and the coordinator recording COMMITTED, the record still says

FROZEN. The dead-letter handler detects the disagreement and raises a drift alert, but it does not repair it, and the compensation path would read FROZEN and release the reservation instead of refunding the value that was actually deducted. **Not exercised by the seven cases**: the injector never kills the coordinator between a participant commit and the phase write. This is the most serious defect we found.

2. *The node's self-check bypasses the transition guard.* The coordinator validates every transition it performs, but the self-check plugin writes the terminal state directly, so the guard protects only one of the two writers. The transition log makes this visible: case 5 reaches ROLLED_BACK without a recorded transition.

3. *The reachability probe is process-level, not shard-level.* The node's health endpoint reports the process as up while a shard is isolated, and reports the shard outage in a separate field that the probe does not read. A shard isolation is therefore detected by the failing operation, not by the probe.

4. *Idempotency de-duplication is a full scan.* It fetches the entire record set and compares keys in the coordinator's process. There is no unique index, and the cost grows with the store.

5. *The durations are script-bound.* The largest term in cases 4, 5 and 7 is the sleep in the case driver, and the largest term in the whole suite is the process restart in case 5. These are not protocol costs and must not be read as such.

6. *The operation order decides which compensation is exercised.* The gateway always places the user operation first, so the user is debited before the merchant's commit is attempted, and case 3 exercises the refund path. With the opposite order, case 3 would have exercised the release path instead.

7. *Manual review is a state, not a resolution.* After the run, the user still holds the reservation of the case 7 transaction: the balance is 470 and 30 units are still reserved against a transaction that is in MANUAL_REVIEW. The design has no automatic path out of that state.

8. *Alerts go to an in-memory list.* They are never exported, paginated, or delivered, so in the artifact nobody is actually notified of a manual-review hand-off.

9. *The reconciler's repairs are unconditional.* It auto-releases an orphan freeze without checking whether the transaction it belongs to might still be in flight, which is safe only because the record store is consulted first and the record is genuinely absent.

7 Threats to validity

Construct validity. The faults are injected by flags, not by breaking the system. A shard outage is a boolean that makes one call raise an error; a node crash is a process termination with the data directory intact. The artifact therefore exercises the protocol's reaction to a fault, not the system's behaviour under a fault. No test involves concurrent writers, partial writes, clock skew, or a network that delays without dropping, and the two-layer design's interaction with concurrency is entirely untested.

Internal validity. The measurement script wraps the framework's own harness and case modules and changes nothing in them, so the control flow under measurement is the shipped control flow. The traffic counter observes only the coordinator process, so requests issued by a node's own self-check are invisible and the request counts are lower bounds. The durations are wall-clock measurements of a Python process on one machine, with no repetitions beyond the three reported and no confidence intervals; the ranges we report are the observed extremes, not statistical intervals.

External validity. The numbers in Section 6 characterise a simulation with JSON files, loopback HTTP and no contention. Nothing in this paper supports a claim about throughput, latency at scale, or behaviour on a real database engine. The design's qualitative claims about the directions of compensation and about the reachability assumptions of the two recovery layers should transfer better than the timings, because they are properties of the control flow rather than of the substrate.

Conclusion validity. We did not compare against an alternative. There is no baseline implementation of two-phase commit, TCC or SAGA under the same faults, so we cannot say whether this design recovers faster, more cheaply, or more often than those patterns would. Our claim is about the structure of the failure mode and about the demonstrable correctness of the compensation direction, not about relative performance.

8 Related work

Table 5 places the design against the four families discussed in Section 2.

On two-phase commit, our position is the standard one: the protocol is correct and well understood [Gray and Reuter, 1992, Garcia-Molina et al., 2002], and we do not use it because holding locks across a global transaction and blocking participants behind a coordinator are unacceptable at the concurrency we target, and because the protocol does not answer the question of which participants may disagree with each other when a partition splits them.

On consensus, we do not dispute the necessity of quorum-based agreement wherever a shared key is genuinely shared [Lamport, 1998, Ongaro and Ousterhout, 2014], and we note the impossibility result [Fischer et al., 1985] that bounds what any such protocol can guarantee. Our design sidesteps the question rather than solving it: no key is shared between the two business-asset databases, so there is nothing for them to disagree about. Where our design does need a replicated,

Table 5: Comparison of coordination patterns. The last column is the distinguishing question: what happens when the participant itself is unreachable.

pattern	coordination	blocking	behaviour when a participant is down
Two-phase commit [Gray and Reuter, 1992]	central coordinator, prepare and commit votes	participants hold locks until the decision; the coordinator blocks them if it fails	the coordinator recovers and decides; participants wait
TCC (try-confirm/cancel)	per-operation Try, Confirm and Cancel, usually coordinated centrally	no long locks, but the business must write three interfaces and a correct Cancel	the coordinator retries the Cancel; a wrong or missing Cancel is a silent loss
SAGA [Garcia-Molina and Salem, 1987]	central orchestration of subtransactions with compensations	no long locks, but a compensation must be authored per subtransaction	the coordinator retries the compensation, which assumes the participant returns
Reliable message / outbox [Richardson, 2018]	asynchronous delivery with retries	none, but the effect is eventually consistent and possibly duplicated	the consumer is retried; correct only if the consumer eventually accepts the message
This work	stateless coordinator, freeze before commit, phase-exact compensation, two recovery layers	no long locks; the record store is written on every step	layer one retries; layer two lets the node settle its own debt when it returns

durable record of decisions, that record is a database with its own local replication, and the same consensus machinery belongs inside it.

On TCC, our design borrows the reserve-in-Try idea and reduces the reservation to a uniform local marker. The reduction is the contribution: it removes the need for a business-specific reserve interface and a business-specific Cancel, at the price of not being able to express reservations over arbitrary business state.

On SAGA [Garcia-Molina and Salem, 1987], our design borrows the central orchestration and the compensation model, and adds what SAGA conventionally assumes away: a mechanism for the case in which the coordinator cannot deliver the compensation because the participant is down. The second recovery layer is the concrete form of that addition, and its restriction to transactions the authority has already condemned is what keeps it from being an uncontrolled actor.

On reliable messages and the transactional outbox [Richardson, 2018], our design is not a substitute: the outbox pattern is simpler, and it is the right answer in most systems. The difference is the failure model. A queue worker that is temporarily down will be retried; a database node that is down may be down for a long time and, more importantly, the compensation is not a message delivery but a mutation of a database that has to be undone correctly, exactly once, and in the right direction.

Two further references bear on the method rather than on the system. The ordering argument of Lamport [Lamport, 1978] is the background to our treatment of "happens before" between a participant's commit and the coordinator's record of it, and the treatment of time and ordering in Kleppmann [Kleppmann, 2017] is the clearest exposition of the same material. The evaluation follows the practice of fault injection described in the chaos-engineering literature [Rosenthal et al., 2017]: we inject faults where a recovery mechanism is supposed to fire, and we report what happened.

9 Limitations

We separate the limitations of the artifact from the limitations of the design, and we use the artifact's own self-critical notes, which were written before this paper existed.

9.1 The artifact is a simulation

The artifact is a teaching simulation: JSON files instead of a database engine, a loopback HTTP call instead of a network, a flag instead of a partition, an in-memory list instead of an alerting system. Its seven cases are the demonstration of a control flow, not a demonstration of a system's behaviour under load. No conclusion about performance, scale, or behaviour on real hardware can be drawn from it, and we draw none.

9.2 Six gaps between the artifact and production

The artifact's own design notes list six gaps, and we repeat them here because they are the honest difference between the code and a deployable system.

1. *The poller is a single process.* A production deployment needs several copies, and it needs leader election or a distributed lock; without one, several copies will advance the same transaction and compensate twice.

2. *The dead-letter handler is not a message queue.* It is a method call. Production needs a real queue, and must handle duplicate consumption.

3. *There is no concurrency control between the poller and the node's self-check.* In the artifact they run in sequence, so they never contend. In production they can act on the same transaction at the same time, which requires a version number or an optimistic lock, and the guard of Section 4.5 does not provide one.

4. *JSON files are not a database.* The file lock emulates a row lock, but concurrency, persistence behaviour and crash recovery would all have to be replaced by a real engine.

5. *Alerts go only to an in-memory list.* They would have to reach a real monitoring and paging system, and the manual-review hand-off would have to be owned by someone.

6. *There is no de-duplication index.* Idempotency keys are matched by scanning the whole table, which must become a unique index.

To this list we add the defect of Section 6.5: the phase write is not atomic with the participant effect, and the code contains a window in which a coordinator restart produces a compensation in the wrong direction. The artifact's drift detector finds the symptom but does not repair it. Closing this window requires a design decision – a per-operation version, a participant-side commit log that the coordinator can consult, or a two-phase reservation-and-confirm at the participant – and we have not implemented any of them.

9.3 Trade-offs the design accepts

Eventual, not serializable. The design guarantees that the coordinator's record and the participants' states converge, not that an external observer sees a serializable history. A reader that does not go through the coordinator can observe a user whose balance has been debited while the merchant's stock has not yet been deducted, or vice versa. For the scenarios this design targets that is the intended trade; for scenarios that require a single consistent view it is disqualifying.

The record store becomes a new critical centre. The design exists to remove distributed failure, and it replaces it with a single component that every transaction depends on. The store is replicated and snapshotted, and the record is small, but the concentration is real and is the price of having one authority.

Write amplification. The measurements in Section 6.4 quantify the cost: seven logical writes to the record store for a transaction that performs four participant-local transactions, each write rewriting the whole record set and each write applied to the main file and both replicas. The cost per write grows with the number of records the store holds.

The freeze TTL must be measured, not chosen. A TTL that is too short kills transactions that are merely slow, and one that is too long holds value hostage. There is no correct value; it has to be derived from the distribution of how long partitions actually last in the deployment, and the artifact's value of 30 seconds is a demonstration value chosen to make the cases quick.

Residual orphan freezes. If the record store is destroyed and no usable backup survives, the reservations it caused remain in place and invisible. The offline reconciler can detect an orphan freeze only by cross-checking markers against records; if the records are gone, it detects that the store is empty, which is a weaker signal. This is the sharpest residual risk of the design, and it is a direct consequence of choosing a single authority.

9.4 When not to use this design

The artifact's author is explicit about this, and we agree. Most business systems do not need any of this. If a single-database transaction suffices, use a single-database transaction. If cross-database traffic is low, a local message table with a queue is enough, and considerably simpler. If the business cannot tolerate an intermediate state at all, then the business needs serializability, and a compensation-based design is the wrong tool. And if the team cannot afford to operate a state machine, a compensation path, a dead-letter path, a self-check path and a reconciler – which is a real operational burden, and a stuck transaction has to be debugged across three processes – then the complexity is not worth its price. This design earns its cost only in the narrow band where a single-database transaction is insufficient and two-phase commit is unacceptable.

10 Conclusion

We argued that cross-database split-brain is better eliminated than defended against, and we built the elimination: two business-asset databases that never exchange a message, a stateless business-layer coordinator that owns every cross-database decision, a freeze-before-commit protocol with expiring local reservations, a guarded global state machine, a four-phase per-operation model that fixes the direction of every compensation, and two recovery layers, the second of which gives a recovered node the authority to settle its own debt. The protocol is executable: seven scripted fault cases all pass, and we report the measured transcript, the durations, and the coordinator's protocol traffic.

The contribution is not that the artifact is production-ready – it is plainly not, and Section 9 lists the six gaps and the one serious defect that separate it from a deployable system. The contribution is a precise, executable account of how a system can remove a failure class from its architecture, together with an honest measurement of what that removal costs: eventual rather than serializable consistency, a write amplification of roughly 1.75 record-store writes per participant-local transaction, a new critical component, a TTL that must be derived from data rather than chosen, and a residual risk of orphaned reservations if the record store is destroyed. We think the most useful thing a reader can take from this paper is the shape of the argument: if the cross-database commit can be designed out of the architecture, then the failure it used to cause does not have to be defended against at all, and the engineering effort can be spent instead on making the remaining, unavoidable failures – a node that is down, a record store that is lost – visible, bounded, and recoverable.

References

M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

H. Garcia-Molina and K. Salem. Sagas. *ACM SIGMOD Record*, 16(3):249–259, 1987.

H. Garcia-Molina, J. D. Ullman, and J. Widom. Database systems: The complete book, 2002.

J. Gray and A. Reuter. Transaction processing: Concepts and techniques, 1992.

M. Kleppmann. Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems, 2017.

L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC '14)*, pages 305–320. USENIX Association, 2014.

C. Richardson. Microservices patterns: With examples in java, 2018.

C. Rosenthal, L. Hochstein, A. Blohowiak, N. Jones, and A. Basiri. Chaos engineering: Building confidence in system behavior through experiments. Online book, 2017.

```
anti-split-brain distributed transaction framework -- three node server processes, process-level simulation
[start] spawning the txn / user / shop node server processes ...
        user -> http://127.0.0.1:6001   shop -> http://127.0.0.1:6002   txn -> http://127.0.0.1:6003
[seed] users u1=100 / u2=500 / u_low=10; merchants m1{s1:10,s2:5} / m2{s1:3}
transaction state machine transition rules:
  INIT         -> FROZEN, MANUAL_REVIEW, ROLLBACK
  FROZEN       -> MANUAL_REVIEW, ROLLBACK, SUCCESS
  ROLLBACK     -> MANUAL_REVIEW, ROLLED_BACK
  any non-terminal    -> MANUAL_REVIEW (retries exhausted, manual review)
case 1 normal order: debit user balance + merchant stock; atomic to the caller
  initial: user u1 balance=100, merchant m1/s1 stock=10
  state transitions:
    INIT           create txn a466d2aa
    INIT           -> FROZEN         a466d2aa
    FROZEN         -> SUCCESS        a466d2aa
  result: balance=70 (expect 70), stock=9 (expect 9)
  idempotency check passed: a replay with the same key returns the existing result, no double debit.
  [PASS] case1_normal
case 2 freeze failure: insufficient user balance, freeze fails midway -> ROLLBACK -> ROLLED_BACK
  initial: user u_low balance=10, merchant m1/s1 stock=9
  state transitions:
    INIT           create txn 98fea3f9
    INIT           -> ROLLBACK       98fea3f9
    ROLLBACK       -> ROLLED_BACK    98fea3f9
  result: balance=10, stock=9 (merchant freeze released, stock unchanged)
  [PASS] case2_freeze_fail
case 3 failure during commit: user balance already debited, merchant COMMIT rejected downstream -> refund balance + release stock freeze
  state transitions:
    INIT           create txn 8fcb4f19
    INIT           -> FROZEN         8fcb4f19
    FROZEN         -> ROLLBACK       8fcb4f19
    ROLLBACK       -> ROLLED_BACK    8fcb4f19
  result: balance=500 (expect refunded to 500), stock=9 (expect unchanged 9)
  [PASS] case3_commit_fail_refund
case 4 network recovery: merchant shard isolated -> stuck in INIT -> resumed by poller/dead-letter after recovery
  submit order: merchant unreachable, txn stuck in INIT (attempt 1), next retry at 1792511456.74
  merchant shard recovered; the poller starts scanning non-terminal transactions ...
  state transitions:
    INIT           create txn 353835d0
    INIT           -> FROZEN         353835d0
    FROZEN         -> SUCCESS        353835d0
  terminal state: SUCCESS (attempts=1)
  result: balance=470 (expect 470), stock=8 (expect 8)
  [PASS] case4_network_recover
case 5 node-autonomous rollback: merchant down so the rollback cannot be pushed; the restarted node self-checks and compensates
  1) both freezes succeed -> FROZEN; merchant COMMIT hits a transient failure, txn stuck in FROZEN (attempts=1)
  2) merchant process is down; forced rollback: the debited user balance is refunded, the merchant release-freeze compensation hangs
     because the node is unreachable, txn stuck in ROLLBACK (attempts=2)
  3) restart the merchant node; the startup self-check plugin queries the unfinished transactions and checks the local freezes ...
  state transitions:
    INIT           create txn 4a8b618b
    INIT           -> FROZEN         4a8b618b
    FROZEN         -> ROLLBACK       4a8b618b
  terminal state: ROLLED_BACK
  result: balance=470 (expect refunded to 470), stock=8 (expect unchanged 8)
  [PASS] case5_node_selfcheck
case 6 offline reconciliation: orphan freeze auto-released + global store loss recoverable from backup
  inject orphan freeze: user/u1 holds the freeze marker orphan_txn_0001 but the global store has no such record
  global transaction state store snapshot written: <FRAMEWORK>/data/txn/backup/snapshot_1792511461479.json
  reconciliation detected the orphan freeze -> auto release + alert.
    alert: reconciliation: released orphan freeze user/orphan_txn_0001
  orphan freeze cleaned up, u1 balance unchanged (=70).
  global transaction state store main data cleared (simulating total loss) -> reconciliation reports that a backup can restore it.
  restored from snapshot: <FRAMEWORK>/data/txn/backup/snapshot_1792511461479.json, 5 transaction records restored
  [PASS] case6_reconcile_orphan
case 7 retries exhausted: merchant permanently unreachable, exponential backoff reaches the limit -> MANUAL_REVIEW with an alert
  submit order: merchant m2 permanently unreachable, txn stuck in INIT, exponential backoff retrying ...
  terminal state: MANUAL_REVIEW (attempts=3, max=3)
  alert: [13c534bcdb564faf] retries exhausted (3), entering manual review: shop_db[shard_1.json] node unreachable/isolated
  infinite retrying stopped, moved to MANUAL_REVIEW, awaiting manual review.
  [PASS] case7_retry_exhaust
run summary
  [PASS] case 1 normal order  -  INIT->FROZEN->SUCCESS, idempotent replay passed
  [PASS] case 2 freeze-failure rollback  -  merchant freeze released, stock unchanged
  [PASS] case 3 refund after mid-commit failure  -  debited balance refunded, stock unchanged
  [PASS] case 4 resume after network recovery  -  SUCCESS after poller+dead-letter reconnect
  [PASS] case 5 node-autonomous rollback  -  self-check compensation after merchant restart, txn ROLLED_BACK
  [PASS] case 6 offline reconciliation  -  orphan freeze auto-released with alert, global store restorable from backup
  [PASS] case 7 retries exhausted -> manual review  -  backoff limit -> MANUAL_REVIEW + alert
alerts raised in this run:
  [orphan_unfreeze] reconciliation: released orphan freeze user/orphan_txn_0001
  [manual_review] [13c534bcdb564faf] retries exhausted (3), entering manual review: shop_db[shard_1.json] node unreachable/isolated
the three node processes have been shut down.
overall result: all passed
```

Figure 5: The driver's verbatim output, with the Chinese prose translated and all identifiers, state names, numbers and assertion values unchanged. The transaction identifiers and the snapshot timestamp are from one particular run and vary between runs.