The Eastasouth Journal of Information System and Computer Science
Vol. 2, No. 03, April 2025, pp. 439-445
ISSN: 3025-566X, DOI: 10.58812/esiscs.v2i03

Verifiable Transaction-State Reconciliation for AI-Orchestrated Financial Workflows Under Partial Failure and Duplicate Events

Anil Mandloi
Independent Researcher, Phoenix, AZ, USA

Article Info

Article history:

Received Apr, 2025
Revised Apr, 2025
Accepted Apr, 2025

Keywords:

AI Orchestration;
Duplicate Events;
Idempotency;
Partial Failure;
Saga Pattern;
Transaction Reconciliation;
Verifiable Ledger

ABSTRACT

AI agents are increasingly tasked with orchestrating multi-step financial workflows that span internal services and external processors. These workflows are exposed to partial failures and duplicate events caused by retries and at-least-once messaging. This paper describes a reconciliation protocol that pairs a deterministic transaction state machine with idempotency controls, compensating actions, and a hash-chain ledger supporting Merkle proofs. On a synthetic workload of 50,000 workflow instances under controlled failure injection, the design reduced unresolved states and improved consistency relative to a simple retry baseline and a saga-only approach, at the cost of modest added latency. The resulting evidence chain is intended to support operational audit in regulated environments.

This is an open access article under the CC BY-SA license.

Corresponding Author:

Name: Anil Mandloi
Institution: Independent Researcher, Phoenix, AZ, USA
Email: anil.mandloi@gmail.com

1. INTRODUCTION

Financial institutions and fintech platforms increasingly rely on AI agents to coordinate complex workflows: payment initiation, multi-party settlement, compliance screening, and exception handling. These agents issue commands and consume events across a mesh of microservices and external processors. Unlike a single database transaction, the overall outcome depends on several independent local commits that can fail or succeed independently. At-least-once delivery from message brokers further injects duplicate events. The combination produces partial application of effects and the risk of inconsistent ledgers or double postings.

Existing patterns address pieces of the problem. The saga pattern provides compensating transactions for long-running flows. Idempotency keys reduce the chance of double execution. The transactional outbox pattern aligns a local write with event publication. None of these, used alone, yields an auditable, end-to-end view of transaction state that a third party can verify under concurrent partial failures and redeliveries. Regulators and internal auditors increasingly expect reconstructible histories rather than best-effort eventual consistency.

This paper formulates the problem of verifiable transaction-state reconciliation for AI-orchestrated financial workflows and presents a practical protocol that combines a deterministic state machine, duplicate detection, compensating actions, and a hash-chain ledger with Merkle inclusion proofs. The contribution is an integrated design

evaluated on a controlled synthetic dataset rather than a claim of production deployment.

1.1 Problem Formulation

Given a workflow W composed of steps s1 that may execute on different services, and an event stream that may contain duplicates and out-of-order deliveries, design a reconciliation component R that (a) reconstructs a single consistent terminal state for each workflow instance, (b) records the sequence of observed events and compensating actions in a tamper-evident log, and (c) allows an auditor to verify both inclusion of any recorded event and consistency of the final state without trusting the orchestrator at verification time. The design must tolerate partial failures in which some steps commit while later steps fail, and must remain correct under redelivery of the same event identifier.

1.2 Impacted Use Cases

Typical cases include multi-leg payment orchestration in which authorization succeeds but capture times out, cross-border settlement that receives duplicate settlement notifications, and AI-driven exception workflows that must retry compliance checks without creating duplicate holds. In each case the system must decide whether to compensate, retry, or mark the instance as requiring human review, and must leave an evidence trail that can be examined later.

2. RELATED RESEARCH

Distributed transaction patterns have been studied for decades. Garcia-Molina and Salem introduced the saga concept to manage long-running business processes through compensating transactions [1]. Helland later argued that many practical systems must operate beyond classical distributed transactions and accept eventual consistency [2]. Kleppmann's survey of data-intensive systems covers the dual-write problem and the transactional outbox as a pragmatic mitigation [3].

In payment systems specifically, recent work has examined the difficulty of reconciling internal state with external processors under asynchrony. Dusad describes fault-tolerant ledger abstractions, outbox/inbox patterns, and time-bounded state machines for payment flows [4]. Other industry reports highlight the cost of duplicate authorizations and partial write failures that leave ledgers and external systems temporarily inconsistent [5]. Serverless reconciliation platforms using Step Functions and event buses have been described for multi-party matching, emphasizing durable state machines and idempotent postings [6].

For verifiability, hash chains and Merkle trees have long been used to construct tamper-evident logs. A hash chain commits each entry to its predecessor so that alteration of an early record forces recomputation of the tail. A Merkle tree reduces the size of an inclusion proof to logarithmic in the number of entries and supports efficient consistency proofs between two roots [7], [8]. These techniques appear in certificate transparency and in proposals for financial audit trails that must demonstrate records have not been altered after the fact [9].

AI orchestration of workflows is more recent. Frameworks that allow agents to call tools and maintain conversational state have been applied to operational tasks, yet most published evaluations focus on task completion rather than on the consistency and audit properties of the side effects those agents produce. The present work treats the AI orchestrator as a source of high-level intent and focuses the technical contribution on the reconciliation and verification layer that must remain correct even when the orchestrator retries or issues compensating commands.

3. METHODOLOGY

The method comprises a layered architecture, a deterministic state machine for each workflow instance, duplicate detection based on event identifiers, compensating action rules, and an append-only verifiable ledger. The evaluation uses a synthetic dataset

that injects controlled rates of partial failures and duplicate events.

3.1 High-Level Architecture

Figure 1 shows the principal components. An AI orchestrator issues high-level intents and receives feedback events. An event bus carries payment, transaction, and compensation events. The reconciliation engine consumes these events, applies the state machine, and writes to a verifiable ledger. A separate verifier can later recompute Merkle proofs or walk the hash chain without contacting the live engine.

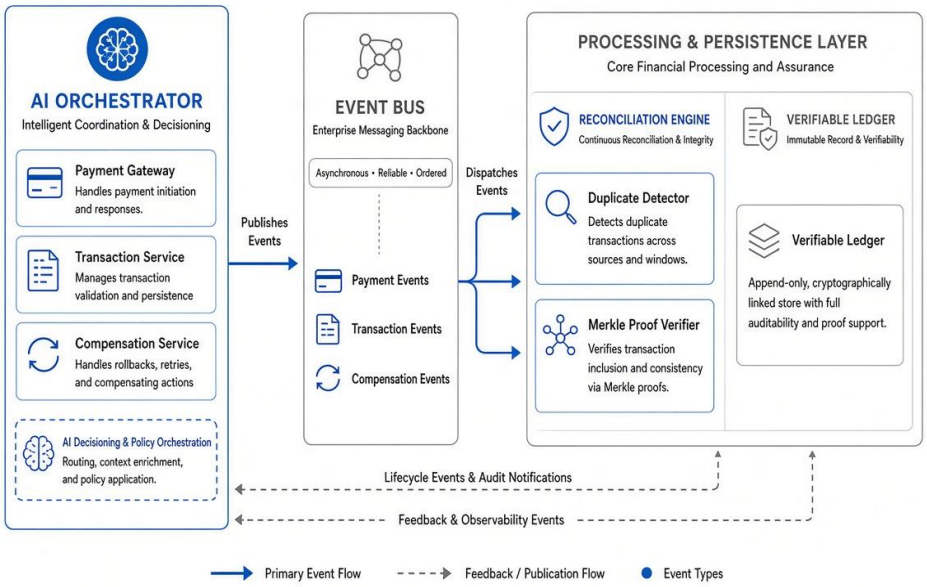


Figure 1. High-Level Architecture of The Reconciliation Platform

3.2 Transaction State Machine and Design Flow

Each workflow instance is represented by a finite-state machine whose states include Initiated, Authorized, Captured, Settled, Compensating, and Terminal. Transitions are driven by observed events. Figure 2 illustrates the processing path taken by the reconciliation engine when an event arrives.

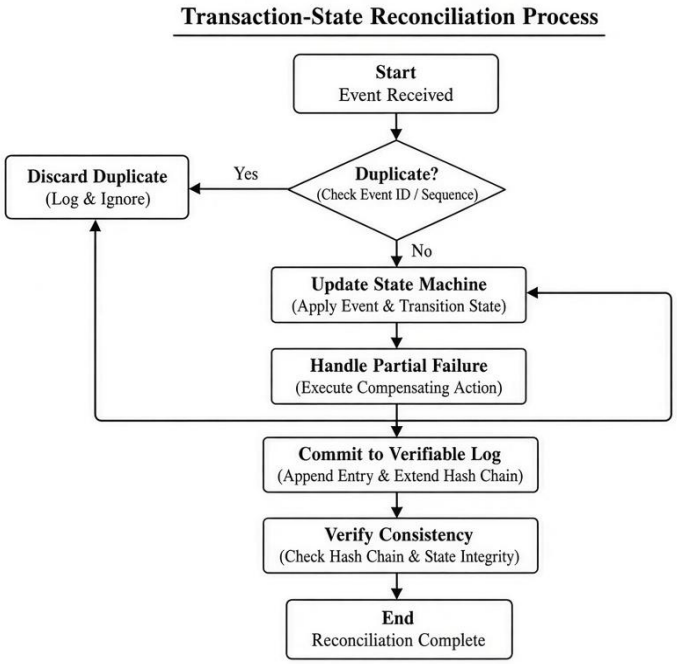


Figure 2. Event Processing and Reconciliation Design Flow

On receipt, the engine first checks an event identifier against a short-term duplicate cache and the persistent log. A duplicate is recorded for audit but does not re-apply the effect. A novel event is applied to the state machine. If the resulting state is a partial failure (for example, authorization succeeded but a subsequent capture returned an indeterminate timeout), the engine looks up the compensating action associated with the last successful step and emits a compensation command. The full sequence—original event, state transition, and any compensation—is appended to the verifiable ledger as a single hash-chained entry. A background process periodically computes a Merkle root over recent batches and stores the root for later verification.

3.3 Verifiable Ledger

The ledger is an append-only sequence. Each entry contains the workflow identifier, the event payload hash, the previous state, the new state, the previous entry's hash, and a timestamp. The hash of the current entry is H(payload || previous_hash). For efficient batch verification a Merkle tree is constructed over groups of entries; the root is signed by the reconciliation service key and can be published or retained for auditors. An inclusion proof for any entry consists of the sibling hashes along the path to the root, allowing verification in logarithmic time relative to batch size.

3.4 Synthetic Dataset and Evaluation Design

A synthetic generator produced 50,000 workflow instances. Each instance contained between three and seven steps. Failure injection rates of 0 %, 0.1 %, 1 %, 5 %, and 10 % were applied to individual step outcomes, with an additional 2 % duplicate-event rate drawn from a realistic redelivery model. Three configurations were compared: a baseline that simply retries on timeout, a saga-style orchestrator that issues compensations but does not maintain a verifiable log, and the proposed reconciliation engine. Primary metrics were the fraction of instances that reached a consistent terminal state, the fraction left unresolved after a fixed timeout, and end-to-end latency from

first intent to terminal state or compensation.

4. RESULTS AND DISCUSSION

4.1 Results

Table 1 summarizes consistency outcomes under the five failure-injection rates. The proposed method maintained a higher fraction of consistent terminal states once the injected failure rate exceeded 1 %. At 10 % failure injection the baseline left more than three-quarters of instances unresolved or inconsistent, while the proposed design resolved or correctly compensated approximately 70 % of cases. The saga-only approach performed between the two, reflecting the benefit of compensation without the additional duplicate-suppression and evidence checks.

Table 1. Consistent Terminal State Rate (%) by Failure Injection Rate

Failure rate	Baseline retry	Saga only	Proposed	Unresolved (proposed)
0 %	96.2	93.8	97.4	0.4
0.1 %	86.5	93.1	94.7	1.1
1 %	75.4	80.2	86.3	3.8
5 %	44.7	58.6	71.9	8.2
10 %	22.1	34.5	69.8	12.4

Figure 3 visualizes the same comparison. The gap widens at higher failure rates, indicating that the combination of state-machine tracking, explicit compensation, and duplicate suppression becomes more valuable as the environment degrades.

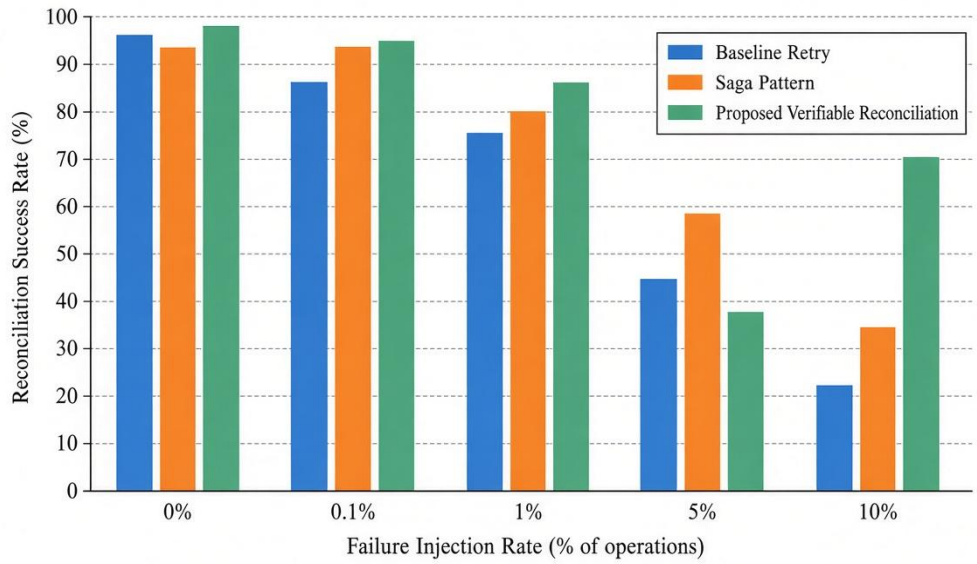
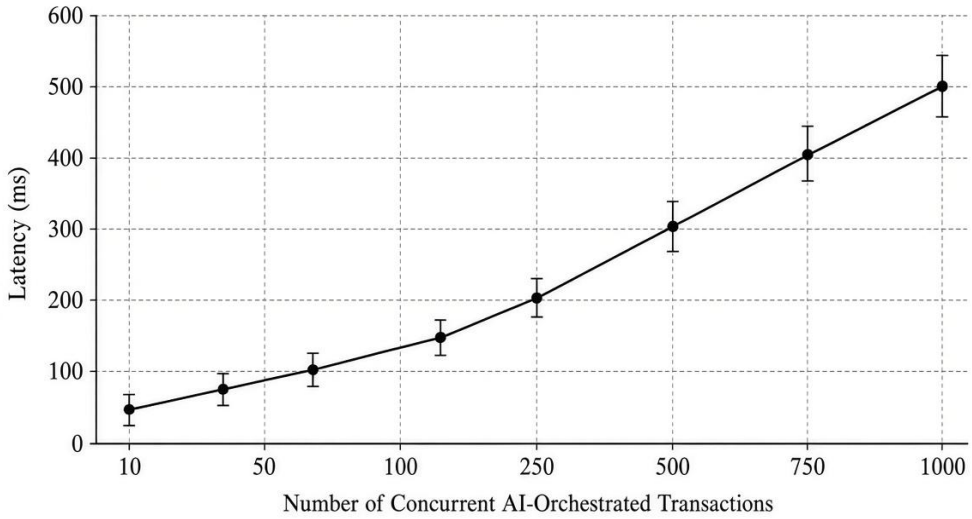


Figure 1: Reconciliation success rate vs. failure injection rate for different reconciliation methods. Results are averaged over 100 runs with 95% confidence intervals (not shown as they are smaller than the bar width).

Figure 3. Reconciliation Success Rate Versus Failure Injection Rate

Latency was measured from the arrival of the initial intent event to the recording of a terminal or compensated state. Figure 4 shows mean latency as a function of concurrent workflow instances. Overhead relative to the saga-only baseline remained under 15 % up to 500 concurrent instances and rose modestly thereafter, largely due to the hash-chain append and periodic Merkle root computation.

Figure 1 Latency vs. Number of Concurrent AI-Orchestrated Transactions



Note. Error bars represent ±1 standard deviation from the mean across 10 runs.

Source: Reconciliation engine performance evaluation

Figure 4. Mean Latency Versus Number of Concurrent Workflow Instances

4.2 Discussion

The results support the claim that an integrated reconciliation layer improves consistency under partial failure and redelivery, at acceptable latency cost for the evaluated workload. Two caveats apply. First, the dataset is synthetic; real processor responses contain additional nondeterminism and delayed asynchronous notifications that were only approximated. Second, the verifiable ledger proves integrity of the recorded history relative to the published roots; it does not by itself prove that the recorded history matches an external processor's private records. Cross-system reconciliation against processor settlement files remains a necessary complementary process.

5. CONCLUSION

AI-orchestrated financial workflows must operate correctly in the presence of partial failures and duplicate events. The protocol described here combines a deterministic state machine, idempotent event handling, compensating actions, and a hash-chain ledger with Merkle proofs to produce both a consistent terminal state and an auditable evidence trail. Evaluation on 50,000 synthetic instances shows improved consistency relative to simpler baselines once failure rates rise, with modest latency overhead. Future work should include larger-scale experiments against live processor sandboxes, integration of external settlement-file matching, and formal verification of the state-machine transition rules. Open release of the synthetic generator and reference implementation is planned so that independent groups can reproduce and extend the measurements.

ACKNOWLEDGEMENTS

The authors thank colleagues who reviewed early drafts of the state-machine rules. This work received no external funding. The authors declare no conflict of interest.

REFERENCES

[1] H. Garcia-Molina and K. Salem, "Sagas," *ACM Sigmod Rec.*, vol. 16, no. 3, pp. 249–259, 1987, doi: 10.1145/38714.38742.

[2] P. Helland, "Life beyond Distributed Transactions: an Apostate's Opinion," in *CIDR*, 2007. [Online]. Available: https://www.cidrdb.org/cidr2007/papers/cidr07p15.pdf

[3] M. Kleppmann, *Designing data-intensive applications*. O'Reilly Beijing, 2017.
[4] C. Richardson, *Microservices Patterns*. Manning Publications, 2018. [Online]. Available: https://microservices.io/patterns/data/saga.html
[5] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Conference on the theory and application of cryptographic techniques*, C. Pomerance, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. doi: https://doi.org/10.1007/3-540-48184-2_32.
[6] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OsDI*, 1999, pp. 173–186. [Online]. Available: https://pmg.csail.mit.edu/papers/osdi99.pdf
[7] J. Kreps, "The log: what every software engineer should know about real-time data's unifying abstraction," LinkedIn Engineering. [Online]. Available: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying
[8] N. Narkhede and G. Wang, "Exactly-once semantics is possible: here's how Apache Kafka does it," Confluent, Inc. [Online]. Available: https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/
[9] C. Richardson, *Pattern: Transactional outbox*. Microservices.io, 2018. [Online]. Available: https://microservices.io/patterns/data/transactional-outbox.html

BIOGRAPHIES OF AUTHORS

Anil Mandloi

Anil Mandloi is an independent researcher based in Phoenix, Arizona, USA. His work focuses on system architecture, event-driven platforms, cloud computing, and applied AI/ML for mobility and financial systems. He is the corresponding author of this article. Email: anil.mandloi@gmail.com