Verifiable Record Integrity Without a Blockchain

Authenticated execution and independently retained evidence for a PostgreSQL transaction workflow

Viktor Khudiaiev / Publication updated September 20, 2026

Disclaimer: The views expressed in this article are my own and do not reflect those of my employer. This article describes independent personal work and is unrelated to my work for my employer.

Implementation reviewed: source snapshot `e2e35de`. This identifies the implementation snapshot, not the commit of this publication. Measurements below retain their original run dates and revisions.

The word blockchain often enters a conversation about tamper-evident records before anyone has defined the attacker. I prefer to begin with a less fashionable question: who can change the database, and what would the application do with those changes?

A database can remain available, answer queries correctly, and still contain a fraudulent payment instruction. An attacker with sufficient database privileges may change a recipient, insert an invented transfer, replay an old operation, or delete an inconvenient record. Ordinary application checks are of little help if the attacker writes beneath them.

The question I wanted to answer was practical: can we make administrative access to a transaction database insufficient to produce an unauthorized financial effect, while retaining evidence that its contents have changed?

This article describes a local reference implementation that combines established cryptographic techniques with a protected execution boundary. It uses ordinary PostgreSQL, three Java services, and simulated transfers. It does not introduce a new cryptographic primitive or claim to replace every ledger database. Its contribution is the integration: authenticate exact operation content, retain independent evidence before publication, execute only the checked operation, and recover from retries without repeating its financial effect. For this explicitly centralized trust model, that does not require distributed consensus.

Start with the attacker, not the algorithm

The attacker in this model controls the primary transaction database, including administrative privileges. They may insert, modify, delete, replay, reorder, or restore its contents and disable its triggers.

The application, key service, processor, Audit/Protected database and embedded broker, service credentials, and underlying host remain trusted. Under those conditions, control of the primary database alone is insufficient to execute a forged, modified, or already settled operation. The tests described below exercise that bounded claim; they are not a proof against every possible attack.

This distinction determines the architecture. If the attacker also controls the authoritative balances, they can change balances directly. If they control the service allowed to authenticate new instructions, they can ask it to authenticate fraudulent instructions. Adding a MAC column cannot repair either boundary.

The integrating application remains responsible for authenticating people and applying business authorization. The integrity module authenticates service requests and protects operation content; it does not introduce another identity provider or establish a human author's entitlement. Nor does it encrypt database contents, prevent database destruction, or guarantee availability.

Two databases, with distinct authority

The current demo uses two PostgreSQL instances. Primary remains attacker-controlled. Audit/Protected combines evidence and authoritative execution state, with separate runtime roles for issuance, audit and accounting. Java computes the debit and credit; protected SQL commits them atomically. The relevant property is that the primary administrator cannot modify this protected state.

Store	Contains	Authority in this model
Primary	Operation records, delivery hints, status projections	Assumed fully writable by the attacker
Audit/Protected: evidence and notification tables	Independent issuance receipts, ordered audit evidence and durable incident-notification outbox	Trusted evidence and alert-delivery state outside primary administration
Audit/Protected: accounting tables	Balances, account holds, paired postings, execution journal, durable jobs and outcome outboxes	Trusted authority for financial effects in the same protected database

The application requests authentication from the key service and publishes the resulting operation to primary. The processor verifies candidates and performs settlement. Separate service credentials restrict issuance, verification, key administration, and checkpoint signing. The application receives no key bytes or settlement SQL access; the processor cannot issue new operation MACs.

Locally, all services still run on one trusted Windows host. Different database instances and credentials demonstrate the intended access boundaries, but the common host administrator can reach every process and secret. Production separation requires independent identities, custody, and operational ownership - not merely extra schemas under the same administrator.

Authenticate bytes, not an informal record description

A hash-based message authentication code, or HMAC, combines a secret key with a message. Someone without that key cannot simply edit the message and recompute a valid authenticator. The construction is established, not invented here; this implementation uses HMAC-SHA256. RFC 2104

The difficult application question is what the message actually contains. Joining variable-length fields without boundaries is ambiguous: `("AB", "C")` and `("A", "BC")` can produce identical bytes.

Currency, correction references, and retry identity are equally dangerous to leave outside the authenticated content.

Version 1 therefore uses a byte-exact type-length-value encoding. Each field has a one-byte numeric field identifier, a one-byte type, an unsigned four-byte big-endian byte length, and its value. Fields appear in this fixed order:

`domain, schemaVersion, keyId, id, ledgerId, type, fromAccountId, toAccountId, amountMinor, currency, createdAtMicros, relatedOperationId, idempotencyKey`

UUIDs occupy 16 bytes. Integers use fixed-width, signed, big-endian encoding. Money is a positive integer in minor currency units, accompanied by its currency. Time is an exact signed 64-bit UTC Unix epoch count in microseconds. Explicit null has its own type; omitting the required relation field is rejected rather than silently equated with null.

Text is UTF-8, but this schema restricts identifiers to specified ASCII alphabets. It does not silently normalize arbitrary Unicode. Unsupported schema versions are rejected; future changes require explicit version support and preserved historical decoding rules. JSON Canonicalization Scheme is another established approach to deterministic serialization, but this protocol uses its documented binary encoding, not JCS. RFC 8785

Both the content hash and HMAC are calculated over those exact bytes. The ordinary hash is useful for comparisons; it does not replace secret-key authentication. The accompanying protocol specification and golden test vectors make the encoding independently reproducible.

A valid MAC protects the recorded timestamp against alteration. It does not establish the actual real-world event time, identify a human author, or prove that external funds arrived.

Commit independent evidence before returning a MAC

Before returning an authenticated operation, the key service commits an issuance receipt to the independent audit store. The receipt binds the exact operation and its authentication envelope. If that commit cannot complete, issuance does not succeed.

This order matters. A record and its MAC stored only in primary can disappear together. The independent receipt provides an expected operation that survives primary manipulation and exists even if the application fails before publishing its row.

Retries are bound by a unique ledger and idempotency-key pair. The business fingerprint includes the domain, schema, ledger, operation type, accounts, amount, currency, related operation, and idempotency key. Server-generated operation UUIDs and creation times are not part of that business comparison. An identical retry returns the original authenticated operation. Reusing the retry key for different business content is rejected.

The processor also scans the independent issuance inventory. It can therefore discover a missing issued operation even if it never observed a primary outbox hint. The implementation applies a five-second operation-age threshold measured from the authenticated `createdAtMicros`, which is assigned before the issuance request. This is not a guaranteed five seconds after receipt commit or a

detection-time guarantee. Periodic reconciliation and retained-history size also affect when a missing operation is observed.

An issued-but-missing record is a discrepancy, not proof of malicious deletion. A crash before publication can produce the same symptom. Nor are gaps in the inventory sequence cryptographic evidence: the sequence is a traversal cursor. PostgreSQL sequence allocation can legitimately leave gaps after failed transactions. PostgreSQL sequence documentation

Completeness therefore depends on protected expected evidence and reconciliation, not on a gapless-looking primary ID column.

Durable delivery inside the processor

The application commits the candidate record and its outbox hint together. A dispatcher in the processor polls the outbox and publishes a persistent UUID-only message to embedded ActiveMQ Classic 6.3.2. After durable broker acceptance, it acknowledges the Primary hint. The broker uses VM-only transport, with no separate network listener, and a KahaDB journal outside build artifacts. ActiveMQ VM transport, KahaDB

The consumer in the same JVM inserts a unique protected SQL job, then commits its JMS receive. Failed SQL insertion rolls back delivery. A crash after either durable write but before its acknowledgment can produce a duplicate; operation identity and protected execution state absorb it. This is at-least-once delivery with idempotent financial execution, not a global exactly-once transaction. Independent reconciliation remains necessary because Primary hints can be deleted.

The queue provides a bounded persistent buffer, not an independent availability domain. Embedded broker and processor share process and host failure. A remote broker or Kafka adapter would require a new deployment and acknowledgment/offset design; horizontal scaling is not established by adding a dependency.

Verification must survive the moment of use

“Execute if a verified record exists” is not a sufficient rule. A legitimate operation can pass verification, then have its primary amount or recipient changed before processing. That is a time-of-check-to-time-of-use, or TOCTOU, problem.

The processor instead resolves the independent receipt, matches the exact authenticated operation, checks its HMAC and ledger domain, then rereads primary immediately before settlement. The current content must still match the checked snapshot. Crucially, settlement receives the fields of that same snapshot - not amount or account fields fetched again from untrusted storage.

A later primary edit cannot change which amount and recipient that snapshot specifies. This does not imply a cross-database lock preventing all subsequent primary edits; reconciliation remains responsible for detecting later discrepancies. A primary `VERIFIED` or `COMPLETED` flag is only a projection and never grants execution authority.

Within the accounting tables of Audit/Protected DB, one SQL transaction locks the relevant accounts, checks authoritative funds and holds, records both debit and credit postings, updates balances, records

the unique execution result, and writes its durable outcome outbox. Identical retries cannot produce another financial effect. Conflicting content cannot reuse the same execution identity.

There is no claim of one ACID transaction spanning both databases and JMS. After settlement commits, durable relays deliver audit evidence and primary status projections with retries and deduplication. A crash between settlement and publication can temporarily delay those projections without authorizing a second debit.

Funding, corrections, and failure behavior

The demonstration does not bypass its own protection by directly assigning arbitrary opening balances. New accounts start at zero. Simulated funding follows the authenticated operation path and settles against a designated treasury account, with balancing postings. In a real integration, evidence that money arrived and authorization to fund an account must come from the integrating system.

Corrections preserve the original operation. For an already completed transfer, an exact compensating `REVERSAL` is executed first, followed by a separately authenticated `CORRECTION`. Both bind the original operation reference into their MAC input, preventing silent redirection of that relationship.

This is a compensating workflow, not an atomic undo of history. Reversal can fail if funds are unavailable or an account is held. Pre-settlement cancellation and supersession are not implemented. Append-only facts also do not mean every table is immutable: balances, projections, processing leases, and retry state have legitimate updates.

Unavailable verification dependencies do not become permission to skip checks. Work remains retryable; confirmed invalid content is quarantined. A dependency outage and a proven mismatch are different outcomes. Quarantine stops the suspicious operation. It does not automatically freeze the named recipient: an attacker could deliberately name an innocent account to cause denial of service. Protected account holds are controlled separately.

Report incidents without making email an execution gate

Detection must also reach an operator. When the processor appends an `INTEGRITY_INCIDENT`, the same Audit/Protected SQL transaction inserts the first notification for that operation into `notification_outbox`. The event and its initial alert are therefore committed together outside Primary administration. Repeated observations can add audit evidence without creating another initial notification for that operation. Ordinary completion or a business rejection does not generate an integrity alert.

A separately scheduled dispatcher claims pending notifications with expiring, fenced leases, sends email outside any database transaction, and records SMTP acceptance or schedules a retry. It has finite SMTP timeouts, exponential retry delay and a per-dispatcher attempt-rate cap. Email is an asynchronous reporting path, not permission to settle: a mail-service outage leaves a backlog but cannot make invalid content executable. A failure to commit protected incident evidence is a different storage failure; Audit/Protected remains required.

Messages contain only an event ID, operation ID, audit sequence, recorded time and a fixed allow-listed reason or generic review instruction. Financial payloads, account identifiers, amounts,

MACs and keys stay out of email. Recipients are BCCed and come only from trusted configuration, not the Primary record. An internal operation ID is still information that belongs in an approved mailbox, not a public issue tracker.

Delivery attempts are at least once. If SMTP accepts a message and the processor stops before recording that result, a retry can duplicate it; a stable event reference supports correlation. The stored delivered marker means SMTP acceptance, not arrival in an inbox or acknowledgement by a person. Deduplication is per operation, so a later distinct discrepancy on that operation may add evidence without another initial email. Historic incidents are not automatically backfilled.

The local demonstration uses a loopback-only Mailpit capture inbox with no forwarding or relay. Even an alert addressed to a public email address remains local; the tests did not deliver to Gmail or another external mailbox. Real SMTP requires separately supplied credentials, authenticated TLS and approved sender/recipient configuration. An alert discovered after settlement does not reverse or invalidate the earlier correctly authenticated effect, and a missing row is not proof of fraud. See the notification design and operational limits.

A second layer for audit history

HMAC protects individual operation content internally. Ordered audit events - including settlement outcomes and account-control evidence - use a separate history mechanism: a Merkle tree and Ed25519-signed checkpoints. These events are not each represented as another operation HMAC, and not every internal job transition becomes an immutable audit event.

A Merkle tree combines event hashes into a root committing to a particular ordered history. An inclusion proof connects an event to that root. The separately signed checkpoint binds log identity, tree size, root, signing-key identity, version context, and recorded time.

The design draws on established verifiable-log techniques. Certificate Transparency's RFC 9162, which supersedes RFC 6962, specifies Merkle inclusion and consistency proofs. This demo is not a Certificate Transparency implementation, and using a tree does not eliminate ordering or operational trust requirements. RFC 9162

The current audit verifier reconstructs the relevant tree or anchored prefix. Compact consistency-proof APIs and incremental large-scale tree maintenance remain future work. The key service signs roots submitted by its authorized audit client; it does not independently rebuild that client's tree or prove consistency between increasing-size roots.

Every signed checkpoint is retained in a separate local anchor file before being returned. These files are outside the audit database, so a database-only rewrite cannot also replace the retained anchor. They remain on the same trusted host, however: they are neither write-once storage nor an independent public witness.

A verifier must obtain the expected public key through a trusted channel and retain appropriate checkpoint evidence. Trusting a replacement key merely because it arrives beside a signature defeats that assurance. A valid old checkpoint also does not establish that the presented history is the latest history. Production freshness requires independently retained, sufficiently recent evidence.

Key custody defines the strength of the boundary

Anyone possessing an HMAC key can generate valid MACs. A client limited to a verification-only API need not possess that key and can be denied generation permission, as the processor is here. This distinction improves service separation without turning HMAC into a publicly verifiable signature.

The local key service uses operating-system-protected software key files. HMAC rotation is implemented: new issuances use the active key, while historical records retain exact key identifiers and remain verifiable with retained old keys. That is an API and custody policy, not a hardware prohibition against misuse by someone who acquires the underlying secret.

Ed25519 provides asymmetric checkpoint signatures: verification uses a public key rather than the signing secret. This makes checkpoint verification possible outside the writer, given trusted key distribution. It does not independently establish truthful business events, human authorship, legal non-repudiation, or trusted time. No external timestamp authority is implemented.

Measured workload: the latest strict throughput check failed

The latest full load run is September 7, 2026, `2026-09-07T01-27-24-67e1f9c6`, before the notification revision. It used Main and Audit/Protected PostgreSQL, three Java services and persistent embedded ActiveMQ on the same Windows host. At 20 offered transfers per second for 120 seconds, all 2,400 distinct operations completed with zero transaction failures; protected accounting and audit checks passed. All 92 Java tests and 15 PostgreSQL scenarios passed, and normal mode was restored.

Nevertheless, the strict steady-window completion criterion **failed**: **19.963636363636365 TPS** was below the required 20 TPS with zero configured tolerance. Whole-run completion including warmup and drain was 19.940251926438954 TPS; end-to-end p95 was 668.2416 ms and p99 was 791.458 ms. The load stage and combined report are failures, not rounded passes. This is a throughput acceptance failure, not an observed unauthorized or duplicate financial effect. See the September 7 verification evidence.

Earlier successful runs remain useful historical evidence, but do not replace that latest result. The first September 6 run, `2026-09-06T23-27-43-c027aafb`, passed 83 Java tests, 15 PostgreSQL scenarios and all 2,400 transfers. Steady completion was 20.3091 TPS, whole-run completion 19.9371 TPS, drain 378 ms, p95 3,990 ms and p99 4,599 ms. The final September 6 run, `2026-09-06T23-48-18-1b9a3dcc`, passed 86 Java tests, 15 scenarios and all 2,400 transfers at 20.0000 steady TPS and 19.9173 whole-run TPS; drain was 498 ms, p95 3,057 ms and p99 3,993 ms. These counts belong to their individual runs, not the current source revision.

The finite steady completion window can include work submitted during warmup, so a rate slightly above 20 does not imply more than 20 offered TPS. Different retained-history sizes and configurations also make these runs unsuitable as a controlled A/B comparison. Embedded messaging is a durability/buffering choice, not a demonstrated performance improvement. Separate dated September 6 recovery and Live Lab checks retain their narrower scope.

Notification-revision verification

On September 8, the notification revision passed 137 Java tests in 19 suites, with zero failures, errors or skips, plus 44 Node checks and 18 PostgreSQL role-isolation checks. The automatic coverage includes atomic incident/outbox rollback, repeated and concurrent incident deduplication, lease fencing, retry/restart behavior, safe message content, SMTP configuration and status-endpoint authorization. These are functional checks, not a new throughput benchmark.

The PostgreSQL-to-Mailpit integration run, `2026-09-08T15-49-53.807Z-28014d6c`, passed four isolated fixture cases: a valid transfer generated no incident alert; a fabricated Primary row was quarantined without a financial effect and generated one captured email; a repeated observation retained the same notification and one captured message during a five-second observation; and changing a completed operation's Primary MAC produced an alert while its original protected result, balances and postings remained unchanged.

The duplicate-observation case does not remove the SMTP acceptance/crash window. This run established local capture only: it did not test external inbox delivery, a real provider's TLS handshake, a mail-service outage or host power loss. No new load benchmark was run for the notification revision. The latest recorded strict 20 TPS result therefore remains failed, and notification throughput and production availability remain unestablished.

Historical baseline and its limits

The historical September 5 acceptance run, before the two-database/ActiveMQ revision, used three Java services and three PostgreSQL instances on one Windows/Docker host, with software keys and simulated money. The machine had an Intel Core i7-13620H, 16 logical processors, and approximately 16 GB RAM.

Measurement	Recorded result
Automated Java verification	75 tests in 10 suites; zero failures, errors, or skips
PostgreSQL functional and attack scenarios	15 of 15 passed
Offered load	20 transfers per second for 120 seconds across 8 accounts
Completed operations	2,400 distinct operations; zero failures
Completion rate, seconds 10-120	20.0091 transactions per second
Whole run, including warmup and drain	19.9181 transactions per second
End-to-end p95 latency	709 ms

Completion required the protected terminal result, the exact intended payload and paired postings, and delivery of the durable audit outcome - not merely an HTTP success response. The harness also checked uniqueness, authoritative balances, and bounded backlog. The post-warmup completion

window can slightly exceed the offered rate because work crosses measurement boundaries; it must not be confused with whole-run throughput.

The attack scenarios included fabricated records, tampering after verification, missing issued records, deletion before execution, replay, and rollback after an injected debit-stage failure. Two additional real JVM outage experiments tested recovery. With the processor stopped, a request returned HTTP 503 after durable source publication; restart drained the operation, and an identical retry caused no second effect. With the key service stopped, new issuance returned 503 without a receipt, source row, or balance change; retry after restart settled once.

These results are retained in the accompanying verification evidence and recovery evidence. They demonstrate this workload, not maximum capacity, comparative cost, a ten-minute soak, a million-record history, or a production service-level commitment. Database outages, host power loss, point-in-time recovery, and key-service failure during already verified in-flight settlement were not covered by these outage experiments.

Related approaches and the production boundary

Blockchain is not the only comparison. Database-native ledger features also address tamper evidence. Microsoft's Ledger is available in SQL Server 2022 and later, Azure SQL Database, and Azure SQL Managed Instance; it combines protected history with externally retained database digests. It is therefore inaccurate to describe this category as requiring Azure alone. Microsoft Ledger overview

Verifiable logs address historical commitments; traditional audit logs, access controls, and backups address other parts of the problem. This implementation focuses on connecting independently authenticated content to protected execution on a concrete PostgreSQL path. It is not a transparent adapter already validated for arbitrary databases or MySQL. No comparative benchmark establishes that it is cheaper or faster than a ledger database or distributed system.

Before production deployment, software custody must be replaced or strengthened with a reviewed key-management-service or hardware-security-module integration, independent service identities and deployment ownership, encrypted authenticated transport, protected key distribution, and independently retained checkpoints. Recovery needs backups, point-in-time recovery exercises, retention controls, and operational playbooks. Notification delivery additionally needs an approved provider and sender identity, protected routing, backlog/age and bounce monitoring, escalation policies and handling of distinct-operation floods. Scalable history verification, broader failure testing, dependency modernization, vulnerability review, and external security assessment remain necessary.

The useful result is not “a database that cannot be changed.” It is a more precise separation: primary data may be changed, but those changes do not automatically become executable instructions, and independent evidence makes discrepancies inspectable. Established cryptography supplies the building blocks. The engineering task is to preserve their meaning through publication, execution, retries, correction, and audit.

Current topology, dated acceptance results and deployment limits are tracked in the architecture reference. Historical measurements remain unchanged and must not be relabeled as benchmarks of a later revision. Neither these tests nor this architecture assert regulatory compliance or legal non-repudiation.