T2 — Kleppmann distributed systems audit

Distributed Systems & Data Layer Audit (MC #10357 T2 — Martin Kleppmann)

Auditor: Martin Kleppmann (DDIA) **Date:** 2026-04-30 **Scope:** ALAI orchestration data plane — kernel, tool scripts, SQLite stores **Method:** Read-only file inspection + sqlite3 queries against live databases

---

1. Event Flow & Consistency

Traced path: task intake ? execution ? completion

Canonical flow as implemented:

1. **Intake** — `mc.js add` inserts into `tasks` (mission-control.db). Status: `open`. No corresponding `task_scheduling` row yet. 2. **Poll** — `pi-orchestrator.js` (daemon, 30s default poll) executes a `getNextTask()` query: LEFT JOIN `tasks` with `task_scheduling`, filtering on `status IN ('open','in_progress')`, `dead_letter=0`, `next_eligible <= now`, `lease_until < now`. Priority ordering: H→M→L, then `updated_at ASC`. 3. **Claim** — `pi-orchestrator` sets `tasks.status = 'in_progress'`, upserts `task_scheduling.lease_until = datetime('now', '+N minutes')`, inserts `active_work(agent, task_id, session_id)`. These are three separate non-atomic DML statements (better-sqlite3 synchronous, but no BEGIN TRANSACTION wrapping the trio in code reviewed). 4. **Execute** — model selection (tier 1–5), agent spawn (Ollama or `claude` CLI subprocess). Execution happens outside the database entirely. 5. **Complete** — `mc.js done "outcome"` sets `tasks.status = 'done'`, clears `task_scheduling.lease_until`, removes `active_work` row. `task_history` append. 6. **Failure path** — after `MAX_TASK_RETRIES = 3`, `task_scheduling.dead_letter = 1`, `tasks.status = 'blocked'`, entry in `dead_letter_queue`.

Events durability: `events.db` has a proper schema with `idempotency_key UNIQUE`, `correlation_id`, `causation_id`, `publisher`, `retry_count`, `dead_letter` table — this is a full transactional outbox pattern. **However, it is underused.** `pi-orchestrator.js` does not write to `events.db` for task transitions; it uses `postHiveMind()` (fire-and-forget to HiveMind daemon) instead. `chain-runner.js` uses `bus.emit()` (in-process event bus, not durable). The durable `events.db` store exists but the primary orchestration path bypasses it. **Source of truth fragmentation (confirmed by query):**

```
tasks:           10,220 rows  (primary state machine)
task_scheduling:  4,211 rows  (covers 41% of tasks — not all tasks have scheduling entries)
active_work:          2 rows  (in-memory live view — only current active agents)
task_history:    36,865 rows  (audit log — orphan FK violations detected, see §2)
```

State is split across four tables plus `active_work`. A task can be `status=in_progress` without a `task_scheduling` row (confirmed: 1 task in this state). The `active_work` table has a stale row (`skillforge|10342|ready_for_review`) — the task is no longer `in_progress` but the `active_work` entry was not cleared. **This is a consistency bug.**

Replayability: The `events.db` schema supports replay (`status`, `retry_count`, `idempotency_key`), but since orchestration transitions are not written there, an event replay would be incomplete. `task_history` provides an immutable audit trail but lacks payload to reconstruct state from scratch.

---

2. Data Layer Health

Schema summary (mission-control.db)

tasks table — 38 columns including `version INTEGER DEFAULT 0` (optimistic locking field exists but usage requires code audit to confirm enforcement), `quality_signals TEXT DEFAULT '{}'`, `dod_evidence`, `builder_model`, `validator_model`, `validation_timestamp`. Schema has grown organically via ALTER TABLE (evidenced by column ordering and `review_requeue_count` trailing addition).

```
-- Indexes:
CREATE INDEX idx_tasks_status_owner ON tasks(status, owner);
CREATE INDEX idx_tasks_status_priority ON tasks(status, priority, created_at);
```

Only two indexes on 10,220 rows. The primary poll query uses `status`, `priority`, `updated_at` — the existing index on `(status, priority, created_at)` covers this well. No index on `delegated_to` despite the primary `getNextTask()` path filtering on `delegated_to = 'pi-orchestrator'` first. This is a **missing index**. **task_scheduling table:**

```
CREATE TABLE task_scheduling (
  task_id INTEGER PRIMARY KEY REFERENCES "tasks"(id),
  cb_state TEXT DEFAULT 'closed' CHECK(cb_state IN ('closed', 'open', 'half_open')),
  attempt_count INTEGER DEFAULT 0,
  last_attempt TEXT, next_eligible TEXT, lease_until TEXT,
  last_outcome TEXT, last_error TEXT, dead_letter INTEGER DEFAULT 0
);
```

No index on `next_eligible` or `lease_until`, both used in hot query path. No index on `dead_letter`. **active_work table:**

```
CREATE TABLE active_work (
  agent TEXT PRIMARY KEY,
  task_id INTEGER DEFAULT NULL,
  FOREIGN KEY (task_id) REFERENCES "tasks"(id)
);
```

`agent TEXT PRIMARY KEY` — means only one task per agent name. This is correct for the current design but implies no parallelism per named agent. **Row counts (actuals):**

| Table | Count | |---|---| | tasks | 10,220 | | tasks (open) | 890 | | tasks (in_progress) | 2 | | tasks (done) | 6,605 | | tasks (paused) | 2,202 | | tasks (blocked) | 492 | | tasks (ready_for_review) | 29 | | task_scheduling | 4,211 | | active_work | 2 | | task_history | ~36,865 | | outbox | 3,262 total, 0 unprocessed | | dead_letter_queue | 22 | | task_scheduling.dead_letter=1 | 52 | | events.db total | 3,710 | | events.db dead | 19 |

Foreign key violations (PRAGMA foreign_key_check):

```
task_history → tasks: 19 orphan rows (task_ids: 962, 982, 983, 984, 1770-1772, 7436-7438,
7446, 14430-14431, 15883, 33500, 36110-36111, 36127, 36133)
task_scheduling → tasks: 1 orphan row (task_id: 9351)
task_metrics → tasks: 7 orphan rows (10053-10054, 10056, 10058-10059, 10061, 10309)
```

Total: **27 dangling foreign key references**. SQLite enforces FK constraints only when `PRAGMA foreign_keys = ON` is set per connection. `pi-orchestrator.js` uses `busy_timeout = 3000` but does not set `PRAGMA foreign_keys = ON` — FK integrity is not enforced at runtime. **WAL state:**

```
mission-control.db      25.6 MB  (main)
mission-control.db-wal  23.3 MB  (WAL = 91% of main — WAL checkpoint is NOT running)
hivemind.db             60.1 MB  (main)
hivemind.db-wal        147.0 MB  (WAL = 245% of main — severely bloated)
flywheel.db            224.2 MB  (main)
flywheel.db-wal        263.6 MB  (WAL = 118% of main)
events.db               14.6 MB  (main)
events.db-wal            4.5 MB  (WAL = 31% — acceptable)
```

`PRAGMA wal_checkpoint` returned `0|1|0` for mission-control.db, confirming the checkpoint ran 0 pages — the WAL has not been checkpointed recently. hivemind.db WAL at 147MB is a **P1 operational risk**: if the process crashes, SQLite must replay the entire WAL on next open, causing multi-second startup delays. Large WALs also inflate Litestream replication bandwidth.

Last vacuum: No `VACUUM` or `ANALYZE` evidence found. The db has `freelist_count = 0` (no free pages), which is normal but may indicate vacuum has never run or ran recently on a growing dataset. **Growth rate:**

```
Week 2026-W09: 798 tasks/week
Week 2026-W13: 1354 tasks/week (peak)
Week 2026-W16: 1145 tasks/week
Week 2026-W17: 790 tasks/week
```

At ~900 tasks/week, mission-control.db will reach 50,000 tasks in ~4 months. At current page size (4096 bytes), the database will reach ~500MB without periodic archival of done/blocked tasks.
---

3. Concurrency & Ordering

Multiple writers

The following processes can write to mission-control.db concurrently:

- `pi-orchestrator.js` daemon (primary writer — poll loop + task state transitions)
- `mc.js` CLI (human-invoked — add, done, block, assign, start)
- `sprint-pipeline.js` (calls `mc.js start` via `execSync` — indirect writer)
- `pipeline-controller.js` (reads MC DB directly readonly + indirect via mc.js)
- `cross-company-bus.js` (creates tasks via `mc.js add`)
- `chain-runner.js` (updates tasks via MC script calls)
- Multiple Claude agent subprocesses calling `mc.js` (spawned by pi-orchestrator)

Concurrency mechanism: better-sqlite3 is **synchronous** — all reads/writes block the Node.js event loop. With WAL mode enabled (`PRAGMA journal_mode = WAL`), readers do not block writers and writers do not block readers. Multiple writers still serialise through SQLite's single-writer lock. `pi-orchestrator.js` sets `busy_timeout = 3000` ms — meaning it will wait up to 3s for the lock, then throw. This is acceptable for a single-machine deployment. **Lease mechanism analysis:**
The lease is implemented in `task_scheduling.lease_until`. The `getNextTask()` query filters `lease_until < datetime('now')` before claiming. Claim path:

```
// From getNextTask() — three separate statements, not in a single transaction:
1. UPDATE task_scheduling SET lease_until = datetime('now', '+N min')
2. UPDATE tasks SET status = 'in_progress'
```

```
3. INSERT OR REPLACE INTO active_work(agent, task_id, ...)
```

Critical bug: These three writes are not wrapped in a single SQLite transaction. If the process crashes between statements 1 and 2, `task_scheduling.lease_until` is set but `tasks.status` remains `open`, and `active_work` has no entry. The task appears "leased" to a dead worker but also appears available to the next poller once the lease expires — which is the correct recovery. **However**, between crash and lease expiry, `getNextTask()` will skip this task unnecessarily, creating transient unavailability without notice. **Confirmed stuck leases (from query):** 3,254 of 3,255 `task_scheduling` rows with a `lease_until` value have an *expired* lease (`lease_until < datetime('now')`). This means nearly all leases are in the past — which is normal if tasks completed (the lease was not cleared to NULL on completion, just left expired). But it also means the lease mechanism provides **zero protection** against concurrent double-claiming, because any new poller will see `lease_until < now` as valid to claim again. This is by design (lease expiry = re-eligibility), but implies: **if two pi-orchestrator instances ran simultaneously, both could claim the same task** because the claim is not atomic (no SELECT...FOR UPDATE, SQLite has no such construct, and no BEGIN IMMEDIATE around the claim trio). **Reconcile mechanism:** `reconcileWorkerCounters()` (pi-orchestrator.js line ~3248) runs every 30s to detect drift between `activeTasks` Map and `activeWorkers` counters. This addresses the in-process ghost slot problem but not the cross-process double-claim problem. **On daemon crash mid-task:** The task remains `status=in_progress` with an expired lease. The next orchestrator cycle (after restart) will pick it up from `getNextTask()` because `lease_until < now`. `attempt_count` is incremented. After 3 retries, it goes to DLQ. This recovery path is correct but relies on the orchestrator restarting — if it does not restart (manual intervention needed), the task is orphaned in `in_progress` indefinitely. Confirmed: MC task #10330 is currently `in_progress` with `active_work` row showing `john` as agent — this appears to be a legitimate in-progress task, not a ghost. **Event log:** `events.db` has a `dead_letter` table with 19 entries — these are events that exhausted retries. The events schema includes `correlation_id` and `causation_id` for lineage tracing. However, as noted in §1, the primary orchestration path does not write to this store.

---

4. Backup & Recovery

What exists

Litestream (`/Users/makinja/system/config/litestream.yml`): Comprehensive — 50+ databases replicated to Azure Blob Storage (`alaibackups0ebb.blob.core.windows.net`, container `system-db-backups`). Tiering:

- **P0-critical** (mission-control.db, hivemind.db, costs.db, events.db, durable-runner.db, knowledge.db, email-inbox.db, alem-directives.db): `sync-interval: 1s`, `retention: 168h` (7 days)
- **P0-financial** (fiken.db, invoices.db, contracts.db, leads.db): `sync-interval: 1s`, `retention: 720h` (30 days)

- **P1** (agent-routing.db, bee-index.db, contacts.db, etc.): `sync-interval: 1s`, `retention: 72h`
- **P2** (prompt-cache.db, baikal-caldav.db, etc.): `sync-interval: 10s`, `retention: 24h`

flywheel.db (224MB main + 263MB WAL): `sync-interval: 30s` — reasonable given size, but WAL bloat means Litestream is continuously replicating a large dirty WAL. **azure-db-backup.sh** (`com.alai.azure-db-backup`, every 4h): Independent Postgres + Docker volume + Qdrant backup to Azure Blob. This covers the cloud products (Bilko, Drop) but is reported as silently failing (daemon-fleet watchdog MC #10173). The error log shows: `rm: /tmp/az-backup-50926: Directory not empty` — non-fatal cleanup failure but indicative of script state pollution. **Point-in-time restore:** Litestream `.ltx` format supports point-in-time restore within the retention window. `litestream.yml.bak-20260425` and restore config at `litestream-restore.yml` confirm the restore procedure has been documented. **Manual snapshots:** `mission-control.db.bak-20260306-175717` (4.5MB) and `mission-control.db.bak-before-7082` (21.6MB) exist on disk — evidence of manual before/after snapshots. The 2026-03-06 snapshot is ~7 weeks old, covering a period when the database was ~1/5 current size. These are not automated. **RPO assessment:**

- mission-control.db: RPO = ~1s (Litestream sync-interval) + Azure write latency (~100ms). **Effective RPO: ~2s**. This is excellent for a SQLite-based system.
- Litestream replication of a 23MB WAL: the WAL is being replicated continuously, but a large un-checkpointed WAL means recovery requires replaying that WAL — **estimated recovery time: 5–15 seconds** on a healthy machine.

RTO assessment:

- If mission-control.db corrupts NOW: restore from Litestream latest snapshot + WAL segments. Litestream restore to local path, then restart pi-orchestrator. Assuming Azure access is available: **RTO ~2–5 minutes**.
- If ANVIL (the Mac Studio) dies: restore requires a new machine, reinstall Homebrew + Node.js + Litestream, then `litestream restore`. **RTO ~30–60 minutes** for full service restoration.

What is lost on corruption NOW:

- `active_work`: 2 current rows — 2 in-flight tasks would need manual status check/reset
- Any `task_history`, `outbox`, or `approval_tokens` rows written in the last ~2 seconds
- The 23.3MB WAL (uncommitted pages) — Litestream replicates WAL pages, so this is covered

Untested restore: No evidence of a restore drill in logs or memory files. `litestream-restore.yml` exists as documentation but there is a note in MEMORY.md that `azure-db-backup` was silently failing. If Litestream's Azure SP credentials (`AZURE_CLIENT_ID`) have rotated or the SP permissions have changed, the replicas may be stale without any alert. This is the **single most dangerous gap** in the backup posture. **ingest-queue.sqlite.corrupt** in `~/system/state/`: A 6.4MB corrupt SQLite file alongside a backup — evidence of a past corruption event. No RCA documented in accessible files.

---

5. Evolution Risk

The 5,251-line pi-orchestrator.js problem

`pi-orchestrator.js` is a 5,251-line monolith that owns: semantic classification, model selection, prompt construction (with 4 RAG sources), worker pool management, quality gating (lint, test, semantic), proof-of-work verification, routing token writing, DLQ writing, HiveMind posting, cost tracking, and the main daemon loop. It has at least 14 conceptually independent subsystems co-located in one file.

Concrete risks:

1. **Single process, single point of failure.** If pi-orchestrator crashes mid-task, all in-flight work is abandoned. The lease mechanism provides recovery but with a gap of `lease_until` duration (typically 5–30 minutes) before re-eligibility.

2. **In-process worker pool counters** (`activeWorkers` object) are process-local. These are not persisted to the database. On restart, `activeWorkers` resets to zero even if tasks are still `status=in_progress` in the DB — the orchestrator will over-claim slots until `reconcileWorkerCounters()` detects the drift (up to 30s).

3. **No command authority boundary.** `chain-runner.js`, `sprint-pipeline.js`, `pipeline-controller.js`, `cross-company-bus.js`, and `mc.js` all write directly to mission-control.db. There is no single command handler. A schema migration or constraint change in `tasks` requires auditing all 14+ writers. The 38-column `tasks` table with columns added via `ALTER TABLE` is the result.

4. **`task_scheduling` not populated for all tasks.** 4,211 scheduling rows for 10,220 tasks means 58% of tasks have no scheduling metadata. `getNextTask()` handles this with `LEFT JOIN ... COALESCE(ts.cb_state, 'closed')` — safe but means circuit breaker and lease state is silently absent for most tasks.

5. **`sprint-pipeline.js` DAG state in JSON column.** `sprints.dag_json TEXT` stores the full DAG serialisation inside a SQLite text column. This means querying or filtering on DAG phase status requires deserialisation in application code, cannot use SQL indexes, and is vulnerable to partial writes leaving malformed JSON.

Modernisation recommendations

Recommendation 1 — Immediate: WAL checkpointing cron (hours) Run `PRAGMA wal_checkpoint(TRUNCATE)` on hivemind.db, flywheel.db, and mission-control.db every 4 hours via LaunchAgent. The 147MB hivemind WAL is a live production risk. Cost: 2 hours, zero architectural change. **Recommendation 2 — Short term: Atomic claim transaction (1–2 days)** Wrap the task claim (lease_until update + status=in_progress + active_work insert) in a single `BEGIN IMMEDIATE ... COMMIT`. This is trivially achievable with better-sqlite3's `db.transaction()` API and eliminates the three-statement race window. Also add `CREATE INDEX IF NOT EXISTS

idx_task_scheduling_next_eligible ON task_scheduling(next_eligible, dead_letter)` and `CREATE INDEX ON tasks(delegated_to)` to fix the missing indexes in the hot query path. **Recommendation 3 — Short term: FK enforcement (1 day)** Add `db.pragma('foreign_keys = ON')` to every better-sqlite3 connection open in pi-orchestrator.js, mc.js, and all tool scripts. Then fix the 27 existing orphan rows (most are safe to delete — they reference tasks that were hard-deleted, violating the soft-delete principle). This prevents a class of data inconsistency bugs at zero performance cost. **Recommendation 4 — Medium term: Extract command boundary (1–2 weeks)** Move all task state mutations (open→in_progress, in_progress→done, etc.) behind `mc.js` as the **single writer**. All tools that currently open mission-control.db directly as a writer (sprint-pipeline.js, pipeline-controller.js, cross-company-bus.js) should call `mc.js` via IPC or HTTP instead. This eliminates the 14-writer problem and makes schema evolution safe. The `outbox` table in mission-control.db is already structurally correct for this — route all mutations through it with `processed=0`, and have mc.js process the outbox as the canonical writer. **Recommendation 5 — Medium term: Decompose pi-orchestrator.js (2–4 weeks)** Split into four services:

- `task-classifier.js` — semantic classification + caching (stateless, ~200 LoC)
- `model-router.js` — model selection + fleet health + routing tokens (~300 LoC)
- `agent-executor.js` — subprocess management + quality gate + proof-of-work (~500 LoC)
- `task-scheduler.js` — the daemon loop + lease management + DLQ + concurrency counters (~300 LoC)

These communicate through the existing `events.db` store (which is already correctly designed for this). This eliminates the single-process SPOF and enables independent restart of each component. **Recommendation 6 — Long term: Event-sourced task store (4–8 weeks)** Replace the multi-table task state machine with an event-sourced design: one `task_events` table (taskId, eventType, payload, timestamp, actor) as the source of truth; `tasks` becomes a materialised view rebuilt from events. This enables full replay, audit trail, and time-travel queries. The existing `task_history` table is a partial implementation of this pattern. The `events.db` schema (with `idempotency_key`, `correlation_id`, `causation_id`) is the correct foundation — extend it to cover task lifecycle events and wire pi-orchestrator to write there instead of posting to HiveMind. **CDC option:** An alternative to event sourcing is Litestream-level CDC: Litestream already replicates WAL pages; a consumer reading those pages could reconstruct a change log. This requires custom Litestream consumer code but avoids changing the write path.

Top 3 Architectural Risks

RISK 1 — Unverified Litestream backup integrity (CRITICAL) Litestream is configured for 1s RPO on mission-control.db, but the azure-db-backup daemon is known-silently-failing (MC #10173). If the Litestream LaunchAgent has also silently stopped (daemon watchdog found 11 failures), the Azure replicas may be hours or days stale. A corruption event today could result in data loss far exceeding the stated 2s RPO. **Action needed: verify last successful Litestream replication timestamp immediately.** Run `litestream snapshots -config ~/system/config/litestream.yml` and check dates. Then run a restore drill to a temp path. **RISK 2 — Non-atomic task claim + 14 concurrent writers (HIGH)** The three-statement claim (lease/status/active_work) is non-atomic.

With 14 identified writers accessing mission-control.db without a command authority boundary, there is no structural guarantee against double-claims, lost updates, or constraint bypass. The `version` column in `tasks` exists for optimistic locking but its enforcement is not verified — if writers do not check `WHERE version = ?` in their UPDATE statements, it provides no protection. **RISK 3 — 147MB hivemind.db WAL + 264MB flywheel.db WAL (HIGH)** Both WALs exceed their main database file sizes. WAL mode accumulates pages when readers hold long-running transactions or when `wal_autocheckpoint` threshold is never reached due to write volume. These databases cannot benefit from Litestream's efficient snapshot mechanism until checkpointed. On an unexpected restart, SQLite must replay the entire WAL — for hivemind.db this means replaying 147MB before any operation can proceed, causing multi-second to minute-level startup delays. Under extreme conditions (full disk), the WAL files could prevent writes to all SQLite databases sharing the same volume.

Top 3 Strengths

STRENGTH 1 — Litestream replication architecture The investment in Litestream with proper tier classification (P0/P1/P2), 7-day retention for critical stores, and Azure Blob backend is genuinely sophisticated for a single-machine SQLite deployment. This puts ALAI's backup posture ahead of many funded startups. The `litestream-restore.yml` and SP-based auth show operational maturity. **STRENGTH 2 — Circuit breaker + dead-letter queue pattern** `task_scheduling.cb_state IN ('closed', 'open', 'half_open')` with `attempt_count`, `next_eligible` backoff, and `dead_letter` flag implements a textbook circuit breaker. The DLQ path (auto-block + Slack alert after 3 retries) prevents runaway retry storms. 52 entries in DLQ with max 5 attempts shows the mechanism is working in production. **STRENGTH 3 — events.db transactional outbox schema** The `events.db` schema — with `idempotency_key UNIQUE`, `correlation_id`, `causation_id`, `status` state machine, `dead_letter` table, and proper indexes — is correctly designed for at-least-once event delivery with deduplication. The schema supports everything needed for an event-sourced architecture. The gap is that the primary orchestration path does not yet write to it; when it does, this becomes a genuine durable, replayable event log.

Appendix: Key File Paths

- `/Users/makinja/system/kernel/pi-orchestrator.js` — 5,251 LoC main daemon
- `/Users/makinja/system/kernel/cross-company-bus.js` — lateral event routing
- `/Users/makinja/system/tools/chain-runner.js` — DAG chain execution
- `/Users/makinja/system/tools/sprint-pipeline.js` — sprint coordination
- `/Users/makinja/system/tools/pipeline-controller.js` — 13-phase lifecycle
- `/Users/makinja/system/databases/mission-control.db` — 25.6MB, 10,220 tasks
- `/Users/makinja/system/databases/mission-control.db-wal` — 23.3MB (needs checkpoint)
- `/Users/makinja/system/databases/hivemind.db-wal` — 147MB (CRITICAL — needs checkpoint)

- `/Users/makinja/system/databases/flywheel.db-wal` — 264MB (needs checkpoint)
- `/Users/makinja/system/databases/events.db` — 3,710 events, 19 dead
- `/Users/makinja/system/config/litestream.yml` — 741 lines, 50+ DBs replicated
- `/Users/makinja/system/state/ingest-queue.sqlite.corrupt` — evidence of past corruption

Revision #2
Created 2026-05-01 08:08:29 UTC by John
Updated 2026-06-07 20:00:44 UTC by John