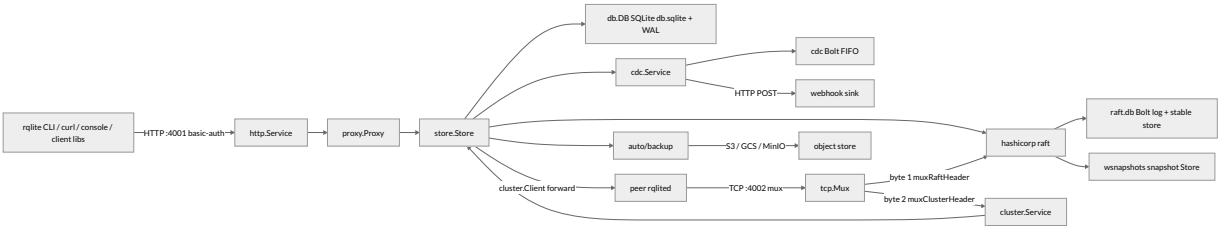
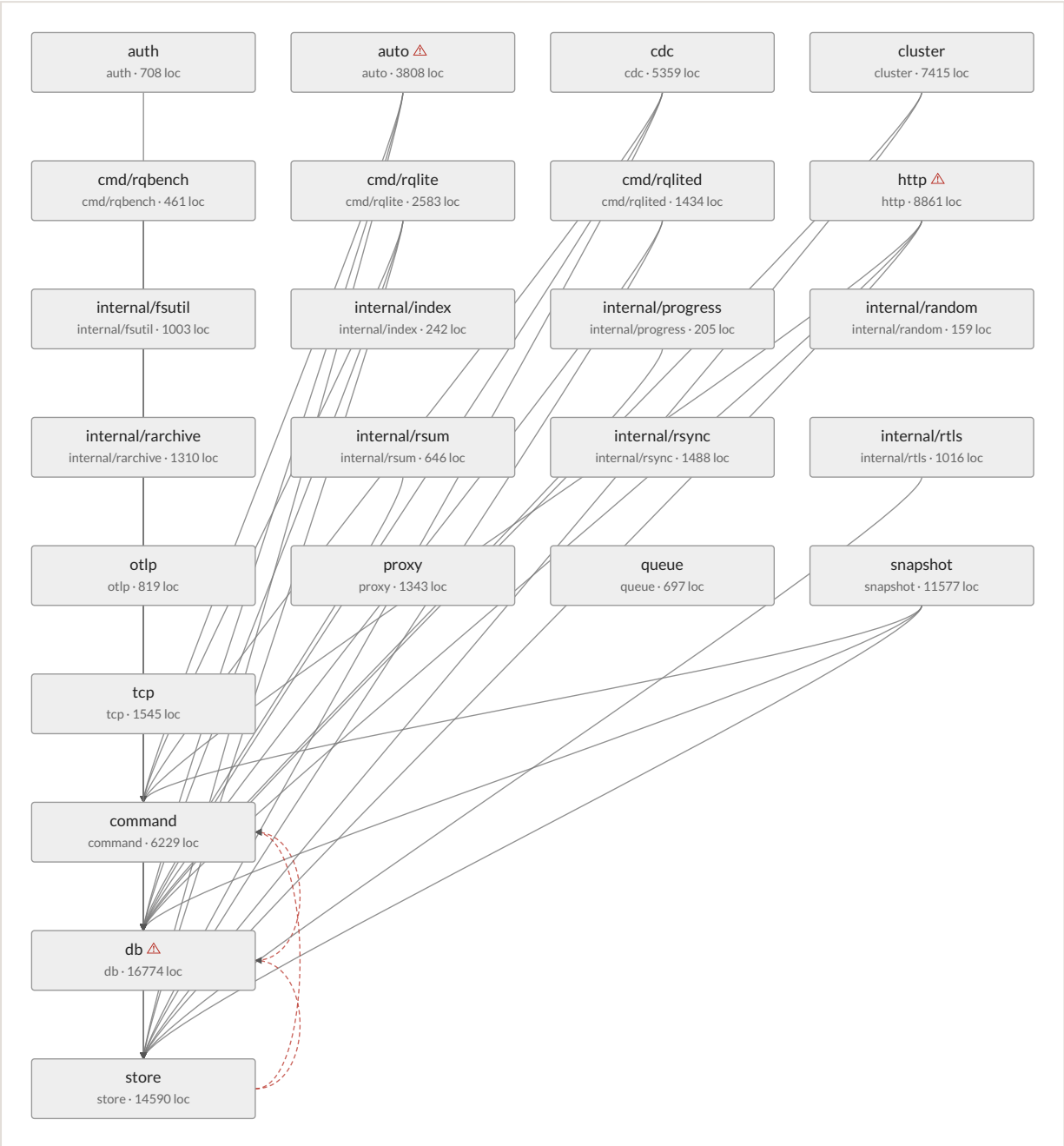
rqlite at 8b10e2d — production-readiness whiteboard tour

rqlite is a distributed SQLite: one Go binary `rqlited` runs a Raft group whose replicated state machine applies SQL to a local SQLite file, and the Raft log itself is stored in BoltDB. This tour reads commit `8b10e2d927ca2835cec46456c5e197968dccb38c` (2026-10-28, module `github.com/rqlite/rqlite/v10`, `go 1.27.0` [observed] go.mod:1,3; version 10, CHANGELOG.md top entry v10.6.6). It shows where state lives, how a request is routed by port and byte, the write and read paths, the internals of each component, the lifecycles, where identity is checked on every port, the deployment shape, and the upgrade path — then the design bets and a comparison with etcd and Litestream. Claims are tagged [observed] (read in code at this commit, cited path:line), [computed] (arithmetic on observed values), or [inferred]. As read at commit `8b10e2d` (2026-10-28). ⚠ marks a structural weak spot.

1. The system on one page

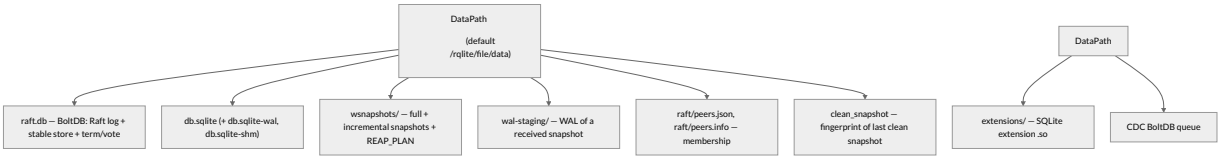


2. Code map



Laid out from build manifests, directory structure and resolved imports (deterministic, not model-drawn). Red dashed = edge that closes a dependency cycle; ⚠ = open high/critical findings.

3. Where each kind of state lives

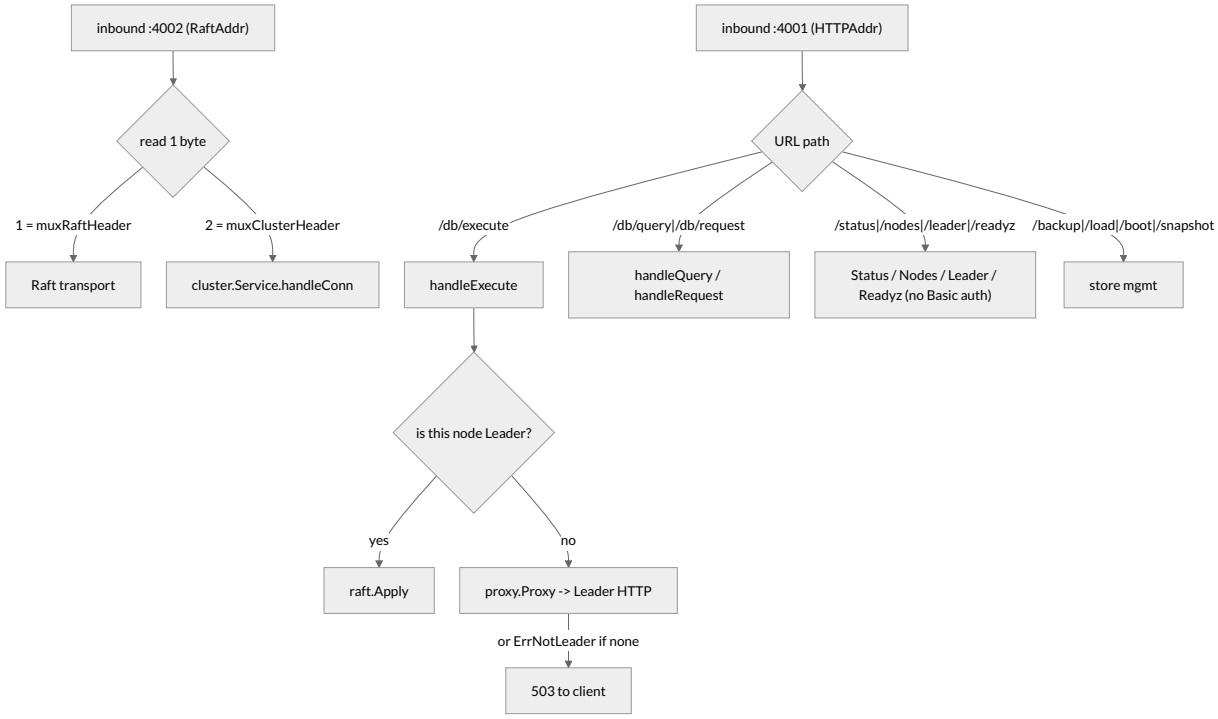


State	File / store	Written by	Anchor
Raft log, term, vote, stable store	`<dir>/raft.db` (BoltDB)	hashicorp raft	[observed] store/store.go:463
Applied database	`<dir>/db.sqlite` + `-wal` + `-shm`	db.DB on each applied command	[observed] store/store.go:461,467
Snapshots (full + incremental)	`<dir>/wsnapshots/`	snapshot.Store	[observed] store/store.go:464
Snapshot WAL staging	`<dir>/wal-staging/`	snapshot receive path	[observed] store/store.go:465
Membership	`<dir>/raft/peers.json`, `peers.info`	cluster/bootstrap + store	[observed] store/store.go:466-467
Clean-snapshot fingerprint	`<dir>/clean_snapshot`	store.Open	[observed] store/store.go:468
SQLite extensions	`<DataPath>/extensions`	loader at start	[observed] cmd/rqlited/main.go:593
CDC outbound queue	BoltDB under DataPath	cdc.Service	[observed] cmd/rqlited/main.go:203

Two files hold everything that matters: `raft.db` (the log, the source of truth) and `db.sqlite` (a derived, reproducible projection). Delete `db.sqlite` and the node rebuilds it by replaying `raft.db` or restoring the newest snapshot [inferred]. Delete `raft.db` and the node loses its vote history and log [inferred].

- ⚠ store/store.go:316-359 — one `Store` struct holds the Raft node, the SQLite DB, the snapshot store, membership, CDC, and about ten atomic counters/indexes in ~80 fields; every lock and lifecycle decision for the node crosses this one struct.
- ⚠ the SQLite file is not itself checksummed against the log; correctness depends on `fsmApply` never returning an error mid-command (see fsmApply ⚠ below).

4. How a key/request is routed



Two listening sockets, one muxed internode port.

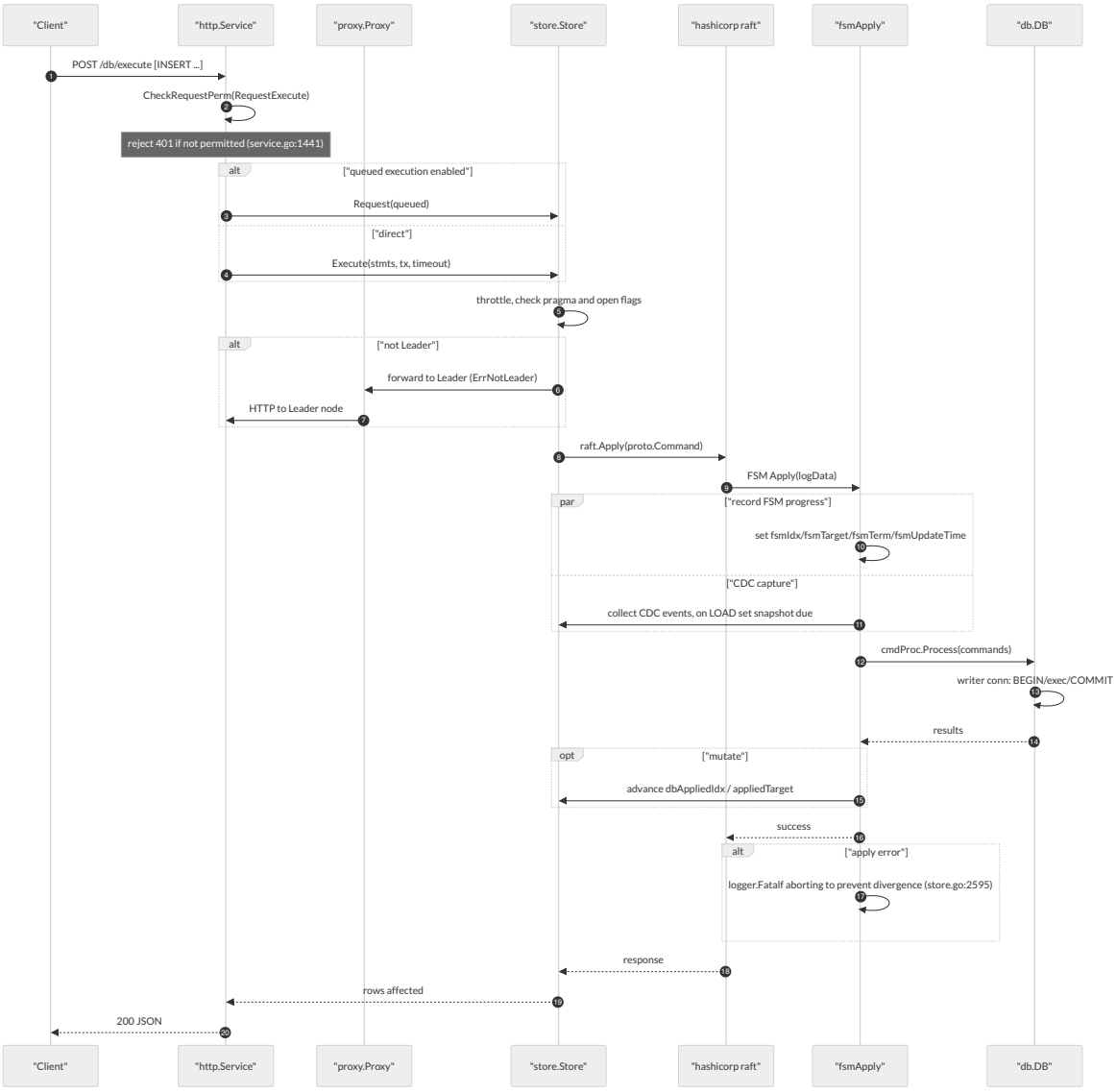
Input	Decision	Result	Anchor
TCP `:4002` first byte `1`	Raft RPC	to `raft.NetworkTransport`	[observed] cluster/service.go:70; tcp/mux.go
TCP `:4002` first byte `2`	cluster RPC	`cluster.Service`	[observed] cluster/service.go:73
HTTP `/db/execute`	write	`handleExecute`	[observed] http/service.go:1438
HTTP `/db/query`, `/db/request`	read	`handleQuery`, `handleRequest`	[observed] http/service.go:1600,1694
HTTP `/status`, `/nodes`, `/leader`, `/readyz`	status	served without Basic auth	[observed] http/service.go:476-537

There is no key hash ring. Any node accepts a write; if it is not the Leader it forwards over HTTP to the known Leader (`proxy.Proxy.Execute` , proxy/proxy.go:134) or returns `ErrNotLeader`

(proxy/proxy.go:23). [inferred] This makes every node a valid entry point but adds one HTTP hop on every non-Leader write.

- ⚠ http/service.go:458-537 — routing is a long chain of `strings.HasPrefix(r.URL.Path, ...)` checks, so any path that merely starts with `/db/executefoo` or `/statusthing` reaches that handler.

5. Main write path



Numbered as in the diagram.

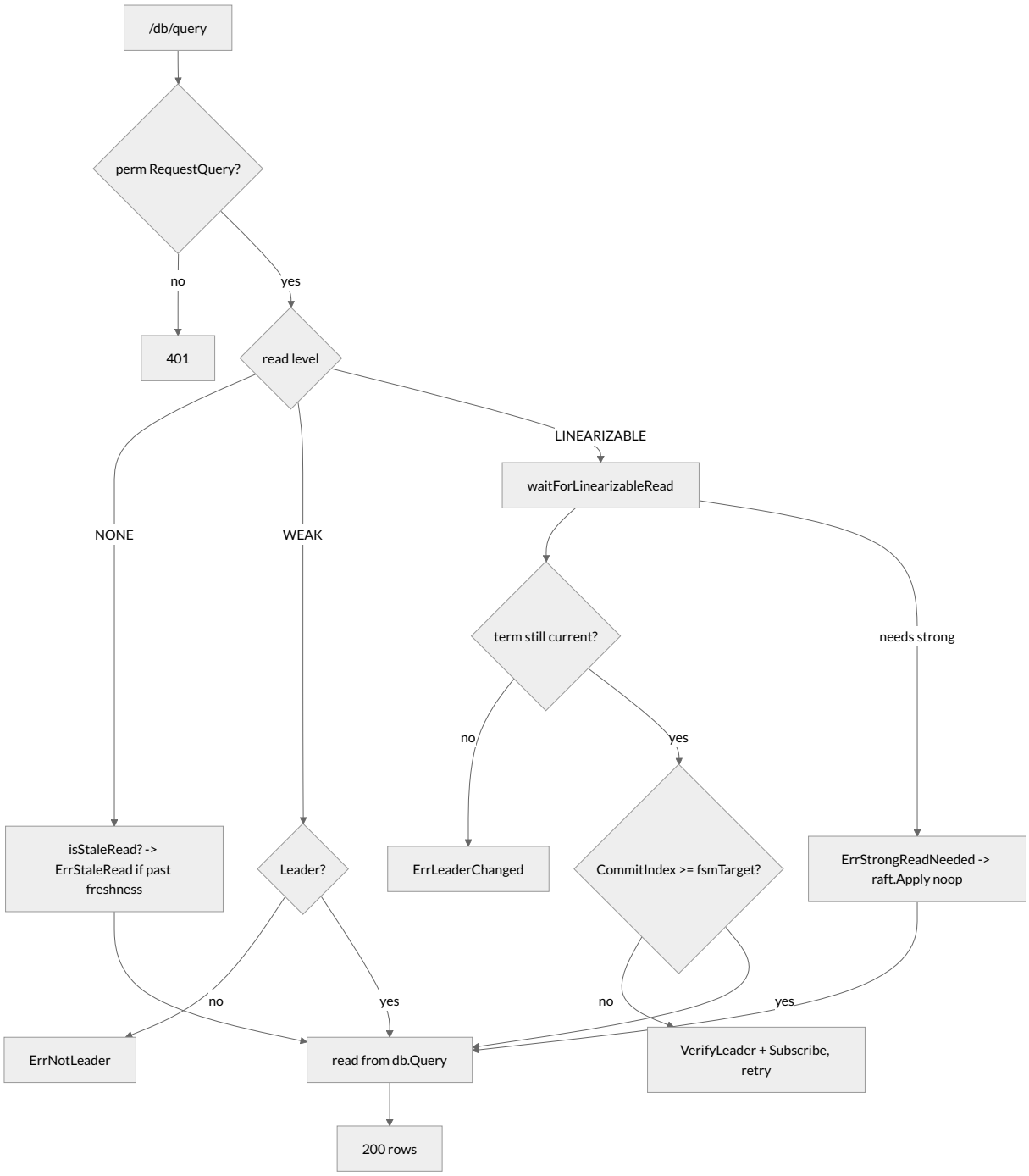
#	Step	Anchor
1	Permission check	[observed] http/service.go:1441
2	`store.Execute` throttles, checks pragma + open state	[observed] store/store.go:1474-1494
3	Non-Leader returns `ErrNotLeader`	[observed] store/store.go:1491
4	Marshal `proto.Command`, `raft.Apply`	[observed] store/store.go:1506-1517

#	Step	Anchor
5	`fsmApply` records index/term then processes	[observed] store/store.go:2562-2592
6	Mutating command advances applied index	[observed] store/store.go:2598-2601
7	Apply error aborts the process	[observed] store/store.go:2595

Key consequence: an error inside the FSM does not fail the request — it kills `rqlited` to stop replicas diverging [observed] store/store.go:2595. A malformed or unsupported statement that reaches the FSM is therefore a node-availability event, not a 500 [inferred].

- ⚠ store/store.go:2595 — `logger.Fatalf` inside `fsmApply` stops the process on any apply error; a single node stops, and every node will stop on the same log entry.
- ⚠ store/store.go:1482-1491 — `Execute` checks `ctx` and leadership before the throttle/pragma gates have fully drained, so a write can still be rejected as non-Leader under load rather than queued.

6. Main read path — healthy and degraded



Four levels, chosen automatically from a live Leader's `CommitIndex − AppliedIndex` lag.

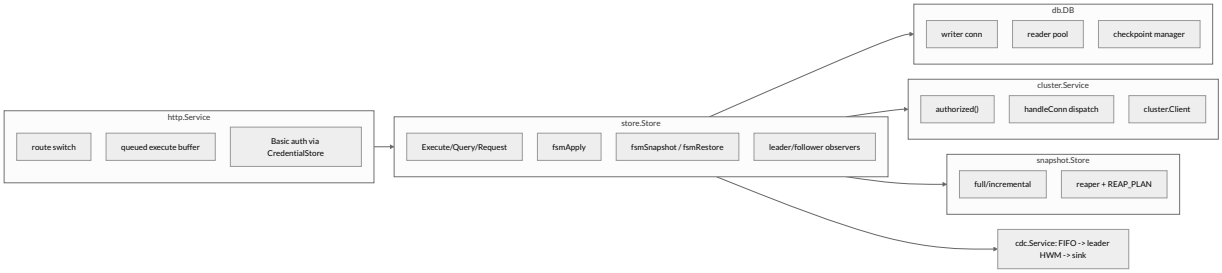
Level	Guarantee	Anchor
NONE	stale read unless `Freshness` set; not stale iff `now−commitTime <= Freshness`	[observed] store/store.go:1608,2533

Level	Guarantee	Anchor
WEAK	must run on a Leader	[observed] store/store.go:1605
LINEARIZABLE	term guard, confirm Leader against quorum, wait for FSM to reach commit index	[observed] store/store.go:2486-2530
STRONG	LINEARIZABLE + a full `raft.Apply` round-trip	[observed] store/store.go:1556-1589

Degradation: as replication lag grows, LINEARIZABLE reads block in `waitForLinearizableRead` and can be upgraded to STRONG, which doubles the round-trips [observed] store/store.go:1556. If the node loses leadership mid-wait it returns `ErrLeaderChanged` and the client must retry [observed] store/store.go:2498,2515.

- ⚠ store/store.go:2533-2560 — `isStaleRead` compares read state against `fsmTarget`; during snapshot restore the target can regress, so a NONE read can serve a pre-snapshot page until restore completes [inferred].

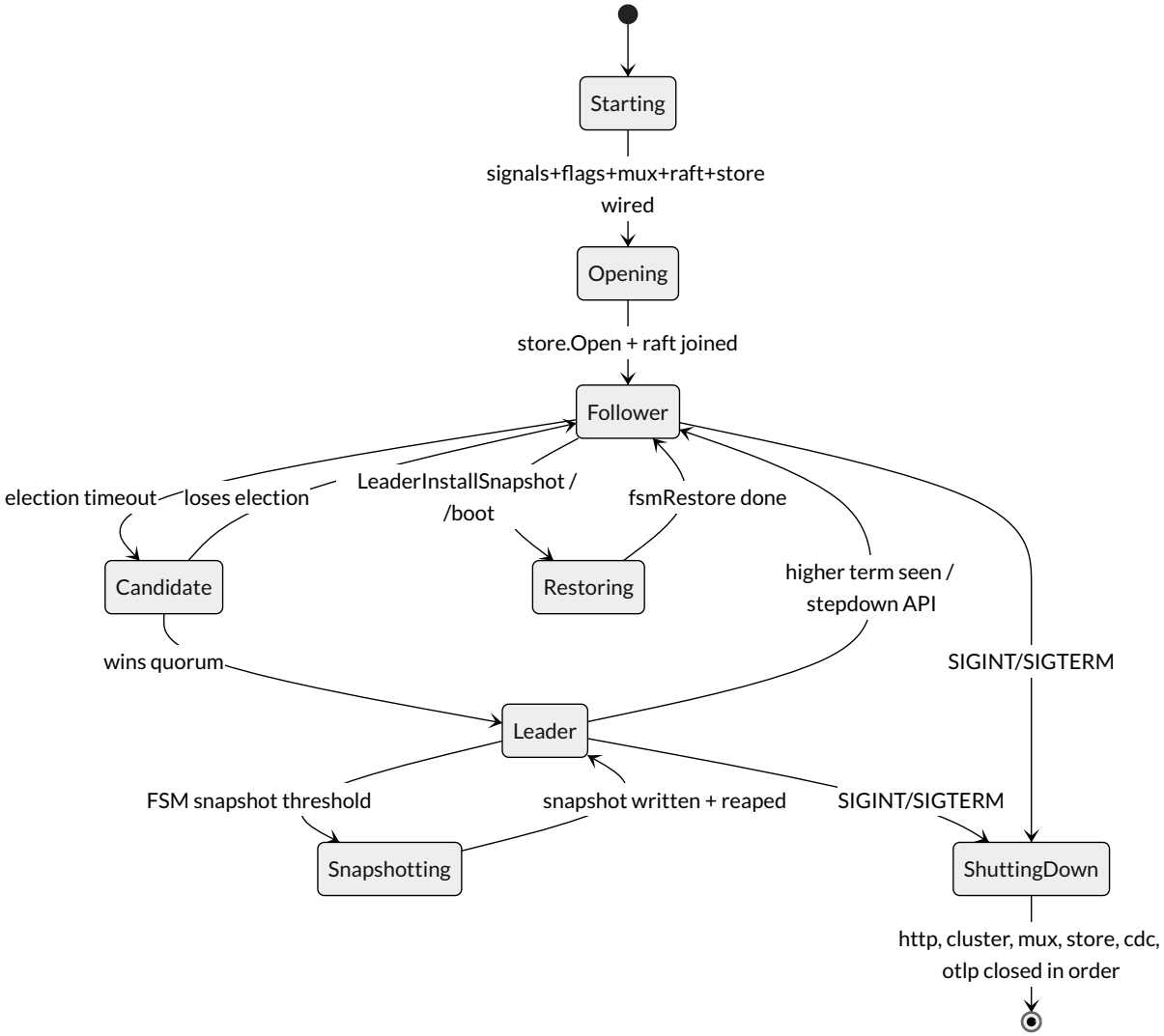
7. Inside each major component



Component	Single responsibility	Anchor
`http.Service`	HTTP surface, per-handler perm check, queued-execute buffer	[observed] http/service.go:1438-1637
`proxy.Proxy`	Forward writes/reads to the Leader, else `ErrNotLeader` / `ErrLeaderNotFound`	[observed] proxy/proxy.go:23,27,110,134
`store.Store`	Owns Raft node + FSM; mediates every read/write level	[observed] store/store.go:316-490
`db.DB`	One writer connection + a reader pool, `DisableAttach` on readers	[observed] db/db.go:189-227
`cluster.Service`	Internode RPC over the muxed port; `authorized()` gate	[observed] cluster/service.go:401,486
`snapshot.Store`	Full + incremental snapshots, reaping with a durable plan	[observed] snapshot/store.go:28,62,857
`cdc.Service`	Leader-only event emission through a bounded FIFO to one sink	[observed] cdc/service.go:326,515,568

- ⚠ db/db.go:189-227 — every reader connection has `DisableAttach = true`; a query that needs `ATTACH` (or a temp table across connections) silently loses that ability on the read path.
- ⚠ cluster/service.go:401-437 — `authorized()` is a hand-maintained switch over command type; a new RPC type that is not added here is denied by the `default` branch, so the failure mode is a rejected call, not an open door.

8. Lifecycle and state machines



Two overlapping machines: the Raft role (Follower/Candidate/Leader) and the data-lifecycle (Snapshot, Restore).

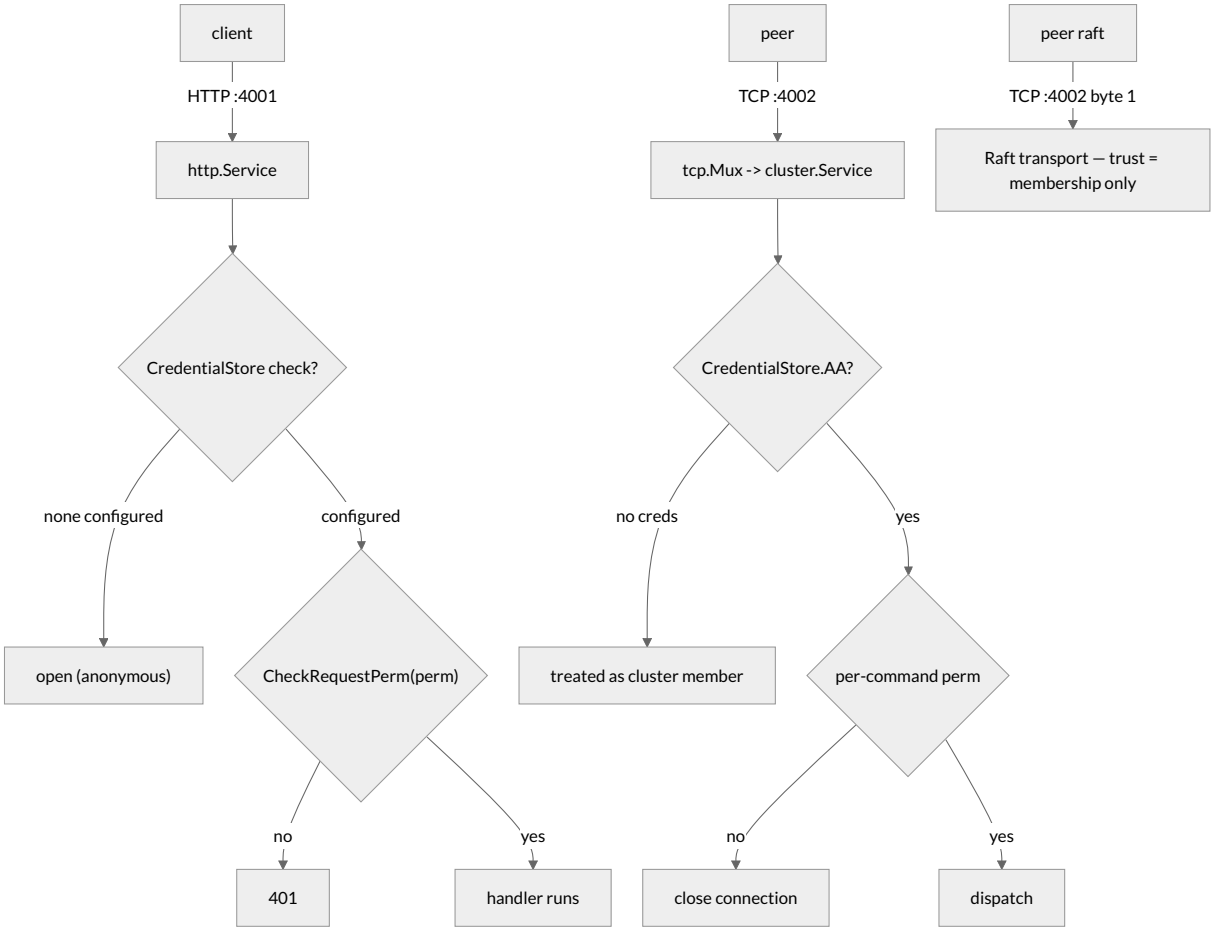
Startup order is fixed [observed] cmd/rqlited/main.go:71-280: signals -> flags -> mux (95) -> raft layer (108) -> extensions (117) -> CDC config (128) -> store (148) -> auto-restore (154) -> credentials (184) -> cluster service (190) -> cluster client (196) -> CDC service (203) -> proxy (211) -> HTTP (219) -> `str.Open()` (225) -> status (230) -> join/create (251) -> auto-backup (262) -> OTLP (271) -> block (280).

Shutdown order is the reverse [observed] cmd/rqlited/main.go:284-327: HTTP (284) -> cluster service (285) -> optional remove (287) -> optional stepdown (299) -> mux listener (303) -> mux (310) -> `str.Close(true)` (314) -> query logger (320) -> OTLP (327).

- ⚠ cmd/rqlited/main.go:284-285 — HTTP is closed before the cluster service; in-flight writes already accepted by HTTP can be mid-`raft.Apply` when the cluster service stops.

- ⚠ snapshot/store.go:857-894 — a crashed reap is resumed from `REAP_PLAN` at the next `Open`; if the plan is unreadable the `Open` fails, so a corrupt plan file keeps the node down.

9. Who checks identity where (every port)



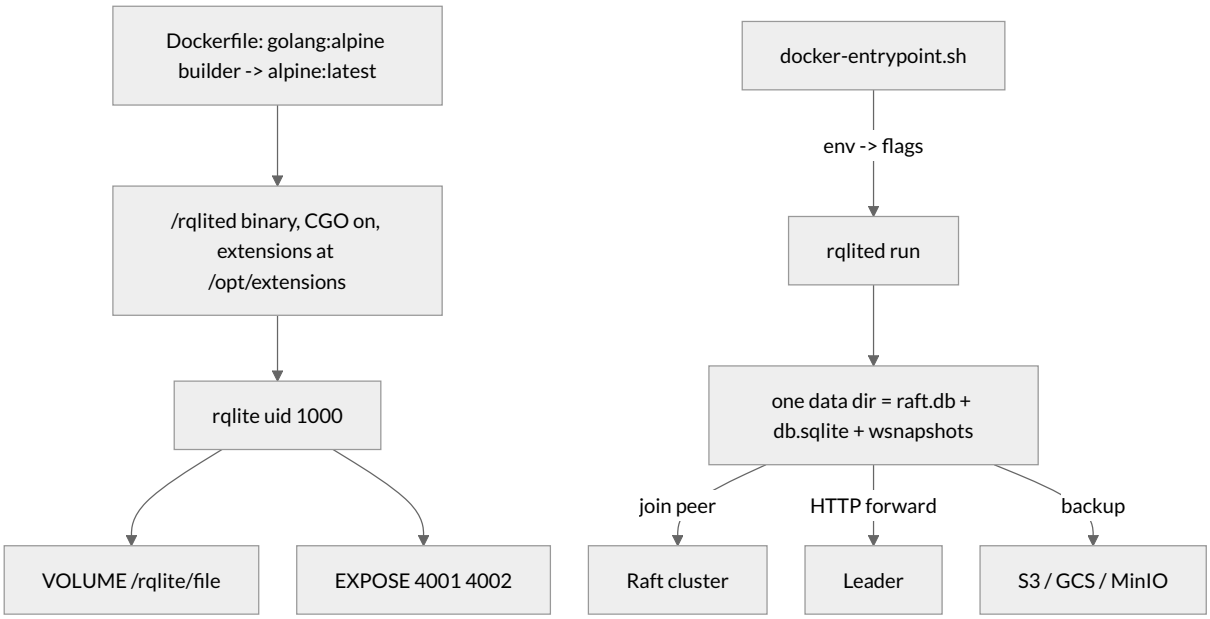
Port	Default	Identity check	Anchor
HTTP `:4001`	`0.0.0.0:4001`	Basic auth via CredentialStore `CheckRequestPerm` per handler	[observed] docker-entrypoint.sh; http/service.go:1441,1603
HTTP status paths	same	served with no credential check	[observed] http/service.go:476-537
Internode cluster RPC	`:4002` byte 2	`CredentialStore.AA` per command, or none if no credentials configured	[observed] cluster/service.go:401,533
Internode Raft RPC	`:4002` byte 1	no application-layer credential; trust is Raft membership + optional mTLS	[observed] cluster/service.go:70; cmd/rqlited/main.go:556-569

Port	Default	Identity check	Anchor
HTTP TLS	off unless certs given	optional mTLS via `ClientVerify`	[observed] cmd/rqlited/main.go:519-523

Constant-time password compare: `subtle.ConstantTimeCompare` [observed] auth/credential_store.go:140; overall auth decision in `AA` [observed] auth/credential_store.go:188.

- ⚠ cluster/service.go:403-404 — `GET_NODE_META` is authorized unconditionally, so any peer that reaches `:4002` can read a node's URL, version, and commit index without credentials.
- ⚠ cmd/rqlited/main.go:567-569 — with no `NodeX509*` flags the mux runs plaintext, and a past advisory in this line (CHANGELOG.md:10-11) was exactly an unauthenticated `:4002` user downloading a backup; the fix added authorization, but the port is still unencrypted by default.
- ⚠ the Raft transport on byte 1 has no password check; anyone with TCP access to `:4002` and the first byte `1` is inside the consensus plane unless mTLS is configured [inferred].

10. Deployment shape

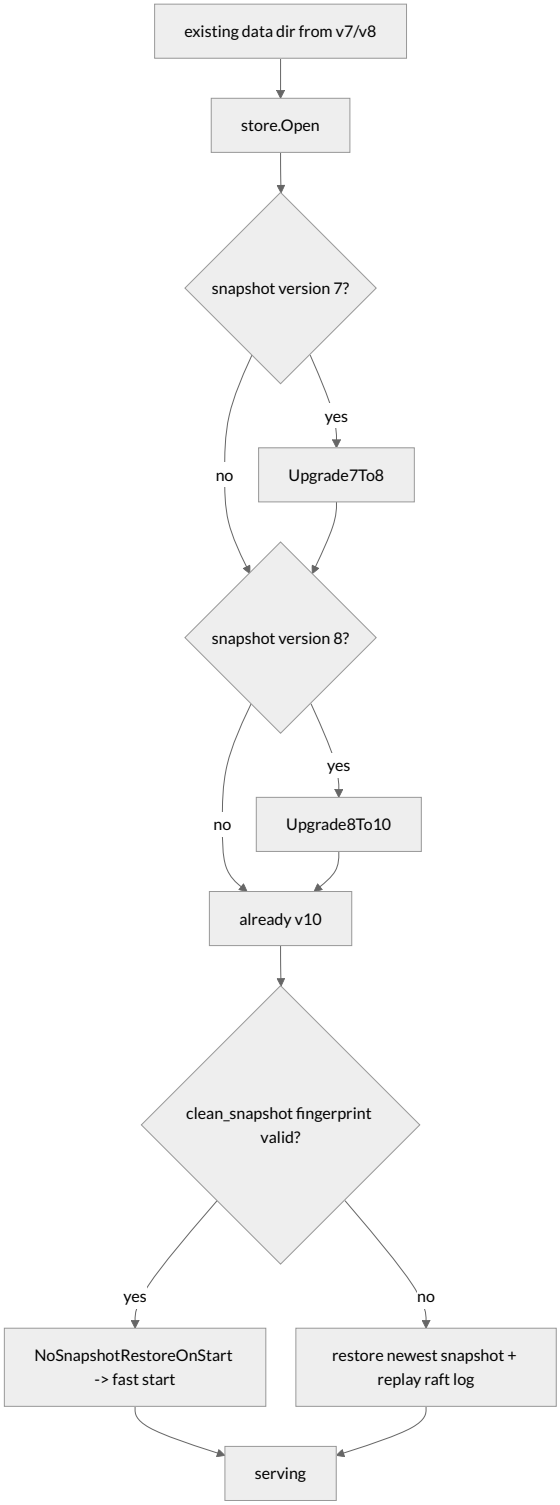


Aspect	Value	Anchor
Build	two-phase, `golang:alpine` -> `alpine:latest`, CGO enabled	[observed] Dockerfile
Runtime user	non-root `rqlite`, uid 1000	[observed] Dockerfile
Persistent state	single `VOLUME /rqlite/file`	[observed] Dockerfile
Ports	`EXPOSE 4001 4002`	[observed] Dockerfile
Entrypoint	`docker-entrypoint.sh` maps env, then `CMD ["run"]`	[observed] Dockerfile; docker-entrypoint.sh
Defaults	`HTTP_ADDR=0.0.0.0:4001`, `RAFT_ADDR=0.0.0.0:4002`	[observed] docker-entrypoint.sh

Operator consequence: one mounted directory holds the log and the database; the volume must be durable and single-writer per node, because two `rqlited` processes on the same `raft.db` corrupt state [inferred].

- ⚠ Dockerfile — the default binds `0.0.0.0` for both ports, so a container started without a network policy exposes the unauthenticated Raft port directly to the network.

11. Upgrade path



Step	What happens	Anchor
7 -> 8	`Upgrade7To8` rewrites the snapshot directory	[observed] store/store.go:582

Step	What happens	Anchor
8 -> 10	`Upgrade8To10` rewrites the snapshot directory	[observed] store/store.go:585
Clean fingerprint	if the last snapshot matches `clean_snapshot`, skip restore	[observed] store/store.go:606-636
Restore	copy newest snapshot into `db.sqlite`, then replay the Raft log	[observed] store/store.go:2829

Go floor is 1.27 [observed] go.mod:3, so an operator upgrading rqlite must also move the build toolchain [inferred]. The snapshot upgrade functions run at `Open`, before serving, so a first start on old data is slower but leaves the node ready [observed] store/store.go:579-587.

- ⚠ store/store.go:582-586 — snapshot upgrades run only at `Open`; there is no dry-run or version precondition check before the rewrite, so a failed upgrade leaves the on-disk snapshots partially converted.

The bets in one table

Bet	What it buys	What it costs
Replicate the Raft log, not the SQLite file.	Deterministic, byte-identical replicas; every node is a full copy; reads and writes go to the same engine.	One writer node does all writes; every statement is serialized through the FSM; an FSM error must abort the node to avoid divergence (store/store.go:2595).
One TCP port muxed by first byte for both Raft and cluster RPC.	A single firewall rule and a single listener for the consensus plane; no separate Raft port to manage.	The muxed port carries unauthenticated Raft traffic by default, so exposure is a single mistake away (cmd/rqlited/main.go:567-569); a past advisory exploited exactly this (CHANGELOG.md:10-11).
Any node is a valid entry point; non-Leaders forward over HTTP.	Clients need no leader discovery; a load balancer in front of every node works.	Every write to a Follower pays an extra HTTP hop; leader churn surfaces as `ErrNotLeader` retries (proxy/proxy.go:23).
Per-command authorization on the internode channel instead of network isolation alone.	A peer that can reach `:4002` still cannot run commands it lacks permission for (cluster/service.go:401).	A hand-maintained switch must list every RPC type; `GET_NODE_META` is deliberately left open (cluster/service.go:403).
Snapshots with a durable, resumable reap plan.	A crash mid-reap is recovered at restart instead of leaking disk (snapshot/store.go:871).	An unreadable `REAP_PLAN` blocks `Open`; the node stays down until an operator clears it (snapshot/store.go:873-875).
At-least-once CDC through a bounded leader-only FIFO.	Events survive a Raft snapshot restore (PR #2990); a slow sink cannot stall writes.	Consumers must dedupe; one sink per node; the queue is BoltDB under DataPath (cmd/rqlited/main.go:203).
Bake extensions into the image at build time.	Reproducible deployments; no runtime downloads.	Adding or updating an extension requires a new image and a rolling restart (cmd/rqlited/main.go:593).

Compared with etcd and Litestream

	etcd	Litestream	rqlite
Data model	key-value tree	streams a SQLite WAL to object storage	replicated SQLite database
Write path	Raft proposes a key mutation	local SQLite write, then async WAL ship	Raft proposes a SQL command, FSM executes on each node
Entry point	any member forwards	single writer; no cluster	any node forwards to Leader over HTTP
Consistency knob	linearizable or serializable	none; replication lag is unbounded	NONE / WEAK / LINEARIZABLE / STRONG per query
Read scale	any member	one node	any node; readers use a separate connection pool
Storage	BoltDB WAL + snapshot	SQLite + object store	BoltDB raft.db + db.sqlite + wsnapshots
Failure on FSM error	n/a (no SQL)	n/a	process aborts to avoid divergence
Version at this commit	v3.x	v0.3.x	v10.6.6 (Go 1.27, CHANGELOG.md)

etcd and rqlite share the same consensus substrate (hashicorp/raft, BoltDB) but differ in the state machine: etcd replicates a key-value tree, rqlite replicates SQL. Litestream keeps one SQLite file and asynchronously ships its WAL, so it gives durability without write scale-out; rqlite gives both, at the cost of an FSM that must never fail mid-statement [inferred].