Downloaded from buildwithruud.com · Kami Document System

# Eliminating Dual-Write Debt Across Salesforce, GTM & D

2026.11.16 · /blog/eliminating-dual-write-debt-distributed-systems

1. The Anatomy of Dual-Write Debt

Whether you are auditing a complex enterprise Salesforce org and GTM pipeline or a modern cloud-native microservice architecture, the most destructive production bugs share a single structural signature: **two uncoordinated processes believing they own the same write**.

In a high-volume Salesforce org or GTM stack (CRM, CPQ, quote-to-cash, and billing), this manifests when a declarative Record-Triggered Flow, a legacy Workflow Rule, and an asynchronous `@future` or Queueable Apex class all mutate `Account.Status` or `Opportunity.StageName` under different transaction boundaries. Under light load, the system appears to work. Under concurrent webhook bursts or bulk data loads, lock contention (`UNABLE_TO_LOCK_ROW`), governor limit exhaustion, and out-of-order overwrites corrupt the system of record.

In a distributed backend, the exact same defect appears when an API handler updates a PostgreSQL row and then fires an HTTP webhook or event message before returning `200 OK`. If the process crashes between the database commit and the network send — or if the downstream consumer processes retries out of order — the two systems diverge permanently.

2. Single-Writer Law & Type-State Domain Modeling

The patterns that untangle enterprise Salesforce and GTM systems translate directly to any software stack. To eliminate dual-write debt, we enforce two architectural laws across both **Folio** (our property operating system) and our enterprise GTM integration architectures:

1. **The Single-Writer Invariant**: Every domain entity and field has exactly one authoritative state machine responsible for mutating it. All other surfaces — UI portals, webhooks, background schedulers, and external CRM syncs — must submit typed commands to that owner rather than writing columns directly.
2. **Append-Only Financial & Event Ledgers**: Mutable balances and status flags destroy auditability. In Folio, account balances, rent charges, commissions, and security deposit holds are never stored as mutable float columns updated in place; they are projections over an immutable, idempotent double-entry ledger (`G-03` payments and period charge idempotency).

3. Transactional Outbox in Rust & PostgreSQL

To bridge PostgreSQL state transitions with external side effects (email dispatch, syndication feeds, CRM synchronization, and R2 vault operations) without dual-write races, Atlas uses a **Transactional Outbox** (`outbox_job` and `atlas_syndication_outbox`):

sql

```
BEGIN;
-- 1. Mutate the authoritative domain state
INSERT INTO atlas_leases (id, tenant_id, asset_id, status, ...)
VALUES ($1, $2, $3, 'active', ...);
-- 2. Enqueue the side-effect in the SAME ACID transaction
INSERT INTO outbox_job (id, tenant_id, job_type, payload, idempotency_key, status)
VALUES (gen_random_uuid(), $2, 'lease.activated', $4, $5, 'pending')
ON CONFLICT (idempotency_key) DO NOTHING;
```