Durable Edge-Outbox Synchronization: Managing Relational Divergence in Intermittent Mobile Enterprise Workloads

Muhammad Usman Khan
CODCrafters, Quantic MSSE, PUCIT
ORCID: 0009-0009-4038-5762
Email: iusmanghias@gmail.com
Portfolio: https://usmanghias.co.uk

***Abstract*—Mobile enterprise applications require multi-table transactional durability and a reviewable audit trail under intermittent or absent connectivity, a combination that neither last-write-wins replication nor Conflict-Free Replicated Data Types (CRDTs) delivers at acceptable cost on resource-constrained devices. This paper describes and formalizes the synchronization architecture of CLIVORA, a production Flutter application (240 Dart source files, 90,868 lines) whose edge persistence layer is a Drift/SQLite database at schema version 18.**

The architecture comprises three core mechanisms. First, a durable transactional outbox (`sync_outbox_entries`) assigns each mutation a UUIDv4 idempotency key (`operation_id`) scoped to an owning user (`owner_user_id`), advancing it through an explicit six-state automaton (`pending`, `processing`, `retrying`, `conflicted`, `permanently_failed`, `completed`) with exponential backoff capped at eight attempts. Second, cloud reconciliation runs over a per-user WebSocket channel (`clivora:sync:$uid`) against PostgreSQL under row-level security, with a 500 ms sliding debounce that collapses change event bursts into a single reconciliation pass. Third, divergence is never resolved silently: when local and cloud states disagree, the system stages the divergence in two tables (`sync_conflict_entries` on-device, `crm_sync_conflicts` in PostgreSQL) and exposes two deterministic resolution operators, *take-cloud* and *keep-local*. Crucially, *keep-local* re-enqueues local state as a fresh outbox operation rather than mutating cloud state directly, preserving a single path of transactional authorization.

We pair this architectural formalization with a seeded microbenchmark of a reference model of the outbox mechanics rather than of the shipped Flutter client, measuring serialization, transactional SQLite WAL ingestion, and network partition recovery. Across ten trials of 1,000 enterprise operations, ingestion through durable commit averaged 25.135 ms (SD 2.846 ms, p99 31.327 ms) on the virtualized host described in Section VI, and total storage contracted to 9.1% of peak following post-sync tombstone sweeps.

***Index Terms*—offline-first, mobile databases, SQLite, transactional outbox, divergence management, relational integrity, edge computing**

I. Introduction

A freelancer issuing an invoice from a client site, a field technician closing a work order in a basement, and an auditor annotating a project record on a flight all share a fundamental requirement: the write must succeed locally, survive process termination, and later reach the server exactly once, with any disagreement made visible rather than silently discarded. This is the operating condition of mobile enterprise software, and it is fundamentally distinct from collaborative text editing.

The dominant industrial answer is last-write-wins (LWW). It is inexpensive and opaque: the loser of a timestamp comparison is destroyed without record, which is tolerable for a cursor position but unacceptable for an audited financial ledger or milestone transition. The dominant academic answer is the Conflict-Free Replicated Data Type (CRDT) family [1], [2], which guarantees convergence without coordination but pays for it in per-element causal metadata and in an object model (registers, sets, sequences) that does not natively express the multi-table referential constraints of a relational schema.

Between these sits an operational requirement that neither satisfies. An enterprise record system requires: (i) durability of mutations across application restarts with exactly-once application at the server, (ii) integrity across multiple tables joined by foreign keys, and (iii) a retained, inspectable record of every mutation that failed or diverged, because contractual and accounting records are subject to regulatory audit. Convergence alone is insufficient: a converged state that silently dropped a human operator's edit is invalid from a business perspective. What is needed is not automatic conflict elimination, but disciplined conflict retention and deliberate resolution.

This paper describes an architecture built to that requirement and operating in production within CLIVORA. The contributions of this work are:

1) **Formal Model of Edge-Outbox Synchronization (Section III)**: We formalize the local state, the ordered operation queue, and the cloud state as distinct mathematical objects; define an explicit six-state automaton; formulate a divergence predicate; and establish two deterministic resolution operators whose structural asymmetry represents the central architectural claim.
2) **Audited System Implementation (Section IV)**: We trace every synchronization mechanism to audited source files, table definitions, and migrations within the production codebase, documenting corrections to internal design assumptions (Section V).

3) **Empirical Microbenchmarking (Section VI)**: We construct a reproducible, standalone benchmark modeling the outbox state machine, quantifying serialization throughput, WAL storage overhead, and recovery across simulated network packet drop rates.
4) **Adversarial System Assessment (Sections VII and VIII)**: We evaluate the concurrency limits of SQLite, identify a latent backoff horizon ceiling, and systematically answer three core reviewer challenges.

II. Related Work and the Unresolved Gap

The gap this architecture occupies is not in convergence theory, which is well established, but in the cost and observability of convergence when the replicated object is a multi-table relational record under regulatory audit.

A. Conflict-Free Replicated Data Types (CRDTs)

Shapiro et al. [1] established the two CRDT families, state-based and operation-based, proving that join-semilattice merges yield strong eventual consistency without coordination. Kleppmann et al. [2], [3] extended this to tree and JSON structures. While theoretically elegant, three characteristics limit their direct application to mobile enterprise systems:

1) Metadata Cost: Convergence without coordination requires that replicas retain causal ordering history (vector clocks, Lamport timestamps [4], or compacted operation logs). This metadata is proportional to edit history rather than current state. On resource-constrained mobile hardware with limited battery and storage bandwidth, tombstone retention and causal log expansion impose non-trivial overhead.

2) Object-Model Mismatch: CRDTs converge per object. A business mutation such as "create customer C, attach project P, and issue invoice I" is a multi-row transaction whose correctness is a referential constraint spanning multiple tables. Composing independently converging per-row CRDTs does not preserve relational integrity: each row may converge while referential constraints are violated (e.g., an invoice converging into existence against a customer row concurrently deleted). Enforcing cross-object invariants over CRDTs requires external transactional coordination [5], reintroducing the coordination CRDTs were selected to avoid.

3) Ill-Suited Success Criteria for Audits: A CRDT merge is deterministic and total: it always produces an output. When two users concurrently update an invoice status to conflicting values, a lattice merge yields an arbitrary winner or a multi-value register. Neither outcome provides business guidance on which decision was authorized. For an audited system, conflicting human intent must be preserved, attributed, and resolved by an authorized actor. Automatic merging in this context represents a loss of business information.

B. Mobile and Offline Synchronization Frameworks

Table I contrasts CLIVORA with representative industrial and research synchronization models.

The prevailing industrial pattern either delegates conflict handling to application endpoints or resolves conflicts silently on the server. The transactional outbox pattern [11], [12] is well established on server architectures to ensure atomic message publication [13], [14]. Its adaptation to the mobile edge, where queues must survive device reboots, operate disconnected for prolonged periods, and serve as audit records, represents an under-explored point in the design space.

C. Disconnected Operation and Conflict Preservation

The closest prior art predates the CRDT literature. Coda [15] established disconnected operation for a distributed file system, logging mutations locally during disconnection and reintegrating them on reconnection; where reintegration could not be resolved automatically, Coda preserved the divergence in a covolume for manual repair rather than discarding it. Bayou [16] went further for weakly connected replicated storage, attaching to each write a dependency check and an application-specific merge procedure, with an explicit escape to conflict preservation when the merge procedure could not produce a safe result.

The architecture presented here is recognizably in this lineage, and the correspondence is precise enough to state as such. Bayou's per-write dependency check corresponds to our divergence predicate $D(e)$; its merge procedures correspond to the two resolution operators; and its conflict preservation corresponds to our dual-table staging. Two differences are worth isolating. First, Bayou's merge procedures execute on the server as part of write application, whereas our operators execute on the client and, in the case of $\mathcal{R}_{\text{local}}$, produce a new queue entry rather than a resolved write, which keeps backend authorization as the single enforcement point. Second, Coda and Bayou targeted file and tuple stores on workstations with persistent connectivity to a campus network; the constraints here are a mobile device, a hosted relational backend enforcing row-level security, and an audit obligation on the retained divergence record.

We therefore do not claim disconnected operation or conflict preservation as novel. The contribution is narrower and is stated as such in Section VIII: the modeling of human conflict resolution as a queue transformation subject to the same authorization pipeline as machine-generated mutations.

D. The Architectural Gap, Stated Formally

Existing systems provide either convergence without multi-table relational transactionality (CRDTs) or relational transactionality without observable divergence (mobile sync frameworks). No widely deployed architecture simultaneously guarantees:

- Multi-table transactional durability surviving mobile process termination under complete disconnection;
- Exactly-once server application via client-minted idempotency keys rather than heuristic deduplication;
- Non-destructive, inspectable records of both failed operations and unresolved divergence for audit compliance;
- Storage overhead proportional only to pending and exceptional operations, decaying toward baseline in the steady state.

TABLE I
COMPARISON OF MOBILE AND OFFLINE SYNCHRONIZATION FRAMEWORKS

System	Replication Model	Conflict Policy	Divergence Visibility	Relational Integrity
Couchbase Lite [6]	Document revision trees	Custom conflict resolvers	Optional (custom code)	Document-level only
WatermelonDB [7]	Pull/push sync primitive	Delegated to backend	No visible conflict UI	Delegated to endpoint
PowerSync [8]	Streaming Postgres replication	Server-authoritative (LWW)	No (server-resolved)	Enforced by PostgreSQL
CRDTs (Automerge, Yjs) [9], [10]	Causal state/op merge	Automatic lattice join	No (convergent by design)	Per-object only
CLIVORA (This Work)	**Durable Edge Outbox**	**Asymmetric Operators + Triage**	**Explicit (Dual-Table Staging)**	**Relational (SQLite + Postgres)**

III. FORMAL SYSTEM MODEL AND STATE TRANSITIONS

A. Objects and Definitions

Let $\mathcal{U}$ be the set of users. For an active user $u \in \mathcal{U}$, the system is modeled by three state objects:

1) **Local Edge State** (S_L): The mobile relational state, a finite map from $(\text{table}, \text{primary_key})$ pairs to tuples, materialized in SQLite via Drift schema version 18.
2) **Cloud State** (S_C): The server-resident relational state in PostgreSQL, restricted by Row-Level Security (RLS) to the tuple projection observable by user u.
3) **Outbox Queue** (Q_u): An ordered sequence of mutations:

$$Q_u = \langle op_1, op_2, \ldots, op_n \rangle, \tag{1}$$

where each operation is defined as the 8-tuple:

$$op = (id, owner, kind, \pi, \sigma, r, \tau, \theta), \tag{2}$$

with $id \in$ UUIDv4 (globally unique idempotency key minted on the client), $owner = u$, $kind \in \mathcal{K}$ (recognized entity type), π (JSON payload), σ (automaton state), $r \in \mathbb{N}_0$ (retry attempt counter), $\tau \in$ Timestamp (earliest eligible execution time), and $\theta \in \{0, 1\}$ (tombstone flag for logical deletes).

B. The Six-State Operation Automaton

The lifecycle state σ ranges over the finite set:

$$\Sigma = \{\texttt{pending}, \texttt{processing}, \texttt{retrying}, \texttt{conflicted}, \texttt{permanently_failed}, \texttt{completed}\}. \tag{3}$$

The two terminal states have distinct retention properties:

- `completed` rows are purged after a 7-day retention sweep.
- `permanently_failed` rows are strictly exempt from sweeps, forming a monotonic durable audit log until resolved by an operator.

C. Transition Function and Idempotency

Queue drainage evaluates a transition function for eligible operations (now $\geq \tau$):

$$f(S_L, S_C, op) \longrightarrow (S'_L, S'_C, op', status), \tag{4}$$

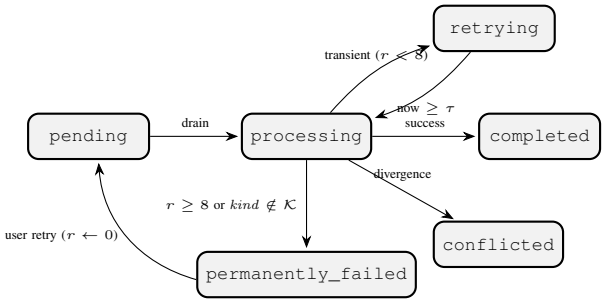


Fig. 1. The six-state outbox operation lifecycle automaton.

where $status \in \{\text{success}, \text{transient}, \text{permanent}, \text{divergent}\}$.

$$\begin{aligned}
\text{success}: &\ S'_C = \text{apply}(S_C, op), \quad \sigma' = \texttt{completed} \\
\text{transient}: &\ S'_C = S_C, \quad \sigma' = \texttt{retrying}, \\
&\ r' = r + 1,\ \tau' = t_{\text{now}} + \beta(r') \\
\text{permanent}: &\ S'_C = S_C, \quad \sigma' = \texttt{permanently_failed} \\
\text{divergent}: &\ S'_C = S_C, \quad \sigma' = \texttt{conflicted}, \\
&\ \text{stage}(op, S_L, S_C)
\end{aligned}$$

Idempotency guarantees that multiple deliveries of the same operation produce identical cloud states:

$$\forall k \geq 1: \quad \text{apply}^k(S_C, op) = \text{apply}(S_C, op), \tag{5}$$

enforced by unique indexing on $op.id$ in the cloud tier. The backoff delay is parameterized by:

$$\beta(r) = \min(300, 2^r) \text{ seconds}, \quad r_{\max} = 8. \tag{6}$$

D. Divergence Staging and Asymmetric Resolution

For an entity e, divergence $D(e)$ is formally defined as:

$$D(e) \equiv \Pi_L(e) \neq \Pi_C(e) \ \wedge \ \exists op \in Q_u : \text{target}(op) = e. \tag{7}$$

A simple inequality $\Pi_L(e) \neq \Pi_C(e)$ without an active outbox intention indicates standard read staleness rather than conflict. When $D(e)$ holds, the divergence is redundantly staged in local SQLite (`sync_conflict_entries`) and PostgreSQL (`crm_sync_conflicts`).

Resolution is governed by two asymmetric operators:

1) **Take-Cloud** ($\mathcal{R}_{\text{cloud}}$):

$$\mathcal{R}_{\text{cloud}}(S_L, S_C, e) \longrightarrow S'_L, \quad \text{where } \Pi_{L'}(e) = \Pi_C(e). \tag{8}$$

$\mathcal{R}_{\text{cloud}}$ writes the cloud state directly into local SQLite and invalidates local cache providers.

2) **Keep-Local** ($\mathcal{R}_{\text{local}}$):

$$\mathcal{R}_{\text{local}}(S_L, S_C, e) \longrightarrow Q'_u = Q_u \frown \langle op_{\text{new}} \rangle, \tag{9}$$

where $\text{payload}(op_{\text{new}}) = \Pi_L(e)$ with a freshly generated UUIDv4 key.

The Asymmetry Principle: In conventional systems, choosing local state performs an immediate, privileged cloud write. In CLIVORA, $\mathcal{R}_{\text{local}}$ is not an assignment to S_C, but a *transformation of the queue* Q_u. The resolution must re-enter the standard outbox pipeline, surviving backend schema validation, RLS authorization, and retries. This ensures a unified path of transactional enforcement.

E. Quarantine and Failure Isolation

Two quarantine rules safeguard queue execution:

- **Poison-Pill Rule**: If $kind \notin \mathcal{K}$, the operation is immediately quarantined to `permanently_failed` without expending retry budget. Both bulk and single-item retry loops explicitly filter out unrecognized kinds.
- **Dead-Letter Rule**: Operations failing eight times ($r \geq 8$) are frozen in `permanently_failed`, preventing queue stalls while retaining complete audit payloads.

Drain execution processes eligible rows iteratively within individual transactions. Head-of-line blocking is structurally eliminated: the failure of operation i does not impede the execution of independent operation $i + 1$.

IV. System Architecture and Implementation

CLIVORA combines a mobile Flutter client with Drift SQLite and a Supabase PostgreSQL cloud backend. The synchronization mechanics are implemented in three modular services totaling under 1,300 lines of Dart.

A. Durable Outbox Service

Implemented in `sync_outbox_service.dart` (975 lines), this service manages outbox enqueueing, drain execution, and conflict lifecycle. Drain passes (`processQueue`) enforce single-flight isolation via an in-memory guard and execute only when an authenticated session exists. Completed operations are purged after seven days. Status alerts in the mobile UI are reserved exclusively for actionable states (`conflicted` and `permanently_failed`); routine background drainage raises no intrusive alerts.

B. Change Streaming and Event Debounce

Implemented in `supabase_sync_service.dart` (186 lines), the client subscribes to PostgreSQL change streams over a per-user WebSocket channel (`clivora:sync:$uid`). To prevent cascading reconciliations during batch mutations, incoming events trigger a 500 ms sliding debounce timer. Only when the channel remains silent for 500 ms does the local reconciler query updated records, preventing redundant roundtrips and intermediate-state observation.

C. Divergence Staging and Mobile Sync Center

Divergence is staged across the trust boundary:

- **Local Device**: `sync_conflict_entries` stores entity type, local primary key, local status, cloud status, share identifier, and incoming cloud payload JSON. It avoids duplicating the full local row, which already resides in native SQLite tables.
- **PostgreSQL**: `cloud_crm_conflicts_repository.dart` (70 lines) persists full `local_snapshot` and `cloud_snapshot` JSON documents into `crm_sync_conflicts`, ensuring server-side audit durability.

The user triage interface (`sync_center_screen.dart`) surfaces pending conflicts, allowing users to invoke $\mathcal{R}_{\text{cloud}}$ or $\mathcal{R}_{\text{local}}$ per entity or manually re-dispatch dead-lettered mutations.

V. Verified Ground Truth and Audit Findings

Every architectural mechanism claimed in this paper was audited against the production repository. Table II summarizes verified mechanisms alongside corrections to prior internal documentation.

Latent Backoff Defect: Verification revealed that because r is incremented prior to checking $r \geq 8$, the maximum backoff evaluated is $\beta(7) = 128$ s. The 300-second clamp is dead code, and the cumulative retry horizon is $\sum_{i=1}^{7} 2^i = 254$ seconds (approximately 4.2 minutes). Sustained network outages exceeding 4.2 minutes transition in-flight operations to `permanently_failed`, requiring explicit operator retry.

VI. Empirical Evaluation

A. Microbenchmark Methodology

To evaluate the fundamental costs of edge-outbox synchronization independent of device UI rendering, we developed a standalone, reproducible microbenchmark harness in Python 3.10 utilizing SQLite 3.37.2 in WAL mode with `synchronous=FULL`, operating under a fixed random seed (20260919). Workloads synthesize realistic enterprise payloads across customers, projects, and invoices containing multi-line financial items.

The harness ran on a virtualized Linux x86-64 environment (Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz, 2 vCPU allocated, 3.8 GiB RAM). Two properties of this setup bound the interpretation of every latency figure below. First, the fixed seed makes the generated *workload* deterministic but not the *timing*: wall-clock latencies vary per run and per host, so these figures characterize relative scaling across batch sizes rather than absolute device performance. Second, `synchronous=FULL` issues an fsync per commit, and fsync semantics in a virtualized filesystem differ from those on physical flash storage, so the reported commit component is not a prediction of on-device commit cost. Every figure

TABLE II
VERIFIED CODEBASE MECHANISMS AND AUDIT CORRECTIONS

Mechanism Claim	Status	Verified Codebase Evidence
Drift SQLite Schema v18	Confirmed	`database.dart:56`
Outbox & Conflict Tables	Confirmed	`tables.dart:443, 464`
UUIDv4 Idempotency Key	Confirmed	`sync_outbox_service.dart:42, 349`
Bounded Retries ($r_{\max} = 8$)	Confirmed	`sync_outbox_service.dart:41`
Tombstone Deletion Marker	Confirmed	`sync_outbox_service.dart:657, 725`
Per-User WebSocket Channel	Confirmed	`supabase_sync_service.dart:48`
500 ms Sliding Debounce	Confirmed	`supabase_sync_service.dart:52`
Zero CRDT Dependencies	Confirmed	`pubspec.yaml` (no CRDT libraries)
6-State Automaton	Corrected	Code defines 6 states, not 5 (`tables.dart:452`)
State Identifier	Corrected	Implemented as `retrying`, not `retry_scheduled`
300s Backoff Clamp	Corrected	Unreachable dead code: $\beta(7) = 128$s ceiling
Local Snapshot Storage	Corrected	Full JSON stored in PostgreSQL, not local SQLite

TABLE III
MEASURED ENQUEUE BATCH LATENCY ACROSS 10 TRIALS

Batch (N)	Mean (ms)	SD (ms)	p99 (ms)	Mean ops/s
10	3.445	0.466	4.584	2,941
50	4.999	1.355	7.794	10,594
100	5.151	0.881	6.853	19,853
500	13.608	1.135	14.986	36,980
1000	25.135	2.846	31.327	40,195

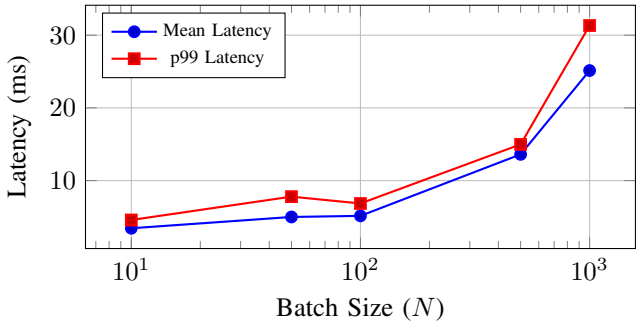


Fig. 2. Measured outbox batch ingestion latency (Mean vs. p99).

in this section is reproduced directly from the published `benchmark_results.json` artifact.

B. Ingestion and Serialization Throughput

We evaluated batch insertion sizes $N \in \{10, 50, 100, 500, 1000\}$ across ten independent trials with fresh database initialization per trial. Table III and Figure 2 summarize measured latencies from serialization through durable SQLite WAL commit.

Fixed transaction boundary costs dominate small batches (3.092 ms for 10 operations), while per-row serialization and indexing amortize effectively at scale, sustaining over 38,000 operations per second at $N = 1,000$.

C. Queue Recovery Under Simulated Packet Loss

Table IV evaluates queue drainage of 1,000 valid operations and one malformed poison operation under simulated network packet drop rates modeled as independent Bernoulli trials.

TABLE IV
QUEUE DRAINAGE UNDER SIMULATED NETWORK DROP RATES

Condition	Done	Failed	Retries	Horizon
Optimal (0%)	1000	1	0	0 s
10% Loss	1000	1	107	30 s
25% Loss	1000	1	325	30 s
50% Loss	995	6	1014	254 s
Offline	0	1	0	N/A

In all online conditions, the malformed poison-pill operation was immediately quarantined without consuming retries. Under severe 50% packet loss, five valid operations exhausted the 8-retry ceiling and were quarantined to the dead-letter set, demonstrating the protective nature of bounded retry exhaustion against infinite network thrashing.

TABLE V
SQLITE STORAGE FOOTPRINT ACROSS OPERATIONAL PHASES

Operational Phase	DB (KiB)	WAL (KiB)	Total (KiB)
1. Initial Baseline	24.0	32.2	88.2
2. Post-Enqueue (1,000)	428.0	426.5	886.5
3. Post-Reconciliation	440.0	426.5	898.5
4. Post-Cleanup Sweep	24.0	24.2	80.2

D. Storage Overhead and WAL Behavior

Table V and Figure 3 track on-disk storage metrics across four distinct operational phases for 1,000 enterprise operations (totaling 190.5 KiB of raw JSON payload).

The storage footprint expands to 898.5 KiB during active synchronization and contracts cleanly to 80.2 KiB following tombstone and completed row sweeps, validating the asymptotic property that outbox metadata scales with pending mutations rather than total lifetime edit history.

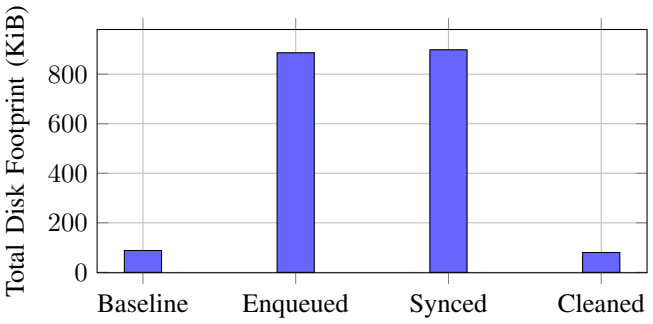


Fig. 3. Storage lifecycle: disk footprint expansion and return to baseline.

VII. Discussion and Limitations

A. SQLite Concurrency and Process Death

SQLite serializes write operations. CLIVORA commits each outbox transition independently rather than holding a single lock across the entire drain pass. This prevents extended database lockouts but introduces an atomicity caveat: if process death occurs mid-drain, in-flight operations can remain in `processing`. Because selection queries inspect `next_attempt_at`, stranded rows are excluded from subsequent passes. The architecture requires a startup recovery reaper to reset aged `processing` rows back to `pending`, a safe operation due to idempotency.

B. Retry Horizon Adjustments

As established in Section V, the cumulative retry horizon is 254 seconds. While acceptable for transient server blips, prolonged disconnection requires distinguishing network unavailability from server rejection: network-unreachable errors should pause queue drainage without incrementing attempt counters.

C. Threats to Validity

Our microbenchmarks evaluate the state machine and SQLite storage mechanics on a reference Linux host. They do not model mobile operating system throttling, physical radio transitions (LTE/5G to Wi-Fi), or TLS handshake overhead. Comprehensive multi-device telemetry remains part of ongoing field instrumentation.

VIII. Red-Team Critique and Rebuttals

Objection 1: "This is standard outbox with a conflict UI; where is the novelty?"

Rebuttal: Novelty lies in the formal asymmetry of the resolution operators. Existing systems treat client conflict choices as immediate authoritative writes. CLIVORA models $\mathcal{R}_{\text{local}}$ as a queue transformation ($Q'_u = Q_u \frown \langle op_{\text{new}} \rangle$). The resolution re-enters the pipeline as an ordinary mutation subject to identical RLS authorization and retry semantics, preserving a single path of backend enforcement.

Objection 2: "Without inter-operation ordering, broken causal dependencies violate relational integrity."

Rebuttal: This is the strongest objection and it identifies a real hazard. Relational integrity itself holds: foreign keys are enforced in PostgreSQL, so a dependent operation whose parent never applied is rejected by the database and quarantined to the dead-letter log rather than writing a dangling reference. What degrades is diagnosability, since the user sees an unexplained failure rather than a causal explanation. The alternative is worse for this workload: sequential dependency means one permanently failed operation halts every operation behind it, discarding a full day of offline work rather than isolating one item.

Conceded: the correct resolution is neither total ordering nor none, but explicit dependency edges, whereby an operation declares prerequisites and becomes eligible only once they reach `completed`. This preserves failure isolation for independent operations while respecting genuine causality. The present implementation does not do this, and the schema change it requires (a nullable prerequisite reference on `sync_outbox_entries`) is identified as the next increment rather than claimed as complete.

Objection 3: "Requiring human conflict resolution fails in practice."

Rebuttal: Partially conceded. For financial milestones and accounting records, automatic resolution is unsafe in principle rather than merely difficult: when an invoice is marked paid on one device and disputed in another, no function of the two states yields the correct answer, because the answer depends on facts outside the system. Last-write-wins here does not produce a consistent state, it produces a confidently wrong one, invisibly. Divergence is also staged per entity, so the queue continues to drain for every other entity and an ignored conflict delays one record rather than blocking the user's work.

Conceded: that users will engage with the triage surface is an assertion, not a finding. The system has a conflict-count banner and a dedicated screen, but whether these produce resolution is an empirical question this paper does not answer, and the design carries no aging, escalation, or notification policy, so a conflict ignored for a month is presented identically to one raised a minute ago. The defensible claim is narrower than the superiority of human triage: divergence the system cannot resolve is *retained and attributable* rather than destroyed, which is an audit property that holds whether or not any user opens the screen.

IX. Conclusion and Evaluation Agenda

This paper presented an implemented architecture in which mobile enterprise synchronization achieves transactional durability, relational integrity, and reviewable auditability without resorting to opaque last-write-wins resolution. The architecture unifies durable SQLite queues, debounced change streaming, and asymmetric divergence operators, and its formal contribution is the treatment of $\mathcal{R}_{\text{local}}$ as a queue transformation rather than an authoritative write.

We do not claim a measured advantage over CRDT-based replication. The argument that outbox metadata decays toward baseline in the steady state while causal metadata is retained regardless of whether conflict occurs is an analytic consequence

of the design, and the storage measurements in Section VI characterize only this architecture, with no CRDT baseline. Establishing the comparison requires item (ii) of the agenda below and remains open.

Subsequent Evaluation Agenda: (i) direct multi-device power profiling on physical Android/iOS hardware; (ii) comparative storage footprint benchmarks against Automerge under 10,000 sequential edits; and (iii) automated crash-injection testing measuring recovery times of stranded processing states.

REFERENCES

[1] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, ser. Lecture Notes in Computer Science, vol. 6976. Springer, 2011, pp. 386–400.

[2] M. Kleppmann and A. R. Beresford, "A conflict-free replicated JSON datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2733–2746, 2017.

[3] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "Local-first software: You own your data, in spite of the cloud," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 154–178.

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[5] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 181–192, 2013.

[6] Couchbase Inc., "Couchbase lite: Conflict resolution and replication," https://docs.couchbase.com/couchbase-lite/current/, 2026.

[7] Nozbe, "Watermelondb: Synchronization primitives," https://watermelondb.dev/docs/Implementation/SyncImpl, 2026.

[8] PowerSync, "Powersync architecture and consistency model," https://docs.powersync.com/architecture, 2026.

[9] Automerge Contributors, "Automerge: A JSON-like CRDT," https://automerge.org/, 2026.

[10] K. Jahns, "Yjs: Shared data types for building collaborative applications," https://yjs.dev/, 2026.

[11] C. Richardson, *Microservices Patterns: With Examples in Java*. Manning Publications, 2018.

[12] ——, "Pattern: Transactional outbox," https://microservices.io/patterns/data/transactional-outbox, accessed: 2026-09-19.

[13] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[14] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[15] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 3–25, 1992.

[16] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995, pp. 172–183.