PRODUCTION READINESS REVIEW

rqlite 8b10e2d production-readiness review — consolidated findings

Prepared for rqlite maintainers · 2026-10-28 · commit `8b10e2d` · engine: XailonCode (json)

We reviewed **rqlite** at commit `8b10e2d` (2026-10-28) for production readiness. This is what needs work, what to plan around, and what to sign off.

Summary

This is a mature, single-binary distributed SQLite (Raft + SQLite) at 95k LOC Go. The core replication and storage design is sound and its trade-offs are written down (db/DESIGN.md, snapshot/DESIGN.md). The problems are not in the data model — they are at the edges: the node-to-node wire, the HTTP listener, and the process's response to its own bad state. Four structural choices sit under most of the confirmed defects (section 3).

Two defects are blockers and reachable without credentials. The Raft/cluster port reads an attacker-chosen 8-byte length and calls make() on it before any auth (cluster/service.go:500-502) — a peer can panic or OOM the node; this is open issue #2619, and A1/K8/K24 are the same bug. The shipped Docker image binds the SQL API to 0.0.0.0:4001 with -auth unset (docker-entrypoint.sh:22,34; config_flags.go:175), so `docker run -p 4001:4001 rqlite/rqlite` exposes arbitrary SQL and full DB exfiltration; the bare binary defaults to localhost (config_flags.go:167).

Beyond those: the HTTP server runs with no read/header/write/idle timeouts and no request-body cap (http/service.go:346-348; snapshot/restore.go:29-32), so a slow client or a large body pins sockets and drives RSS; queued writes that exhaust their retries are still answered 200 and then dropped (http/service.go:1980-1996); and every integrity/durability failure calls logger.Fatalf with no break-glass, so a rotted snapshot or a full disk crash-loops the node until an operator deletes files by hand (snapshot/store.go:276-278,961-970; snapshot/sink.go:212-223).

What is acceptable and only needs to be written down: the queued fire-and-forget path loses its in-flight batch on crash (S1), _sync=0 means a power-lost node rebuilds from Raft and can lose unacknowledged writes (S2), SIGTERM resets in-flight HTTP by design (C4), and Windows skips the directory fsync behind atomic renames (C3). None of these is a surprise in the design docs; each is a line an operator must sign.

Process is the weak signal: from the GitHub facts every listed PR merged with an empty reviewer list, all authored by one person, and defects like the unbounded make() and the missing HTTP timeouts survived dozens of them. There is effectively no independent review gate.

- **Architecture gaps** (4). Structural choices underneath most remaining defects; they need design decisions.
- **Needs work** (9 items). Defects a user or operator will run into. The urgent ones: A1, A2, A3, A4.
- **Deployment and use-case constraints** (5 items). Limits to plan the rollout around.
- **SLA considerations** (2 items). Behaviour that is fine once written down and signed off.

> **Recommendation:** Run it on a trusted, firewalled network only — single node or a small cluster — with -auth set on the API and -node-cert/-node-key on internode links, never on a public port, and do not treat a queued-write 200 as durable.

1. How to read this report

Every claim says how we know it:

- [observed]: read in the code, configuration or a merged pull request at `8b10e2d`.
- [computed]: arithmetic on observed values.
- [inferred]: reasoning from what was read. Not run.

This was a code review. Nothing was run against a live system. Every file reference was checked mechanically against the repository at this commit: 76 resolved, 0 did not (marked †; an [observed] claim whose citations all fail is downgraded to [inferred]).

Open items sort into: **(a) Needs work**, visible to users or affecting operations; **(b) Constraint**, a limit on deployment or use; **(c) SLA**, hard-failure behaviour you accept in writing. One rule governs the sorting: a failure with no recovery path never goes in (c). An SLA can put a price on data loss or downtime, but not on "the system does not come back".

2. Architecture gaps

Most items in the sections below come from these. Each one is a design decision, not a patch.

#	Gap	What it causes	How known
G5	**The node-to-node trust boundary is opt-in, and the internode wire is unguarded** The Raft/cluster lane treats the network as trusted. cluster/service.go:500-502 reads an 8-byte length and makes a buffer of that size before the auth check at :533, and handleConn has no recover, so a peer reaching port 4002 can panic or OOM the node ([observed] lines 486-539). Encryption and peer auth are off unless -node-cert/-node-key are set and -node-verify-client is true (config_flags.go:176-180), checkCommandPerm returns true when the store is nil (cluster/service.go:360-364), and GET_NODE_META is authorized unconditionally (cluster/service.go:403-404), leaking each node's API URL and commit index. With -auth on and node TLS off, user passwords cross port 4002 in cleartext (makeCredentials at http/service.go:2168; conditional dialer at main.go:599-609). The container image compounds this by binding 0.0.0.0 with no auth. Fixing this is one decision — is internode mTLS default-on, and is there a hard message-size cap — not a scatter of patches.	A1, A5, G1, A2	[observed] the unbounded make() and auth ordering at cluster/service.go:486-539, the nil-store short-circuit at :360-364, unconditional GET_NODE_META at :403-404, opt-in flags at config_flags.go:176-180, cleartext credential forwarding, and 0.0.0.0 container defaults; the crash/OOM and cleartext capture follow from those ([inferred])
G6	**HTTP ingress has no resource limits** The API server is built with no ReadTimeout, ReadHeaderTimeout, WriteTimeout or IdleTimeout (http.Server{Handler: s} at http/service.go:346-348), and no handler wraps the body in http.MaxBytesReader — /db/load buffers the whole file with io.ReadAll (http/service.go:908), and the restore path allocates a header buffer from an unvalidated uint32 (snapshot/restore.go:29-32, 4 GiB at 0xFFFFFFFF). Queue.Write blocks on the channel under seqMu with no full/return-error path (queue/queue.go:131-152), and the handler calls it with no request context or deadline (http/service.go:1501), so a stalled consumer makes producers hang open instead of shedding a 503. One decision — the maximum accepted request size and the always-on timeouts — closes the Slowloris, the OOM-via-body, and the unbounded backpressure together.	A3, G3, G4	[observed] the bare http.Server literal, io.ReadAll with no limit, make([]byte, hdrLen) from a wire uint32, and the blocking channel send under lock; the exhaustion and hang effects are [inferred] from net/http and channel semantics

#	Gap	What it causes	How known
G7	**Every integrity or durability failure is fatal, with no break-glass path** Integrity and durability errors call logger.Fatalf and exit: corrupt snapshot CRC (snapshot/store.go:276-278, via ensureVerified at :961-970), a failed incremental snapshot write (snapshot/sink.go:212-223), and a clean-snapshot CRC mismatch at startup (store/store.go:676-680). verifyOnce is per-Store, so each restart re-reads the same bad artifact and exits again — a crash loop whose only fix is deleting a file or directory by hand. An unreadable REAP_PLAN makes NewStore fail outright (snapshot/store.go:282-284, 871-876), so the node will not start. With str.NoVerifyDB = true hardcoded (cmd/rqlited/main.go:462), the reaper skips the post-checkpoint PRAGMA integrity_check (snapshot/store.go:675-678) while recomputing the CRC from that same file, so a torn checkpoint is blessed and replicated. A single documented degraded-start mode (boot and re-fetch from Raft) plus a non-fatal quarantine of bad artifacts is the missing decision.	A6, A7, A9, C1, G2	[observed] each logger.Fatalf site, the per-process verifyOnce guard, the NewStore error on bad plan, and NoVerifyDB=true; the crash-loop on restart and inherited corruption are [inferred]
G8	**The queued-write durability contract is not modelled** The queued-write path separates acknowledgement from commit without representing the gap. Without ?wait, POST /db/execute?queue returns 200 as soon as stmtQueue.Write returns (http/service.go:1501-1506), before runQueue applies the batch. When retries are exhausted, runQueue logs, breaks, and still runs the post-write step — atomic.StoreInt64(&s.seqNum, ...) and req.Close() (http/service.go:1980-1996) — so seqNum (exposed as sequence_number in GET /status, http/service.go:1092) advances and even a ?wait caller gets a normal 200 for a write that never applied. The queue is in-memory and runQueue uses context.Background() (:1958), so a SIGTERM drops the in-flight batch (:1953-1954). Deciding what a queued 200 means — and giving the caller a per-write failure signal — is the design answer.	A4, S1	[observed] the early 200, the unconditional seqNum store and req.Close() on the retry-limit break, and the in-memory queue; the client-visible success despite loss is [inferred] from req.Close() releasing the flush channel

3. (a) Needs work

#	Item	Why it is (a)	How known
A1	**The Raft service allocates a buffer from an unauthenticated wire length, so a peer can crash the node** `cluster/service.go:500–502` reads an 8-byte little-endian length off the connection and calls make([]byte, sz) with no upper bound, and this happens before the authorization check at `cluster/service.go:533`. A peer that reaches the Raft port and sends sz near 2^64 makes make() panic with "len out of range"; handleConn has no recover, so the panic ends the process. A merely large but legal sz (for example 1<<40) makes the node reserve and fault in gigabytes within the 30s connReadTimeout window. `cluster/client.go:973–976` has the same unbounded make(); its recover at `cluster/client.go:952–959` catches the range panic but not an allocation out-of-memory throw. `cluster/service.go:500–502`, `cluster/service.go:533`, `cluster/client.go:952–976`	Any host that can open the Raft port can take the node down.	[observed] the length read, the unbounded make(), and the missing recover on the server path; the process exit is [inferred]
A2	**The official Docker image binds the SQL API to 0.0.0.0 with authentication off by default** `docker-entrypoint.sh:22` and :34 default HTTP_ADDR to `0.0.0.0:4001` † and RAFT_ADDR to `0.0.0.0:4002` †, and Dockerfile:80 runs that entrypoint with no -auth. The flag default is empty (`config_flags.go:175` †), so `main.go:184–187` † returns a nil credential store, and every HTTP handler then treats CheckRequestPerm as pass (`http/service.go:1862–1864`). A container started with `docker run -p 4001:4001 rqlite/rqlite` therefore exposes /db/execute, /db/query, /db/load and /db/backup to anyone who can reach the port — arbitrary SQL and full DB exfiltration. The bare binary is safer (localhost:4001, `config_flags.go:167` †), so the risk is specific to the container path that most operators use. `docker-entrypoint.sh:20–42`, `Dockerfile:75–81`, `cmd/rqlited/config_flags.go:167–183`, `http/service.go:1861–1871`	Anyone who can reach the published port gets unauthenticated read/write/backup of the database.	[observed] defaults at docker-entrypoint.sh:22,34, config_flags.go:167-175, main.go:184-187, http/service.go:1862-1864; the open exposure [inferred] from those defaults.

#	Item	Why it is (a)	How known
A3	**The HTTP server starts with no read, header, idle, or write timeouts** Start() builds the server as http.Server{Handler: s} at `http/service.go:346–348`, leaving ReadTimeout, ReadHeaderTimeout, IdleTimeout and WriteTimeout at zero. A client that opens a connection and sends bytes slowly holds its goroutine and file descriptor for as long as it likes; the same applies to a /db/load body that stalls. A few thousand such connections pin all sockets open and new requests cannot be served. `http/service.go:346–348`	A handful of clients trickling bytes can hold every connection open and starve real traffic.	[observed] the http.Server literal has no timeout fields; the exhaustion effect is [inferred]
A4	**A queued write that exhausts its retries is still reported as committed** runQueue retries a failed execute and, when nRetries exceeds queueRetry, logs the failure and breaks at `http/service.go:1980–1985`, then unconditionally runs the post-write step at `http/service.go:1992–1996`: atomic.StoreInt64(&s.seqNum, req.SequenceNumber) and req.Close(). seqNum is documented "Last sequence number written OK" (`http/service.go:314`) and is exposed as sequence_number in GET /status (`http/service.go:1092`), and req.Close() closes the flush channel (`queue/queue.go:46–49`), which unblocks a ?wait caller. So a caller using the queued-write protocol sees success for a write that was never applied, because the only record of the failure is a log line and a counter. `http/service.go:1980–1996`, `http/service.go:314`, `http/service.go:1092`, `queue/queue.go:46–49`	A dropped write is acknowledged as committed, so the caller cannot tell it lost data.	[observed] the control flow and the unconditional seqNum store and Close; the caller seeing success is [inferred]
A5	**When auth is enabled on a cluster but node TLS is not, user passwords are forwarded to the leader in cleartext** makeCredentials (`http/service.go:2168–2176`) copies the request's BasicAuth username and password into a cluster proto.Credentials, which the follower marshals onto the internode connection (`cluster/client.go:199–206` for Execute, similar in Query/Request/Load). The cluster dialer only gets a *tls.Config when -node-cert or -node-ca-cert is set (`main.go:599–609`†), so with the default no-TLS internode link the operator's plaintext password traverses the network on port 4002. Turning on -auth does not turn on encryption, so a passive listener on the internode network captures credentials for every write that lands on a follower. `http/service.go:2168–2176`, `cluster/client.go:195–206`, `cmd/rqlited/main.go:595–619`	Enabling authentication gives operators false confidence while credentials still cross the wire in cleartext.	[observed] makeCredentials and client marshalling, and the conditional TLS dialer at main.go:599-609.

#	Item	Why it is (a)	How known
A6	**A corrupt snapshot file crash-loops the node with no built-in override** When a node must restore at startup the Store calls snapshotStore.EnsureVerify() (`store/store.go:838–840`), which runs checkCRCs and, on any CRC32 mismatch, calls fatalFn (`snapshot/store.go:961–970`). The default fatalFn is logger.Fatalf (`snapshot/store.go:276–278`), which terminates the process. verifyOnce is reset on every new process, so a restart re-reads the same corrupt snapshot and exits again: the node crash-loops. The only fix is a human deleting the offending snapshot directory. The same fatalFn also fires lazily the first time snapshot data is used for a reap or a snapshot transfer to a peer (`snapshot/store.go:950–970`), so a bit-flip in a stored snapshot can kill a running, otherwise healthy node. `store/store.go:838–840` , `snapshot/store.go:961–970` , `snapshot/store.go:276–278`	One corrupt byte in the newest snapshot leaves the operator with a crash-looping node and no supported flag to skip or quarantine it.	[observed] for the call sites and the Fatalf default; [inferred] that restart re-runs the check because verifyOnce is process-local
A7	**Snapshot integrity check is disabled in production and the CRC is computed after the checkpoint, so corruption is blessed then replicated** `cmd/rqlited/main.go:462` hardcodes str.NoVerifyDB = true. In the reaper this skips step 6, the PRAGMA integrity_check of the consolidated database (`snapshot/store.go:675–678` , 'if s.noVerifyDB.IsNot() { p.AddVerifyDB(dbPath) }'). The CRC32 sidecar is recomputed from that same post-checkpoint file (`snapshot/store.go:640–641` , AddCalcCRC32(dbPath, ...)), so the verify-at-restore CRC check always matches even if the checkpoint produced a bad file. A torn checkpoint can therefore be consolidated into the 'full' snapshot that followers then fetch and restore, propagating the damage instead of catching it. `cmd/rqlited/main.go:462` , `snapshot/store.go:675–678` , `snapshot/store.go:640–641`	The one integrity check that could catch a bad checkpoint is turned off in the shipped binary, and the checksum that remains is derived from the suspect file.	[observed] for all four lines; [inferred] that a follower restoring the snapshot inherits the corruption

#	Item	Why it is (a)	How known
A8	**Graceful shutdown has no overall deadline and a second SIGTERM is ignored, so the process can hang on shutdown** On the first signal main cancels mainCtx (`cmd/rqlited/signals.go:31–37`) and runs the shutdown sequence (`cmd/rqlited/main.go:280–332`). The only bounded step is the pre-shutdown self-removal RPC (10s, `cmd/rqlited/main.go:291–295`); the pre-close snapshot `s.Snapshot(0)` (`store/store.go:1005–1015`) and `s.raft.Shutdown()` waited on with no timeout (`store/store.go:1030–1035`) are unbounded. `CreateContext` reads exactly one signal, so a second Ctrl-C/SIGTERM does nothing. Consequence: a node shutting down during an in-flight snapshot transfer or restore never exits, ignores further SIGTERM, and is only killed by Kubernetes SIGKILL after terminationGracePeriodSeconds — this is open issue #1695. `cmd/rqlited/main.go:280–332` , `cmd/rqlited/signals.go:31–37` , `store/store.go:1005–1015,1030–1035`	An operator cannot complete a normal rolling restart and is forced to SIGKILL the node.	[observed] the shutdown code paths and single-signal context; the hang effect [inferred] from the un-timed `raft.Shutdown` / `Snapshot` waits, consistent with issue #1695.
A9	**An unreadable reap plan stops the Snapshot Store — and the node — from opening** check() runs inside NewStore. If REAP_PLAN exists and plan.ReadFromFile fails to parse it (`snapshot/store.go:871–876`), check returns an error and NewStore fails (`snapshot/store.go:282–284`), so rqlited does not start. The plan write itself is atomic (`snapshot/plan/plan.go:117–138` , temp file + sync + rename), so a torn write is unlikely, but filesystem corruption or a truncation of the live plan file yields this state directly. There is no fallback that logs the bad plan and continues; the operator must manually delete REAP_PLAN to boot. `snapshot/store.go:871–876` , `snapshot/store.go:282–284` , `snapshot/plan/plan.go:104–115`	A single unreadable control file converts a recoverable snapshot-store inconsistency into a node that cannot start.	[observed]

4. (b) Deployment and use-case constraints

#	Constraint	Rationale	How known
C1	**A full disk during an incremental snapshot or checkpoint exits the process, producing a crash loop until space is freed** On Close, if an incremental snapshot did not finish and a local WAL staging dir was used, the Sink calls fatalFn (`snapshot/sink.go:212–223`), whose default is logger.Fatalf (`snapshot/sink.go:75–76`). The process exits rather than returning an error. If the cause was ENOSPC, the restart re-runs the same snapshot into the same full volume and exits again, so the node stays down until an operator frees or grows disk. This makes free-disk headroom a hard precondition, not a soft one. `snapshot/sink.go:212–223` , `snapshot/sink.go:75–76`	Operators must budget disk headroom for a full WAL snapshot plus the rollover; hitting zero exits the node and it will not self-heal.	[observed] for the fatal path; [inferred] that a still-full disk reproduces the exit on restart
C2	**Credentials are loaded once at process start and cannot be reloaded, so a leaked password cannot be revoked without restarting every node** `main.go:184–187` † calls credentialStore(cfg) a single time and the resulting *auth.CredentialsStore is handed to the HTTP service, the cluster service and the discovery/join clients; nothing watches cfg.AuthFile for changes (`auth/credential_store.go:80–90` reads the file once). To revoke a compromised operator account, an operator must edit the JSON and restart every rqlited in the cluster in turn. This matches the still-open request 1553 in the facts. `cmd/rqlited/main.go:183–219` , `auth/credential_store.go:80–132`	Incident response for a leaked credential is a rolling restart of the whole cluster, not an edit-and-reload.	[observed] single load at main.go:184-187 and auth/credential_store.go:80-90; no reload path found by grep.

#	Constraint	Rationale	How known
C3	**Windows skips the directory fsync behind every atomic rename, so those renames are not power-loss durable** SyncDirMaybe and SyncDirParentMaybe return nil on Windows (`internal/fsutil/fsutil.go:198–220`). They are the durability step after the temp+rename in the snapshot Sink (`snapshot/sink.go:261–263`), the reap plan writer (`snapshot/plan/plan.go:137`), and the file backup client. The code comment (`internal/fsutil/fsutil.go:209–214`) argues the rename is atomic so a rollback after power loss is acceptable, and the state left is internally consistent. The operator consequence stands: on Windows a power loss can roll a snapshot, plan, or backup-metadata rename back to the pre-rename state, so a just-completed snapshot or backup may silently not be there after reboot. `internal/fsutil/fsutil.go:198–220` , `snapshot/sink.go:261–263` , `snapshot/plan/plan.go:137`	Windows operators cannot rely on a completed snapshot or backup rename surviving an unclean power loss; the surrounding state stays consistent, so it is a durability bound, not corruption.	[observed]
C4	**SIGTERM closes the HTTP listener abruptly instead of draining, resetting in-flight requests** `Service.Close` calls `s.httpServer.Close()` , not `Shutdown()` (`http/service.go:418–431`), and main invokes it first thing on shutdown (`cmd/rqlited/main.go:284`). By design — CHANGELOG v10.5.0 line 123 (PR #2833, "Do not perform graceful shutdown of HTTP server, just close it"). Consequence: every deploy, scale-in or node restart severs in-flight queries and any in-progress `/backup` stream with a connection reset; clients must retry. This is a deliberate choice, not a bug, so plan terminationGracePeriodSeconds and client retries around it. `http/service.go:417–431` , `cmd/rqlited/main.go:284` , `CHANGELOG.md:123`	Rollouts are not request-draining, so callers must tolerate connection resets on every node recycle.	[observed] the Close implementation, the call site and the changelog entry naming the decision.

#	Constraint	Rationale	How known
C5	**Container image builds on unpinned bases, ships no HEALTHCHECK, and publishes no image provenance** The Dockerfile builds from `golang:alpine` and runs on `alpine:latest` (Dockerfile:1,50) — both floating tags, so the same source can produce different runs. The image defines no `HEALTHCHECK` (Dockerfile:1-81) and the entrypoint adds a fixed 5s delay under Kubernetes (`docker-entrypoint.sh:111-114`). The release workflow pushes multi-arch images to Docker Hub and GHCR by digest with no `provenance` , SBOM or signing step (`.github/workflows/build-docker-ghcr.yml:61-87`). Consequence: operators cannot pin or attest the image they deploy, and have no container-level health signal. `Dockerfile:1,50-81` , `docker-entrypoint.sh:111-114` , `.github/workflows/build-docker-ghcr.yml:61-87`	Supply-chain and reproducibility are weaker than a production operator can rely on from the published artifact alone.	[observed] the Dockerfile base tags, the absence of HEALTHCHECK, and the build-push steps lacking provenance/SBOM.

5. (c) SLA considerations

#	Behaviour to sign off	Precondition	How known
S1	**Fire-and-forget queued writes are acknowledged before they are durable** POST /db/execute?queue without wait returns 200 with the sequence number as soon as stmtQueue.Write returns (`http/service.go:1501-1506`) and skips the flush wait (`http/service.go:1508-1518`). The queue is in memory only; if the node exits before runQueue applies the batch (`http/service.go:1958` , which uses context.Background()), those statements are gone while the client already saw success. Queued writes are a rate-absorbing path, so losing the in-flight batch is acceptable if an operator relies on ?wait or non-queued writes for durability. `http/service.go:1501-1518` , `http/service.go:1958`	Loss is bounded to writes the client chose not to wait for, which is the documented trade-off of the queue API.	[observed] the early 200, the skipped wait, and the in-memory queue; loss on crash is [inferred]

#	Behaviour to sign off	Precondition	How known
S2	**Steady-state SYNCHRONOUS=OFF means a crashed node's SQLite file may be stale but is rebuilt from the Raft log** rqlite opens every SQLite connection with _sync=0 (`db/state.go:295–308` , MakeDSN always adds opts.Add("_sync","0")), and `db/DESIGN.md:11` ,140 document that committed writes are durable in the Raft log first and the local SQLite file is treated as rebuildable. createDBOnDisk always deletes the WAL and SHM files on open regardless of the fast path (`store/store.go:3318–3331`), so any frames not yet checkpointed are dropped and reapplied from Raft. This is priced, not a defect: an acknowledged write survives on a quorum of Raft logs; an unacknowledged write can be lost; a node that lost power restarts and pays a full restore (fsmRestore deliberately hard-exits after the swap point, `store/store.go:2881–2895`) before serving again. `db/state.go:295–308` , `db/DESIGN.md:11` , `db/DESIGN.md:140` , `store/store.go:3318–3331` , `store/store.go:2881–2895`	Data loss is bounded to unacknowledged writes and never to acknowledged ones, because the Raft log on a quorum is the durability anchor.	[observed] for the DSN, the WAL removal and the hard-exit; [inferred] for the recovery-time cost

6. Process

From the GitHub facts, every listed merge has an empty reviewer list — the mergedPRs entries all carry "reviewers": [], and unreviewedMerges lists ~80 PRs — all authored and merged by the same person (otoolep). There is no second reviewer on any change in the window. The telling part is survival, not count: the unbounded make() on the cluster socket (A1, open issue #2619) and the HTTP server with zero timeouts (A3) are single-line omissions that sat through dozens of these PRs, and the recent commits are dominated by cosmetic fixes ("fix typos", "tidy locals", "fix stale comments" — PRs #2924-2962) alongside real ones. The process signal is a single-maintainer, no-independent-review gate: neither a peer read nor an automated check caught two defects that are the top of this list.

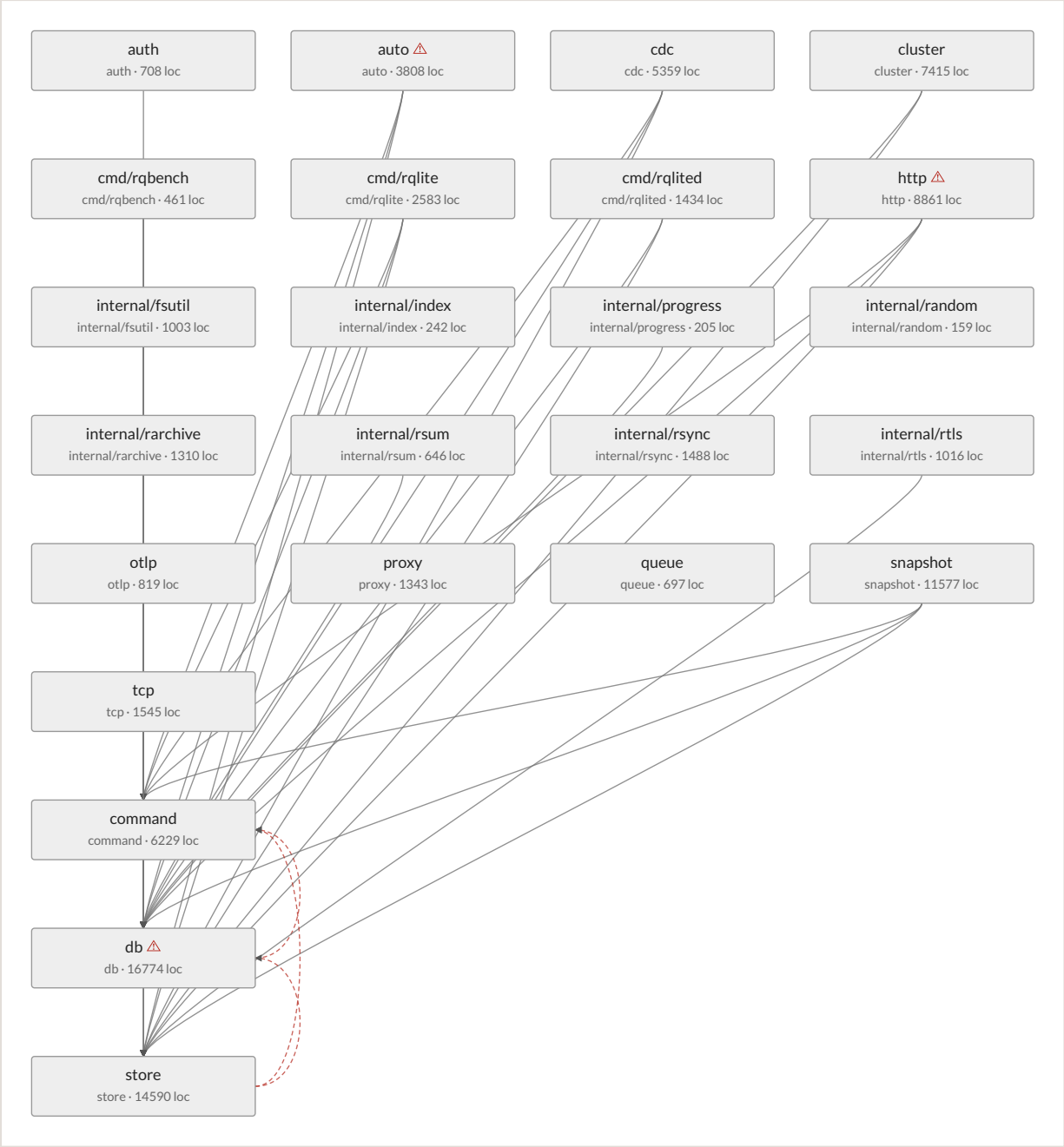
7. What is already good

- Read connections block ATTACH by setting SQLITE_LIMIT_ATTACHED to 0 (db/conn_config.go:125-129), closing the read-only-versus-ATTACH bypass.
- Snapshot restore verifies CRC32 while streaming, so corruption is caught during the copy rather than after it (snapshot/restore.go:52-60).
- Cluster authorization is decided once, before any command acts, and a rejected command closes the connection (cluster/service.go:529-538).
- Password comparison is constant-time via subtle.ConstantTimeCompare (auth/credential_store.go:140), and an empty password for a non-AllUsers user is rejected at

load (auth/credential_store.go:112-117).

- The cluster service fails closed: an unknown command type is never authorized (cluster/service.go:434-435) and a rejected command closes the connection so no handler can act on it (cluster/service.go:529-539).
- TLS configs pin MinVersion to TLS 1.2 (internal/rtls/config.go:146-153), and the container runs as non-root (Dockerfile:78).
- The Raft log is the durability anchor and SQLite is treated as a rebuildable cache: connections open with _sync=0 (db/state.go:295-308) and startup always discards the WAL and re-applies from Raft (store/store.go:3318-3331), so a torn SQLite file is a recovery cost, not a data-loss event.
- Destructive snapshot work is atomic and crash-safe: temp+fsync+rename for snapshots (snapshot/sink.go:245-263) and for the reap plan (snapshot/plan/plan.go:117-138), with an idempotent, resumable reap plan that check() replays or skips on restart (snapshot/store.go:848-918).

8. Code map



Modules from build manifests or directory structure; edges from resolved imports. Red dashed = edge that closes a dependency cycle. ⚠ = open high/critical scanner findings.

9. Recommended next steps

1. Cap and validate the internode length prefix and add a recover on the cluster server and client paths (cluster/service.go:500-502, cluster/client.go:973-980); this is issue #2619 and removes the top blocker. Set a sane maximum message size and reject larger frames with the connection dropped, not the process.

2. Change the container defaults so the image is not insecure by default: bind the bare-binary default (localhost) or require -auth, and document that `-p 4001:4001` must not be published. (docker-entrypoint.sh:22,34; config_flags.go:167,175)
3. Set ReadHeaderTimeout, ReadTimeout, IdleTimeout and WriteTimeout on the HTTP server and wrap bodies in http.MaxBytesReader, including the /db/load and snapshot-restore paths (http/service.go:346-348, 908; snapshot/restore.go:29-32).
4. Make the queued-write retry-limit break return a real failure to the caller and stop advancing seqNum / closing the flush channel as success (http/service.go:1980-1996), so a ?wait client cannot be told a dropped write succeeded.
5. Introduce one documented break-glass start mode and make integrity failures quarantine the bad artifact instead of Fatalf crash-looping; stop hardcoding str.NoVerifyDB = true (cmd/rqlited/main.go:462; snapshot/store.go:276-278, 961-970; store/store.go:676-680).
6. Add a required second reviewer (or an automated gate) for merges — every PR in the window merged unreviewed by a single author, which is how a one-line unbounded allocation and a timeout-less server reached the release.