Zaxonlite

Replicated SQLite on Multi-Paxos: WAL frames as the unit of consensus, a journal you can trust, and one implementation from an embedded durable node to role-aware clustered deployments

```
execute -> capture frames -> persist payload -> append
journal + fsync -> confirm -> committed -> acknowledge
```

Vikrant Rathore · Ronak Rathore

with the paxos-zig library and the `zaxon` command line

July 2026

About this book

This book documents Zaxonlite. Zaxonlite is an embeddable SQLite service replicated by the paxos-zig Multi-Paxos library. Its front door is the `zaxon` command line, one binary that is both server and client. The book runs the system before it explains it. Chapter 1 takes you from a fresh build to a three-voter cluster that survives a killed leader. Chapter 2 covers every `zaxon` command. Chapter 3 states what those commands promise. Part II then explains how the machine works. Part III moves the same machine into your own process, in Zig or through the C ABI. Part IV covers operating it. Part V is reference and evidence.

Five readers use this book, and each gets a guide of their own. All five start with the quickstart in chapter 1. Running the system first is the fastest way to build a correct mental model of it.

1. **The Zig developer embedding one durable node** wants a crash-consistent SQL store in-process, with no network. Chapter 9 is the guide, built around `Node.open`.
2. **The Zig developer embedding a cluster member** owns the listener, peers, and role inside one process. Chapter 10 is the guide, built around `Embedded.open`.
3. **The C ABI embedder** links `zaxonlite.h` from another language. Chapter 11 carries the same guarantees across the opaque-handle boundary.
4. **The operator** deploys `zaxon serve` with voters and learners. Chapters 13 and 15 supply the failure playbook and the file formats.
5. **The client or gateway developer** speaks the TCP RPC protocol. Chapter 12 covers leader redirects and the current shared-secret streams.

The implemented vertical slice is tested, not merely described. Its evidence includes unit tests, single-process durability tests, and a three-process loopback cluster scenario with a SIGKILL failpoint. It also includes a CLI contract test, a C ABI smoke test, seeded property fuzzing, a soak run, role, gateway, and adverse-network integration tests, and benchmarks. Protocol v6 uses mutual TLS for production, binds peer node IDs, encrypts traffic, confines the disclosed PSK-only development mode to numeric loopback, quorum-confirms transferred checkpoint proofs, and pipelines durable phase-two storage. It also carries the bounded one-time token/CSR enrollment exchange for nodes already in the static registry. The 10,000-crash, 100-run, and 1-GiB stress targets are explicitly deferred. Where this book states a current guarantee, chapter 17 names its evidence.

> **Reading order**
>
> Run first, understand second, embed third. Part I gets a cluster running on your machine and states its guarantees. Part II explains the machine: architecture, WAL replication, storage, clusters, and consistency. Part III embeds it in your own process. Part IV operates it. Part V holds the formats, the desk reference, verification, and conformance.

Contents

How to Read This Book

Learning contract

By the end of this chapter you should be able to pick the reading route that matches how you will use Zaxonlite, name the entry-point symbol or command your route is built around, and state what the repository supplies versus what your deployment must still add.

Zaxonlite is one implementation serving five kinds of user. The five do not need the same chapters. They do share one starting point. Everyone begins with chapter 1, the quickstart, at a shell. Ten minutes there gives you a durable database, a three-voter cluster, and a leader kill that lost nothing. That experience is the mental model every later chapter refines. Reading about failover is abstract. Having caused one is not.

Chapters 2 and 3 finish the shared ground. Chapter 2 is the full `zaxon` command reference. Chapter 3 turns what you saw at the shell into stated guarantees. After chapter 3 the routes diverge. Each route is organized around a symbol or command that exists in the repository today. When a chapter names an API, the API is real.

The five readers

Reader	Entry point	Route after chapters 1–3
Embedded Zig, one node	`Node.open`	Chapter 5 for WAL replication and chapter 6 for storage and recovery. Then chapter 9, the embedded-node guide. Then chapter 8 for sessions and read levels, and chapter 13 for backup and integrity checking.
Embedded Zig, cluster	`Embedded.open`	The one-node route, plus chapter 7 on role-aware clusters before the chapter 10 guide. The facade owns the listener, peer senders, and tick loop. Chapter 7 explains what those threads are doing.
C ABI embedder	`zaxonlite_open`	Chapter 8 first, because the session contract survives the ABI unchanged. Then chapter 11, the C ABI guide. The Zig guides are optional background. The C surface is a strict subset with JSON results and `zaxonlite_last_error`.
Operator of `zaxon serve`	`zaxon serve`	Chapter 7 on clusters, then chapter 13 on operations, then chapter 15 so you recognize every file in a data directory. Chapter 17 says what the failure playbook has actually been tested against.
RPC client or gateway developer	`Connection.open`	Chapter 8 on consistency, then chapter 12, then the wire-frame section of chapter 15. The gateway is byte-transparent. Authentication and frame integrity stay end-to-end between client and storage node.

API anchor: `Node.open`

Opens one durable node in-process: a data directory, a journal, and a SQLite image, with no network. Empty `members` means a single-member configuration of just `node_id`. in `zaxonlite/src/node.zig`

API anchor: `Embedded.open`

> Opens the transport-owning facade: one node plus its TCP listener, peer connections, and tick thread, configured from a member list with roles. The same API serves one through nine voters plus non-voting replicas. in `zaxonlite/src/embedded.zig`

The C entry points are `zaxonlite_open` and `zaxonlite_cluster_open` in `zaxonlite/src/capi.zig`. They are declared in `zaxonlite/include/zaxonlite.h`. The client library entry point is `Connection.open` in `zaxonlite/src/client.zig`, and the gateway is `gateway.serve` in `zaxonlite/src/gateway.zig`.

> **Predict before reading on**
>
> Decide now: which of the five readers are you, and which guarantee do you most depend on? Durability, exactly-once retry, or read freshness? Write the answer down. Chapter 8 will test it.

What the repository supplies, and what you must build

The honest boundary matters as much as the feature list. The repository supplies the replicated database itself. That means the node and facade libraries, the C ABI, and the `zaxon` command line with its serve and client modes. It also means shared-secret, integrity-protected TCP framing, a mutual TLS 1.3 transport with per-node certificates, snapshots, logical backup streaming, `zaxon integrity-check`, `zaxon recover`, and `zaxon status --json` for automation. The shared secret alone is not a production identity boundary; the TLS mode is, once you provision its certificates.

Your deployment must still provide four things. The release limits state them directly.

1. **Application authentication.** Zaxonlite has one database principal. Your application authenticates its users and decides which operations and SQL they may issue.
2. **A production transport.** Use the embedded API in process, the owner-only Unix-domain socket (`--listen unix:<path>`) for local service, or mutual TLS with per-node certificates for TCP clusters. Bootstrap the CA and one issuer identity out of band; an authenticated operator can then issue a short-lived one-time bundle so a configured node generates its key and CSR locally. The PSK mode proves only shared secret possession and does not encrypt; `--dev-psk` confines that tradeoff to a numeric-loopback local development cluster.
3. **Monitoring and alerting.** The node reports status; nothing in the repository watches it. After a fatal storage error the host must stop voting and serving. Something of yours must notice and page a human.
4. **Orchestration.** Process supervision, restart policy, and voter replacement are operator work. Automatic voter replacement is roadmap, not product. A failed voter stays failed until you act.

> **Scope is part of correctness**
>
> The guarantees in this book are conditional on that boundary. "Acknowledged means durable and decided" holds on POSIX filesystems with working fsync, and on NTFS from Windows 10 1809 onward, which a node verifies before it opens. Wire confidentiality holds only inside TLS, and offline-media confidentiality depends on OS disk or filesystem encryption. Availability after a voter loss holds only if your orchestration replaces the voter.

How the chapters build on each other

The book has five parts. Part I (chapters 1–3) gets you running: the quickstart, the `zaxon` CLI, and the product guarantees. Part II (chapters 4–8) explains how it works: architecture, WAL replication, storage, clusters, and consistency. Part III (chapters 9–12) embeds it: one node in Zig, a cluster in Zig, the C ABI, and clients and gateways. Part IV (chapters 13–14) operates it, with a playbook and worked examples. Part V (chapters 15–18) holds the formats, the desk reference, verification and benchmarks, and conformance.

The dependency structure is deliberate. Part I gives you the experience and the vocabulary: descriptor, payload, journal, chosen slot. Within Part II, storage and clusters are independent of each other, but both feed chapter 8. Consistency is the last chapter that states guarantees. Everything after it shows how to use them or how to check them. The guides assume their Part II chapters. Chapter 10 leans on chapter 7, and chapter 12 leans on chapter 8. The format chapter is normative. When prose and format disagree, chapter 15 and the frozen format document win. Chapter 17 closes the loop by mapping every stated guarantee to the test that exercises it. Chapter 18 states what a compatible implementation must do.
Read the callouts the way the theme presents them. A `decision` callout records a design commitment and its consequence. A `warning` callout marks a boundary you can silently cross. Checkpoints and exercises exist for one reason. Recognizing a guarantee on the page is not the same as rebuilding it from memory during an incident at 3 a.m.

This book and the zig-paxos book

Zaxonlite is a host of the paxos-zig library, and this book deliberately does not re-prove consensus. Why a quorum intersects, why a promised ballot never moves backward, and why a later leader must adopt the highest accepted value are all proved in the zig-paxos book in `docs/book/`. So are the effect-machine design of `paxos.Protocol` and the sealed epochs of `paxos.ReplicatedLog`. This book treats the library as a component with a contract. Effects come out in order. Every dependent message is sent only after its write is durably synced. When a Zaxonlite chapter says "the log chose slot n", the zig-paxos book is where "chose" is defined and defended. Read it when you want the invariant, not just the interface. The two books share their pedagogy and their style, so moving between them costs little.

Teach it back. A colleague asks: "We already run SQLite; which parts of Zaxonlite do we get for free, and which parts are still our job?" Answer in plain language. Name your route's entry-point symbol and the three items the repository leaves to you. If you cannot name all three, reread the boundary section before continuing.

PART I

Getting started

We run Zaxonlite before we explain it. One binary gives us a durable SQL database, then a three-voter cluster that survives a killed leader.

1 Quickstart with the `zaxon` CLI

> **Learning contract**
> By the end of this chapter you should be able to build the `zaxon` binary, run one durable database from a shell, bring up a three-voter cluster on your machine, kill its leader and watch a write succeed anyway, and take a verified backup.

Zaxonlite ships as one standalone executable, the way rqlite ships `rqlited`. The binary is called `zaxon`. It is both the server and the client. With a `--data` directory it works on a local node directly. With `--connect` it speaks the replication protocol to a running cluster. Nothing else is needed: no external SQLite install, no separate shell tool, no agent.

1.1 Build the binary

Zaxonlite lives inside the paxos-zig monorepo and builds with Zig 0.16:

```
$ cd zaxonlite
$ zig build -Doptimize=ReleaseSafe
$ ./zig-out/bin/zaxon version
```

`ReleaseSafe` keeps runtime safety checks on. That is the build we recommend for a database. Copy `zig-out/bin/zaxon` anywhere on your `PATH`.

1.2 One durable database in two minutes

Every `zaxon` command that takes `--data` runs against a local node directory. The directory is created on first use.

```
$ zaxon sql --data ./mydb
zaxon> create table notes(id integer primary key, body text)
0 row(s) changed
zaxon> insert into notes(body) values ('first durable row')
1 row(s) changed
zaxon> select * from notes
id | body
---+-----
1 | first durable row
zaxon> .quit
```

That looked like plain SQLite. It was not. Every write you just made was decided by Multi-Paxos in a single-member configuration, recorded in a checksummed journal, and synced to disk before `ok` came back.

> **Predict before reading on**
> Delete the file `mydb/current.db` and run the `select` again. What happens? Decide before you try it.

The database file is disposable. On the next start Zaxonlite discards the materialized image, copies the last verified snapshot, and replays the committed journal suffix. The journal is the truth. The `.db` file is a cache of it. Chapter 6 walks through this recovery in detail.

1.3 Three voters on one machine: PSK first

A real cluster needs one `serve` process per node. Open three terminals, or background each command. Each node gets its own directory, ID, and port. Each `--peer` flag names one other member. Start with the deliberately small local development transport: one owner-only pre-shared key (PSK), restricted by `--dev-psk` to numeric loopback addresses.

Create the provider file, then define two Bash conveniences for the client commands below:

```
$ openssl rand -hex 32 > demo.psk
$ chmod 600 demo.psk
$ cluster=127.0.0.1:7001,127.0.0.1:7002,127.0.0.1:7003
$ zc() { zaxon "$@" --auth-file ./demo.psk; }
```

Start one command in each of three terminals:

```
$ zaxon serve --data ./n1 --node 1 --listen 127.0.0.1:7001 \
    --peer 2@127.0.0.1:7002 --peer 3@127.0.0.1:7003 \
    --auth-file ./demo.psk --dev-psk
$ zaxon serve --data ./n2 --node 2 --listen 127.0.0.1:7002 \
    --peer 1@127.0.0.1:7001 --peer 3@127.0.0.1:7003 \
    --auth-file ./demo.psk --dev-psk
$ zaxon serve --data ./n3 --node 3 --listen 127.0.0.1:7003 \
    --peer 1@127.0.0.1:7001 --peer 2@127.0.0.1:7002 \
    --auth-file ./demo.psk --dev-psk
```

Each terminal immediately prints the node ID, role, data directory, listen address, transport, durability mode, peer connections, and the stable leader. There is no silent-daemon guessing step.

> **Development boundary**
>
> `--dev-psk` is a quickstart convenience, not production transport. It is refused unless the listener and every peer are `127.0.0.1` or `::1`. The PSK authenticates one shared secret and protects frame integrity, but does not encrypt traffic or give nodes distinct identities. Use mTLS for any cluster that leaves one machine.

Wait for election, write, and read. In PSK-only mode give leader-only commands the whole endpoint list: a shared key cannot safely authenticate an advertised node ID, so the client rotates only through seeds you supplied.

```
$ zc wait --connect "$cluster" --leader
applied 0, leader 3
$ zc leader --connect "$cluster"
leader: node 3 at 127.0.0.1:7003
$ zc exec --connect "$cluster" \
    --sql "create table orders(id integer primary key, item text)"
0 row(s) changed
$ zc exec --connect "$cluster" \
    --sql "insert into orders(item) values ('espresso machine')"
1 row(s) changed
$ zc query --connect "$cluster" --sql "select * from orders"
id | item
---+-----
1 | espresso machine
```

1.4 Move the same cluster to mTLS

Stop all three nodes with Ctrl-C. Their data stays in `n1`, `n2`, and `n3`; we are changing only the transport. Production TCP uses a small cluster CA, one identity per node, and an operator/client identity. Create them now:

```
$ mkdir -p demo-pki
$ openssl ecparam -name prime256v1 -genkey -noout -out demo-pki/ca.key
$ chmod 600 demo-pki/ca.key
$ openssl req -new -x509 -key demo-pki/ca.key -sha256 -days 1 \
    -subj /CN=zaxon-demo-ca -addext basicConstraints=critical,CA:TRUE \
    -addext keyUsage=critical,keyCertSign,cRLSign -out demo-pki/ca.crt
$ for n in 1 2 3; do \
    openssl ecparam -name prime256v1 -genkey -noout -out demo-pki/n$n.key; \
    chmod 600 demo-pki/n$n.key; \
    openssl req -new -key demo-pki/n$n.key -subj /CN=zaxon-node-$n \
      -out demo-pki/n$n.csr; \
```

```
    openssl x509 -req -in demo-pki/n$n.csr -CA demo-pki/ca.crt \
      -CAkey demo-pki/ca.key -CAcreateserial -days 1 -sha256 \
      -out demo-pki/n$n.crt; \
  done
$ openssl ecparam -name prime256v1 -genkey -noout -out demo-pki/client.key
$ chmod 600 demo-pki/client.key
$ openssl req -new -key demo-pki/client.key -subj /CN=zaxon-client \
    -out demo-pki/client.csr
$ openssl x509 -req -in demo-pki/client.csr -CA demo-pki/ca.crt \
    -CAkey demo-pki/ca.key -CAcreateserial -days 1 -sha256 \
    -out demo-pki/client.crt
```

The server certificates' canonical common names bind their configured node IDs. Redefine the Bash helper for mTLS client commands:

```
$ zc() { zaxon "$@" --tls-cert demo-pki/client.crt \
    --tls-key demo-pki/client.key --tls-ca demo-pki/ca.crt; }
$ zaxon serve --data ./n1 --node 1 --listen 127.0.0.1:7001 \
    --peer 2@127.0.0.1:7002 --peer 3@127.0.0.1:7003 \
    --tls-cert demo-pki/n1.crt --tls-key demo-pki/n1.key --tls-ca demo-pki/ca.crt
$ zaxon serve --data ./n2 --node 2 --listen 127.0.0.1:7002 \
    --peer 1@127.0.0.1:7001 --peer 3@127.0.0.1:7003 \
    --tls-cert demo-pki/n2.crt --tls-key demo-pki/n2.key --tls-ca demo-pki/ca.crt
$ zaxon serve --data ./n3 --node 3 --listen 127.0.0.1:7003 \
    --peer 1@127.0.0.1:7001 --peer 2@127.0.0.1:7002 \
    --tls-cert demo-pki/n3.crt --tls-key demo-pki/n3.key --tls-ca demo-pki/ca.crt
```

Now wait for the restarted cluster to elect a leader, from a fourth terminal:

```
$ zc wait --connect 127.0.0.1:7001,127.0.0.1:7002,127.0.0.1:7003 --leader
$ zc leader --connect 127.0.0.1:7001
```

Client mode takes a comma-separated endpoint list and follows leader redirects on its own. Under mTLS you can point it at any member: the client accepts the advertised address only when the new server certificate names the advertised node ID. Send a write to node 1 even if node 3 is leader, then read through node 2:

```
$ zc exec --connect 127.0.0.1:7001 \
    --sql "insert into orders(item) values ('mTLS redirect')"
1 row(s) changed
$ zc query --connect 127.0.0.1:7002 --sql "select * from orders"
id | item
---+-----
1 | espresso machine
2 | mTLS redirect
```

Both commands may begin at followers. A follower redirects the authenticated client to the leader. The default read level is `linearizable`: before answering, the leader proves to a quorum that it is still the leader. You never read stale data by accident.

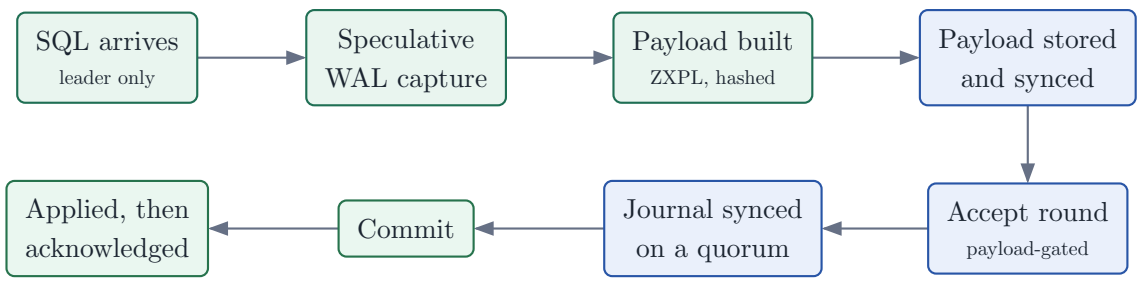

Figure 1: One replicated write. The payload is stored and synced on a quorum before any voter's Paxos state may reference it, and the client is acknowledged only after the decided transaction is applied.

1.5 Kill the leader

Time to earn the word "replicated". Find the leader, press Ctrl-C in that node's terminal, and immediately write again:

```
$ zc leader --connect 127.0.0.1:7001
leader: node 1 at 127.0.0.1:7001
$ zc exec --connect 127.0.0.1:7002,127.0.0.1:7003 \
    --sql "insert into orders(item) values ('written during failover')"
1 row(s) changed
$ zc query --connect 127.0.0.1:7002 --sql "select count(*) from orders"
count(*)
--------
3
```

The two survivors elect a new leader and the write lands. Restart the killed node with its original `serve` command. It rejoins, catches up from its peers, and the cluster is back to three.

Step	Actor	Event and reason
1	You	Kill the leader process. Two voters remain, which is still a quorum of three.
2	Voter 2 or 3	An election timeout fires. The survivor campaigns with a higher ballot and wins both remaining votes.
3	You	Send `exec` to any survivor. The client follows the redirect to the new leader.
4	New leader	Stores the payload, gathers quorum acceptance, commits, applies, and only then acknowledges.
5	You	Restart node 1. It discards its image, rebuilds from its own journal, and asks its peers for the slots it missed.

1.6 Retries that cannot double-apply

Kill-the-leader has one sharp edge: if the leader dies after deciding your write but before answering you, a blind retry would insert the row twice. Sessions close that hole. Open one, then give every write a sequence number:

```
$ zc session --connect 127.0.0.1:7001
session 4211843370881911185
$ zc exec --connect 127.0.0.1:7001 --session 4211843370881911185 \
    --sequence 1 --sql "insert into orders(item) values ('exactly once')"
```

If that command times out, run it again with the same session and the same sequence. A decided write replays its saved result instead of applying twice. An undecided write applies normally. Chapter 8 explains the contract; the rule to remember is: same session, same sequence, same statement.

1.7 Snapshot and backup

```
$ zc backup --connect 127.0.0.1:7001 --to ./orders-backup.db
$ zaxon integrity-check --data ./n1
```

`backup` streams a consistent logical copy from the leader and verifies an end-to-end SHA-256 before installing it at the destination. It is a plain SQLite file you can open anywhere. `integrity-check` verifies the SQLite image, the descriptor chain, and every referenced payload of a stopped node, and exits 3 on any mismatch.

1.8 Where to go next

You have now used everything the rest of the book explains. The full command reference is the next chapter. Chapter 3 states the guarantees precisely. Parts II and III explain how the machine works and how to embed it in your own process, in Zig or through the C ABI.

Exercise 1. Start the three-voter cluster again and kill two voters instead of one. Try a write and a `--level linearizable` read, then a `--level any` read against the survivor. Explain each result. Restart one killed voter and watch the cluster recover without any repair command.

Hint: One voter out of three is not a quorum. Writes and fenced reads must fail. A local read may still answer.

Teach it back. Explain to a colleague why deleting `current.db` on a stopped node loses nothing, but deleting one journal file can refuse to start the node. Use the words journal, snapshot, and materialized image.

2 The zaxon command line

Learning contract

By the end of this chapter you should be able to pick the right mode for any task, run every `zaxon` subcommand and read its output, wire a node up from a config file and environment variables, script the CLI through `--json` and its exit codes, and read any diagnostic without guessing.

The quickstart used `zaxon` in a hurry. This chapter is the reference. One binary carries three modes that share one command surface:

- Embedded mode (`--data <dir>`): the command opens the node in-process, exactly as an embedding application would. No server runs.
- Client mode (`--connect <endpoint>[,...]`): the command speaks the replication protocol to running `zaxon serve` processes. It walks the endpoint list and follows leader redirects on its own.
- Server mode (`zaxon serve`): the process hosts one role-aware node behind a TCP endpoint, alone or in a cluster, or behind a local Unix-domain socket for a single node.

Use embedded mode for a stopped node: local inspection, integrity checks, recovery, one-machine databases. Use client mode whenever a `serve` process owns the directory. A data directory has exactly one owner at a time. If a server owns it, an embedded command is refused:

```
$ zaxon status --data ./srv1
-- NODE DIRECTORY LOCKED --

Another process owns this data directory.

Hint: Stop that process or choose a different --data directory.
```

That refusal exits with code 4. The lock protects safety: two writers on one journal would fork history.

2.1 Embedded mode

Every embedded command opens the node, does its work, applies any journal suffix it finds, and closes. The directory is created on first use. No daemon runs. `zaxon exec --data ./mydb --sql "..."` is a complete durable write from a shell. Embedded mode is the only home of `recover`, and the natural home of `integrity-check`, because both want exclusive ownership of the files they judge.

2.2 Client mode

`--connect` takes a comma-separated endpoint list. An endpoint is `host:port` or `unix:<path>`; the latter dials a local server's Unix-domain socket. The same syntax applies to the config file's `connect` field and to `ZAXON_CONNECT`. For a command that needs the leader, the client tries each endpoint until one answers, and follows the redirect the follower sends back. You never need to know who leads.

Step	Actor	Event and reason
1	You	Run `zaxon exec --connect a,b,c`. The client picks the first reachable endpoint.
2	Follower	Answers "not leader" and names the current leader.
3	Client	Reconnects to the advertised leader and replays the request there.
4	Leader	Commits the write through Paxos, applies it, and returns the result the command prints.

Commands that do not need the leader, such as `status`, `members`, `wait`, and `stop`, are answered by whichever endpoint you reached. So is a `query --level any` read. Under mTLS, a leader hint may name a node outside the seed list: the client follows it only when the new connection presents `zaxon-node-`

<advertised-id> from the trusted CA. Under PSK-only development transport, a shared secret cannot prove that per-node binding, so hints are followed only when they exactly match a configured endpoint; supply every member in `--connect`. When the cluster did advertise a leader but policy forbade following it, the client prints `-- LEADER REDIRECT REFUSED --` naming the advertised address and the fix: add it to `--connect` or switch to mTLS. When no endpoint answered at all, the client prints one `-- NO REACHABLE LEADER --` diagnostic instead. Both exit 4.

When the servers require mutual TLS, client mode takes the same three certificate flags as `serve`: `--tls-cert`, `--tls-key`, and `--tls-ca`. Chapter 13 covers provisioning.

2.3 Server mode: `serve`

`serve` needs three things: a directory, a node ID, and a listen endpoint. Peers are optional; without them you get a single-voter cluster.

```
$ openssl rand -hex 32 > demo.psk && chmod 600 demo.psk
$ zaxon serve --data ./n1 --node 1 --listen 127.0.0.1:7001 \
    --peer 2@127.0.0.1:7002 --peer 3@127.0.0.1:7003 \
    --auth-file ./demo.psk --dev-psk
```

After binding, `serve` writes a compact startup card to stderr: node and role, data and listen paths, transport, member count, configuration, and durability. It then reports peer connections and only stable leader changes. A terminal that shows `writes are ready` is ready for leader-only client work; a leader loss says explicitly that the node is waiting for voter quorum.

Each `--peer` is `id@host:port[/role]`. A peer whose ID equals `--node` is ignored, so all members can share one peer list. The optional role suffix and the node's own `--role` accept `data-voter`, `witness`, `standby`, `read-replica`, or `gateway`. A node run with `--role gateway` owns no database state at all: it only routes client traffic to members that serve reads or writes. Chapter 7 explains what each role may do.

With more than one member, `serve` derives a database identity from the membership, plus `--cluster-id` when given. Two clusters with the same member list but different `--cluster-id` values refuse to mix. That fence protects safety against cross-cluster replay.

For a single local node that should not open a TCP port at all, `--listen unix:<path>` serves over a Unix-domain socket instead. Filesystem permissions are the local authorization boundary: the socket is restricted to owner-only permissions (mode 0600) immediately after binding. A pre-existing file at the socket path is refused, never silently unlinked, so a stale socket left by a crash needs explicit operator removal; an orderly shutdown removes the path itself. The mode is single-node only. Configured peers are rejected, because cluster links require TCP, and gateway mode is TCP-only. Clients reach the node with `--connect unix:<path>`.

`serve` can also carry a mutual TLS identity: `--tls-cert <pem>`, `--tls-key <pem>`, and `--tls-ca <pem>`, always all three together; a partial set is a usage error, exit 2. Every TCP connection the server accepts or dials, peer and client alike, then runs TLS 1.3 with mutual certificate verification against the cluster CA, and a peer's certificate must name exactly the node id it claims. TLS and the PSK compose: when both are configured, the PSK handshake runs inside the TLS channel. Chapter 13 covers certificate provisioning and the identity rules; chapter 7 states what each transport mode proves.

`--dev-psk` is the one explicit PSK-only mode. It requires `--auth-file` and refuses startup unless the listener and every peer use numeric loopback. It exists for the one-machine quickstart and local development: it has frame authentication and integrity, but neither confidentiality nor distinct node identity. It cannot be set on a gateway or combined with mTLS.

> **TCP transport boundary**
>
> Every non-loopback TCP listener requires mTLS. Loopback also defaults to mTLS unless the operator explicitly selects `--dev-psk` with an owner-only PSK provider. A single local node can instead use `unix:<path>`. Credential flags take file paths, never literal secrets, so keys stay out of shell history and

> process listings. PSK-only protocol v6 remains a local development transport, not a production trust boundary.

`--enable-failpoints` makes the server honor fault-injection RPCs. It exists for test controllers only. Never set it in production.

`--enrollment-ca-key <path>` deliberately turns a storage node into a certificate issuer. The file must be a regular, non-symlink, owner-only CA private key matching `--tls-ca`; most nodes should omit it. It does not enable dynamic membership. Chapter 13 gives the two-command enrollment procedure and the crash semantics.

2.4 Data commands: `sql`, `exec`, `query`

`sql` opens the interactive shell in either embedded or client mode. A statement starting with `select`, `with`, `values`, or `explain` runs as a read; every other statement runs as a replicated write.

On a terminal the shell is a rich REPL (ZDS 0005): statements end with `;` and may span lines under a `...>` continuation prompt, the input line is edited in place with readline keys (`ctrl+a`/`ctrl+e`, word movement, `ctrl+w`/`ctrl+u`/`ctrl+k`, and `ctrl+y` yank), SQL keywords highlight as you type, the up/down arrows walk history, and `ctrl+r` is reverse incremental history search. Results render as aligned tables with `NULL` shown dimmed; results taller than the screen open in a pager (arrows scroll, `q` returns).

```
$ zaxon sql --data ./mydb
zaxonlite unreleased — interactive shell
Statements end with ';'. Type .help for commands and keys.
zaxon> select id, author from notes where id < 3;
┌────┬──────────┐
│ id │ author   │
├────┼──────────┤
│  1 │ vik      │
│  2 │ NULL     │
└────┴──────────┘
(2 rows)
zaxon> .quit
```

`.help` lists the dot commands, which include `.tables`, `.schema [name]`, `.status`, `.members`, `.mode table|expanded|auto|json|csv` (`expanded` prints one block per record for wide rows; `auto` switches to it when a table would overflow the terminal), `.timer on|off`, and `.history [off|clear]`. `ctrl+c` cancels the statement being typed. During a remote statement it abandons the local wait and warns that the statement may still apply server-side; check its effects before retrying a write. `ctrl+d` on an empty line leaves. `--no-color`, `NO_COLOR`, or `TERM=dumb` disable styling.

When stdin or stdout is not a terminal — pipes, scripts, CI — the shell keeps its historical plain form, byte for byte: a bare `zaxon>` prompt, one statement per line with no `;` requirement, dot commands `.status`, `.tables` (embedded only), and `.quit`/`.exit`, and header-width tables:

```
$ echo "select count(*) as n from notes" | zaxon sql --data ./mydb
zaxon> n
-
1
```

`exec` runs one write and prints how many rows it changed, in the `1 row(s) changed` form you have seen. A SQL failure prints the SQLite message and exits 1:

```
$ zaxon exec --data ./mydb --sql "insert into missing values (1)"
-- SQL ERROR --

no such table: missing

Hint: Correct the statement and retry it as a new request.
```

`query` runs read-only SQL and prints a table: a `col | col` header, a dashed underline, then the rows, with `NULL` for null cells.

```
$ zaxon query --connect 127.0.0.1:7001 --sql "select a, b from t"
a | b
--+--
1 | x
```

A write statement given to `query` is rejected with "statement is not read-only; use exec", exit 1. The read level defaults to `linearizable`; `--level` also accepts `leader` and `any`, and `--freshness-ms` bounds the staleness of a local learner read. Chapter 8 defines what each level promises. In embedded mode there is no other node to consult, so every read is local.

2.5 Session commands: `session`, `exec --session`, `expire-sessions`

`session` opens a client session and prints its ID on one line, as `session 1`.

> **Predict before reading on**
>
> You pass `--session` to `exec` but forget `--sequence`. Does the write run without retry protection, or does the command refuse? Decide before reading on.

It refuses, exit 2: "–session and –sequence go together". A session without a sequence number cannot deduplicate anything, so `zaxon` will not pretend it can. With both flags, a retry of a decided write replays its recorded result instead of applying twice:

```
$ zaxon exec --data ./mydb --session 1 --sequence 1 --sql "insert into notes(body) values
('exactly once')"
1 row(s) changed
$ zaxon exec --data ./mydb --session 1 --sequence 1 --sql "insert into notes(body) values
('exactly once')"
replayed: 1 row(s) changed (recorded result)
```

Skipping ahead exits 1 with a `-- SESSION ERROR --` diagnostic naming the cause: `SequenceGap`, `UnknownSession`, or `ResultExpired`. The hint is the rule: use the same live session and its next monotonic sequence.

`expire-sessions --retain <n>` deletes idle sessions, keeping the `n` with the newest activity, and reports the count as `0 session(s) expired` when nothing was idle enough to remove. Expiring a session a client still retries against turns its safe retry into a double apply, so retain generously. Chapter 8 covers sizing.

2.6 Cluster commands: `status`, `members`, `leader`, `wait`, `stop`

`status` on an embedded node prints an aligned field list: node id, database id, configuration id, role, node type, decided and applied slots, journal records, epoch capacity, the journal chain hash, page size, and the current snapshot name or `(none)`. Against a server, `status` and `members` print the raw JSON response even without `--json`; the server's answer includes fields such as the current ballot that the embedded view does not have.

`leader` names the coordinator, or reports `leader: none` when no leader is known:

```
$ zaxon leader --connect 127.0.0.1:7001
leader: node 1 at 127.0.0.1:7001
```

`wait` blocks until conditions hold, then reports where the node stands on one line, as `applied 3, leader 1`. `--applied <slot>` waits for the apply point to reach that slot and defaults to 0. `--leader` also waits for a known leader. `--timeout-ms` defaults to 10000. Use `wait` in scripts between "start the cluster" and "first write". `stop` asks one served node to shut down cleanly and prints the bare acknowledgement `{"ok":true}`. It stops one process, not the cluster.

2.7 Identity bootstrap: `enroll-token`, `enroll`

`enroll-token` is a client-mode operator command. It uses an existing mTLS identity to ask one configured issuer for a token bound to `--node`, then writes an owner-only opaque bundle to `--to`. `--ttl-seconds` defaults to 600 and cannot exceed 86400:

```
$ zaxon enroll-token --connect 10.0.0.1:9901 --node 2 --to node2.token \
    --tls-cert operator.crt --tls-key operator.key --tls-ca ca.crt
```

Transfer that bearer file securely to node 2. There, `enroll` pins the bundled CA and issuer identity, creates the node key and CSR locally, redeems the token once, verifies the certificate, and installs a new identity directory with one atomic rename:

```
$ zaxon enroll --token-file node2.token --identity-dir /etc/zaxon/node2
```

The new directory contains `node.key`, `node.crt`, and `ca.crt`; its private key and directory are owner-only. An existing destination is never replaced, and the bundle is removed after success. The issuer consumes the token before signing, so a lost response or installation failure is deliberately fail-closed: issue a new token rather than retrying the old one.

2.8 Maintenance: `snapshot`, `backup`, `integrity-check`, `recover`

`snapshot` compacts: it installs a verified snapshot and seals the current journal epoch, printing the new configuration ID. `backup --to <path>` streams a consistent logical copy, a plain SQLite file, verified end to end with SHA-256 before it is installed at the destination. Against a cluster it streams from the leader. If the leader dies mid-stream the command reports "backup interrupted" and installs nothing, so a half-written backup can never be mistaken for a good one.

`integrity-check` verifies three layers and prints one verdict per layer:

```
$ zaxon integrity-check --data ./n1
sqlite: pass
chain: pass
payloads: pass
```

Any `FAIL` makes the exit code 3. The client-mode form prints a single `integrity: pass` or `integrity: FAIL` line. `recover` runs the same checks after rebuilding the node from its authoritative state, and ends with `recovery rebuild complete`. It is embedded-only: running it with `--connect` is refused, exit 2, because rebuilding a node that a live server owns would race the server. Stop the node, then recover it.

2.9 The sync mode: `--sync`

Every command takes `--sync <mode>`, which sets the process-wide durability policy before any storage I/O runs. The mode decides how far a sync must reach before the node treats bytes as durable. `full`, the default, flushes the drive's volatile write cache on macOS through `fcntl(F_FULLFSYNC)`, so an acknowledged write survives power loss, not just a process crash. `os` keeps the plain platform `fsync(2)`, which on macOS does not reach the drive cache; it is for development and benchmarks only there. On Linux and the other supported platforms plain `fsync` already flushes the cache, so the two modes are identical. Any other value is a usage error, exit 2: `--sync must be os or full`. Chapter 6 explains why a consensus voter must not run `os` on macOS in production, and chapter 13 prices the difference.

2.10 Configuration file and environment

Every mode can read a JSON config file, named by `--config <path>` or the `ZAXON_CONFIG` environment variable. The file accepts exactly these fields, each optional:

Field	Meaning
`data`	Node data directory, as for `--data`.
`connect`	Comma-separated endpoint list, as for `--connect`.

Field	Meaning
`node`	This node's integer ID.
`role`	Node role name, as for `--role`.
`listen`	Listen endpoint for `serve`.
`peers`	Array of peer specs, each `id@host:port[/role]`.
`cluster_id`	Extra entropy for the derived database identity.
`auth_file`	Path to the transport secret provider.
`tls_cert`	Node certificate PEM for mutual TLS.
`tls_key`	Node private key PEM for mutual TLS.
`tls_ca`	Cluster CA PEM that peer certificates must chain to.
`enrollment_ca_key`	Owner-only CA private key enabling token/CSR issuance on this server. Omit it on ordinary nodes.
`revocation_file`	Node-ID denylist reloaded by a serving node.
`sync`	Durability sync mode, as for `--sync`: `full` (the default) or `os`.

An unknown field is an error, not a warning: the command reports `-- CONFIGURATION UNREADABLE --` with `UnknownField` and exits 2. A typo in a config file should fail loudly, not silently misconfigure a database. Each field also has an environment variable: `ZAXON_DATA`, `ZAXON_CONNECT`, `ZAXON_NODE`, `ZAXON_ROLE`, `ZAXON_LISTEN`, `ZAXON_PEERS` (comma separated), `ZAXON_CLUSTER_ID`, `ZAXON_AUTH_FILE`, `ZAXON_TLS_CERT`, `ZAXON_TLS_KEY`, `ZAXON_TLS_CA`, and `ZAXON_SYNC`. `ZAXON_ENROLLMENT_CA_KEY` and `ZAXON_REVOCATION_FILE` name the two additional server-only provider paths.

Precedence is fixed: a command-line flag beats an environment variable, and an environment variable beats the file. A complete node config:

```
{
  "data": "/var/lib/zaxon/n2",
  "node": 2,
  "role": "data-voter",
  "listen": "10.0.0.2:7001",
  "peers": ["1@10.0.0.1:7001", "3@10.0.0.3:7001"],
  "cluster_id": "orders-prod",
  "auth_file": "/etc/zaxon/psk",
  "tls_cert": "/etc/zaxon/node.crt",
  "tls_key": "/etc/zaxon/node.key",
  "tls_ca": "/etc/zaxon/ca.crt"
}
```

With that file in place, `ZAXON_CONFIG=/etc/zaxon/n2.json zaxon serve` is a complete node start.

> **The secret provider file**
>
> The provider file must hold at least 32 bytes and at most 4096. One trailing line ending is stripped; every other byte, including spaces, is part of the key. Give every node the same provider content. A short file is rejected with `SecretTooShort`, exit 2.

2.11 `--json` for automation

`--json` switches stdout to machine-readable output. Embedded commands emit compact objects:

```
$ zaxon exec --data ./mydb --json --sql "insert into notes(body) values ('json row')"
{"changes":1,"slot":7,"replayed":false}
$ zaxon status --data ./mydb --json
```

```
{"node_id":1,"database_id":"a13f203d26d80813d0834ff231269878",
 "configuration_id":1,"role":"leader","node_type":"data-voter","leader":1,
 "decided_slot":7,"applied_slot":7,"journal_records":31,
 "epoch_capacity":2048,"chain":"8f6a...94f8f","page_size":4096,
 "snapshot":null}
```

The status object is one line on a real terminal, and the chain field is the full 64-hex-digit hash. `query --json` emits `{"columns":[...],"rows":[[...]]}` with every cell as a string or `null`. In client mode `--json` passes the server's response through untouched, including error responses, which arrive as `{"ok":false,...}` on stdout while the exit code still reports the failure class. Parse the exit code first, then the body.

2.12 Diagnostics and exit codes

Every failure prints one diagnostic to stderr in a fixed shape: an uppercase header between `--` markers, a plain-language description, and a `Hint:` line naming the next action. The hint is chosen by failure class. A `not_leader` error says to retry the advertised leader or restore a quorum; a `stale` read says to wait for catch-up, relax freshness, or query the leader. The exit code states the failure class:

Code	Meaning
0	Success.
1	SQL error or session error. The statement or retry was wrong; the node is healthy.
2	Usage error: bad flags, bad config, bad request, or a command not available in this mode.
3	Integrity failure. `integrity-check` or `recover` found a layer that does not verify.
4	Node unavailable: directory locked, no reachable leader, corrupt state, or a refused insecure listen.

Scripts should branch on the code, not on the text. The text is for humans and may improve; the codes are the contract.

Exercise 2. Write a shell script that takes an endpoint list, waits up to 30 seconds for a leader, runs one idempotent insert inside a fresh session, and retries it once. Assert with the `--json` output that the retry reports `"replayed":true` and the same slot both times.
Hint: You need `wait --leader --timeout-ms`, `session --json`, and two identical `exec --session --sequence --json` calls. Compare the two `slot` values.

Teach it back. Explain to a colleague why `recover` works only with `--data`, why an embedded command on a served directory exits 4, and why both rules follow from the same single-owner invariant.

3 The product

Learning contract

By the end of this chapter you should be able to state what an acknowledged write promises, explain why replicating WAL frames makes nondeterministic SQL safe, name the five node types and what each one may do, and list the things Zaxonlite refuses to do.

You have already run the product. In chapter 1 you built `zaxon`, wrote through a cluster, and killed its leader. This chapter states precisely what those commands promised you. Hold the book to these statements. Chapter 17 maps each one to the test that exercises it.

3.1 What Zaxonlite is

Zaxonlite is SQLite with a replicated, crash-consistent history. Your application links one Zig library, or the C ABI, and opens one data directory. It gets a full SQL database. Every committed write transaction is decided by Multi-Paxos and recorded in a checksummed journal before it is acknowledged. One library and one on-disk layout serve two deployments.

1. **One durable node** is a single process with no network. Every write is fsynced through the journal. This is the upgrade path for an application that outgrew plain SQLite's durability story.
2. **Role-aware clusters over TCP** run Paxos across one to nine configured voters. Witnesses vote without campaigning. Standbys and read replicas learn chosen entries without changing the quorum. Stateless gateways route clients.

The companion CLI is `zaxon`, the binary from chapter 1. With `--data` it embeds a node directly. With `--connect` it speaks the client RPC protocol to running servers and follows leader redirects on its own. Chapter 2 documents every command.

Security model and current status

Zaxonlite is a single-application database, not a multi-user SQL server. The application authenticates its users and decides what SQL to issue. Production TCP is mutual TLS 1.3 only. Protocol v6 can still layer the optional shared-PSK challenge inside TLS. An explicit PSK-only development mode is restricted to numeric loopback, while plaintext TCP exists solely behind the failpoint-gated test switch. Mutual TLS (`--tls-cert`/`--tls-key`/`--tls-ca`) gives every node a certificate chained to one cluster CA, binds a peer connection to the node id its certificate names, and encrypts the wire. After the initial CA and issuer identity are provisioned, the one-time token/CSR flow automates issuance for a node already in the static registry without sending its private key. A single local node can instead serve over an owner-only Unix-domain socket (`--listen unix:<path>`), where filesystem permissions are the boundary. A narrow SQLite invariant guard screens every application statement: transaction control, `ATTACH`/`DETACH`, the reserved `__zaxon_*` namespace, and capture-critical pragmas are denied at prepare time, and loadable SQLite extensions are compiled out. The guard protects replication invariants for a trusted application; it is not a sandbox or multi-user RBAC. The detailed security plan is in `docs/zds/records/0003-zaxonlite-security-remediation-plan.typ`.

3.2 The one idea

Most replicated-SQLite systems replay SQL text on every node, and hope the replicas compute identical results. Hope is the weak point. One call to `random()` and two replicas can disagree forever.

Zaxonlite replicates the bytes SQLite itself committed: the page frames of the write-ahead log. The leader executes a transaction once. It captures exactly the frames SQLite appended for that transaction, and it replicates those frames. Nondeterministic SQL is therefore safe by construction. `random()`, `randomblob()`,

and date functions all resolve on the leader, one time. The thing being decided is the page images, not the recipe that produced them.

> **Consequence: byte-identical replicas**
>
> Applying a committed frame payload is a deterministic page write plus a truncate. Given the same base image and the same decided history, every member's database file converges to the same bytes. The verification suite checks this literally. It compares SHA-256 digests of `VACUUM INTO` images across all three cluster members.

3.3 User-visible guarantees

Five statements carry the product. Each is a claim you can test.

1. **Acknowledged means durable and decided.** A write is acknowledged only after three facts hold. Its descriptor is committed by the configured voter quorum. Its journal records are fsynced. The slot has been applied locally.
2. **Exactly-once retry.** A client session executes sequence n at most once. Retrying the last sequence returns the recorded result without executing SQL. That holds across process crashes, leader changes, and restarts. Gaps and expired sequences fail without side effects. You used this in chapter 1; chapter 8 states the full contract.
3. **One-writer serializability.** All writes flow through the current leader, one replicated transaction at a time. A dependent slot is never proposed before its predecessor is chosen.
4. **Read levels you can name.** `any` reads locally, and may be stale on a follower; the response says so. `leader` reads the leader's applied state. `linearizable` first completes a quorum read fence.
5. **The journal is the database.** The SQLite file is a materialized image. Delete it, or restore a stale copy, and the node converges back to the decided state from snapshot plus journal suffix. You did this to `current.db` in chapter 1. Chapter 6 walks through the machinery.

> **Predict before reading on**
>
> A `linearizable` read must prove to a quorum that the leader still leads. Does that proof append to the log or sync any disk? Decide before reading on.

It does neither. The read fence is a quorum round with no log append and no disk sync. Chapter 8 shows the fence message flow and what a failed fence means for the client.

3.4 Node types

Zaxonlite defines five node types. `data-voter` proposes, votes, materializes SQLite, and serves SQL. `witness` votes and stores durable payloads, but it cannot campaign and cannot serve SQL. `standby` and `read-replica` are non-voting learners of the chosen log, each holding a SQLite copy. Only the standby is marked promotion-eligible. `gateway` is a byte-transparent router. It holds no Paxos state and no SQLite state.

3.5 Quorum and scaling

For a voter set V, the majority is $q = \left\lfloor |V| \frac{1}{2} \right\rfloor + 1$. Learners are absent from V, so adding them does not change the write quorum. The implementation profile bounds voters at nine but allocates the total member registry at runtime. That is a scaling boundary. It is not a promise that millions of all-to-all sockets are practical.

3.6 Explicit non-goals

Zaxonlite does not do multi-master writes. It does not do cross-database transactions or SQL-visible replication controls. It does not replace a failed voter automatically. It bounds one epoch at 2,048 slots and

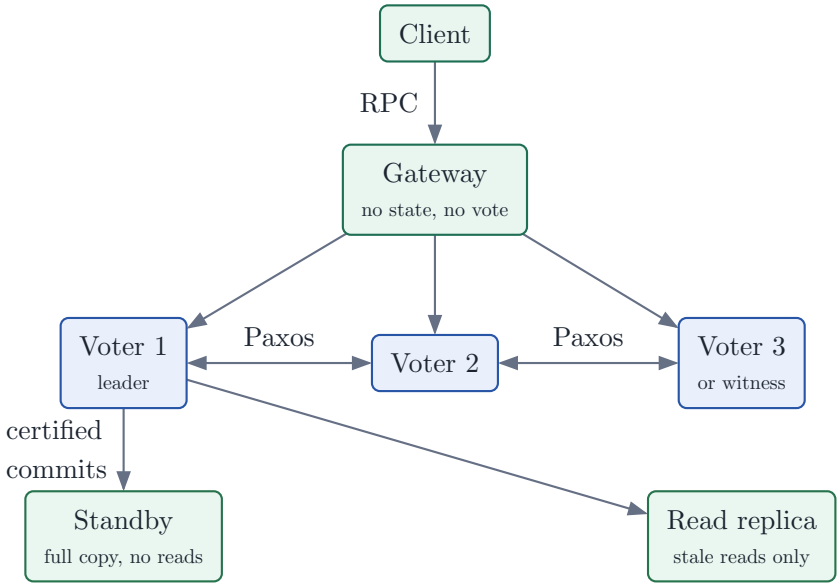

Figure 2: Only the voters decide writes. The learners below them receive certified commits, so adding a standby or a read replica never changes the write quorum.

one consensus group at nine voters. Both bounds are policy choices of the `ReplicatedLog` instantiation, not Paxos theorems.

Teach it back. Explain to a colleague why `select random()` can never make two replicas diverge, and exactly what the acknowledgment promised when your chapter 1 write returned `1 row(s) changed`. Use the words frames, quorum, and journal.

PART II

How it works

We open the machine you drove in chapter 1. One ordering contract moves every write from SQL text to a durable, replicated, acknowledged row.

4 Architecture

Learning contract

By the end of this chapter you should be able to name each module of a node and its single responsibility, trace one write from SQL text to acknowledgement in order, explain why Paxos decides a fixed-size descriptor instead of transaction bytes, and say how client sessions ride inside the replicated state machine.

In chapter 1 you killed a leader and a write still landed. This part of the book explains why that worked. We start with the shape of one node. Chapter 5 then follows the transaction bytes. Chapter 6 follows the files on disk. Chapter 7 grows the cluster beyond three voters. Chapter 8 states exactly what a client may assume.

4.1 One node, small parts

One node owns one data directory. It is assembled from small parts, and each part is tested on its own.

Module	Responsibility
`node.zig`	The host: lifecycle, the write path ordering contract, effect handling, recovery, snapshots, epoch rollover, and the leader/follower split of the materialized image.
`journal.zig`	The framed, CRC-checksummed, append-only protocol journal; replay with torn-tail truncation and interior-corruption rejection.
`payload_store.zig`	Content-addressed immutable payloads under `payloads/aa/<hash>`, installed with write-temp/sync/link.
`wal.zig`	WAL frame capture from SQLite's `-wal` file and the deterministic offline page apply; the payload wire format.
`command.zig`	The fixed-size replicated descriptor (`TransactionBatch`) and the cumulative chain-hash formula.
`types.zig`	The one `ReplicatedLog` instantiation and canonical encodings of its entries and durable writes.
`sqlite.zig`	Narrow bindings over the pinned SQLite amalgamation: exactly the C API subset the product needs.
`guard.zig`	The SQL invariant guard: a narrow authorizer screening application statements at prepare time, plus the capture-contract check run before every payload extraction.
`server.zig` / `client.zig` / `wire.zig`	The TCP or Unix-socket host behind `zaxon serve`, the RPC client, and the shared frame codec.
`capi.zig`	The C ABI (`libzaxonlite` + `zaxonlite.h`).

The Paxos core is the parent `paxos` library's `ReplicatedLog`. It is a bounded effect machine and it allocates nothing. The host never mutates protocol state directly. It calls `append`, `step`, and `tick`, and each call hands back an explicit `Effects` batch. The contract on that batch is the durability ordering itself: persist `writesSlice()`, fsync, call `confirmWritesDurable()`, and only then send `messagesSlice()` and apply `committedSlice()`. Chapter 6 turns this contract into the full set of ordering rules and names the invariant each one protects.

4.2 The replicated command

Paxos never carries transaction bytes. A committed slot holds a fixed-size descriptor:

```
TransactionBatch {
    database_id:       u128,   // cluster-wide database identity
    batch_id:          u128,   // random identity of this execution
    base_data_slot:    u64,    // previous data slot in the epoch
    base_chain_hash:   [32]u8, // cumulative history before this batch
    result_chain_hash: [32]u8, // cumulative history after this batch
    payload_hash:      [32]u8, // SHA-256 name of the frame payload
    payload_bytes:     u64,
    transaction_count: u32,
    frame_count:       u32,
}
```

The real bytes live somewhere else. The payload holds the header, the per-transaction records, the frame descriptors, and the raw page images. It sits in the content-addressed store under its own SHA-256, and it is persisted before the descriptor is ever proposed. That ordering protects a safety invariant: no counted vote can ever name bytes that are not durable somewhere. Chapter 5 walks through the payload format itself.

> **Chain hashes: history identity in constant space**
>
> `result_chain_hash = H(base_chain_hash, database_id, batch_id, base_data_slot, payload_hash, payload_bytes, transaction_count, frame_count)` with a domain-separated SHA-256. Two nodes with equal chain values have applied the same batches in the same order. Recovery, commit accounting, and `integrity-check` all validate the chain. A mismatch is fatal. It is never repaired silently.

4.3 The write path

Every write follows one ordering contract. The contract is identical in a one-member database and a full cluster. The `./mydb` directory you created in chapter 1 walked this exact path with a quorum of one.

Step	Actor	Event and reason
1	Leader	Executes the SQL inside `BEGIN IMMEDIATE ... COMMIT` on its capture connection. The replicated session row and the `batch_id` marker are updated inside that same SQLite transaction.
2	Leader	Captures the committed frames from the `-wal` file. The `wal_hook` count says exactly how many frames belong to this commit.
3	Leader	Persists the payload in the content-addressed store: write to a temporary name, fsync, link into place.

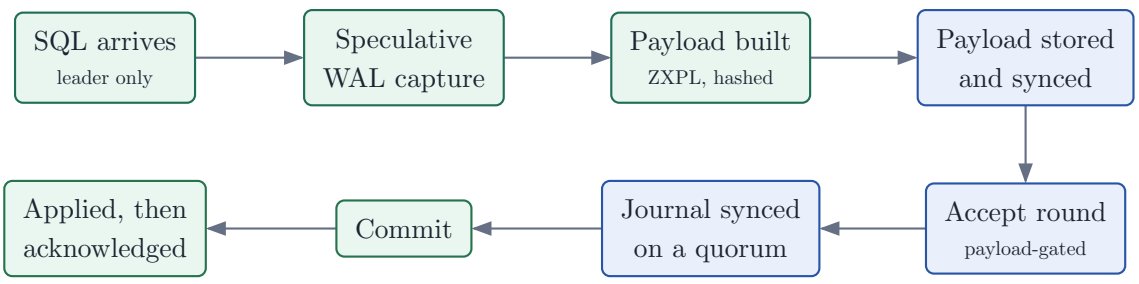

Figure 3: One replicated write, in the order the bytes move. The payload is durable on a voter before that voter's vote may count, and the client hears nothing until the decided transaction is applied.

Step	Actor	Event and reason
4	Leader	Appends the descriptor through `ReplicatedLog`, then journals the resulting durable writes, fsyncs, and confirms them. Only now may accept messages leave the process.
5	Each voter	Stores and syncs the payload, then acknowledges it with `payload_stored`. The leader releases the value-bearing accept to a voter only after that acknowledgement.
6	Each voter	Journals its acceptance and fsyncs before replying. The slot commits once a quorum has accepted.
7	Leader	Applies the committed batch. When the committed slot carries this `batch_id`, and only then, it acknowledges the client.

What about a write that fails? A failed SQL statement rolls back before capture. A rolled-back transaction never advances the WAL hook count, so nothing about it is ever captured, proposed, or replicated. The failure stays local to the leader, exactly as it would in plain SQLite.

4.4 The read path

The leader holds the only live SQLite connection: the capture connection. Its applied state is the database. Followers keep the image materialized offline and open short-lived read connections per query.

Reads are labeled, and the label is the contract. A read at `any` may be stale on a follower. A read at `leader` returns the leader's applied state. A read at `linearizable` completes a quorum fence before it answers, which is the default you used in chapter 1. Chapter 8 defines each level precisely and shows the fence protocol.

4.5 Sessions inside the state machine

Bounded client sessions are rows in a replicated table, `__zaxon_sessions`. The session row is updated inside the same captured transaction as the user's SQL. This placement is the whole trick. Exactly-once retry needs no separate log, because wherever the frames go, the session state goes with them, atomically.

Application SQL can neither read nor write the `__zaxon_*` namespace: the invariant guard (chapter 5) denies the reserved prefix at prepare time, so the replicated metadata stays Zaxonlite's alone.

The `__zaxon_meta.batch_id` marker travels the same way. On restart, recovery compares it against the last committed descriptor to prove the materialized image is the one the log says it should be. Chapter 6 shows where that check sits in the recovery sequence, and chapter 8 states the session contract from the client's side.

5 WAL-frame replication

Learning contract

By the end of this chapter you should be able to explain why Zaxonlite replicates page images instead of SQL text, walk one committed transaction from `COMMIT` to a named payload, apply a captured payload to an offline database file by hand, and state the determinism property that recovery, follower apply, and resync all rely on.

Checkpoint: write path

Place "capture the committed frames" inside chapter 4's write-path transcript without looking back. If you cannot say what is already durable at that step, reread the write path first.

This is the one-idea chapter of the book. Everything else in Part II is careful engineering around a single decision: consensus decides page images, not SQL. Take the time to understand this chapter and the rest of the machine follows from it.

5.1 Why frames, not SQL

Suppose we replicated the SQL text instead, and had every replica execute each statement locally. Now consider one statement:

```
insert into audit(token) values (randomblob(16));
```

Every replica computes a different blob. The databases diverge, and no error tells you so. Nondeterministic functions are only the most obvious hazard. To make SQL replay safe, every statement must be deterministic, every schema hook must fire identically, and every SQLite version in the cluster must behave the same. Each of those is a standing obligation on every future application and every future upgrade.

WAL-frame replication removes the entire class. The leader's SQLite computes the transaction once. Consensus then decides the resulting page images. A replica applies bytes; it never re-executes anything, so determinism of the SQL stops mattering. dqlite and rqlite arrived at related designs by patching or wrapping SQLite. Zaxonlite's Phase-0 spike proved the capture can be done with public, stable SQLite interfaces only.

5.2 The capture technique

First, a fact about SQLite in WAL mode: a commit never rewrites a page in the main database file. It appends the new page images to `current.db-wal` as frames. A later checkpoint folds frames back into the main file. Zaxonlite's node runs one writer connection configured so that capture can trust this behavior:

```
pragma journal_mode = wal;        -- append frames, never rewrite pages
pragma wal_autocheckpoint = 0;    -- checkpoints only at snapshots
```

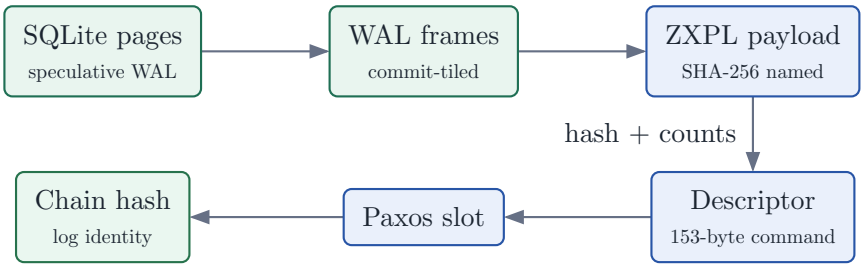

Figure 4: One committed transaction, from SQLite pages to log identity. The page images travel as a SHA-256-named payload. Paxos decides only the fixed-size descriptor that names them.

```
pragma synchronous = normal;      -- journal fsync is the durability
sqlite3_wal_hook(db, hook, ...)   -- committed frame count per commit
```

With automatic checkpoints disabled, the WAL only grows between snapshots, and a written frame is never moved or rewritten. That stability is what makes the next step legal.

> **These pragmas are replication invariants**
>
> Application SQL runs inside Zaxonlite's `BEGIN IMMEDIATE`. SQLite refuses changing `journal_mode` or `synchronous` there, but a `wal_autocheckpoint` write would replace the installed WAL hook. The invariant guard below therefore denies capture-changing statements at prepare time, and the node re-verifies the hook and mode contract before every payload extraction. Loadable extensions are omitted at SQLite compile time.

After every commit, the hook reports the total number of committed frames now in the WAL. We remember the previous total. The frames between the old count and the new count are exactly this transaction's committed pages. We read their bytes directly from the file, and the layout is fixed:

1. the file starts with a 32-byte WAL header;
2. each frame is a 24-byte frame header followed by the raw page image;
3. the frame header stores the page number big-endian at offset 0 and the commit size, the database size in pages, at offset 4.

A rolled-back transaction never advances the hook count, so its frames are invisible to capture. Rollbacks cost the cluster nothing.

Here is one insert moving through the whole pipeline. The frame numbers are illustrative; the mechanics are exact.

Step	Actor	Event and reason
1	You	Run `insert into notes(body) values ('hello')` through the leader.
2	SQLite	Appends the changed pages to `current.db-wal` as frames 13 and 14, then returns from `COMMIT`.
3	Hook	Fires with a committed frame count of 14. The previous count was 12, so this transaction owns frames 13 and 14.
4	Capture	Reads both frames straight from the file: two 24-byte headers and two raw page images.
5	Capture	Builds the `ZXPL` payload and hashes it. The SHA-256 becomes the payload's name in the store and the `payload_hash` in the descriptor.
6	Consensus	Decides the descriptor. The page images travel to voters through the payload store, never inside a Paxos message.

> **Why reading the file directly is safe**
>
> Only the capture connection writes the WAL. Checkpoints are disabled. Capture runs synchronously after `COMMIT` returns and before the next write begins. So the `[from, to)` frame range is immutable by the time it is read. The WAL format has one famous subtlety: frame checksums are cumulative and seeded per WAL file. It does not matter here, because Zaxonlite never splices frames into another WAL. It applies pages offline instead.

5.3 The invariant guard

Everything above assumes the writer connection stays configured exactly as the node opened it. A single application statement could break that. `COMMIT` would end the outer transaction capture depends on. `ATTACH` would produce WAL frames in a file Zaxonlite never replicates. A write to `wal_autocheckpoint` would replace the installed hook. So every application statement — `exec`, `query`, prepared statements,

transactions, the CLI, the C ABI, and the network surface alike — is screened by a narrow SQLite authorizer (`guard.zig`) at prepare time, before any side effect. Zaxonlite's own metadata statements run in a separate internal scope that the authorizer waves through.

The denied list is short and exact: transaction control (`BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`, `RELEASE`), `ATTACH` and `DETACH`, any read or write of the reserved `__zaxon_*` namespace, `pragma wal_checkpoint` in both its forms, and writes to `wal_autocheckpoint`, `journal_mode`, `synchronous`, `page_size`, `locking_mode`, `auto_vacuum`, `writable_schema`, and `query_only`. Reads of those pragmas stay allowed, and everything else SQLite permits stays available: DDL, triggers, views, and ordinary pragmas such as `user_version` and `foreign_keys`. A denied statement fails at prepare with SQLite's authorization error, which reaches the caller as an ordinary SQL error.

The authorizer already denies every statement that could change the capture configuration; a runtime check keeps the invariant robust anyway. After each application batch, and before commit and payload extraction, the node re-verifies the capture contract: the outer transaction is still open, the journal mode is WAL, the page size is the expected one, and the frame-counting WAL hook is still the installed one. Any mismatch fails the write with `error.CaptureContractViolated` instead of capturing the wrong bytes.

A build invariant completes the fence: the bundled amalgamation keeps `SQLITE_OMIT_LOAD_EXTENSION`, and a test asserts both the compile option and that `load_extension`, `readfile`, and `writefile` are absent from the SQL surface.

> **A guard, not a sandbox**
>
> The application is trusted; it is the single database principal. The guard protects Zaxonlite's replication invariants against accidental interference. It is not a sandbox for hostile tenants and not multi-user RBAC.

5.4 The apply technique

A committed payload is applied offline. No SQLite connection is open on the file while it happens. The whole algorithm is three steps:

1. for each frame, write the page image at byte offset `(page_number - 1) * page_size`;
2. truncate the file to the commit frame's database size in pages;
3. fsync.

SQLite is not consulted at apply time. We are performing the same page placement a checkpoint would perform, with nothing else in the way.

Now the property everything depends on. Given the same base image and the same frames, this transition is deterministic. It is also idempotent: applying any decided prefix again, from any intermediate state, converges to the same bytes. One property, three payoffs. It powers recovery, which rebuilds from a snapshot plus the committed suffix. It powers follower apply, which materializes the image offline. It powers resync after a leadership change. It also makes crash timing across the entire apply path harmless, because a half-applied batch is repaired by applying it again.

5.5 The proof: byte-identical rebuild

> **Predict before reading on**
>
> The test workload below includes `randomblob()` and `random()`. Can a second database built only from captured frames still match the leader's file byte for byte? Decide, and say why, before reading on.

It can, and the reason is the point of this chapter. The nondeterminism is resolved exactly once, on the leader, when SQLite executes the transaction. The frames carry the outcome, not the recipe. A replica applying those frames cannot diverge, because it never rolls the dice.

The spike test, kept as a permanent unit test, runs a hostile workload on a live database while capturing every committed transaction: DDL with triggers and indexes, DML, 20-KiB blobs, savepoints with partial

rollbacks, `ALTER TABLE`, full rollbacks, and nondeterministic `randomblob()`/`random()` traffic. It then applies the captured payloads offline to a second image, checkpoints the original with truncation, and compares the two files byte for byte. The seeded fuzz harness extends the same oracle to randomized workloads with restarts, torn journal tails, and image deletion.

5.6 The fallback that was not needed

The plan reserved a custom VFS as a fallback: intercept `xWrite` on the WAL file if the public interfaces proved insufficient. They did not prove insufficient. The VFS remains a documented escape hatch in case a future SQLite release changes the WAL contract.

Exercise 5. Watch a WAL frame with your own eyes. In a scratch directory, open a database with the `sqlite3` shell, set `pragma journal_mode = wal` and `pragma wal_autocheckpoint = 0`, create a table, and insert one row. Hexdump the `-wal` file. Find the 32-byte WAL header, the first 24-byte frame header, and the big-endian page number at its offset 0. Then insert a second row and explain which bytes changed.

Hint: Each frame occupies 24 bytes plus one page. `pragma page_size` tells you the page size, so frame n starts at byte $32 + (n - 1) \times (24 + \text{page\_size})$.

Teach it back. Explain to a colleague why `randomblob()` is fatal to SQL-statement replication and harmless to WAL-frame replication. Use the words execute, capture, and apply, and name the one place in the cluster where nondeterminism is allowed to run.

6 Storage and recovery

Learning contract

By the end of this chapter you should be able to name every file in a node's data directory and say which ones are authoritative, state the five write-ordering rules and the invariant each protects, walk the recovery sequence in order and predict its behavior at any crash point, and explain how a snapshot seals an epoch.

6.1 The file set

One data directory holds everything a node knows:

```
data/
  LOCK                     exclusive process lock (flock)
  identity                 node id, database id, current configuration
  paxos-<config16hex>.log  framed, checksummed protocol journal (per epoch)
  payloads/aa/<62hex>      immutable frame payloads, named by SHA-256
  snapshots/<config>/      db image + manifest per sealed epoch
  snapshots/tmp-*          in-progress generations (crash debris, GC'd)
  CURRENT                  installed snapshot pointer
  current.db               materialized SQLite image (rebuildable)
```

Hold onto one rule: the journal, the payloads, and the snapshots are the database. `current.db` is a cache of applying them. The `-wal` and `-shm` files are working artifacts of the live connection, and every open deletes them before rebuilding. This is why the prediction exercise in chapter 1 was safe. Deleting `current.db` on a stopped node deletes a cache, and recovery rebuilds it from the authoritative files.

6.2 The five ordering rules

The host enforces five write-ordering rules, and the crash tests exist to catch any violation. Each rule exists to protect one invariant, so we state them as pairs.

1. **Payload before vote.** A sender queues payload bytes immediately before a dependent voter envelope on one ordered TCP/TLS stream. The receiver verifies and fsyncs the object before reading the next frame; its bounded missing-value gate covers reordering and reconnects. `payload_stored` caches readiness for later sends rather than serializing the normal accept. The invariant: every vote that can count toward a quorum is backed by a payload that is durable on the voter that cast it. A committed slot can therefore always be materialized. This is safety.
2. **Sync before durable claim.** Promise evidence, accepted replies, recovered values, commits, and client replies remain behind the journal barrier. The sole early class is a phase-two `accept` request: it asks another voter to persist a vote but does not claim the sender's own vote is durable. The host stays serialized until its barrier completes. The invariant: a node never claims durable state it could forget in a crash. A forgotten promise would let two quorums stop intersecting, so this rule is safety too.
3. **Name with the bytes.** Every authoritative create, link, or rename is followed by a sync that persists the new name. The invariant: an authoritative file survives a crash together with its directory entry. Without this rule, journal names, payload objects, `identity`, `CURRENT`, and snapshot generations could sync their contents and still vanish from the directory. Which handle carries that sync is a platform question, answered later in this chapter.
4. **Commit before apply.** A batch is applied only after its slot commits, and batches are applied contiguously in slot order. The invariant: the materialized image only ever reflects a decided prefix. Combined with chapter 5's deterministic apply, any snapshot plus any committed suffix rebuilds the same image.

5. **Acknowledge after session update.** The session row is updated inside the captured transaction, and the client is acknowledged only after that transaction is decided and applied. The invariant: exactly-once retries. If a client ever saw `ok`, the decided log contains the session row that records it, so a retry replays the saved result instead of applying twice.

6.3 How far a sync reaches

The five rules say **when** to sync. The sync policy says what a sync means. On Linux and the other supported platforms, `fsync(2)` flushes the drive's write cache, so a confirmed sync is durable against power loss. On macOS it does not: `fsync` hands the bytes to the drive but leaves them in its volatile cache, and a power cut can drop writes the kernel already confirmed. For rule 2 that is not mere data loss. A voter that forgets an acknowledged promise or accept can vote again, and two quorums stop intersecting — the same amnesia the interior-corruption rule below refuses to open with, inflicted by hardware instead of a damaged file.

The policy therefore has two modes, set once at startup for the whole process. `full`, the default for real binaries, makes every authoritative barrier on macOS issue `fcntl(F_FULLFSYNC)`, which flushes the drive cache; a filesystem that refuses the request falls back to plain `fsync`. `os` keeps plain `fsync` and is development-only on macOS: process-crash recovery is identical under both modes, and only power-loss durability differs. On the other supported platforms the two modes are the same syscall. The CLI sets the policy with `--sync` (chapter 2), embedded hosts call `zaxonlite.durability.setSyncMode` before opening a node, and test builds default to `os`, because the crash campaigns simulate process death, which loses nothing either way.

Full mode does not flush the drive cache once per file. `F_FULLFSYNC` empties the drive's entire cache, so one barrier per commit point covers every block already handed to the drive: a payload install flushes its object and directory entries with plain `fsync` — into the drive, not yet to stable media — and the journal sync that follows is the single full barrier that lands both together. That journal barrier precedes every vote acknowledgement, recovered value, and client acknowledgement (the sync-before-durable-claim rule above), so every counted vote still implies durable payload bytes at its consumer. Rarer transitions that create their own commit points — snapshot generations, epoch installs, the `CURRENT` pointer, backups — keep their own full barriers.

The steady-state cluster path overlaps those per-node barriers. The sender queues an immutable payload immediately before its phase-two accept. TCP order means the follower finishes the payload file and shard-directory fsyncs before it reads the accept; the receiver's bounded missing-payload gate covers reordering and reconnect races. Meanwhile the leader performs its own journal barrier. The leader mutex prevents accepted replies from entering Paxos until that barrier completes. Only an `accept` request is eligible for this early release: promises and accepted replies assert durable facts and remain behind the barrier. A later commit-only journal marker is derived from the durable accepting quorum and is omitted; phase one reconstructs it after a crash. The materialized follower database is likewise a rebuildable cache, so page apply does not add another full barrier.

6.4 Where a name becomes durable

Rule 3 says the name must survive with the bytes. It does not say which handle carries that promise, because the two platform families disagree.

POSIX keeps the name in the parent directory, so the parent is what gets synced, and it is synced after the rename. Windows keeps it somewhere else. NTFS is write-ahead logged: `$LogFile` is one sequential metadata journal per volume, so flushing it through a given record persists every record before it, and flushing a file forces the log through that file's last update. Microsoft documents the file rather than the directory as the durable unit — the way to be sure a newly created empty file has reached disk is to flush the file. There is no directory sync on Windows and none is needed.

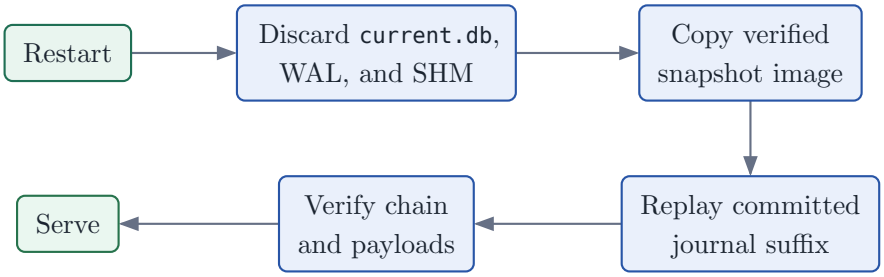

Figure 5: Restart never trusts the materialized database file. The image is discarded and rebuilt from the snapshot and the committed journal suffix, then checked against the log before the node serves.

The consequence is an ordering flip, and it is easy to get wrong. POSIX syncs the file, renames, then syncs the parent. Windows syncs the file, renames, then syncs the file again. Turning the directory sync into a no-op without moving the barrier leaves nothing flushed after the rename: it compiles, it passes the crash matrix — process death loses nothing under any sync mode — and it drops acknowledged writes on power loss. `syncPathnameTransition` hides the difference so no call site has to remember which way round it goes.

A directory sync may still be skipped where a later barrier is named, which is the `...BeforeBarrier` contract from the previous section. Both snapshot renames qualify: a snapshot becomes authoritative only when the stop sign naming it is journaled and synced, and an installed snapshot is followed at once by the `CURRENT` write. Transitions with no such successor — enrollment tokens, published identities, backups, journal creation — take their barrier immediately.

The model needs POSIX rename semantics and a metadata log, so Windows is supported from release 1809 and Server 2019 on NTFS. Rather than read a version number, a node probes: it creates a file, holds it open, and renames a second file over it. Replacing an open file is exactly what the older Windows rename cannot do, so one operation rejects old releases, FAT volumes, and network filesystems that quietly degrade. ZDS 0006 records the reasoning, and is candid that the log-ordering property is an inference from how NTFS is built rather than a documented guarantee.

6.5 The recovery sequence

`Node.open` performs these steps, in this order:

1. take the exclusive directory lock;
2. load or create `identity`;
3. open the payload store;
4. open the epoch journal and replay it into `DurableState`, truncating a torn final record and refusing to open on interior corruption;
5. restore the protocol node from the replayed state;
6. validate the installed snapshot: its identity, its epoch relation, its manifest fields, and its image digest, resuming an interrupted snapshot install if one is pending;
7. campaign, in a one-member configuration only;
8. rebuild the materialized image: always delete `current.db`, `-wal`, and `-shm`, copy the snapshot base, then chain-validate and offline-apply every committed batch;
9. complete a pending epoch rollover if a decided stop sign was replayed;
10. validate that the image's recorded `batch_id` equals the last committed descriptor's.

The last step closes the loop from chapter 4. The `batch_id` marker was written inside the captured transaction, so a rebuilt image that passes this check is provably the image the log describes.

> **Predict before reading on**
>
> Step 4 says the node may truncate the tail of its own journal during replay. Deleting protocol state sounds dangerous. When is it safe, and when would it violate a promise? Decide before reading the rule.

Figure 6: An epoch ends at a decided stop sign. Everything before the seal is covered by the installed snapshot, and the next epoch starts a fresh journal at slot 1.

Torn tail versus interior corruption

A record that fails to parse and touches end-of-file is a torn append. It is truncated and recovery proceeds, because rule 2 guarantees that an unconfirmed record was never mentioned to any peer. Nothing outside this process depends on it. A record that fails inside the valid prefix is different: it is corruption of state the node may have promised. The node refuses to open rather than vote with amnesia.

6.6 Snapshots and epoch rollover

The journal cannot grow forever. An epoch holds at most 2,048 slots, and the host checkpoints before the bound, reserving four slots so the stop sign always fits. A snapshot is not a local maintenance action here. It is a **decided** object, agreed through the same log it seals.

Rollover runs in four steps:

1. materialize: the leader runs a truncating checkpoint on its capture connection, while followers are already fully materialized offline;
2. build `snapshots/tmp-<config>/` with the database copy and a manifest recording the database id, the sealed configuration, the applied slot, the chain value, and the image's SHA-256, then rename the generation into place;
3. propose `checkpoint(metadata)` with metadata `zx1 <name> <manifest-sha256>`, the stop sign that seals the epoch;
4. when the stop sign commits, every member verifies its local generation against the decided digest, installs `CURRENT`, bumps `identity`, starts the next epoch's journal, and re-elects.

Why does a follower's digest match the leader's? Chapter 5 again: followers build their generation from the offline image, and byte-deterministic apply makes the two images identical, so their digests agree. Consensus decides one manifest hash, and every member can check itself against it.

Rollover is crash-resumable at every step. The decided stop sign lives in the sealed journal, so a restart replays it, and the completion re-runs idempotently until it succeeds.

6.7 Garbage collection

After a rollover the node keeps the new epoch's journal, the sealed epoch's journal as a fallback generation, the two newest snapshot generations, and every payload referenced by a retained journal. Everything older is covered by the installed snapshot and deleted. A payload is collected only when no retained journal references it. Age and ballot changes never delete a payload.

6.8 The journal format

Each record is framed so that replay can tell a torn tail from corruption:

Offset/size	Field	Meaning
0 / 4	`magic`	`0x315a584a` ("ZXJ1"), little-endian
4 / 1	`version`	format version, 1
5 / 1	`kind`	write tag: promise 0, accept 1, commit 2
6 / 2	`reserved`	zero
8 / 8	`sequence`	strictly increasing from 1 per epoch

Offset/size	Field	Meaning
16 / 4	`payload_len`	encoded write length
20 / 4	`crc32`	over header-sans-crc plus payload
24 / n	`payload`	canonical little-endian `Write` encoding

Writes encode ballots (`round:u64, priority:u32, node:u32`), slots, and entries. An entry is either a command descriptor or a stop sign carrying a configuration id, the members, and the metadata string.

Teach it back. Walk a colleague through one epoch rollover, from "epoch nearly full" to "next epoch elected", using the words stop sign, manifest, `CURRENT`, and `identity`. Then name the point in the sequence after which a crash can no longer lose the snapshot, and say which ordering rule makes that true.

7 Role-aware clusters

Learning contract

By the end of this chapter you should be able to name the five node roles and say which of them vote, trace every connection in a running cluster, explain why a vote is never processed before its payload bytes are durable, and follow a chosen value from the leader to a learner that never voted on it.

Checkpoint: storage

Chapter 6 defined the journal, the content-addressed payload store, and the snapshot generation. This chapter assumes all three. If "payload by digest" does not ring a bell, read chapter 6 first.

In chapter 1 you started three voters, killed the leader, and watched a write land anyway. This chapter explains the machinery behind that demo. Who talks to whom. Who votes. And what every byte must survive before a node is allowed to act on it.

7.1 Topology

`zaxon serve` hosts one node behind one TCP endpoint. Peer traffic and client traffic share that endpoint. The first frame on any connection is a `hello` that says which kind of connection this is. A single local node may instead listen on a Unix-domain socket (chapter 2); cluster links always require TCP, so a unix-mode server rejects configured peers.

Connections follow the roles. Voters dial every storage node. Learners dial only voters, so they can return payload ACKs and payload requests. A link between two learners would carry no protocol information, so it does not exist. Each connection sends in one direction only; the inbound side is receive-only. Every direction therefore has one unambiguous stream order. Gateways proxy client streams and never appear in the peer topology. Every link reconnects on its own with backoff, and every re-establishment runs protocol repair (`reconnected`).

The thread model is direct. There is one accept loop, one reader per connection, one sender per peer, and one tick thread. Each sender owns a bounded frame queue. The tick thread drives elections, heartbeats, and retransmission. A single mutex guards the node. The write path's fsyncs serialize under that mutex,

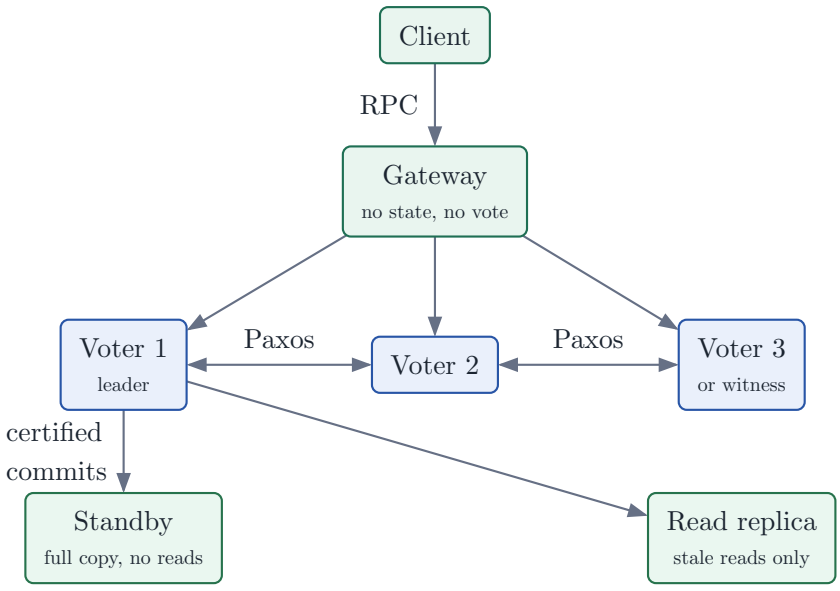

Figure 7: Who does what in a full cluster. The three voters vote and decide. The standby and the read replica learn chosen values but never vote. The gateway routes client connections and stores nothing.

exactly as the ordering contract requires. Admission is bounded. The server caps concurrent connections — peers, clients, and transfer streams together — at four per configured member plus sixteen by default, sized for a small cluster; an over-limit connection is closed at accept, and admission is refused during shutdown. An accepted connection must complete hello and authentication within a handshake deadline, 10 seconds by default, or the tick loop closes it. Established connections have a five-minute idle deadline and each peer is capped at two overlapping inbound connections. Remote SQL additionally has the row, byte, text, and VM-step budgets described in chapter 13.

7.2 Registry, voter membership, and scale

Every node is configured with the same role registry. The shared `database_id` is derived deterministically from the sorted voter ids, plus an optional `--cluster-id`. A mis-configured process therefore cannot join the wrong database. The `hello` handshake rejects it. Election priority is the node id. That gives deterministic tie-breaking and does not affect safety.

Only two roles enter Paxos quorums: `data-voter` and `witness`. A data voter may campaign, and it materializes SQLite. A witness votes but cannot campaign and does not serve SQL. The other storage roles are learners. A `standby` and a `read-replica` both store the chosen log and materialize SQLite, but they never send a promise or an accepted vote. A standby is promotion-eligible by an operator. A read replica is not. A `gateway` stores nothing and routes end-to-end authenticated client connections; chapter 12 covers it in detail.

Two registry rules are enforced at startup. Every registry must contain at least one data voter. An all-witness quorum would be safe, but it could never make progress, so it is rejected.

The first release bounds one Paxos group to one through nine voters. The bound keeps the fixed-size core state reviewable. The product registry is allocator-backed and may hold any runtime-sized number of learners and gateways, subject to host resources. Paxos has no theorem that limits a deployment to three or nine nodes. The implementation bound is a choice. Be clear about what extra replicas buy you: better read placement and better failure placement. They do not buy write throughput. SQLite has a single writer. Aggregate write scale requires independent databases or shards.

7.3 Payload gating: votes never outrun bytes

Chapter 5 split a transaction into a small descriptor and a large payload. The descriptor travels inside Paxos messages. The payload travels beside them. That split creates a hazard: a message can name bytes its receiver does not have.

> **Predict before reading on**
>
> A follower's connection drops and comes back. The leader holds an `accept` for slot 40 whose payload the follower may have missed. May the leader send the `accept` first and the payload bytes afterwards? Decide what could go wrong before reading on.

The transport closes the hazard with one correctness-critical rule: a member never processes a value-bearing promise, accept, or commit whose payload bytes are not durable and digest-verified in its own local store. A vote is a durable promise about a value. The rule makes sure the value's bytes are at least as durable as the vote.

On the normal path, the sender enforces the rule:

Step	Actor	Event and reason
1	Leader	Builds an `accept` for slot 40 that names payload hash `h`. It holds no storage ACK for `h` from voter 2 yet.
2	Leader	Queues `payload_data` carrying `h` and then the dependent `accept` on the same ordered stream. Its own journal barrier runs in parallel.

Step	Actor	Event and reason
3	Voter 2	Recomputes the digest over the received bytes. On a match it fsyncs and atomically installs the object in its payload store.
4	Voter 2	Only after storage completes does its read loop reach the adjacent `accept`. It may also return `payload_stored` to cache readiness.
5	Voter 2	Verifies `h` against its store, journals the accept, crosses its own barrier, and only then returns `accepted`. The vote cannot outrun the bytes.
6	Leader	Processes `accepted` only after the leader's own barrier has completed, so no volatile local vote can count toward the quorum.

Races can still deliver an envelope early. A reconnect or a dropped frame can hand a receiver an `accept` whose payload it lacks. Then the receiver enforces the rule itself:

1. It holds the envelope in a bounded queue. It does not step it.
2. It sends `payload_request` for the missing hash.
3. When `payload_data` arrives, it stores and fsyncs the bytes, then steps every held envelope for that hash.
4. Bounded means bounded. On overflow the envelope is dropped, and Paxos retransmission recovers it later.

Why are promises gated too? Because the core may complete phase one and emit recovery accepts inside the same `step` transition. A promise can carry a value the new leader must re-propose. Its bytes must be present before the step runs.

The reward for this discipline is a strong invariant. A follower stores the payload before it votes. So any chosen value has its bytes durable at a write quorum. Recovery can always fetch the payload by digest from some surviving voter. And a node missing a payload for a committed slot refuses to serve rather than invent state.

> **Production TCP is mTLS**
>
> Protocol v6 can authenticate possession of a provider-file PSK with a challenge-response: a fresh nonce, a connection-unique session key, and a monotonically sequenced HMAC on every post-handshake frame. A wrong proof, a replay, tampering, or a version downgrade closes the stream. The same PSK is used by all nodes and clients, while the plain `hello` supplies the claimed peer ID and connection kind. It therefore does not authenticate a distinct configured node, and HMAC does not encrypt SQL or page data. The mutual TLS transport closes both gaps: with `--tls-cert`/`--tls-key`/`--tls-ca`, every TCP connection — peer and client — runs TLS 1.3 with certificates verified against the cluster CA in both directions, and a peer connection's certificate common name must be exactly `zaxon-node-<id>` for the node id its hello claims, checked on the accepting and the dialing side alike. The mechanisms compose: when both are configured, the PSK challenge runs inside the TLS channel. Production startup rejects every TCP listener without TLS. The narrow `--dev-psk` exception is explicit and numeric-loopback-only; it is useful for the one-machine quickstart but retains the PSK's identity and confidentiality limits. Plaintext TCP remains behind a failpoint-gated test option. A local single-node service can use the owner-only Unix-domain socket instead (chapter 2).

Certificate bootstrap is intentionally smaller than cluster membership. An existing mTLS operator may request a one-time token for a non-revoked node already named in this registry. That node creates its key and CSR locally, and the configured issuer signs only the exact `zaxon-node-<id>` identity. The exchange neither changes the voter set nor grants a new role. Chapter 13 gives the operational sequence.

7.4 How a learner learns

A standby or a read replica holds a full copy of the database, yet it never votes. So how does it find out what was chosen? The leader tells it, with a certificate.

The message is `learner_commit(configuration, slot, entry)`. It is not a vote, and the learner never treats it as one. It is a statement from a configured voter that this slot was chosen with this entry. The leader sends it only after the referenced payload has received that learner's durable storage ACK. Learners pass through the same payload gate as voters.

Step	Actor	Event and reason
1	Leader	Applies slot 41. Its cursor for the standby still reads slot 40, so the standby is behind.
2	Leader	Reads the entry at slot 41. It names payload hash `h`, and the standby has not ACKed `h`. Sends `payload_data` first.
3	Leader	Encodes `learner_commit` for slot 41 and gates it behind the storage ACK for `h`. Advances the cursor once the certificate is queued.
4	Standby	Stores and fsyncs the payload, then replies `payload_stored`.
5	Leader	The ACK releases the certificate, and `learner_commit` goes out.
6	Standby	Checks that the sender is a configured voter and that the epoch matches. Verifies `h` in its store. Journals the entry and applies it in contiguous slot order. Remembers the sender as its leader hint.

A learner checks the claimed sender against its registry. It rejects certificates from claimed non-voters. Under the PSK transport this is a protocol invariant, not a hostile-peer security boundary, because any PSK holder can claim a configured voter ID in the hello; under mutual TLS the claimed peer ID is additionally bound to the sender's certificate name. It rejects other epochs and conflicting duplicates. It applies slots only contiguously and buffers at most a compile-time-bounded reorder window. A reconnect resets the leader's cursor and replays the chosen prefix; the replay is idempotent. The certifying voter also serves as the learner's redirect hint when a client sends it a leader-only request (chapter 12).

A learner also needs to know how fresh it is. The leader emits a learner heartbeat every 20 ticks, which is 500 ms at the default tick, carrying its current decided slot. A local `any` read may include `freshness_ms`. The learner rejects the read if leader contact is older than that bound, or if the heartbeat says the learner is behind. Without a freshness bound, `any` explicitly permits an arbitrarily stale local snapshot. Chapter 8 places `any` in the full read-level contract.

7.5 Commit, apply, and the follower image

Only the leader executes SQL. A follower never replays SQL text. It learns commits and applies the payload pages offline to `current.db`, using the same deterministic apply that recovery uses. Every follower image is therefore a function of the decided log alone.

Leadership changes create one subtle hazard. The old leader's live WAL may hold frames Paxos never decided, because an in-flight write lost its slot to the new leader. The host detects every such case in commit accounting. It closes the capture connection, discards the WAL, and rebuilds the image from the decided log before serving again. No undecided frame can leak into the served database.

> **Epoch fencing on the wire**
>
> Every envelope frame carries the sender's configuration id. A frame from an older epoch is dropped. A frame from a newer epoch means this member missed a sealed rollover, and it triggers a snapshot transfer. The Paxos messages themselves never cross epochs.

7.6 Catch-up and snapshot transfer

A member can fall behind in two ways. Each way has its own repair.

Within an epoch, the core protocol repairs a lagging voter. Reconnection sends `learn`. The leader resends missing commits. The tick loop requests catch-up whenever heartbeats reveal a decided slot ahead of the local one. A lagging learner is repaired the same way it learns everything else: the leader replays voter-certified chosen entries.

Across a sealed epoch, the journal alone is not enough. The member requests a snapshot instead. The peer streams the installed generation: the manifest first, then the database image in 1-MiB chunks. The receiver verifies the digest, installs `CURRENT` and `identity`, starts the new epoch's journal empty, rebuilds its image from the snapshot, and catches up the remaining suffix normally. The cluster test exercises exactly this path: a member stopped across a rollover rejoins and converges to byte-identical content.

> **Snapshot transfer confirms the existing proof**
>
> Normal rollover already gets consensus on the physical snapshot: the `zx1 <name> <manifest-sha256>` metadata is the decided Paxos stop sign, and every caught-up member independently requires the same digest. The receive path carries a canonical `ZXP1` encoding of that stop sign. The authenticated source counts as one matching voter report, and the receiver obtains enough independent matching probe replies to form a read quorum before replacing state. It then verifies the manifest and physical image. This is neither a second consensus phase nor signatures over SQLite files.

7.7 Failpoints

How do we know the crash matrix in chapter 8 holds on a real cluster? We kill real processes at chosen lines. A failpoint calls `_exit(137)` with no cleanup, which behaves like a SIGKILL at that line. The sites bracket the write path: before and after payload sync, after journal append, after journal sync, before follower apply, and before the client reply. A test controller arms them at runtime through an RPC. Only servers started with `--enable-failpoints` honor it. Zaxonlite intentionally has no internal administrator role, so every caller admitted to this single-principal interface could invoke an enabled failpoint. Never enable it outside disposable tests.

The cluster scenario's centerpiece is the retry you will meet again in chapter 8. It kills the leader after quorum choice and before the client reply. It retries the same session sequence at the new leader. And it proves the write applied exactly once.

Exercise 7. Extend the chapter 1 cluster with a read replica: start a fourth node with `--role read-replica` and list it on every voter as `--peer 4@.../read-replica`. Write through the cluster, stop the replica, write again, and restart it. Explain which messages repair it, why it never affected the write quorum, and what an `any` read against it can and cannot promise.

Hint: The repair is the learner path: payload data, storage ACK, then voter-certified `learner_commit` replay from the leader's reset cursor.

Teach it back. Explain payload gating to a colleague in four sentences. Name the rule, name who enforces it on the normal path, name who enforces it after a reconnect race, and finish with the invariant it buys: every chosen value has its bytes durable at a write quorum.

8 Consistency and the client contract

Learning contract

By the end of this chapter you should be able to classify any write outcome as success, failure, or ambiguous, retry an ambiguous write without double-applying it, choose a read level and state exactly what it guarantees, and reconstruct the safety argument for the quorum read fence step by step.

Checkpoint: the cluster

Chapter 7 showed how voters decide, and how payload bytes reach a quorum before any vote references them. This chapter states what a client may rely on because of that machinery. You first used sessions in chapter 1; chapter 12 shows the same contract at the wire level.

8.1 Write acknowledgement: three outcomes, not two

We would like every write to either succeed or fail. A distributed system cannot promise that. A reply can be lost after the work is done.

Predict before reading on

Your `exec` call times out. One second later the cluster is healthy again. Did your insert happen? Decide which answers are possible, and what a safe client may do next, before reading on.

A client write returns one of three outcomes. The distinction is the contract:

1. **Success.** The slot committed, and the decided value at that slot is this client's `batch_id`. The transaction is durable at a quorum and applied.
2. **Failure.** The SQL failed and rolled back, or the session sequence was rejected. Provably nothing replicated. There is no side effect.
3. **Ambiguous.** An `ambiguous` reply, a timeout, or a connection loss. The write may or may not have committed. Both answers to the predict question are possible, and the client cannot tell which one happened.

The ambiguous case has exactly one safe response: retry the same session sequence. Never re-execute the SQL blindly. If the write did commit and you resend it as a fresh statement, it applies twice. Sessions exist to make the disciplined retry cheap.

8.2 Exactly-once sessions

A session is a replicated row: `(id, next_sequence, last_sequence, last_changes, last_activity_slot)`. Every sessioned `exec` carries `(session, sequence)`. The leader checks the row before it touches SQL:

1. `sequence == last_sequence`: this write already committed. Return the recorded result, marked `replayed`. No SQL runs.
2. `sequence == next_sequence`: this is the next write. Execute it. The row update rides inside the captured transaction, so the result and the data commit atomically or not at all.
3. `sequence < next_sequence - 1` returns `ResultExpired`. `sequence > next_sequence` returns `SequenceGap`. An unknown id returns `UnknownSession`. None of these execute SQL.

The atomicity in the second rule carries the whole guarantee. The session row is not bookkeeping beside the data. It commits inside the same decided value as the data. Whoever holds the data also holds the proof that sequence `n` was spent.

Walk through the worst case. This is the exact schedule the failpoint suite kills on purpose (chapter 7):

Step	Actor	Event and reason
1	Client	Opens a session, then sends `exec` with sequence 7, an insert.
2	Leader	Checks the row: 7 equals `next_sequence`, so it executes the insert. The session row update is captured inside the same transaction payload.
3	Cluster	The slot reaches quorum choice. The write is decided.
4	Leader	Crashes before sending the reply. The client sees a timeout. The outcome is ambiguous.
5	Voters	The survivors elect a new leader. Phase-one recovery finds the chosen slot, commits it, and applies it. The new leader's image now holds the inserted row and the session row with `last_sequence = 7`.
6	Client	Retries with the same session, the same sequence 7, and the same statement, now at the new leader.
7	New leader	Checks the row: 7 equals `last_sequence`. It returns the recorded result, marked `replayed`. The insert exists exactly once.

The session table travels inside the replicated frames. So the retry lands correctly at any future leader, not only at the process that executed the write. The cluster failpoint test demonstrates this full loop.
Sessions are bounded. Every session write bumps a replicated activity counter (`write_seq`) and stamps the session. `expire-sessions --retain n` deletes sessions idle for more than the last `n` session writes, as a normal replicated write. Storage stays proportional to active sessions plus the result window, not to history.

8.3 Read levels

You choose how much a read costs and how much it promises:

Level	Semantics	Cost
`any`	The local applied state of whichever member answered. It may lag the leader, and the response says so in a label.	none
`leader`	The leader's applied state. It is serializable against the single writer, and it can be stale only across an unnoticed leadership change.	redirect
`linearizable`	It includes every write acknowledged before the read began, and it stays current across leader changes.	one fence round

The default is `linearizable`, as you saw in chapter 1. Chapter 7 explains the `freshness_ms` bound that tightens `any` on a learner, and chapter 12 shows how a client requests each level.

8.4 The quorum read fence

How do we make a read linearizable without writing to the log? The leader proves that it is still the leader. The proof is a fence. It performs no log append and no disk sync:

1. The leader records its current ballot b and a fence slot $s = \text{decidedThrough}$.
2. It probes every peer with one question: is b still exactly the ballot you have promised?
3. Each peer answers from its durable `promised`. The answer is an equality check. Nothing is written.
4. The fence completes when a read quorum confirms b and the leader has applied through s. For three voters, that quorum is the leader plus one.
5. Only then does the query run, on applied state.

Why is this safe? We argue it in five steps.

1. Ballots are totally ordered, and a member's promise only moves upward.

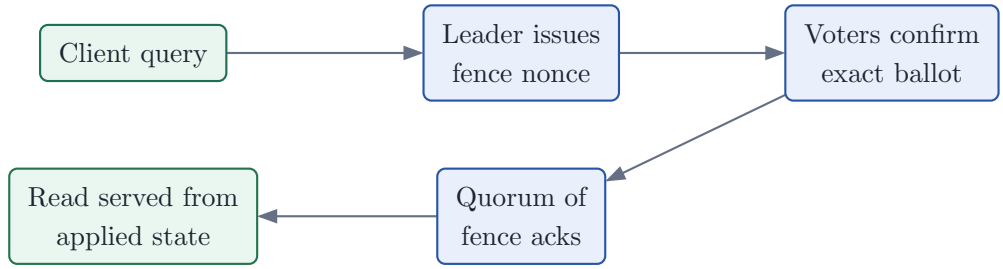

Figure 8: A linearizable read costs one fence round and no disk writes. The leader may answer only after a quorum confirms its exact ballot and it has applied through the fence slot.

2. Suppose some write was acknowledged after our fence began but is missing from our state applied through s. That write completed under a higher ballot $b' > b$, and completing it required a write quorum of promises at b'.
3. Every write quorum intersects every read quorum. So at least one member of our fence quorum had already promised b' when we probed it.
4. That member answers with an equality check against its durable promise. Having promised b', it no longer answers "exactly b". It would have failed our fence.
5. Our fence completed, so every probed member did answer "exactly b". The supposed write cannot exist. The state applied through s holds every acknowledged write, and the read is linearizable.

A fence also fails immediately if the leader's own ballot or role moves while the fence is outstanding.

> **The committed barrier stays as the oracle**
>
> The log still carries a `read_barrier` command. Appending one and waiting for it is the slow, obviously correct reference read. The test suites use it as a differential oracle for the fence path. Production reads use the fence.

8.5 The crash matrix

Chapter 4 gave a write fixed stations: store the payload, append the accept, release the accept request while the local journal sync runs, make a quorum of votes durable, reach quorum choice, apply, reply. The leader does not sync a separate commit marker: a durable accepting quorum is the recovery proof, and the materialized SQLite file is rebuildable state. Now pull the power cord between each pair of stations and ask two questions. What does the client see? And what does recovery do with the pieces?

Three answers repeat across the matrix, so learn them once:

1. Before any vote references the payload, the write simply does not exist. Nothing can be inferred from leftover bytes, and the client never saw success.
2. After a vote is durable but before quorum choice, the write's fate is open. A later leader's phase one may recover it or may not. That is exactly why the client outcome is "unknown" and why the retry uses the session sequence.
3. After quorum choice, the write's fate is sealed. Recovery must preserve and commit it, apply is contiguous and exactly-once, and the session retry turns "unknown" into the recorded result.

The full matrix pins each boundary:

Crash point	Client outcome	Recovery oracle
Before payload sync	failure or unknown, never success	No vote references the incomplete payload. The store's temp-file install leaves no partial object.
After payload sync, before accept append	unknown	The payload may be orphaned garbage. GC collects it. Nothing infers it was chosen.

Crash point	Client outcome	Recovery oracle
After accept append, while the local barrier runs	unknown	The local torn tail is truncated. A pipelined phase-two request may already have made another acceptor's vote durable; phase one preserves it if a quorum chose it. No client success has been returned.
After accept sync, before a durability-bearing reply	unknown	The local vote survives replay and is available to a later leader. Promise evidence and accepted replies never leave before this point.
After quorum choice, before client reply	unknown	Election recovery preserves and commits the chosen value. The session retry replays the recorded result.
After chosen, before apply	unknown or delayed	The local commit marker is derived rather than separately synced. Phase-one recovery reconstructs the choice from durable accepts and contiguous replay applies it once.
After apply, before reply	unknown	The same sequence returns the stored result. It never applies twice.
During snapshot install	existing traffic unaffected	Restart picks a complete generation, old or new, and never a `tmp-*` directory.

One honesty note. The hooks and harnesses exist, but the current automated suite does not yet exercise every row in both one-node and three-node roles. It covers torn tails, stale and corrupt image replacement, both snapshot transition prefixes, and the cluster leader kill after choice and before reply. Completing the remaining rows with the required schedule counts is a release blocker in the product plan.

Exercise 8. Take the crash matrix and mark every row where the write may actually have committed even though the client outcome says unknown. For each marked row, state what a retry of the same session sequence returns at the recovered cluster, and why the answer is never a second apply.
Hint: Split the rows at quorum choice. Before it, recovery may lawfully drop the write, and the retry executes fresh. From quorum choice onward, the retry must return the recorded result.

Teach it back. Explain the fence to a colleague, then answer their follow-up: why must the peer confirm the exact ballot b, and not "any ballot at least b"? Use the quorum intersection step of the safety argument in your answer.

PART III

Embedding

The same engine now runs inside your process, first as one durable node and then as a full cluster member. We embed it from Zig, then cross the C ABI so any language can follow.

9 Embedding a node in Zig

Learning contract

By the end of this chapter you should be able to open a durable single-member node from Zig, say what is synced before each call returns, choose between `exec`, `execPrepared`, and `execTransaction`, retry a write safely through a session, own the memory of every value the node returns, and name the extra surface a transport host must drive.

Checkpoint: the write path

This chapter assumes you know what a captured WAL payload is (chapter 5) and why the journal outranks the SQLite image (chapter 6). The API below is those two chapters turned into function calls.

This chapter serves the single-durable-node embedder. That means one process, one data directory, and no network. The same `Node` type also serves cluster members. Where a cluster member has an extra obligation, we say so explicitly.

9.1 Lifecycle: `open` and `close`

API anchor: `Node.open(gpa, io, options) !*Node`

Creates or recovers one node from one data directory and returns a heap-allocated handle owned by the caller. in `zaxonlite/src/node.zig`

```
const zaxonlite = @import("zaxonlite");

var node = try zaxonlite.Node.open(gpa, io, .{ .directory = "./data" });
defer node.close();
```

That is the whole lifecycle for a single durable node. The options struct has six fields, and five of them have defaults:

Field	Default	Meaning
`directory`	required	The node's data directory. It is created when missing.
`node_id`	`1`	This node's Paxos identity.
`members`	empty	The full voting membership, including this node. Empty means a one-member configuration of just `node_id`.
`leader_priority`	`0`	The election priority carried in this node's ballots.
`database_id`	`null`	The database identity for a fresh directory. `null` draws a random one.
`role`	`.data_voter`	The product role from chapter 7.

Three of these fields are identity: `node_id`, `database_id`, and `role`. The first `open` persists them in the directory's identity file. Every later `open` checks the options against that file. Cluster members must agree on `database_id`; a random one is only right for a single node. `open` refuses to proceed when anything disagrees:

Open-time error	When `open` returns it
`error.NodeLocked`	Another live process holds the directory's exclusive `LOCK` file. One process per directory is the rule.
`error.NodeIdMismatch`	The `node_id` option differs from the persisted identity.

Open-time error	When `open` returns it
`error.DatabaseMismatch`	The `database_id` option differs from the persisted identity.
`error.NodeRoleMismatch`	The `role` option differs from the persisted identity.
`error.TooManyMembers`	The membership names more than nine voters.
`error.RoleHasNoLocalStore`	The role is `.gateway`. A gateway owns no data directory, so it cannot be opened as a `Node`.
`error.RoleMembershipMismatch`	A voter is missing from `members`, or a non-voter appears in it.

The directory is the database. It holds the identity file, the exclusive `LOCK`, one `paxos-*.log` journal per epoch, the content-addressed `payloads/` store, the `snapshots/` directory with its `CURRENT` pointer, and the materialized `current.db` image. The journal and the payload store are authoritative. The SQLite image is not. Every `open` deletes the image and rebuilds it from the verified snapshot plus the committed journal suffix. A tampered or deleted image therefore never outranks Paxos state.

A one-member node campaigns during `open`. A one-member quorum completes immediately, so `open` returns ready to serve. The schema-bootstrap write runs on the first open.

> **Predict before reading on**
>
> `close` performs no flush of its own. Is that a durability hole? Decide before reading on.

It is not. Nothing is ever unflushed: every acknowledged write was journaled and fsynced before its call returned. `close` releases everything `open` acquired, including the `*Node` allocation itself.

9.2 Your first write: `exec`

> **API anchor: `Node.exec(sql) !ExecResult`**
>
> Executes one write transaction and replicates its captured WAL frames. in `zaxonlite/src/node.zig`

The SQL must be a null-terminated slice, typed `[:0]const u8`. The node borrows it only for the duration of the call. Every write follows one durability ordering, and `exec` does not return success until the required prefix of it is done:

1. The SQL executes inside `BEGIN IMMEDIATE ... COMMIT` on the capture connection.
2. The committed WAL frames are captured and persisted as one payload in the content-addressed store. The payload is written, fsynced, and linked before anything may reference it.
3. The fixed-size descriptor is appended through the replicated log. The resulting durable writes are journaled and fsynced.
4. Only now may protocol messages leave the node. Only now is the slot committed and applied.

What the returned `ExecResult` means depends on the configuration. In a one-member configuration the result carries `.changes` and `.slot`, and the slot is already committed and applied locally. Acknowledged means durable. In a cluster the same call returns once the local journal write is durable. Commitment

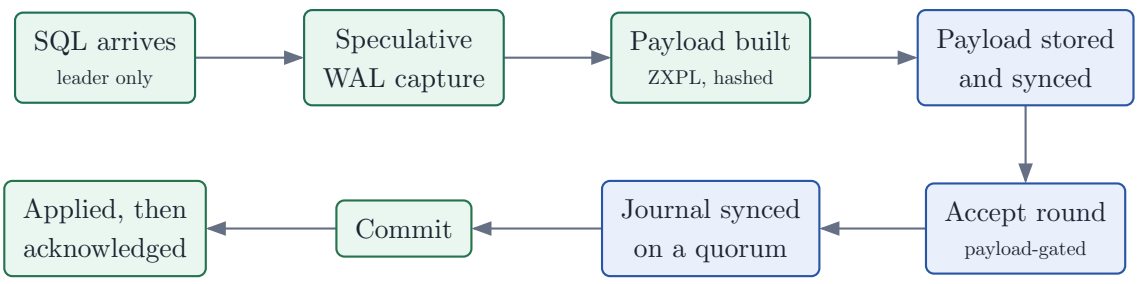

Figure 9: Every `exec` walks this path. Nothing leaves the node, and nothing is acknowledged, before the payload and journal syncs complete.

still needs a quorum. The transport host must await the slot's commitment and check `pendingBatchId` before acknowledging a client. Chapter 8 states that acknowledgement rule precisely.
Failure has two sides, and they are not symmetric. A failed SQL statement rolls back before capture. Nothing about it is replicated. The SQLite message is readable through `lastSqliteMessage()`, bounded at 512 bytes and valid until the next SQL call. An error between the local commit and the log append marks the image for resync. The next call repairs it automatically.

> **A failed fsync is fatal and sticky**
> A journal or payload fsync failure poisons the node. Every later call returns `error.StorageFailed`. This protects the effects contract: a node that cannot make its state durable must stop talking. Stop using the node, repair the storage, and open the directory again. Recovery rebuilds from the verified journal.

One more refusal matters here. A sealed epoch returns `error.LogSealed`. A one-member node avoids it by checkpointing automatically as its 2,048-slot epoch fills.

9.3 Prepared statements: `execPrepared`

String SQL is fine for schema work. Data should travel as typed parameters. `execPrepared` binds values of type `prepared.Value`, exported as `zaxonlite.Value`. The variants are `null_value`, `integer: i64`, `real: f64`, `text: []const u8`, and `blob: []const u8`. Parameter bytes are borrowed only for the duration of the call. The value count must match the statement's placeholders, or the call fails with `error.ParameterCountMismatch`.

```
const result = try node.execPrepared("insert into items(v) values (?1)",
    &.{.{ .text = "tea" }});
std.debug.assert(result.changes == 1);
```

9.4 Multi-statement transactions: `execTransaction`

> **API anchor: `Node.execTransaction(*Transaction) !ExecResult`**
> Commits a completed transaction builder as one replicated WAL transition. in `zaxonlite/src/node.zig`, `zaxonlite/src/prepared.zig`

A `zaxonlite.Transaction` is a builder, not a live SQLite transaction. Each `txn.exec` copies its SQL and parameter bytes into the builder's own arena. Nothing touches the database until you commit the builder. This is deliberate. Application think-time never holds database locks. No speculative SQLite state can survive a leadership change, because none exists until commit.

```
var txn = zaxonlite.Transaction.init(gpa);
defer txn.deinit();
try txn.exec("insert into items(v) values (?1)", &.{.{ .text = "a" }});
try txn.exec("update items set v = ?1 where id = ?2",
    &.{ .{ .text = "b" }, .{ .integer = 1 } });
const result = try node.execTransaction(&txn);
```

You own the builder. Call `deinit` on it exactly once; the node keeps no reference. The bounds are hard. A builder accepts at most 1024 statements (`error.TooManyStatements`) and 64 MiB of copied input (`error.TransactionInputTooLarge`). An empty builder cannot be committed (`error.EmptyTransaction`). The replicated payload itself is bounded at 64 MiB minus framing (`error.TransactionTooLarge`).
A builder is single-use, even when execution fails. Reusing one returns `error.TransactionFinished`. This is a safety choice, not a convenience gap. Retry belongs to an idempotent session, never to accidental re-submission.

9.5 Exactly-once sessions

Chapter 8 defined the problem: an ambiguous outcome, such as a timeout, leaves you unsure whether a write committed. Sessions are the node-level answer. `openSession()` creates a replicated session row and

returns its `u64` id. A session permits one outstanding sequence, starting at 1. `execIdempotent(session_id, sequence, sql)` then decides among five cases:

1. The sequence is the permitted next one. The SQL executes exactly once. The session-row update rides inside the same captured transaction as your SQL, so the result and the data commit atomically.
2. The sequence equals the last executed one. The recorded result returns with `.replayed = true` and `.slot = 0`. No SQL executes.
3. The sequence is older than that: `error.ResultExpired`, and no SQL executes.
4. The sequence skips ahead: `error.SequenceGap`, and no SQL executes.
5. The session id is unknown: `error.UnknownSession`, and no SQL executes.

Only the last result is retained. That bound is why exactly one sequence may be outstanding. The retry rule follows from the case list. On an ambiguous outcome, retry the same session and the same sequence with the same statement. Never re-execute blindly. The retry either lands as the first execution or replays the recorded result.

```
const session = try node.openSession();
const sql = "insert into items(v) values ('x')";
const first = try node.execIdempotent(session, 1, sql);
// Ambiguous outcome? Retry the SAME sequence. Never re-execute blindly.
const again = try node.execIdempotent(session, 1, sql);
std.debug.assert(again.replayed and again.changes == first.changes);
```

Two helpers round out the surface. `checkSession(session_id, sequence)` runs the same validation without executing anything. `expireSessions(retain)` bounds the session table: as a normal replicated write, it deletes every session more than `retain` session writes behind the newest one. An expired id fails closed with `error.UnknownSession`.

9.6 Reads: `query` and `queryPrepared`

`query(gpa, sql)` and `queryPrepared(gpa, sql, values)` run one read-only statement. A write hiding inside a read is rejected with `error.WriteInReadQuery`. The returned `QueryResult` owns its memory through an internal arena on the `gpa` you pass. Column names and cell bytes are copies. Nothing borrows from the node. You call `rows.deinit()` exactly once. Cells arrive as SQLite text conversions, typed `?[]const u8`, with `null` preserved.

```
var rows = try node.query(gpa, "select id, v from items order by id");
defer rows.deinit();
for (rows.rows) |row| std.debug.print("{s}\n", .{row[1] orelse "null"});
```

Which connection serves the read depends on the member. A node holding the live capture connection reads through it. Any other member opens a short-lived connection against the materialized image. A fresh member with no image yet reports `error.NoDatabaseImage`.

Read levels (`any`, `leader`, `linearizable`) are the transport host's vocabulary, defined in chapter 8. A `Node`-level query always answers from the local applied state. On a single durable node that state is by definition current, so every read is linearizable.

9.7 Maintenance surface

Call	Contract
`snapshot()`	Online checkpoint. It materializes every committed frame, seals the epoch with a stop sign, installs the snapshot generation, and starts the next epoch. One-member only (asserted). Cluster hosts call `prepareCheckpoint()` and finish with `completeClusterRollover()` once the stop sign is decided. No write may be in flight (`error.WriteInFlight`).

Call	Contract
`backup(destination)`	Streams a consistent logical backup (`VACUUM INTO`) to a path. `openBackup()` instead returns a `BackupHandle` carrying the file, its size, and its SHA-256 for bounded streaming. The caller must `close` the handle, which deletes the temporary image.
`contentHash() ![32]u8`	Deterministic digest of the logical database content. Identical applied history gives an identical digest.
`integrityCheck() !IntegrityReport`	Verifies three things: the SQLite image (`sqlite_ok`), the descriptor chain across the decided epoch (`chain_ok`), and payload availability plus descriptor agreement (`payloads_ok`). `report.ok()` folds the three.
`status() Status`	A point-in-time view returned by value: node and database identity, configuration id, protocol role and product node type, leader, ballot, decided and applied slots, journal record count, the 2,048-slot epoch capacity, chain hash, page size, and installed snapshot name. The `role` and `node_type` strings are static.

9.8 Advanced: hosting the transport yourself

Everything above assumed the node drives itself. A cluster member does not. Its host owns the sockets, so the host must also drive the protocol, exactly as `server.zig` does. Most embedders should reach for the `Embedded` facade in chapter 10 instead. Read this section when you are building your own transport.

Call	What it does
`stepEnvelope(envelope)`	Processes one protocol message from a peer.
`tickProtocol()`	Advances election, heartbeat, and retransmission timers. The host calls it on a steady cadence.
`peerReconnected(peer)`	Repairs protocol traffic after the transport re-establishes a peer connection.
`requestCatchUp(peer)`	Requests decided entries this node is missing, starting after its applied slot.
`learnChosen(...)`	Accepts a voter-certified chosen entry on a non-voting learner. The arguments are the sender, the slot, and the entry.
`campaign()`	Starts phase one explicitly.

Each of these journals and fsyncs its durable effects before any outbound envelope is queued. Sync-before-send is enforced inside the node. It is not left to the host. Outbound messages accumulate in `node.outbox`. The host drains the outbox after every protocol or write call, then calls `clearRetainingCapacity()` on it. This is exactly what `server.zig` does.

The remaining obligations are flags the host must poll and obey:

1. `needsResync()` demands `resyncImage()` before further service.
2. `ensureWriter()` opens the capture connection when leadership is gained.
3. `epochNearlyFull()` demands a checkpoint before the next append.
4. `storageFailed()` means stop voting and stop serving. Talking after a failed fsync would violate the effects contract.

`isLeader()`, `currentLeader()`, `pendingBatchId()`, and `lastAppend()` complete the read-only view.

Exercise 1. Write a program that opens a node, opens a session, and inserts one row with `execIdempotent` at sequence 1. Run the same call a second time. Predict the `changes`, `slot`, and `replayed` fields of both results before you run it. Then call sequence 3 and explain the error you get.
Hint: Only one sequence may be outstanding, and only the last result is retained.

Teach it back. Without rereading, write down the ordered steps between `node.exec(sql)` being called and its `ExecResult` returning on a one-member node. Mark which steps are fsynced. Then reread the module comment at the top of `zaxonlite/src/node.zig` and list any step you missed or reordered. Most first attempts misplace the payload sync relative to the log append.

10 Embedding a cluster in Zig

Learning contract

By the end of this chapter you should be able to start a cluster member inside your own process with `Embedded.open`, say which parts of the stack the facade owns and which parts your host owns, route writes to the leader without knowing who the leader is, and state plainly what the facade does not do for you.

Checkpoint: the embedded node

Everything here wraps chapter 9's `Node`. You should know what `exec` syncs before it returns, and who owns a `QueryResult`, before we add a network to the picture.

Chapter 9's `Node` leaves networking to you. The `Embedded` facade, exported as `zaxonlite.Embedded`, is the batteries-included member. Each facade owns one node, its TCP listener, its peer senders, its tick loop, and client routing. An application creates one facade per process. The same API serves one through nine voters, plus any number of non-voting replicas and gateways.

10.1 The member registry and `open`

API anchor: `Embedded.open(gpa, io, options) !*Embedded`

Starts a full cluster member, server thread included, and returns once its listener answers, or fails within the startup timeout. in `zaxonlite/src/embedded.zig`

Every member passes the same registry: a slice of `zaxonlite.EmbeddedMember` values. Each entry carries an `id`, an `address` in `host:port` text, and a `role` that defaults to `.data_voter`. `OpenOptions` adds `directory` and `node_id`, which say which member this process is. It also takes an optional `cluster_id`, an optional `tls` provider-path identity, an optional defense-in-depth `auth_secret`, and `startup_timeout_ms` with a default of 10,000. Production TCP requires `tls`; the plaintext switch exists only for failpoint-gated tests. The CLI's loopback-only `--dev-psk` convenience is deliberately not exposed by this production embedding facade.

There is no `database_id` field here. The database identity is derived from the member registry plus `cluster_id`. Agreeing on the registry is agreeing on the database. Changing either later names a different database.

Validation happens before any thread starts, and the error names are the contract:

Open-time error	When `open` returns it
`error.InvalidMemberCount`	The registry is empty.
`error.InvalidNodeId`	A member id is zero.
`error.DuplicateNodeId`	Two registry entries share an id.
`error.InvalidVoterCount`	The registry has zero voters, or more than nine. Witnesses count as voters; learners do not.
`error.CampaignerRequired`	No voter can campaign. Such a cluster would be leaderless forever, by construction, so `open` refuses it.
`error.NotMember`	Your `node_id` does not appear in the registry.

After validation the facade copies every string into its own arena. You may free your registry and address buffers the moment `open` returns. `open` then spawns the server thread and polls its own endpoint until

a status call answers. A server that dies first surfaces as `error.ServerStartupFailed`. Silence past the timeout surfaces as `error.ServerStartupTimeout`.

10.2 Who owns what

The facade owns	Your host owns
The TCP listener and the server thread behind it.	The allocator and the `Io` you pass in. Both must outlive the facade.
The peer senders and the protocol tick loop.	The data directory chosen for each member.
The node, with its journal, payloads, snapshots, and image.	The retry policy around ambiguous write outcomes (chapter 8).
Client routing, including leader redirects.	Freeing every response body returned by `call`.

10.3 `exec`, `query`, and `call`

The facade speaks to its own cluster through the client RPC protocol. That is what makes leader routing uniform: your process is a client of itself.

1. `exec(sql) !ExecResult` runs one replicated write, routed to the leader. The result is the decided outcome. Acknowledged means committed by a voter quorum and applied.
2. `query(gpa, sql) !QueryResult` runs a `linearizable` read, using the quorum fence from chapter 8, routed to the leader. The result owns its memory through an arena on the `gpa` you pass. Call `deinit()` exactly once.
3. `call(request, leader) ![]u8` sends any raw JSON request from the RPC vocabulary: `session`, `wait`, `status`, `members`, `snapshot`, `backup`, `integrity`, `expire-sessions`, and the rest. With `leader = true` the request routes to the leader. With `false` the first reachable member answers. You own the returned body. Free it with the `gpa` the facade was opened with.

Routing tries endpoints in rotation and follows the `not_leader` redirect hint each declining member returns. It retries for up to twelve attempts with 150 ms pauses, which is enough to ride out an election. After that it gives up with `error.NoLeaderReachable`.

Step	Actor	Event and reason
1	You	Call `exec` on member 2's facade. Member 2 is not the leader, but you do not need to know that.
2	Facade	Sends the RPC to the first endpoint in its rotation.
3	Member	Declines with `not_leader` and names the leader in the redirect hint.
4	Facade	Retries against the hinted leader.
5	Leader	Runs the replicated write and answers with the decided result once the slot is committed and applied.

`exec` and `query` flatten any other error response to `error.RemoteOperationFailed`. Sometimes you need more than that. The structured error code (`sql`, `ambiguous`, `stale`, `forbidden`, and the rest) and the `replayed` marker of a session retry only appear in the JSON body. For those, use `call` and parse the response yourself. Idempotent sessions in particular are reached through `call`. The facade adds no session helper.

When `tls` is set, every connection the facade makes or accepts runs mutual TLS 1.3. The facade copies the three provider paths, creates separate server and client TLS contexts, binds peer certificate common names to configured node IDs, and keeps one client connection open across calls. `auth_secret`, when also set, adds the sequenced PSK/HMAC exchange inside TLS. It is not a substitute for TLS. Each member

supplies its own node certificate; applications may use the same identity for the facade's internal client calls because the application is the single database authority.

10.4 Walkthrough: three voters in one process

This is the shape of `zaxonlite/src/role_cluster_test.zig`, reduced to three data voters. A deployed cluster normally uses three processes on three machines; production TCP uses the facade's mutual TLS option. First we write the registry. Every member will pass this same slice:

```
const zaxonlite = @import("zaxonlite");

const members = [_]zaxonlite.EmbeddedMember{
    .{ .id = 1, .address = "127.0.0.1:9701" },
    .{ .id = 2, .address = "127.0.0.1:9702" },
    .{ .id = 3, .address = "127.0.0.1:9703" },
};
```

Next we open the three members. Each gets its own directory and its own `node_id`. Everything else is shared:

```
var nodes: [3]*zaxonlite.Embedded = undefined;
for (&nodes, 0..) |*node, index| {
    node.* = try zaxonlite.Embedded.open(gpa, io, .{
        .directory = directories[index], // one directory per member
        .node_id = @intCast(index + 1),
        .members = &members,
        .cluster_id = "example",
        .tls = .{
            .cert_path = certificate_paths[index],
            .key_path = private_key_paths[index],
            .ca_path = "./pki/ca.crt",
        },
    });
}
defer for (nodes) |node| node.close();
```

The cluster is now electing its first leader. A write issued before that election finishes fails cleanly, so the first write retries until a leader wins:

```
var elapsed: u64 = 0;
while (elapsed < 20_000) : (elapsed += 100) {
    if (nodes[0].exec("create table t(v text)")) |_| break else |_| {}
    io.sleep(.fromMilliseconds(100), .awake) catch {};
}
```

From here, any member's facade accepts any operation. Routing finds the leader for us:

```
_ = try nodes[1].exec("insert into t values ('chosen')"); // any member
var rows = try nodes[2].query(gpa, "select v from t");
defer rows.deinit();
```

Mixed roles use the same registry. The six-member test adds a `.witness`, a `.standby`, and a `.read_replica`. A member whose own role is `.gateway` runs the stateless gateway loop instead of a node. It keeps no data directory contents and no Paxos state. It routes bytes, unmodified, to the members whose roles serve reads or writes, with TLS remaining end-to-end to a storage backend.

10.5 Shutdown ordering in `close`

`close` is deliberate about order. A normal member first asks its own server to stop through a client `stop` RPC. A gateway instead flips its shutdown flag and pokes the listener awake. `close` then joins the server thread. The listener, peer senders, and node therefore close on the server's own path, with the journal already durable. Only then does `close` free the arena-held registry and the facade itself.

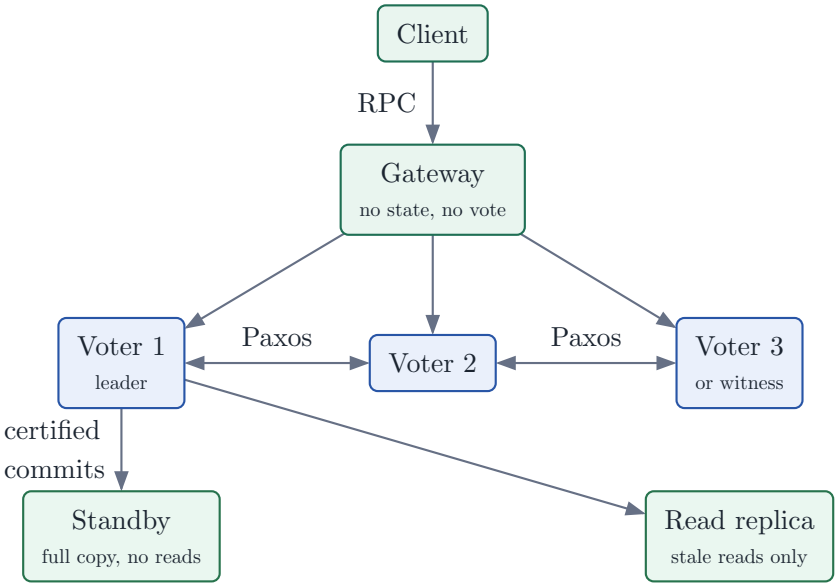

Figure 10: One registry can describe every role at once. Voters replicate, the gateway routes without storing anything, and learners receive certified commits.

Close members in any order. Once fewer than a quorum of voters remain, the survivors keep serving `any`-level reads. Writes and linearizable reads block until a quorum returns. A request still in flight when `close` runs may or may not have committed. Its caller must treat the outcome as ambiguous, exactly as in chapter 8. `close` never returns an error, and it is safe to call after the server has already exited on its own.

10.6 What the facade does not provide

Stated directly, per the release limits in `docs/zds/records/0004-zaxonlite-format.typ`:

1. **No automatic membership change.** The voter set is fixed at open. Replacing a failed voter is an operator procedure, not an API call.
2. **No dynamic registry.** Adding even a read replica means restarting members with the new registry, keeping the same voters and the same `cluster_id`.
3. **No dynamic identity platform.** `OpenOptions.enrollment_ca_key` can expose the same narrow one-time token/CSR issuer used by `zaxon serve`, while `enrollment.zig` supplies the join-side primitives. It does not add members, rotate certificates, escrow keys, or replace the host's operator workflow. The node-ID denylist is an operational option rather than an online membership-change API.
4. **No session sugar and no read-level knob.** `exec` is leader-routed and `query` is linearizable. Every other combination goes through `call`.
5. **No multi-database support.** One facade serves one replicated database. There are no cross-database transactions.

Exercise 1. Extend the three-voter example with a fourth member of role `.read_replica`. Predict, before running: which validation error changes if you instead give all four members role `.witness`? Then use `call` with a `{"op":"query", ..., "level":"any"}` request against the replica's endpoint. Explain why that response may be labeled stale when the same query through `query` never is.

Hint: Count campaigners, then reread the `open` validation order in `zaxonlite/src/embedded.zig` and the read-level table in chapter 8.

Teach it back. Explain to a colleague which failures of a three-voter embedded cluster your application code must still handle itself. Use the words ambiguous, quorum, and registry at least once each.

11 The C ABI: libzaxonlite

Learning contract

By the end of this chapter you should be able to link `libzaxonlite` into a C program, open a node and run a replicated write, bind typed parameters, batch statements into one atomic transaction, retry a write exactly once through a session, read results as JSON, and state who owns every buffer that crosses the boundary.

This chapter is for the dqlite-style embedder. Any language with a C FFI can host a replicated SQLite node inside its own process. The whole surface is one header, `zaxonlite/include/zaxonlite.h`, implemented by `zaxonlite/src/capi.zig`. We walk the surface in the order you will use it: link, open, write, batch, retry, read, maintain, cluster. The rules about memory and threads come last, once you have seen every buffer they govern.

11.1 Link and open

Build the library first. `zig build` inside `zaxonlite/` produces the static library `zig-out/lib/libzaxonlite.a` and installs the header as `zig-out/include/zaxonlite.h`. Compile your program against both:

```
$ cc -I zig-out/include embedder.c zig-out/lib/libzaxonlite.a -o embedder
```

The ABI is narrow on purpose, because narrow is what stays stable. `zaxonlite`, `zaxonlite_transaction`, and `zaxonlite_cluster` are `typedef void`. No struct layout leaks into your binary, so the implementation can change without recompiling you. Every scalar that crosses the boundary is a fixed-width type from `<stdint.h>` or `<stdbool.h>`. Protocol quantities are never a bare `int`. Incompatible changes require a new symbol suffix or a major library version. Additive JSON response fields are compatible, and you must ignore fields you do not use. `zaxonlite_version()` returns the library version string. It says `"unreleased"` until the release owner assigns one.

API anchor: `zaxonlite_open` / `zaxonlite_close`

Opens (or creates) one node data directory and releases it. in `capi.zig`

One handle owns one data directory: journal, payload store, snapshots, and the materialized SQLite image. Open performs full journal-authoritative recovery before returning. The handle you receive is therefore already at the decided state. The directory is locked, so a second `zaxonlite_open` on the same path returns 4 and leaves `*out_handle` null. `zaxonlite_close` flushes nothing extra, because every acknowledged write was already durable when its call returned. It frees the handle.

11.2 The return-code contract

Every fallible function returns a `c_int` from one fixed set. The set is mirrored in the header comment, in `capi.zig`, and in the CLI's exit codes. Learn it now, because every later section uses it.

Code	Meaning and what maps to it
0 (ok)	The operation completed.
1 (SQL or session)	SQLite rejected the statement (`SqliteError`, `SqliteBusy`), or a session failed (`UnknownSession`, `SequenceGap`, `ResultExpired`). The failed statement had no durable effect.
2 (misuse)	A caller bug. Examples: a null required argument, a write statement on the read path (`WriteInReadQuery`), a bind-count mismatch, an invalid value

Code	Meaning and what maps to it
	type, or a transaction-builder violation (`TransactionFinished`, `EmptyTransaction`, `TooManyStatements`, `TransactionInputTooLarge`).
3 (integrity)	Only `zaxonlite_integrity_check` returns it. The image, descriptor chain, or payload store failed verification.
4 (unavailable)	Everything else. The directory is locked by another handle, durable state is corrupt, or I/O failed. The node is not safe to use for this operation.

> **API anchor: `zaxonlite_last_error`**
> Returns the most recent error message for a handle as a borrowed, NUL-terminated string. in `capi.zig`

The message lives in a fixed buffer inside the handle. The next failing call overwrites it, and `zaxonlite_close` frees it. Never call `zaxonlite_free` on it. For code 1 from SQLite the message is SQLite's own text, for example `no such table: missing`. Otherwise it is the Zig error name. The cluster facade has its own `zaxonlite_cluster_last_error` with the same lifetime rules.

11.3 Your first replicated write

`zaxonlite_exec` runs one SQL statement batch as one replicated write transaction. On success `*changes_out` receives the affected-row count:

```
int64_t changes = 0;
int rc = zaxonlite_exec(db, "insert into c(b) values ('tea')", &changes);
```

That one call proposed a slot, stored and synced the payload, applied the decided transaction, and only then returned. When `rc` is 0 the row is durable, not merely buffered.

11.4 Prepared values

String-splicing SQL invites injection and quoting bugs. `zaxonlite_exec_prepared` adds typed parameter binding through an array of `zaxonlite_value` structs. Each value names its type and the fields that type reads:

`type`	Fields read	SQLite binding
`ZAXONLITE_NULL`	(none)	NULL
`ZAXONLITE_INTEGER`	`integer`	64-bit integer
`ZAXONLITE_REAL`	`real`	64-bit float
`ZAXONLITE_TEXT`	`bytes`, `length`	Text, bounded by `length`, never read to a NUL.
`ZAXONLITE_BLOB`	`bytes`, `length`	Blob

TEXT and BLOB borrow `bytes` for the duration of the call only. A `length` of zero binds an empty value and ignores `bytes`. Otherwise a non-null `bytes` is required. The value count must equal the statement's parameter count, and an unknown `type` is misuse (code 2). The bound is 65536 values per call.

11.5 Transactions

> **API anchor: `zaxonlite_transaction_begin` / `_exec` / `_commit` / `_close`**
> Collects copied statements across calls, then executes and replicates them atomically at commit. in `capi.zig`

An explicit transaction is a builder, not a live SQLite transaction. This is a safety decision. A live transaction would hold database locks while your application thinks, and it would leave speculative SQLite state behind across a leadership change. The builder holds neither.

Each `zaxonlite_transaction_exec` copies the SQL and all TEXT and BLOB bytes, so you may release your buffers immediately. Nothing touches SQLite until commit. Commit executes every statement under the one-writer gate and proposes exactly one captured WAL transition, so the batch is atomic: all of it is decided, or none of it.
The bounds are 1024 statements and 64 MiB of copied input. Exceeding either is misuse.

> **Predict before reading on**
> You call `zaxonlite_transaction_begin`, then `_commit` with no statements in between. Is that an error or a harmless no-op? Decide before reading on.

Committing an empty transaction is misuse, not a no-op. An empty commit almost always means a logic bug upstream, and the library refuses to replicate a write that does nothing. A transaction is also single-use. After `_commit`, whether it succeeded or failed, only `_close` is valid. `_close` is always required, because it frees the builder.

11.6 Sessions: exactly-once retry

> **API anchor: `zaxonlite_session_open` / `zaxonlite_exec_idempotent`**
> Replicated client sessions. Each (session, sequence) executes at most once. in `capi.zig`

Suppose your process crashes after proposing a write but before seeing the result. A blind retry could apply the write twice. Sessions close that hole. `zaxonlite_session_open` returns a `uint64_t` session id, and creating it is itself a replicated write. That matters: because the session table is part of the replicated state, the exactly-once guarantee survives process crashes and restarts.
`zaxonlite_exec_idempotent(handle, session, sequence, sql, &changes, &replayed)` executes `sequence` exactly once. Retrying the last sequence sets `*replayed_out = true` and returns the recorded change count without executing any SQL. A sequence gap or an expired result fails with code 1 and has no side effects. `zaxonlite_expire_sessions(handle, retain, &expired)` deletes sessions idle for more than `retain` recent session writes and reports how many were removed.

11.7 Queries and JSON results

`zaxonlite_query_json` and `zaxonlite_query_prepared_json` run read-only statements. Each returns one JSON object of the shape `{"columns":[...],"rows":[[...]]}`. Every cell is a string or `null`. The returned buffer is owned by you and released with `zaxonlite_free`.
A write statement on the read path is misuse. The reason is safety, not pedantry. The read path performs no replication, so a write that slipped through it would change this node's image and no one else's. The replicas would fork. Rejecting the statement with code 2 is the only correct answer.

11.8 Maintenance

Three calls keep a node healthy over months, not minutes. `zaxonlite_snapshot` takes an online snapshot and seals the current journal epoch. `zaxonlite_backup` streams a consistent logical backup to `path`. `zaxonlite_integrity_check` verifies the SQLite image, the descriptor chain, and payload availability. It returns 0 only when all three pass, and 3 otherwise.

11.9 The cluster facade

Everything so far ran one local node. The cluster facade turns your process into a full cluster member with its own transport.

> **API anchor: `zaxonlite_cluster_open`**
> Opens a transport-owning cluster member from a runtime role registry. in `capi.zig`

You describe the cluster in a `zaxonlite_cluster_options` struct. It names the data directory, this process's `node_id`, and the member registry: an array of `zaxonlite_member { id, address, role }` with `host:port` addresses. It also takes an optional `cluster_id` mixed into the derived database identity, an optional PSK (`auth_secret` plus `auth_secret_length`) for the authenticated transport, and `startup_timeout_ms`, where 0 selects the 10000 ms default.

Roles are `ZAXONLITE_DATA_VOTER`, `ZAXONLITE_WITNESS`, `ZAXONLITE_STANDBY`, `ZAXONLITE_READ_REPLICA`, and `ZAXONLITE_GATEWAY`. The registry has rules: it holds at most 36 members, at most nine entries may be voting roles, at least one member must be a campaigning data voter, ids must be unique and non-zero, and `node_id` must appear in the registry. Open starts the node, or the stateless router when the local role is gateway, then blocks until the local endpoint answers. If the endpoint never answers, open fails with 4 at the timeout.

The facade routes through the client RPC protocol and follows leader redirects for you. `zaxonlite_cluster_exec` handles writes. `zaxonlite_cluster_query_json` handles linearizable reads. `zaxonlite_cluster_call_json(handle, request_json, require_leader, &json_out)` sends any RPC op from the next chapter. Cluster-surface failures return misuse only for null arguments and invalid registries. Remote errors surface as code 4 with the error name in `zaxonlite_cluster_last_error`, and for `_call_json` you also receive the `{"ok":false,...}` response body itself. `zaxonlite_cluster_close` requests a graceful stop and joins the server thread.

11.10 Memory ownership across the boundary

You have now seen every buffer the ABI moves. Three rules cover all of them.

1. **You lend inputs.** SQL strings, `zaxonlite_value` bytes, paths, and the cluster options struct are borrowed only for the call. Transactions copy at `_exec`, and cluster open copies the registry before returning, so your buffers are free the moment each call returns.
2. **You own returned JSON.** Every `char **json_out` result is a heap buffer. Release it with `zaxonlite_free`, which uses the C allocator the library links. Your own `free` is the wrong allocator.
3. **You borrow error strings.** `zaxonlite_last_error` and `zaxonlite_cluster_last_error` return handle-internal storage. Do not free them. Copy them out if you need them past the next call.

11.11 The boundary contract

The header states the argument rules once, and every exported function follows them. `NULL` is accepted only where a parameter is documented optional; a non-zero length always requires a non-null pointer, and a null pointer always requires a zero length. Declared counts and lengths are validated against product limits before any memory is read, sliced, or allocated from them: the cluster registry holds at most 36 members, a secret is at most 4096 bytes, and one bound TEXT or BLOB value is at most 64 MiB. Every fallible function sets its output handles, pointers, and scalar out-parameters to a safe empty value — `NULL`, 0, or `false` — before doing any other work, on success and on every error path, so a caller that ignores a return code still never reads an uninitialized output. Violating the argument rules is misuse, code 2.

11.12 Threading discipline

Each handle owns its own event-loop instance and node. `capi.zig` guarantees independence between handles and nothing within one. Use a handle, and any transaction created from it, from one thread at a time, or under your own lock. This is SQLite's own connection discipline. The library adds no internal cross-call synchronization on the single-node surface, so two threads in one handle are a data race, not a queued wait.

11.13 A complete embedder

The following condenses the shipped smoke test (`zaxonlite/test/capi_smoke.c`), which exercises every exported function:

```
#include "zaxonlite.h"
#include <stdio.h>
#include <string.h>

int main(void) {
    zaxonlite *db = NULL;
    if (zaxonlite_open("./data", &db) != 0) return 1;

    int64_t changes = 0;
    if (zaxonlite_exec(db, "create table c(a integer primary key, b text)",
                       &changes) != 0) {
        fprintf(stderr, "exec: %s\n", zaxonlite_last_error(db));
        zaxonlite_close(db);
        return 1;
    }

    const zaxonlite_value row[] = {
        {.type = ZAXONLITE_TEXT, .bytes = "tea", .length = 3}};
    zaxonlite_exec_prepared(db, "insert into c(b) values (?1)", row, 1,
                            &changes);

    uint64_t session = 0;
    zaxonlite_session_open(db, &session);
    bool replayed = false;
    zaxonlite_exec_idempotent(db, session, 1,
                              "insert into c(b) values ('z')", &changes,
                              &replayed);           /* retry-safe forever */

    char *json = NULL;
    if (zaxonlite_query_json(db, "select a, b from c order by a",
                             &json) == 0) {
        puts(json);       /* {"columns":["a","b"],"rows":[["1","tea"],...]} */
        zaxonlite_free(json);
    }

    zaxonlite_snapshot(db);
    if (zaxonlite_integrity_check(db) != 0) return 3;
    zaxonlite_backup(db, "./data.backup.db");
    zaxonlite_close(db);
    return 0;
}
```

Exercise 9.1. `zaxonlite_query_json(db, "delete from c", &json)` returns 2, yet `zaxonlite_exec` accepts the same string. Explain, in terms of the replication path each function takes, why the read path must reject writes rather than execute them locally.

Hint: a local write never produces a decided slot.

Teach it back. Without reopening the header, list every buffer your embedder receives from `libzaxonlite`, who frees each one, and what happens if you call `zaxonlite_free` on the string from `zaxonlite_last_error`. Then check yourself against the memory-ownership rules above.

12 Clients and gateways: the wire protocol in practice

Learning contract

By the end of this chapter you should be able to implement a Zaxonlite client in any language from the frame format up, authenticate it, name every RPC op the server accepts and its request and response fields, follow a `not_leader` redirect, and deploy against a gateway knowing exactly what the gateway does and does not protect.

Anything that can frame bytes over TCP can be a client. The reference implementation is `zaxonlite/src/client.zig`, which the `zaxon` CLI uses. The server side is `clientLoop` and `dispatch` in `zaxonlite/src/server.zig`. This chapter is the contract between them. We build it up in the order a connection does: frame, hello, handshake, request, response, redirect.

12.1 Connection bring-up

Every frame on the wire is `u32 total_after_length` (little-endian), then `u8 kind`, then the body. The length counts the kind byte plus the body, so it is never zero. The bound is 64 MiB, and anything larger is a protocol error. The first frame on any connection must be a `hello`:

Offset/size	Field	Meaning
0, 2	`version`	Protocol version. Must equal 4 exactly. Any other value is rejected (`UnsupportedProtocolVersion`) and the connection closes. There is no downgrade negotiation, because a silent fallback would turn a configuration error into a security downgrade.
2, 1	`kind`	`ConnectionKind`: 0 = peer, 1 = client. Clients send 1.
3, 4	`node_id`	The sender's node id. Clients send 0.
7, 16	`database_id`	The cluster's derived database identity. Clients send 0.
23, 8	`configuration_id`	The sender's epoch. Clients send 0.

The encoded hello is 31 bytes. The server answers nothing on success. After a peer or client hello, and after the handshake below when one is configured, it waits for the frames appropriate to that connection. An enrollment hello is the narrow exception described below: it permits exactly one bounded `enrollment_request`, not arbitrary RPC.

12.2 Authentication: the PSK handshake and per-frame protection

When the server is configured with a pre-shared secret, the client's `hello` is immediately followed by a challenge-response handshake. The implementation is `zaxonlite/src/transport_auth.zig`, specified in the format contract under Network protocol v6. From the client's perspective:

1. Read one `auth_challenge` frame. It carries a fresh 32-byte nonce and the server's proof `HMAC-SHA256(secret, "zaxon.auth.server.v1" || hello || nonce)`, where `hello` is exactly the 31 bytes the client just sent. Verify the proof. A mismatch means the server does not hold the secret.
2. Send one `auth_response` frame carrying the 32-byte client proof over the domain `"zaxon.auth.client.v1"`. Both sides then derive the connection key with a third domain, `"zaxon.auth.session.v1"`.

Every later frame body in each direction is then
`u64 sequence_le || application_body || hmac_sha256`
where the MAC covers the frame domain `"zaxon.auth.frame.v1"`, the frame kind, the sequence, and the body. Each direction starts at sequence zero, and the receiver accepts exactly the next sequence. An invalid tag closes the connection (`AuthenticationFailed`). A wrong sequence closes it too (`ReplayDetected`). The fresh responder nonce prevents replay of an earlier handshake. The sequence prevents replay, reorder, and truncation within a connection.

> **Shared-secret integrity, not per-node identity**
> The PSK handshake proves that both endpoints possess the same PSK and provides frame integrity and replay rejection. It does not bind the hello's `node_id` to a distinct credential. It also does not encrypt: every body crosses the network in the clear. Zaxonlite intentionally treats an admitted application caller as having full database authority; end-user permissions belong in that application, not in this RPC protocol. Production TCP never relies on this PSK alone. The CLI permits it only with explicit `--dev-psk`, and then only when the listener and all peers are numeric loopback. A single local node can serve over an owner-only Unix-domain socket instead of TCP (chapter 2); for production TCP, the mutual TLS transport below adds per-node identity and confidentiality.

12.3 Mutual TLS

When the server carries a TLS identity, the TCP connection completes a mutual TLS 1.3 handshake before the first frame. The client presents a certificate and key (`--tls-cert`, `--tls-key`) and verifies the server against the cluster CA (`--tls-ca`); the server verifies the client certificate against the same CA and refuses a connection without one. A client certificate needs only to chain to the CA — its common name is not interpreted. Peer connections are stricter: a peer's certificate common name must be exactly `zaxon-node-<id>` for the node id its hello claims (chapter 7). Everything in this chapter is unchanged inside the channel: the same hello, the same optional PSK handshake, the same frames. A plaintext client dialing a TLS listener fails the handshake and never reaches the frame protocol.
The sole exception bootstraps that missing node certificate. A server explicitly configured with `--enrollment-ca-key` may complete TLS without a client certificate, but the first frame must be an enrollment hello bound to a configured target and the next and only frame must be an `enrollment_request`. The joiner pins the CA carried in its owner-only bundle and requires the exact issuer common name, so this exception does not weaken ordinary peer or client connections. The request contains a one-time secret, the database and target bindings, and a signed CSR; it cannot execute SQL or any RPC. Chapter 13 describes token issuance and failure recovery.

12.4 The RPC contract

One `rpc_request` frame carries one JSON object. The server replies with one `rpc_response`. Success responses carry `"ok":true`. Failures are `{"ok":false,"error":"<code>","message":"<text>"}`. Additive response fields are compatible, so ignore fields you do not use. The dispatch in `server.zig` accepts exactly these ops:

op	Purpose
`status`	Reports one node's identity, role, ballot, and progress.
`members`	Lists the member registry with per-node capabilities.
`leader`	Names the current leader, if one is known.
`exec`	Executes a replicated write, optionally under a session.
`query`	Runs a read-only query at a chosen consistency level.

op	Purpose
session	Opens a replicated client session.
wait	Blocks until the node reaches a condition you name.
snapshot	Takes an online snapshot and seals the journal epoch.
integrity	Verifies the image, chain, and payload store.
hash	Reports the state hashes for cross-node comparison.
expire-sessions	Deletes idle sessions.
issue-enrollment-token	Issuer-only: creates a short-lived bundle secret for one non-revoked configured peer. The caller is already authenticated by normal mTLS.
failpoint	Arms a named fault. Test builds only.
stop	Requests a graceful shutdown.
backup	Streams a verified copy of the database. Not an rpc_response. See the backup section below.

The subsections that follow give each op's request fields and success response. A field not marked optional is required.

Protocol v6 applies no permission matrix to this list. That matches the single-application design: possession of the embedded handle or access to the service means full database authority. An application that serves unrelated users must authenticate them and expose only its own permitted operations. Failpoints remain a test-only capability and must be disabled in real data directories.

12.4.1 Observing the cluster: status, members, leader, hash

These four ops take no request fields beyond op itself.

status answers with node_id, database_id, configuration_id, role, node_type, leader, ballot (an object with round, priority, and node), decided_slot, applied_slot, journal_records, epoch_capacity, chain, page_size, and snapshot. It describes the one node you asked, so it is the op you send with require_leader = false when you want a follower's view.

members answers with voter_membership:"static" and a nodes array. Each entry carries id, host, port, role, votes, campaigns, stores_log, serves_reads, serves_writes, promotion_eligible, self, and leader.

leader answers with leader:{id,host,port}, or leader:null when no leader is known.

hash answers with chain, content, and applied_slot. Two nodes that report the same applied slot must report the same hashes. That is how the test suites catch divergence.

12.4.2 Writing: exec

The request carries sql, and optionally session and sequence. The session fields come as a pair: both or neither. The success response carries changes, slot, and replayed. Chapter 8's exactly-once contract rides on those two request fields, and we return to them below.

12.4.3 Reading: query

The request carries sql, an optional level, and an optional freshness_ms that is valid only with level any. The success response carries columns (an array of names), rows (an array of rows, each cell a string or null), and the level that was actually served.

12.4.4 Sessions and waiting: session, wait

session takes no request fields and answers with session_id. Opening a session is a replicated write, which is why the guarantee it backs survives leader changes.

`wait` takes three optional fields: `applied` (a minimum applied slot), `leader` (a boolean requiring this node to hold leadership), and `timeout_ms`. It answers with `applied_slot`, `decided_slot`, `leader`, and `configuration_id`. This is the op behind `zaxon wait` in the quickstart.

12.4.5 Maintenance: snapshot, integrity, expire-sessions

`snapshot` takes no fields and answers with `configuration_id`, the new epoch. `integrity` takes no fields and answers with `ok`, `sqlite_ok`, `chain_ok`, and `payloads_ok`. `expire-sessions` takes `retain` and answers with `expired`, the count removed.

12.4.6 Operating: failpoint, stop

`failpoint` takes `name` and answers `{"ok":true}`. The server rejects it unless it was started with `--enable-failpoints`; when enabled, every admitted application caller can fault the node over the wire. `stop` takes no fields, answers `{"ok":true}`, and then shuts down gracefully.

12.4.7 The error codes you must handle

Error	Meaning and what to do
`bad_request`	Malformed JSON or fields. Fix the client.
`not_leader`	This node cannot serve the request. Follow the embedded leader hint.
`sql`	SQLite rejected the statement. The statement had no durable effect.
`session`	Unknown session, sequence gap, or expired result.
`ambiguous`	The write's fate is unknown. Retry idempotently with the same session and sequence.
`timeout`	The operation ran out of time. Retry.
`retry`	The epoch is rolling over, or leadership changed during a read fence. Retry.
`too_large`	The payload exceeds the 64 MiB wire limit.
`stale`	A freshness-bounded `any` read could not be served fresh enough.
`forbidden`	This node type serves no SQLite reads.
`unavailable`	The node is not in a state to serve this.
`internal`	An unexpected server error. Inspect the message.

12.5 Redirects, read levels, and freshness

Writes, sessions, and non-`any` reads are answered only by the leader. Any other node replies
`{"ok":false,"error":"not_leader","leader":{"id":1,"host":"10.0.0.5","port":9901}}`
with `leader:null` when no leader is known. The reference client (`callCluster`) tries endpoints round-robin, follows the embedded leader hint, sleeps 150 ms between attempts, and gives up after 12 attempts with `NoLeaderReachable`. A hint is followed only when it names one of the caller's configured endpoints under PSK-only transport. With mTLS, an unmatched numeric address can become a target, but only after its certificate common name proves the advertised node ID under the trusted cluster CA. A wrong name fails before the request is replayed. With `require_leader = false` the first reachable node's answer is returned as-is. That is how you `status` a follower.
`query` takes three levels. `any` is a local read with no leadership required. `leader` reads the leader's applied state. `linearizable`, the default, first completes an exact-ballot quorum read fence, so the leader proves it is still the leader before answering.

> **Predict before reading on**

> You send a `level:"any"` read with no `freshness_ms` to a replica that has been partitioned from the cluster for an hour. Does the read fail? Decide before reading on.

It succeeds. Without a freshness bound, `any` explicitly permits an arbitrarily stale local snapshot. If you need a bound, send `freshness_ms`. A learner then rejects the read with `stale` in three cases: it has never heard from a leader, its last leader contact is older than the bound, or the leader's last reported decided slot is ahead of the learner's applied slot.

12.6 Sessions over the wire

`{"op":"session"}` performs a replicated write and returns `session_id`. Subsequent writes carry `session` and `sequence` inside `exec`. The response's `replayed` field reports whether the recorded result was returned instead of executing SQL. After `ambiguous` or a connection loss, the recovery is always the same: reconnect anywhere, follow redirects, and resend the **same** session and sequence. Exactly-once holds across leader changes because the session table is replicated state.

12.7 A worked exchange

Frame kinds elided. Each line is one JSON body as produced by `server.zig` and consumed by `client.zig`:

```
-> {"op":"exec","sql":"insert into c(b) values ('tea')"}
<- {"ok":false,"error":"not_leader","leader":{"id":1,"host":"127.0.0.1","port":9901}}
   (client reconnects to 127.0.0.1:9901, repeats the hello + handshake)
-> {"op":"session"}
<- {"ok":true,"session_id":7}
-> {"op":"exec","sql":"insert into c(b) values ('tea')","session":7,"sequence":1}
<- {"ok":true,"changes":1,"slot":42,"replayed":false}
-> {"op":"exec","sql":"insert into c(b) values ('tea')","session":7,"sequence":1}
<- {"ok":true,"changes":1,"slot":42,"replayed":true}
-> {"op":"query","sql":"select count(*) from c","level":"linearizable"}
<- {"ok":true,"columns":["count(*)"],"rows":[["1"]],"level":"linearizable"}
```

Read the two `exec` lines closely. The retry carries the same session and the same sequence, and the server answers with the same slot and `replayed:true`. No SQL ran the second time.

12.8 Backup streaming

`{"op":"backup"}` is the one request not answered by an `rpc_response` on success. The leader replies `backup_begin`, carrying a `u64 size` and a 32-byte SHA-256 of the whole image. Then come ordered `backup_chunk` frames, each a `u64 offset` plus bytes, contiguous from zero. Then `backup_end`. The reference client writes to an exclusive temporary file, verifies the declared digest over every received byte, fsyncs, and only then renames into place and syncs the directory. Receiving an `rpc_response` instead of `backup_begin` means the node declined, with `not_leader` or `retry`. Under an authenticated session every backup frame is sequence-protected like any other.

12.9 Gateways

A gateway (`zaxonlite/src/gateway.zig`) is a stateless TCP router that deliberately operates **below** the wire protocol. It never parses frames and never terminates authentication. It holds no secret, no Paxos state, and no SQLite state. Each inbound connection is proxied byte-for-byte to one backend, chosen round-robin with failover to the next backend when a dial fails, and copied in both directions until either side closes. Five consequences are worth stating directly.

- **Authentication and encryption are end-to-end.** The TLS handshake, hello, optional PSK exchange, and protected frames pass through unmodified. The client authenticates the storage node itself. A compromised gateway can drop or delay traffic, but cannot read, forge, replay, or alter it undetected.

- **Redirects still happen.** A gateway does not track the leader. If it routes your write to a follower you receive `not_leader` with the leader's real address, and the reference client then dials that address directly.
- **Backup streams need no special path.** They are bytes like everything else.
- **TLS is passthrough, not terminated.** The gateway does not take its own TLS identity. A client starts TLS with the selected storage node through the raw proxy; that backend still requires a CA-valid client certificate, and the client still requires a canonical `zaxon-node-<id>` server certificate. Passing `--tls-*` to the gateway itself is therefore a usage error. A PSK, when configured, also remains inside the end-to-end TLS stream.
- **Admission is bounded.** At most 128 raw connections are proxied at once. Storage backends independently enforce their global and per-peer limits and handshake/idle deadlines. The gateway never turns into an unbounded list of sockets while a client stalls its TLS handshake.

To deploy against one, list the gateway's `host:port` as an endpoint and give the client the same TLS CA and client identity it uses when dialing a storage node directly. Restarting a gateway costs only open connections. In the role registry a gateway is `ZAXONLITE_GATEWAY` (`gateway` on the wire). The embedded facade starts one automatically when the local member has that role, backending onto every registry member whose role serves reads or writes.

Exercise 10.1. Your client sends `{"op":"query","sql":"select 1","level":"any", "freshness_ms":200}` to a read replica and receives `{"ok":false,"error":"stale",...}`. List the three distinct conditions in the server that produce this, and state which of them can clear without any new client action.

Hint: two involve the learner heartbeat, and one involves never having heard from a leader at all.

Teach it back. From memory, write down the byte layout of an authenticated `exec` request frame, from the `u32` length to the final HMAC byte, and state what the receiver checks and in which order before the JSON body reaches `dispatch`. Then check yourself against `transport_auth.zig`'s `Session.readFrame`.

PART IV

Operating

We turn the laptop cluster from chapter 1 into a deployment someone can be paged for. Configuration, security, monitoring, and rehearsed runbooks come first; the worked examples then turn those habits into reflexes.

13 Operations

Learning contract

By the end of this chapter you should be able to configure a member from a file instead of a pile of flags, state the exact limitations of the legacy pre-shared key, execute one-time certificate enrollment, read `status` and `members` as monitoring signals, act on the failure playbook without improvising, run the backup and disaster recovery runbooks, and roll a new binary through a live cluster.

Checkpoint: bring-up

Chapter 1 built the binary, ran one durable node, and drove a three-voter cluster through a killed leader. We assume you can do all of that without looking. Chapter 2 holds the full command reference. This chapter teaches only what an operator needs beyond both.

13.1 What changes off the laptop

The quickstart ran everything on `127.0.0.1`. Production TCP now fails closed unless mutual TLS is configured. Automated token enrollment is available for members already in the static registry; it is bootstrap, not a hidden shared secret or a membership protocol. A realistic deployment changes at least these four things.

1. Listen and peer addresses become real network addresses, and `zaxon` then refuses to start until you configure the mutual-TLS identity. A PSK may be layered inside TLS but cannot replace it.
2. Each `--peer` may carry a role suffix, as in `2@10.0.0.2:9901/data-voter`, and every storage node must receive the same registry.
3. Two clusters whose members use the same ids would derive the same database identity, so each deployment gets its own `--cluster-id`.
4. Flag piles give way to one configuration file per host.

The rest of this chapter walks through each change, then through the runbooks you should rehearse before you need them.

13.2 Roles, from the operator's chair

Earlier chapters explained why the role system exists. The operator's view fits in five sentences. A `data-voter` votes, campaigns, stores SQL, and serves clients. A `witness` votes but refuses every SQL request, so a small third failure domain can break ties without holding data. A `standby` and a `read-replica` follow the log without voting; the standby is promotion material, the replica serves reads. A `gateway` stores nothing and forwards clients to the nodes that do. Every node is told its own role with `--role` and learns everyone else's from the peer suffixes.

One rule matters operationally: an existing data directory pins its role. Restarting a node with a different `--role` is rejected. Changing a member's role is a controlled migration, not a flag edit.

13.3 The command surface

Chapter 2 is the complete command and flag reference. Operators need three facts on top of it.

1. Client mode accepts a comma-separated endpoint list and follows `not_leader` redirects on its own, so scripts should name every member and ignore leadership entirely.
2. Commands that take `--data` open the node in-process and hold the directory lock, so run them on a stopped node; a directory locked by a running server exits with code 4 and never risks corruption.
3. Runbooks branch on exit codes, so learn them once.

Exit code	Meaning
0	The command succeeded.
1	The SQL statement or the session request failed.
2	The invocation or the configuration was malformed.
3	An integrity check found a mismatch.
4	The node was unavailable: locked, corrupt, or without a reachable leader.

13.4 The shell history file

The interactive `zaxon sql` shell records executed statements so `ctrl+r` search survives restarts, and statements routinely contain material an operator would not commit to disk knowingly — tokens pasted into inserts, credentials in `pragma` or setup statements. Four controls bound the exposure; runbooks that handle secrets should name which one they rely on.

- The file is written with owner-only permissions (`0600`). In embedded mode it lives inside the data directory as `.zaxon_history`, so the protections already required for the data directory cover it. In client mode nothing is persisted unless `ZAXON_HISTORY` names a path.
- `--no-history` disables persistence for one session; `.history off` disables it from inside the shell; `.history clear` empties it.
- A statement typed with a leading space is never recorded — the standard escape hatch for one sensitive statement.
- Scripted (non-terminal) invocations never read or write history at all.

13.5 Configuration: flags, environment, file

Every option can come from three places. The precedence is strict: command-line flags override `ZAXON_*` environment variables, and those override the JSON configuration file. The file is named by `--config <path>` or by the `ZAXON_CONFIG` environment variable. Without either, no file is read. `version` and `help` never load configuration.

The file is one strict JSON object of at most 1 MiB. Unknown fields are rejected. This is a deliberate safety choice: a typo fails loudly with exit code 2 instead of being silently ignored. Every field is optional.

Field	Flag and environment	Meaning
`data`	`--data`, `ZAXON_DATA`	The node data directory, created when missing; selects embedded mode.
`connect`	`--connect`, `ZAXON_CONNECT`	Comma-separated server endpoints, each `host:port` or `unix:<path>`; selects client mode.
`node`	`--node`, `ZAXON_NODE`	This node's integer id, for `serve`.
`role`	`--role`, `ZAXON_ROLE`	`data-voter`, `witness`, `standby`, `read-replica`, or `gateway`.
`listen`	`--listen`, `ZAXON_LISTEN`	The listen endpoint for `serve`: `host:port`, or

Field	Flag and environment	Meaning
		`unix:<path>` for single-node local service.
`peers`	`--peer` (repeat), `ZAXON_PEERS`	An array of `id@host:port[/role]` peer specifications.
`cluster_id`	`--cluster-id`, `ZAXON_CLUSTER_ID`	Extra entropy for the derived database identity.
`auth_file`	`--auth-file`, `ZAXON_AUTH_FILE`	The path to the pre-shared-key provider file. Always a path, never the key.
`tls_cert`	`--tls-cert`, `ZAXON_TLS_CERT`	The node certificate PEM for mutual TLS. All three TLS fields go together.
`tls_key`	`--tls-key`, `ZAXON_TLS_KEY`	The node private key PEM.
`tls_ca`	`--tls-ca`, `ZAXON_TLS_CA`	The cluster CA PEM that every peer certificate must chain to.
`enrollment_ca_key`	`--enrollment-ca-key`, `ZAXON_ENROLLMENT_CA_KEY`	Owner-only CA key enabling the deliberately configured token/CSR issuer. Omit on ordinary nodes.
`revocation_file`	`--revocation-file`, `ZAXON_REVOCATION_FILE`	Reloaded configured-node-ID denylist.
`sync`	`--sync`, `ZAXON_SYNC`	Durability sync mode: `full` (the default) or `os` (development only on macOS).

A complete member configuration looks like this:

/etc/zaxon/n1.json

```
{
  "data": "/var/lib/app/n1",
  "node": 1,
  "role": "data-voter",
  "listen": "10.0.0.1:9901",
  "peers": ["2@10.0.0.2:9901/data-voter", "3@10.0.0.3:9901/data-voter"],
  "cluster_id": "prod-eu-1",
  "auth_file": "/etc/zaxon/psk",
  "tls_cert": "/etc/zaxon/node.crt",
  "tls_key": "/etc/zaxon/node.key",
  "tls_ca": "/etc/zaxon/ca.crt"
}
```

With `ZAXON_CONFIG=/etc/zaxon/n1.json` exported, starting the member is just `zaxon serve`. Operator commands on the same host, such as `zaxon status` or `zaxon integrity-check`, find the directory the same way. A malformed value from any source is a usage error with exit code 2, never a silent default. That covers an unreadable file, a non-integer `ZAXON_NODE`, and an unknown role alike.

13.6 Durability against power loss: the sync mode

The default sync mode, `full`, is the safe one, and production needs no flag at all. On macOS it makes every authoritative sync flush the drive's volatile cache with `F_FULLFSYNC`, so an acknowledged write survives a power cut and a voter can never forget a promise it made. Chapter 6 explains why that is a safety property of the consensus, not a data-loss preference. On Linux plain `fsync` already reaches stable media, the two modes are equivalent, and this section changes nothing.

The safety has a measured price on macOS. Each replicated write pays one full flush per voting node at its commit point — the journal barrier, which the payload install rides (chapter 6). Protocol v6 queues the payload and phase-two accept before the leader barrier, so follower storage and flush overlap it without allowing an accepted reply into the core early. On this Apple-silicon host the current 256-byte, three-voter mTLS run records roughly 15.94 ms p50 and 63 writes per second under `full`; chapter 17 reads the exact row from JSON. Reads are untouched. Sustained full-mode write load can also stall the leader's tick loop long enough for leadership to move, visible as occasional large latency maxima. Use `--sync os` only for development loops and benchmarks on macOS, never for a directory whose votes you intend to keep.

13.7 The PSK layer

The optional inner layer authenticates with a pre-shared key. Both sides prove possession of the secret in a challenge-response handshake. The responder contributes a fresh 32-byte random nonce, so an earlier handshake cannot be replayed. After the handshake, every frame carries an HMAC-SHA256 tag over the frame kind, a strictly increasing sequence number, and the body. A bad tag, a skipped sequence, or a peer that never authenticates closes the connection.

Know exactly what that buys—and what it does not.

- Protected: proof that the remote endpoint holds the cluster-wide PSK, frame integrity, and replay rejection on one connection.
- Not protected: node identity. The unauthenticated hello supplies the claimed node ID and peer/client kind. Any PSK holder can impersonate a configured voter.
- Not provided: database-user authorization. This is intentional: the embedding or socket-owning application is the single database principal and applies end-user policy before calling Zaxonlite.
- Not protected: confidentiality. SQL text, results, pages, snapshots and backups travel in cleartext.

An encrypted tunnel can hide bytes from the network, but it does not bind the PSK holder to a configured node identity. Production startup therefore refuses PSK-only TCP. The explicit `--dev-psk` exception is restricted to numeric loopback for the one-machine quickstart and local development; it requires an owner-only provider and cannot be combined with mTLS or gateway mode. Because leader advertisements cannot be identity-bound in this mode, clients supply every member in `--connect` and rotate only through that seed list. Local single-node service already has its production form: a Unix-domain socket protected by filesystem permissions, described below. For production TCP, the mutual TLS transport in the next section supplies both per-node identity and confidentiality; the one-time enrollment flow below automates certificate issuance after initial CA and issuer bootstrap. Application authorization stays outside Zaxonlite; see the security remediation plan.

The secret comes only from a provider file, named by `--auth-file <path>`, by `ZAXON_AUTH_FILE`, or by the `auth_file` configuration field. The file may be at most 4096 bytes. One conventional trailing line ending is not part of the key; every other byte is, including spaces. The key must be at least 32 bytes, and shorter providers are rejected before any socket opens. Every node and every client of one cluster must present the same secret.

> **Secrets are paths, never values**

> There is no flag that accepts a literal key, and there never will be: command lines leak through process listings and shell history. Give the provider file tight permissions, readable by the service user only. The loader requires a regular non-symlink file with no group/world permission bits (mode 0600). The CLI loader zeroes its original allocation, but embedded mode keeps an arena copy that is not explicitly wiped on close.

Every production TCP listener requires the complete TLS identity and exits 4 otherwise. There is no silent fallback to PSK or plaintext. `--dev-psk` is an explicit opt-in and accepts only the exact numeric loopback hosts `127.0.0.1` and `::1` for the listener and every peer. The internal `--insecure-test-tcp` escape hatch is rejected unless failpoints are enabled; it exists only for deterministic fault harnesses. The refusal diagnostic names the Unix-socket and local-development alternatives.

13.8 The mutual TLS transport

The second transport mode gives every node its own certificate. When `serve` is started with `--tls-cert <pem>`, `--tls-key <pem>`, and `--tls-ca <pem>` — always all three together — every TCP connection the process accepts or dials runs TLS 1.3 at minimum with mutual verification: both sides must present a certificate that chains to the cluster CA, and a connection without one is refused. Client commands take the same three flags, the same `tls_cert`/`tls_key`/`tls_ca` configuration fields, and the same `ZAXON_TLS_CERT`, `ZAXON_TLS_KEY`, and `ZAXON_TLS_CA` environment variables.

Identity is bound by certificate common name. A peer connection must present the certificate issued for the node id it claims in its hello, with common name exactly `zaxon-node-<id>`; the check runs on both the accepting and the dialing side, so neither direction trusts a claimed id on its own. A client certificate needs only to chain to the cluster CA; its name is not interpreted. TLS also encrypts the wire, which the PSK mode never did, and the two modes compose: when both are configured, the PSK challenge-response runs inside the TLS channel. Gateway mode does not support `--tls` yet and rejects the flags with a usage error.

Bootstrap is deliberately small. Create the cluster CA once, then use it out of band to issue one initial `zaxon-node-<id>` identity for the issuer and one operator client identity. The issuer automates all later configured-node certificates through short-lived, one-time token/CSR enrollment. The `openssl` CLI is sufficient for that initial step:

```
$ openssl ecparam -name prime256v1 -genkey -noout -out ca.key
$ chmod 600 ca.key
$ openssl req -new -x509 -key ca.key -sha256 -days 365 \
    -subj '/CN=zaxon-cluster-ca' \
    -addext basicConstraints=critical,CA:TRUE \
    -addext keyUsage=critical,keyCertSign,cRLSign -out ca.crt
$ openssl ecparam -name prime256v1 -genkey -noout -out n1.key
$ chmod 600 n1.key
$ openssl req -new -key n1.key -out n1.csr -subj '/CN=zaxon-node-1'
$ printf '%s\n' \
    'basicConstraints=critical,CA:FALSE' \
    'keyUsage=critical,digitalSignature' \
    'subjectAltName=DNS:zaxon-node-1' \
    'extendedKeyUsage=serverAuth,clientAuth' > n1.ext
$ openssl x509 -req -in n1.csr -CA ca.crt -CAkey ca.key \
    -CAcreateserial -sha256 -extfile n1.ext -out n1.crt -days 365
```

Issue the operator client certificate the same way with a distinct common name such as `zaxon-client`; it needs `clientAuth` and, because the same reference identity may serve embedded members, may also carry `serverAuth`. A Zaxon client refuses a server certificate whose common name is not the canonical `zaxon-node-<positive-id>` shape, so a CA-issued client principal cannot be reused as a storage endpoint identity.

With one such identity per node, the member from the configuration file above starts as:

```
$ zaxon serve --data /var/lib/app/n1 --node 1 --listen 10.0.0.1:9901 \
    --peer 2@10.0.0.2:9901 --peer 3@10.0.0.3:9901 \
    --tls-cert /etc/zaxon/n1.crt --tls-key /etc/zaxon/n1.key \
    --tls-ca /etc/zaxon/ca.crt \
    --enrollment-ca-key /etc/zaxon/ca.key
```

The enrollment CA key must be a regular, non-symlink file with no group or world permissions and must match `--tls-ca`. It is not needed for consensus or normal mTLS, so keep it on as few designated issuer nodes as your availability needs require. Issuance is not a membership operation: the target must already appear in the issuer's static peer registry and must not be denied by the node-ID revocation file.

An operator who already has a valid client identity creates an owner-only bearer bundle for node 2:

```
$ zaxon enroll-token --connect 10.0.0.1:9901 --node 2 \
    --to node2.token --ttl-seconds 600 \
    --tls-cert operator.crt --tls-key operator.key --tls-ca ca.crt
```

The bundle contains the issuer endpoint, full CA certificate, database and node bindings, expiry, and a random 256-bit secret. The issuer stores only a domain-separated hash of that secret. Transfer the file through an authenticated operational channel and keep mode 0600; possession is sufficient to redeem it until expiry. On node 2:

```
$ zaxon enroll --token-file node2.token --identity-dir /etc/zaxon/node2
$ zaxon serve --data /var/lib/app/n2 --node 2 --listen 10.0.0.2:9901 \
    --peer 1@10.0.0.1:9901 --peer 3@10.0.0.3:9901 \
    --tls-cert /etc/zaxon/node2/node.crt \
    --tls-key /etc/zaxon/node2/node.key \
    --tls-ca /etc/zaxon/node2/ca.crt
```

`enroll` pins the bundled CA and exact issuer common name, generates the P-256 private key and signed CSR locally, and accepts only a pinned-CA-signed certificate matching that key and `zaxon-node-2`. It atomically installs `node.key`, `node.crt`, and `ca.crt` into a new owner-only directory and removes the token bundle after success. The private key never crosses the network.

The issuer verifies the CSR and every binding, atomically moves its token record from pending to used, syncs that directory, and only then signs. Thus a crash after consumption can lose the response but cannot issue twice. Treat any ambiguous enrollment failure as consumed and request a new token; the issuer intentionally keeps no result cache or retry protocol.

and a client reaches it with the client certificate:

```
$ zaxon status --connect 10.0.0.1:9901 --tls-cert client.crt \
    --tls-key client.key --tls-ca ca.crt
```

An unreadable certificate, a key that does not match, a private-key symlink or broad private-key mode, or a missing CA file fails startup with `-- TLS IDENTITY FAILED --`. Keep the key files readable by the service user only; like the PSK, they are named by path and never by value. The implementation links the system OpenSSL 3 libraries rather than bundling a TLS stack; macOS builds default to the Homebrew `openssl@3` prefix. Other targets take `-Dopenssl-prefix` pointing at an OpenSSL 3 SDK built for that target. Windows SDKs are linked by their posix names, `ssl` and `crypto`, except under the MSVC ABI, which expects `libssl` and `libcrypto`.

`--revocation-file <path>` (or `ZAXON_REVOCATION_FILE`) supplies the simple static-membership revocation override: one configured node ID per line, with `#` comments. The server reloads it once per second, rejects new handshakes, and closes matching live inbound and outbound sockets immediately. A malformed reload retains the last valid set. Revocation is by node ID, not certificate serial. A canonical `zaxon-node-<id>` certificate is checked even when used on a client connection; unrelated operator-client common names remain the single application authority described above. Admitting a replacement under a denied node ID requires the operator to remove it from the denylist after replacing credentials. Removing a voter remains an operator-controlled offline replacement procedure; the book does not claim online consensus membership management.

Current release restriction

Do not treat PSK-only TCP as the production trust boundary: it has no per-node identity and no confidentiality. `--dev-psk` merely confines that tradeoff to one host. The mutual TLS transport supplies both properties. One-time issuance is automated only for a configured target; initial CA/issuer provisioning and later rotation remain operator work. The node-ID denylist provides immediate revocation. Admission is globally and per-peer capped, handshakes and established idle sockets have deadlines, transfers are bounded, and remote queries default to 10,000 rows, 16 MiB, 1 MiB of SQL text, and a 10-million SQLite VM-instruction budget. Zaxonlite does not require separate operator credentials: the application is the one database authority. Follow the release gates in `docs/zds/records/0003-zaxonlite-security-remediation-plan.typ`.

Plaintext at rest

The current database, captured payloads, journals, snapshots, and backups are plaintext. Zaxonlite reads the WAL and applies page images through direct I/O outside SQLite's VFS, so SQLCipher or an encrypted VFS is not presently a drop-in option. Use platform full-disk or filesystem encryption when powered- off media theft is in scope. An encrypted edge profile is future work and needs its own direct-I/O, recovery, snapshot, backup, key, and rekey tests.

13.9 Local service over a Unix-domain socket

`zaxon serve --listen unix:<path>` serves a single local node over a Unix-domain socket instead of TCP, and filesystem permissions become the local authorization boundary. The server restricts the socket to owner-only permissions (mode 0600) immediately after binding, so only the owning user can connect; widening access is a deliberate host configuration, not a default. A pre-existing file at the socket path is refused — never silently unlinked or taken over — so a stale socket left by a crash requires explicit operator removal after confirming no server owns it. An orderly shutdown removes the socket path. The mode serves exactly one node: configured peers are rejected, because cluster links require TCP, and gateway mode is TCP-only. Clients name the socket the same way everywhere an endpoint is accepted: `--connect unix:<path>`, the `connect` configuration field, or `ZAXON_CONNECT`.

13.10 Monitoring

`status --json` is the machine surface. It reports `node_type`, the Paxos `role`, the current leader, `decided_slot` and `applied_slot`, the journal record count, the epoch capacity, the chain hash, and the installed snapshot generation. A served node also reports its ballot. In client mode, `members --json` returns the runtime registry with one entry per node: id, address, role, the capability flags (`votes`, `campaigns`, `stores_log`, `serves_reads`, `serves_writes`, `promotion_eligible`), plus `self` and `leader` markers. Embedded `members` lists only the static member ids.

Watch four signals.

1. No leader means no writes, so alert on a failing `zaxon wait --leader` quickly.
2. A growing gap between `decided_slot` and `applied_slot` means the node decides faster than it applies, and reads on that node fall behind.
3. The `chain` value must be equal across members at the same applied slot, because equality means identical applied history; a mismatch is an emergency, not a curiosity.
4. The journal record count against the epoch capacity shows how full the current epoch is, which tells you when a `snapshot` is due.

13.11 Failure playbook

Predict before reading on

> A power cut tears the last journal record in half on one member. On restart, should that node refuse to start, repair the record, or drop it? Decide before you read the table.

Symptom	What happens, and what to do
One member down	Writes and linearizable reads continue on the remaining two. Restart the member; it catches up on its own, from the journal suffix or through a snapshot transfer across a sealed epoch.
Two members down	No quorum. Writes and fenced reads refuse, while `--level any` reads still answer locally. Add `--freshness-ms` when a disconnected or lagging node should refuse instead. Restore a member.
`current.db` lost or replaced with an old copy	Nothing is lost. On the next start the node discards the image and rebuilds it from the snapshot plus the journal, validating the batch marker. Just restart it.
Journal tail torn by power loss	The tail is truncated automatically on open. This is safe because the lost suffix was never acknowledged to any client.
Journal corrupt in the middle	The node refuses to open, because an interior gap would mean serving history it cannot prove. Reimage from the healthy quorum: empty the directory and restart, and snapshot transfer plus catch-up rebuild everything.
Disaster of last resort	`zaxon backup --to app.db` on any surviving member yields a plain SQLite file that opens anywhere.

The torn-tail row answers the prediction: the node drops the half-written record. Refusing would sacrifice liveness for data no client was ever promised, and repairing would mean inventing bytes.

13.12 Backup, restore, and disaster recovery

Take backups from the live cluster with `zaxon backup --to <path>`. In embedded mode this is a local `VACUUM INTO`. In client mode the command locates the leader and streams the copy over the authenticated transport, verifying an end-to-end SHA-256 before installing the destination file. An interrupted stream installs nothing, so a retry against the endpoint list is always safe. Either way the result is a plain, consistent SQLite database that any SQLite tool can open.

> **Predict before reading on**
>
> You have last night's `backup.db`. You copy it into a member's data directory as `current.db` and start the node. Which rows does the node serve? Decide, then read on.

The node serves the cluster's rows, not the backup's. Every open discards the materialized image and rebuilds it from the snapshot plus the journal. The journal is authoritative; the `.db` file is a cache. There is no import command, so restoring a logical backup means building a new cluster.

1. Provision fresh data directories on every member, which creates a new database identity.
2. Start the new cluster and wait for a leader.
3. Replay the backup's schema and rows through `zaxon sql` or through your application's own loader.
4. Verify row counts, then run `zaxon integrity-check` before pointing clients at it.

For a node whose directory survives but whose health is in doubt:

1. Copy the directory and work only on the copy, never on the sole original.
2. Run `zaxon recover --data <dir>`, which rebuilds the image from the verified snapshot plus the committed journal suffix and then runs the full integrity check, reporting pass or fail per section.

3. Exit code 0 means the node is sound, so restart it.
4. Exit code 3, or a node that refuses to open because its durable journal prefix is damaged, means the local history is gone; restore a verified logical backup or reimage the member from the healthy quorum.

Journal editing is unsupported

Never fill a gap, patch a checksum, or truncate an interior record by hand to make a node open. The journal is the authoritative history. A hand-edited journal can vote, and a voting node with invented history can silently diverge the cluster. That is a safety violation, not an inconvenience. A node that cannot replay its own prefix must be restored from backup or reimaged. Those are the only supported paths.

13.13 Rolling upgrade

The upgrade contract is frozen in the format chapter. The runbook form:

1. Verify first: run `zaxon integrity-check` on every member and take one remote logical backup that you have actually opened.
2. Record the baseline: capture each member's `zaxon version` and, from `status --json`, its database id, configuration id, and applied slot.
3. Stop one follower, install the new binary, restart it, and wait for it to catch up with `zaxon wait --applied <slot>`.
4. Repeat for the remaining followers, one at a time.
5. Upgrade the leader last: stopping it triggers a normal election, and writers follow the redirect.
6. Verify again: run `zaxon integrity-check` on every member and compare `chain` values, which must be equal at the same applied slot.

If any step fails, stop that node and diagnose. Never delete or rewrite a journal to force a member to join.

Wire-version bridge

Wire compatibility is exact-major: protocol version 6 speaks only to version 6. A release that changes the wire version cannot use this rolling procedure. It requires an explicitly dual-version bridge release, and there is no automatic downgrade. Downgrading is supported only when the older binary declares every installed durable format and wire version compatible.

Exercise 13.1. Rehearse both recovery paths on the chapter 1 loopback cluster. First reimage: stop one voter, delete its entire data directory, restart it with its original command, and prove with `status --json` that its `chain` converges to the other members' value. Then restore: take a `backup`, stop all three voters, delete all three directories, rebuild the cluster, and reload the backup through `zaxon sql`. State exactly which writes each path preserved, and why the second path produced a new database identity.

Hint: Reimaging keeps the replicated history and the identity, because the quorum still holds both. A logical restore replays only rows into a brand-new history.

Teach it back. Explain to a colleague why Zaxonlite ships no journal repair tool. Use the words authoritative, acknowledged, quorum, and reimage. Then explain what a hand-patched journal could do to the cluster that a deleted one cannot.

14 Worked examples

Learning contract

By the end of this chapter you should be able to embed one durable node in a Zig program and name the durability effect of every call, run a failure-drill lab against a five-member cluster and predict each drill's outcome before you run it, and outline how an application with its own event loop embeds the C ABI without violating its threading contract. Guidance fades from example to example.

14.1 Small example: one durable node embedded in Zig

Runnable example, heavy guidance. Every call below is exercised, with the same argument shapes, by `zaxonlite/src/integration_test.zig`. The program mirrors how `zaxon` itself opens a node in `zaxonlite/src/main.zig`. Run it against a fresh directory. On a second run, `create table` fails by design, because the schema is already part of the replicated history.

examples/durable_node.zig (assembled from tested calls)

```
const std = @import("std");
const zaxonlite = @import("zaxonlite");

pub fn main(init: std.process.Init) !void {
    const gpa = init.gpa;
    const io = init.io;

    // Step 1: open (or create) the node data directory.
    const node = try zaxonlite.Node.open(gpa, io, .{ .directory = "./inventory" });
    defer node.close();

    // Step 2: replicated writes. Each exec is one journal-durable slot.
    _ = try node.exec("create table items(id integer primary key, v text)");
    const insert = try node.exec("insert into items(v) values ('tea'), ('coffee')");
    std.debug.print("{d} row(s) changed\n", .{insert.changes});

    // Step 3: read the applied state.
    var rows = try node.query(gpa, "select id, v from items order by id");
    defer rows.deinit();
    for (rows.rows) |row| {
        std.debug.print("{s}: {s}\n", .{ row[0].?, row[1].? });
    }

    // Step 4: exactly-once retry through a replicated session.
    const session = try node.openSession();
    const first = try node.execIdempotent(session, 1,
        "insert into items(v) values ('water')");
    const retry = try node.execIdempotent(session, 1,
        "insert into items(v) values ('water')");
    std.debug.assert(!first.replayed and retry.replayed);
    std.debug.assert(first.changes == retry.changes);

    // Step 5: snapshot. Seal the epoch, start the next one.
    try node.snapshot();
}
```

Let us name what each step guarantees, in order.

1. **Open.** `Node.open` acquires the directory lock, and a second open fails with `error.NodeLocked`. It verifies identity and the `CURRENT` snapshot pointer, truncates a torn journal tail, discards `current.db` with its WAL and SHM, and rebuilds the image from the verified snapshot plus the committed journal suffix. Nothing the previous process acknowledged can be missing. Nothing it never decided can appear.
2. **Write.** When `exec` returns, the transaction's descriptor is committed, its journal records are fsynced, and the slot is applied locally. In a one-member configuration all of that happens before the call returns. A failed statement rolls back and replicates nothing; the test proves this with a `not null` violation, after which the decided slot count has not moved.
3. **Read.** `query` accepts only read-only SQL. A write statement fails with `error.WriteInReadQuery` instead of sneaking around the replicated write path. The result set is arena-backed, and `deinit` frees all of it.
4. **Retry.** `openSession` is itself a replicated write, so the session and its last recorded result survive crash and restart. Retrying sequence 1 replays the recorded result without executing SQL. A skipped sequence fails with `error.SequenceGap`, an unknown session with `error.UnknownSession`, and a sequence older than the last with `error.ResultExpired`. None of those failures has side effects.
5. **Snapshot.** `snapshot` materializes a verified generation, installs `CURRENT`, advances the configuration ID, and starts an empty journal for the new epoch. Recovery afterward is the snapshot base plus the epoch suffix. Old epochs and unreferenced payloads become garbage.

> **Checkpoint: the journal is the database**
>
> The integration suite deletes `current.db` outright after step 2's writes and reopens the node: the rows come back and `integrityCheck()` passes. Try the same against your program's directory. The materialized image is a projection; the fsynced journal and payload store are the authority.

14.2 Middle example: a failure-drill lab

Runnable lab, medium guidance. Chapter 1 already taught bring-up, kill-the-leader, sessions, and backup, so we do not repeat them. This lab practices the failures those basics do not cover. Drills 1 and 2 mirror what `zaxonlite/src/role_cluster_test.zig` asserts about witnesses and read replicas. Drills 3 and 4 mirror the wipe, resync, and restart steps that `zaxonlite/src/cluster_test.zig` runs against real `zaxon serve` processes.

The lab needs a registry the quickstart cluster does not have. A witness votes, and the database identity is derived from the voting member ids plus the cluster id, so adding a witness describes a different database than the one your chapter 1 directories hold. That is a reconfiguration, not a restart with new flags. We sidestep it with fresh directories and a five-member registry: three data voters, one witness, one read replica.

```
zaxon serve --data ./lab/n1 --node 1 --listen 127.0.0.1:7101 \
    --peer 2@127.0.0.1:7102 --peer 3@127.0.0.1:7103 \
    --peer 4@127.0.0.1:7104/witness --peer 5@127.0.0.1:7105/read-replica &
```

Nodes 2 and 3 repeat that shape with their own directory, id, and listen address. The witness and the replica name their own roles:

```
zaxon serve --data ./lab/n4 --node 4 --role witness \
    --listen 127.0.0.1:7104 \
    --peer 1@127.0.0.1:7101 --peer 2@127.0.0.1:7102 \
    --peer 3@127.0.0.1:7103 --peer 5@127.0.0.1:7105/read-replica &
zaxon serve --data ./lab/n5 --node 5 --role read-replica \
    --listen 127.0.0.1:7105 \
    --peer 1@127.0.0.1:7101 --peer 2@127.0.0.1:7102 \
    --peer 3@127.0.0.1:7103 --peer 4@127.0.0.1:7104/witness &
```

Set an endpoint list for the voters, wait for a leader, and create a table:

```
V=127.0.0.1:7101,127.0.0.1:7102,127.0.0.1:7103
zaxon wait --connect $V --leader
zaxon exec --connect $V --sql "create table drills(id integer primary key, note text)"
```

14.2.1 Drill 1: what a witness buys you

Ask the registry what node 4 can do:

```
zaxon members --connect $V --json
```

Node 4 reports `votes` true and `serves_reads` false; node 5 reports the reverse. The flags are the whole story. A witness stores the replicated log so it can vote honestly, but it never materializes a SQL image, so it has nothing to answer a query with:

```
zaxon query --connect 127.0.0.1:7104 --level any --sql "select 1"
```

The witness answers with the error `forbidden`, and the command exits 4. This refusal is not a missing feature. A node that voted on history it never stored as SQL would invite reads it cannot serve consistently.

Now use the vote. Find the leader, then stop one data voter that is not the leader (here we assume the leader is node 1):

```
zaxon leader --connect $V
zaxon stop --connect 127.0.0.1:7103
zaxon exec --connect $V --sql "insert into drills(note) values ('one voter down')"
```

The write succeeds. Count the voting set: nodes 1 through 4 vote, so quorum is three, and voters 1 and 2 plus the witness reach it. The witness kept the cluster writable while holding no data. Restart node 3 with its original `serve` command before the next drill.

14.2.2 Drill 2: stale and fresh reads at the replica

Write one row, then read it back at the replica with a freshness bound:

```
zaxon exec --connect $V --sql "insert into drills(note) values ('fresh')"
zaxon query --connect 127.0.0.1:7105 --level any --freshness-ms 2000 \
    --sql "select count(*) from drills"
```

The replica answers. A freshness-bounded read passes two checks: the replica heard from a leader inside the bound, and its applied slot has caught up to the decided slot it last observed. Note that `--freshness-ms` requires `--level any`; a bound on a fenced read would be redundant, and the server rejects the combination as a usage error.

> **Predict before reading on**
>
> Stop all three data voters. The replica still holds every applied row. Should the freshness-bounded read above still answer? Decide before you run the next block.

```
zaxon stop --connect 127.0.0.1:7101
zaxon stop --connect 127.0.0.1:7102
zaxon stop --connect 127.0.0.1:7103
zaxon query --connect 127.0.0.1:7105 --level any --freshness-ms 500 \
    --sql "select count(*) from drills"
```

The read fails with the error `stale` and exit code 4, even though the rows are sitting right there. The bound is a promise about recency, not about possession, and with no leader the replica cannot prove recency. Two follow-ups sharpen the boundary. A plain `--level any` query still answers from the local image, because you asked for no promise. A `--level linearizable` query against the same endpoint exhausts the redirect budget and exits 4, because no quorum can fence a read. Restart the three voters and the bounded read recovers on its own.

14.2.3 Drill 3: wipe a follower, then resync across a sealed epoch

Stop node 2 and destroy its materialized image:

```
zaxon stop --connect 127.0.0.1:7102
rm ./lab/n2/current.db
```

Restart node 2 with its original `serve` command. It rebuilds the image from its snapshot plus its journal, then rejoins. Prove it: read `applied_slot` from `zaxon status --connect 127.0.0.1:7101 --json`, wait with `zaxon wait --connect 127.0.0.1:7102 --applied <slot>`, and compare `chain` in both nodes' `status --json`. Equal chains mean identical applied history.

Now make the follower miss a whole epoch instead of a file:

```
zaxon stop --connect 127.0.0.1:7102
zaxon exec --connect $V --sql "insert into drills(note) values ('pre-roll')"
zaxon snapshot --connect $V
zaxon exec --connect $V --sql "insert into drills(note) values ('post-roll')"
```

Restart node 2 again. Its journal ends in an epoch the cluster has sealed, so a journal suffix alone cannot catch it up. Watch `status --json` on port 7102: the configuration id jumps when the transferred snapshot installs, and `applied_slot` then climbs through the new epoch. Both writes are present at the end, and the chains converge again.

14.2.4 Drill 4: the full-cluster restart

Record the current row count with a fenced read:

```
zaxon query --connect $V --sql "select count(*) from drills"
```

Now kill all five processes with `kill -9`. Not `stop`, not Ctrl-C. The automated suite does the same, because recovery must not depend on graceful shutdown. Then restart all five with their original commands and verify:

```
zaxon wait --connect $V --leader --timeout-ms 20000
zaxon query --connect $V --sql "select count(*) from drills"
zaxon integrity-check --connect 127.0.0.1:7101
```

The count is unchanged, and the integrity check passes on each data voter. Every write that was acknowledged before the kill is present exactly once afterward. That sentence is the product; the drills exist so you believe it.

14.3 Large example: the C ABI inside your own event loop

Design sketch, light guidance. The API surface below is `zaxonlite/include/zaxonlite.h`, and every function named here is exercised by the C smoke test in `zaxonlite/test/capi_smoke.c`, built by `zig build test-cabi`. The event loop integration around it is yours to design.

One constraint shapes the whole design: every call is synchronous. A single-node handle from `zaxonlite_open` blocks in `zaxonlite_exec` until the write is journal-durable and applied. A cluster facade from `zaxonlite_cluster_open` owns its own listener, peer, and tick threads and blocks the calling thread until quorum commit. Use each handle from one thread at a time. An application with a latency-sensitive loop, such as a game server, a UI, or an async runtime, should therefore never call either handle from the loop thread.

One workable shape:

1. A dedicated database thread owns the handle for its whole lifetime, from `zaxonlite_open(dir, &db)` through `zaxonlite_close(db)`.
2. The loop enqueues request records, SQL text plus a completion callback or future, on a bounded queue; the database thread executes them in order and posts results back through the loop's own wake-up mechanism.
3. Reads return owned JSON from `zaxonlite_query_json` or `zaxonlite_query_prepared_json`; hand the buffer to the loop and let the consumer call `zaxonlite_free`.
4. Writes that must survive an ambiguous crash use `zaxonlite_session_open` once, then `zaxonlite_exec_idempotent` with a monotonic sequence; on restart, replaying the last sequence is safe by construction, and the smoke test proves it by reopening the directory and asserting `replayed`.
5. Multi-statement writes go through `zaxonlite_transaction_begin`, `_exec`, `_commit`, and `_close`; statements and bound values are copied, so the loop may free its buffers as soon as each call returns.

6. Every return code maps to a policy: 0 continue; 1 surface the SQL or session error from `zaxonlite_last_error` to the caller; 2 is a bug in your marshalling, such as a null argument or a write on the read path; 3 and 4 stop accepting work and page an operator.

Gaps deliberately left for you: the queue's backpressure rule when the database thread falls behind; whether reads bypass the queue on a second read-only path; how sequences are assigned when several loop tasks share one session; and when periodic `zaxonlite_snapshot` and `zaxonlite_expire_sessions` calls run so they never race a burst of writes.

Exercise 14.1. Extend the chapter 1 quickstart cluster with a read replica. Tear the three voters down, then restart all four processes with the same extended registry: node 4 serves with `--role read-replica` and the three voter peers, and each voter adds `--peer 4@127.0.0.1:7004/read-replica`. Write through the voters, then compare three reads against port 7004: `--level any` immediately after the write, the same query with `--freshness-ms 500`, and `--level linearizable`. Explain which answered locally, which was rejected as stale or answered fresh, and which redirected. Then explain why adding node 4 changed no write quorum and no database identity.

Hint: A learner rejects a freshness-bounded read until it has heard a leader heartbeat inside the bound and its applied slot has caught up. A linearizable request to a learner returns `not_leader` with the certifying voter as the redirect hint. Identity and quorum are derived from voting members only.

Teach it back. Using only the words votes and serves_reads, explain to a colleague why the witness in drill 1 kept writes alive but could not answer a query, while the replica in drill 2 could answer a query but could not keep writes alive. Then explain which drill would break if the two flags were ever combined carelessly on one node.

PART V

Reference and evidence

This part collects the facts you look up rather than reread: byte formats, wire frames, and the desk tables for commands, errors, and limits. It closes with the verification work that gives us the right to print them.

15 Format reference

> **Learning contract**
> By the end of this chapter you should be able to decode a payload header from a hex dump, read a snapshot manifest and an identity file, name every wire frame kind and what its body carries, and state the exact size and layout of one replicated command.

A format is a contract. Once a byte layout has reached a disk or a socket, every future version of Zaxonlite must still read it. This chapter states each contract precisely enough to check a hex dump against it. Two conventions hold everywhere. Integers are little-endian unless a field says otherwise. Every hash is SHA-256.

15.1 Payload ("ZXPL")

A payload is the immutable object that a descriptor's `payload_hash` names. Chapter 5 explains how one is captured from the SQLite WAL. Here is its header:

Offset/size	Field	Meaning
0 / 4	`magic`	`0x4c50585a` ("ZXPL")
4 / 1	`version`	1
5 / 3	reserved	zero
8 / 4	`page_size`	power of two, 512–65536
12 / 16	`database_id`	must match the node's identity
28 / 4	`transaction_count`	≥ 1
32 / 4	`frame_count`	≥ 1

Three regions follow the header. First come `transaction_count` records of 32 bytes each: `session_id:u64`, `sequence:u64`, `first_frame:u32`, `frame_count:u32`, `change_count:i64`. Then come `frame_count` records of 8 bytes each: `page_number:u32` and `commit_size:u32`. SQLite stores those two values big-endian in the WAL. The capture path converts them, so the descriptors in a payload are already native. The raw page images close the object.

Validation is strict, because a payload becomes evidence. The transactions must tile the frame range in order, with no gap and no overlap. Each transaction must end on a commit frame. A payload that fails any check is rejected before Paxos state may reference it.

15.2 Snapshot manifest

Every snapshot generation carries a manifest of plain `key=value` lines. The whole file is hashed for the stop-sign digest:

```
format=1
database_id=<32 hex>
sealed_configuration_id=<decimal>
applied_slot=<decimal>          # slot before the stop sign
chain=<64 hex>
db_sha256=<64 hex>
```

The stop-sign metadata is the string `zx1 <snapshot-name-16hex> <manifest-sha256>`. During normal local rollover, `completeClusterRollover` requires the manifest to hash to this decided value. Followers reproduce the physical file from the same decided WAL page frames and require the identical manifest digest. The network install carries a canonical `ZXP1` proof containing the

database ID, sealed and next configurations, stop/applied slots, chain, manifest digest, voter set, and exact stop metadata. The receiver validates those bindings and requires matching proof-digest reports from a voter read quorum over mTLS before activation. This adds neither snapshot signatures nor another Paxos phase; it confirms the stop sign Paxos already chose.

15.3 Identity file

```
format=2
node_id=<decimal>
database_id=<32 hex>
configuration_id=<decimal>
role=<data-voter|witness|standby|read-replica>
```

Format 1 omitted `role`. A node reads such a file as `data-voter` and upgrades it to format 2 on the next identity write. Opening a directory under a different role is refused. That refusal protects safety. A restart must never silently turn a voter into a learner, or the reverse. Gateways keep no identity file because they hold no state.

15.4 Wire frames

Every connection speaks one framing: a `u32 total_len` (body length plus one), a `u8 kind` byte, then the body. A body of 64 MiB or more is a protocol error. The largest legal body is one byte under 64 MiB. One declared snapshot or backup transfer — the size announced by `snapshot_begin` or `backup_begin` — is bounded by `wire.max_transfer_bytes`, 4 GiB by default, sized for the small embedded database profile. The server enforces its configured bound (`ServeOptions.max_transfer_bytes`), and the client enforces the same default on backup downloads.

Kind	Name	Body
1	`hello`	`version:u16`, `kind:u8` (0 peer, 1 client, 2 enrollment), `node_id:u32`, `database_id:u128`, `configuration_id:u64`.
2	`envelope`	`configuration_id:u64`, then the encoded Paxos envelope: `from:u32`, `to:u32`, `tag:u8`, message fields.
3	`payload_data`	`hash:[32]u8`, then the payload bytes. The receiver verifies the hash.
4	`payload_request`	`hash:[32]u8`.
5	`fence_request`	ballot as `u64`, `u32`, `u32`, then `fence_id:u64`, `fence_slot:u32`.
6	`fence_ack`	`fence_id:u64`, `ok:u8`, the promised ballot.
7	`snapshot_request`	Empty.
8	`snapshot_begin`	`configuration_id:u64`, `name:[16]u8`, `db_size:u64`, `manifest_len:u32`, the manifest, `proof_len:u16`, and the bounded `ZXP1` proof.
9	`snapshot_chunk`	`offset:u64`, then image bytes.
10	`snapshot_end`	Empty.

Kind	Name	Body
11, 12	`rpc_request`, `rpc_response`	One JSON object.
13	`payload_stored`	`hash:[32]u8`. Sent only after verified durable installation.
14, 15	`auth_challenge`, `auth_response`	A fresh nonce and mutual HMAC-SHA256 proofs.
16	`backup_begin`	`size:u64`, `sha256:[32]u8`.
17	`backup_chunk`	`offset:u64`, then backup bytes.
18	`backup_end`	Empty.
19	`learner_commit`	`configuration_id:u64`, `slot:u32`, then one canonical chosen entry. Accepted only from a configured voter.
20	`learner_heartbeat`	`configuration_id:u64`, `decided_through:u32`. Drives bounded-staleness checks. Carries no vote.
21, 22	`checkpoint_proof_request`, `checkpoint_proof_reply`	`nonce:u64`, `sealed_configuration_id:u64`, `proof_sha256:[32]u8`. Matching configured-voter replies form the install read quorum.
23	`enrollment_request`	`secret:[32]u8`, `node_id:u32`, `database_id:u128`, `csr_len:u32`, then at most 16 KiB of CSR PEM.
24	`enrollment_response`	`status:u8`, then on success at most 64 KiB of certificate PEM.

The current `hello` version is 6. Older versions are rejected outright, never silently downgraded. Version 2 added the storage-ACK gate and applied payload materialization to value-bearing phase-one promises. Version 3 added mutual authentication and backup streaming. Version 4 added durable voter-certified chosen-entry delivery and freshness heartbeats for non-voting learners. Version 5 added checkpoint-proof transfer and quorum confirmation. Version 6 added the bounded one-time token/CSR enrollment exchange. A certificate-less TLS connection is accepted only by a deliberately configured issuer, only for connection kind 2, and only for this single request. The opaque owner-only `ZXET` bundle binds the random token to the CA, endpoint, issuer, database, target node, and expiry; the issuer's `ZXER` record stores only its domain-separated hash. Chapter 13 gives the operational contract and `docs/zds/records/0004-zaxonlite-format.typ` freezes both version-1 encodings.

After authentication, every application body is wrapped as `sequence:u64 || body || hmac:[32]u8`. The receiver requires the exact next sequence. That rule rejects replayed frames.

The envelope message tags are: prepare 0, promise 1, promise_done 2, accept 3, accepted 4, commit 5, learn 6, nack 7, heartbeat 8. Each carries exactly the fields of the core protocol's message type. Log entries inside them use the same canonical codec the journal uses. One encoder serves the disk and the wire, so they can never disagree.

15.5 The replicated command encoding

A `Command` encodes into a fixed 153-byte canonical form: one tag byte, two `u128` fields (32 bytes), two `u64` fields (16 bytes), three 32-byte hashes (96 bytes), and two `u32` fields (8 bytes). Fields a tag does not use must be zero, and decode enforces that padding. Tag 0 is `noop`. Tag 1 is `transaction_batch` and uses every descriptor field in order. Tag 2 is `read_barrier` and uses the nonce. Non-canonical padding and unknown tags are decode errors. One byte pattern has one meaning. That property is what lets equal chain hashes mean equal history.

Exercise 15.1. A payload header begins `5a 58 50 4c 01 00 00 00 00 10 00 00`. Read off the magic, the version, and the page size. Then explain which later check would still reject this payload if its one transaction did not end on a commit frame.

Hint: The integers are little-endian. `0x1000` is 4096.

Teach it back. Explain to a colleague why payload validation runs before Paxos may reference the payload, and what could go wrong if a node accepted a descriptor whose payload bytes it had never verified. Use the words payload gate and safety.

16 Desk reference

These are the tables to keep beside a terminal. Every row was verified against `zaxonlite/src` (`main.zig`, `server.zig`, `client.zig`, `tls.zig`, `types.zig`, `prepared.zig`, `command.zig`, `durability.zig`), against `include/zaxonlite.h`, and against the format contract.

16.1 CLI commands and flags

Chapter 2 is the full command reference. It shows one worked example per command, so we do not repeat the commands here. Two flags select the mode. `--data <dir>` runs the command against a local node, in process. `--connect <endpoint>[,...]` speaks the client RPC protocol to a running cluster and follows `not_leader` redirects on its own. `--json` switches any command to machine-readable output on stdout. An endpoint is `host:port`, or `unix:<path>` for a local server's Unix-domain socket. The remaining flags are a short list. Each belongs to one or two commands.

Flag	Meaning
`--config <path>`	JSON configuration file. `ZAXON_CONFIG` names the same file through the environment.
`--sql <text>`	The statement for `exec` and `query`.
`--session <id>`	Session identity for an idempotent `exec`. Use it together with `--sequence`.
`--sequence <n>`	Monotonic per-session sequence for `exec`.
`--level <level>`	Read level for `query`: `any`, `leader`, or `linearizable` (the default).
`--freshness-ms <n>`	Maximum age for a local learner read. Valid only with level `any`.
`--to <path>`	Backup destination for `backup`, or owner-only token-bundle destination for `enroll-token`. Existing destinations are refused.
`--retain <n>`	Activity window for `expire-sessions`: keep sessions inside the `n` most recent session-write activities. Required.
`--applied <slot>`	Slot that `wait` waits for. The default is 0.
`--leader`	Make `wait` also wait for a known leader.
`--timeout-ms <n>`	Deadline for `wait`. The default is 10000.
`--node <id>`	This server's node ID for `serve`, or the configured target node for `enroll-token`.
`--listen <endpoint>`	This server's endpoint: `host:port`, or `unix:<path>` for owner-only single-node local service. Required by `serve`.
`--role <role>`	`data-voter` (the default), `witness`, `standby`, `read-replica`, or `gateway`.
`--peer <spec>`	One peer as `id@host:port[/role]`. Repeat the flag once per peer (`serve`).
`--cluster-id <text>`	Extra entropy for the derived database identity.
`--auth-file <path>`	Transport secret provider, or `ZAXON_AUTH_FILE`. Never a literal key on the command line.
`--dev-psk`	Allow PSK-only TCP for a local development cluster. Requires an owner-only `--auth-file`; listener and peers must use numeric loopback.

Flag	Meaning
`--tls-cert <path>`	Node certificate PEM for mutual TLS (`serve` and client mode). All three TLS flags go together.
`--tls-key <path>`	Node private key PEM for mutual TLS.
`--tls-ca <path>`	Cluster CA PEM that every peer certificate must chain to.
`--enrollment-ca-key <path>`	Owner-only CA private key enabling the narrow token/CSR issuer on `serve`; omit it on ordinary nodes.
`--token-file <path>`	Owner-only opaque bearer bundle consumed by `enroll`.
`--identity-dir <path>`	New directory atomically receiving `node.key`, `node.crt`, and `ca.crt` from `enroll`.
`--ttl-seconds <n>`	Enrollment lifetime: 600 by default, at most 86400.
`--revocation-file <path>`	Reloaded node-ID denylist for `serve`.
`--sync <mode>`	Durability sync mode, any command: `full` (the default; `F_FULLFSYNC` on macOS, survives power loss) or `os` (plain `fsync`, development only on macOS). Any other value is a usage error, exit 2: `--sync must be os or full`.
`--enable-failpoints`	Honor failpoint RPCs. Test controllers only.
`--json`	Machine-readable output on stdout.

Configuration precedence is fixed: explicit flags win, then environment variables, then the `--config`/`ZAXON_CONFIG` JSON file. The environment names are `ZAXON_DATA`, `ZAXON_CONNECT`, `ZAXON_LISTEN`, `ZAXON_NODE`, `ZAXON_ROLE`, `ZAXON_PEERS`, `ZAXON_CLUSTER_ID`, `ZAXON_AUTH_FILE`, `ZAXON_TLS_CERT`, `ZAXON_TLS_KEY`, `ZAXON_TLS_CA`, and `ZAXON_SYNC`. Server provider paths also have `ZAXON_ENROLLMENT_CA_KEY` and `ZAXON_REVOCATION_FILE`.

16.2 Exit codes

Code	Meaning
0	Success.
1	SQL or session error, including `SequenceGap`, `UnknownSession`, and `ResultExpired`.
2	Usage: an unknown command or option, a missing flag, an unreadable configuration or secret provider, or an invalid registry.
3	Integrity failure reported by `integrity-check` or `recover`.
4	Unavailable: a locked directory, corrupt state, a failed open or listen, no reachable leader, or a malformed peer response.

16.3 Client RPC operations

One JSON object travels per `rpc_request` frame and one per `rpc_response`. Every success body carries `"ok":true`. New fields are only ever added, so additive change stays compatible.

Op	Request fields	Success response fields
`exec`	`sql`; optional `session` and `sequence`, always together	`changes`, `slot`, `replayed`

Op	Request fields	Success response fields
`query`	`sql`; optional `level` (default `linearizable`) and `freshness_ms` (level `any` only)	`columns`, `rows`, `level`
`session`	none	`session_id`
`wait`	`applied`, `leader` (bool), `timeout_ms`	`applied_slot`, `decided_slot`, `leader`, `configuration_id`
`status`	none	The full status record: identity, role, `leader`, `ballot`, decided and applied slots, journal, chain, `snapshot`.
`members`	none	`voter_membership`, `nodes` with role and capability booleans, `self`, `leader`
`leader`	none	A `leader` object (`id`, `host`, `port`), or `null`.
`snapshot`	none	`configuration_id`. Leader only.
`integrity`	none	`ok` mirrors the report; `sqlite_ok`, `chain_ok`, `payloads_ok`
`hash`	none	`chain`, `content`, `applied_slot`
`expire-sessions`	`retain`	`expired`
`issue-enrollment-token`	`node_id`; optional `ttl_seconds`	`node_id`, `issuer_node_id`, `database_id`, `expires_unix_seconds`, `token`. Issuer-only; the caller already passed normal mTLS.
`backup`	none	Not JSON: a `backup_begin` frame (size and SHA-256), ordered `backup_chunk` frames, `backup_end`. A JSON error means the request was rejected.
`failpoint`	`name`; the server must run with `--enable-failpoints`	`ok`
`stop`	none	`ok`, then the server shuts down.

An error response is `{"ok":false,"error":code,"message":text}`. The `not_leader` code adds the advertised `leader` endpoint so a client can redirect.

Error code	Meaning	CLI exit
`bad_request`	Malformed JSON, an unknown op, or a missing or invalid field.	2
`sql`	The statement failed, or a write reached the read path.	1
`session`	`UnknownSession`, `SequenceGap`, or `ResultExpired`.	1
`not_leader`	A leader-only request was refused. Follow the hint.	4
`stale`	A learner read exceeds `freshness_ms` or the learner has no leader contact.	4
`forbidden`	This node type does not serve SQLite reads. Witnesses answer this way.	4
`ambiguous`	The write's fate is unknown after a leader change. Retry idempotently with the same session and sequence.	4

Error code	Meaning	CLI exit
`retry`	Transient: an epoch rollover or a leadership change during a fence.	4
`timeout`	An operation, fence, or wait deadline passed.	4
`too_large`	The transaction payload exceeds the 64 MiB wire limit.	4
`unavailable`	The node failed or durable storage stopped.	4
`internal`	An unexpected error. The name travels in `message`.	4

16.4 C ABI

Every `int` function returns the same codes: 0 ok, 1 SQL or session error, 2 misuse (a null argument, or a write on the read path), 3 integrity failure, 4 unavailable. `zaxonlite_last_error` holds the most recent message per handle. JSON buffers are released with `zaxonlite_free`. Declared counts and lengths are validated against product limits before use (registry ≤ 36 members, secret ≤ 4096 bytes, value ≤ 64 MiB), and every out-parameter is set to a safe empty value on every path; chapter 11 states the full boundary contract.

Function	Purpose
`zaxonlite_version`	Return the library version string.
`zaxonlite_open`, `_close`	Own one node data directory. A second open of a locked directory returns 4.
`zaxonlite_exec`	Run one replicated write transaction and report its change count.
`zaxonlite_exec_prepared`	Run one prepared statement with `zaxonlite_value` bindings. TEXT and BLOB values are borrowed for the duration of the call.
`zaxonlite_transaction_begin`, `_exec`, `_commit`, `_close`	Collect copied statements, then replicate them atomically. Each transaction object is single-use.
`zaxonlite_session_open`	Open a replicated exactly-once session.
`zaxonlite_exec_idempotent`	Execute one session sequence at most once. Sets `replayed` when it returns the saved result of the last sequence.
`zaxonlite_query_json`	Run a read-only query. Returns a JSON buffer of columns and rows.
`zaxonlite_query_prepared_json`	The same read with `zaxonlite_value` bindings.
`zaxonlite_free`	Release a returned JSON buffer.
`zaxonlite_snapshot`	Seal the epoch and install a snapshot generation.
`zaxonlite_backup`	Write a consistent logical backup to a path.
`zaxonlite_integrity_check`	Verify the image, the descriptor chain, and every referenced payload.

Function	Purpose
`zaxonlite_expire_sessions`	Delete idle sessions outside the retained activity window.
`zaxonlite_last_error`	Return the most recent error message for the handle.
`zaxonlite_cluster_open`, `_close`	Own a transport-owning member built from a runtime registry (`zaxonlite_cluster_options`, at most 36 members, nine of them voters).
`zaxonlite_cluster_exec`, `_query_json`	Cluster writes and reads through the facade.
`zaxonlite_cluster_call_json`, `_last_error`	Generic JSON RPC and error text through the facade.

16.5 Diagnostic catalogue

Every operator-facing failure reaches stderr in one fixed shape, produced by `zaxonlite/src/diagnostic.zig`: an uppercase boundary header, a plain description, and a resolution hint.

```
-- NODE DIRECTORY LOCKED --

Another process owns this data directory.

Hint: Stop that process or choose a different --data directory.
```

Client mode renders a server error code in the same shape, with the code as the header (for example `-- NOT_LEADER --`) and a code-specific hint. These are the diagnostics you will meet, with the exit code each one produces.

Header	Emitted when
`-- UNKNOWN OPTION --`	An argument is not a recognized flag. Exits 2.
`-- INVALID COMMAND --`	A required flag or value is missing or malformed. Exits 2.
`-- UNKNOWN COMMAND --`	The command word is not one `zaxon` knows. Exits 2.
`-- UNSUPPORTED CLIENT COMMAND --`	An embedded-only command was given `--connect`. Exits 2.
`-- CONFIGURATION UNREADABLE --`	`--config` or `ZAXON_CONFIG` names a file that cannot be read or parsed as JSON. Exits 2.
`-- AUTHENTICATION PROVIDER UNREADABLE --`	The `--auth-file` provider is missing, unreadable, or shorter than 32 bytes. Exits 2.
`-- INVALID CLUSTER CONFIGURATION --`	`serve` was given a bad registry or role set. Exits 2.
`-- INVALID GATEWAY CONFIGURATION --`	The gateway's registry or role set is invalid. Exits 2.
`-- INVALID LISTEN ADDRESS --`	The listen endpoint cannot be parsed. Exits 2.
`-- SQL ERROR --`	A write statement failed. Exits 1.
`-- QUERY ERROR --`	A read failed, or a write reached the read path. Exits 1.
`-- SESSION ERROR --`	Session misuse: `UnknownSession`, `SequenceGap`, or `ResultExpired`. Exits 1.

Header	Emitted when
`-- BACKUP FAILED --`	The backup destination is unusable. Exits 1.
`-- NODE DIRECTORY LOCKED --`	Another process owns the data directory. Exits 4.
`-- NODE OPEN FAILED --`	Identity, journal, payload, or filesystem verification failed on open. Exits 4.
`-- MUTUAL TLS REQUIRED --`	A TCP storage listener was configured without a complete TLS identity and without explicit loopback-only `--dev-psk`. Exits 4.
`-- TLS IDENTITY FAILED --`	The TLS certificate, key, or CA cannot be loaded, or the key does not match the certificate. Exits 4 from `serve`, 2 from client mode.
`-- LISTEN FAILED --`	The server endpoint cannot be bound. Exits 4.
`-- UNIX SOCKET LISTEN FAILED --`	The Unix-socket path already exists, cannot be bound, or its permissions cannot be restricted. Exits 4.
`-- GATEWAY LISTEN FAILED --`	The gateway endpoint cannot be bound. Exits 4.
`-- NO REACHABLE LEADER --`	No endpoint completed a leader-only request. Exits 4.
`-- LEADER REDIRECT REFUSED --`	A leader was advertised, but the transport cannot authenticate the redirect target (PSK or plaintext) and the address is not in `--connect`. Exits 4.
`-- MALFORMED RESPONSE --`	A peer answered outside the JSON contract. Exits 4.
`-- BACKUP INTERRUPTED --`	The streaming backup lost its server. The destination was not installed. Exits 4.
`-- REMOTE BACKUP FAILED --`	The server rejected or aborted the backup. The destination was not installed. Exits 4.
`-- COMMAND FAILED --`	Any unclassified error. Preserve the error and the node logs. Exits 4.
`-- UNKNOWN SHELL COMMAND --`	A shell dot command is unknown. The shell continues.
`-- INPUT LINE TOO LONG --`	A shell input line exceeds 64 KiB. The shell continues.

16.6 Limits

Limit	Value	Source
Voters per consensus group, witnesses included	1–9	`types.zig`
Epoch capacity	2,048 slots, 4 reserved for the stop sign	`types.zig`, `node.zig`
Protocol append batch	16 entries	`types.zig`
Stop-sign metadata	256 bytes	`types.zig`
Wire frame body	64 MiB	format contract
Declared snapshot or backup transfer	4 GiB default, server configurable	`wire.zig`, `server.zig`
Concurrent server connections	4 × configured members + 16 default	`server.zig`

Limit	Value	Source
Handshake completion deadline	10 000 ms default, 0 disables	`server.zig`
Mutual TLS protocol version	TLS 1.3 minimum	`tls.zig`
Zaxon wire protocol version	6, exact match	`wire.zig`
Enrollment token lifetime	600 s default; 86400 s maximum	`enrollment.zig`
Peer certificate common name	`zaxon-node-<id>`, under 64 bytes	`tls.zig`
C ABI cluster registry	36 members, at most 9 voters	`embedded.zig`
One transaction payload	64 MiB minus 73 bytes	`command.zig`
Explicit transaction	1024 statements, 64 MiB copied input	`prepared.zig`
Idempotent sessions	one outstanding sequence, last result only	format contract
Writers per database	1, serialized through the leader	format contract
Learners and gateways	runtime-sized registry	format contract
Shell input line	64 KiB	`main.zig`

16.7 Glossary

Term	Meaning
Batch	The unit one slot decides: the WAL frames SQLite committed for one write request, named by a random 128-bit batch ID.
Descriptor	The fixed 153-byte replicated command that names a payload by content hash. Paxos never carries transaction bytes.
Payload	The immutable `ZXPL` object holding one batch's transaction records, frame metadata, and page images.
Chain hash	The cumulative SHA-256 identity of the decided history. Equal chains mean identical applied history. It is not a file hash.
Journal	The per-epoch fsynced record of protocol writes. It is the authoritative state from which everything else rebuilds.
Materialized image	`current.db`, a rebuildable projection of the decided history. It is never accepted as evidence over Paxos state.
Epoch	One bounded configuration of the replicated log: at most 2,048 slots under one configuration ID, sealed by a stop sign.
Generation	One installed snapshot, manifest plus database image. The newest fully installed one is named by `CURRENT`.
`CURRENT` pointer	The atomically replaced file naming the single installed snapshot generation that recovery may start from.
Data voter	A voter that may campaign, materializes SQLite, and serves SQL.
Witness	A voter that stores durable payloads but never campaigns and serves no SQL.
Campaigner	A node whose role permits starting elections. In this release that means exactly the data voters.

Term	Meaning
Standby, read replica	Non-voting learners that keep SQLite copies. An operator may promote a standby. A read replica is never promoted.
Gateway	A stateless byte-transparent router. It holds no Paxos state, no SQLite state, and no identity file.
Learner commit	A configured voter's certificate that a slot's entry was chosen. It is sent to a learner only after the learner's durable payload ACK.
Payload gate	The transport rule that holds a value-bearing envelope until its payload bytes are stored and digest-verified locally.
Fence	The quorum read barrier. The leader confirms its exact ballot with a read quorum before a linearizable read. No append, no sync.
Session, sequence	A replicated exactly-once identity and its monotonic counter. Only the next sequence executes. The last one replays its saved result.
Database identity	The shared 128-bit ID derived from the sorted voter IDs, plus the optional cluster ID. The `hello` handshake enforces it.

Teach it back. Close the book. Write down the four nonzero exit codes and one situation that produces each. Then reopen this chapter, find the first mismatch, and reread only the row that corrects you.

17 Verification

Learning contract

By the end of this chapter you should be able to name each test layer and the question it answers, follow the mandatory cluster scenario step by step, point at the enforcement and the oracle for every persistence assertion, read the benchmark dashboard against rqlite without being misled by it, and state which claims this release does not verify.

Checkpoint: recovery and read levels

This chapter leans on chapter 6 for the recovery sequence, and on chapter 8 for the crash matrix and read levels. The benchmark section uses both without re-deriving them.

A guarantee without a test is a claim. This book has made many guarantees, and this chapter is where we pay for them. We verify Zaxonlite in layers, in the same order we built it: bytes first, then one process, then three processes, then hostile networks, then time and load. Each layer has its own suite and its own oracle. The last sections measure the finished product against rqlite and say plainly what those numbers mean. Chapter 18 then closes the book by tracing every guarantee to its code and its oracle.

17.1 The test layers

Each suite exists to answer one question that the suites below it cannot. A unit test cannot prove that three real processes elect a leader. A cluster test cannot pin the exact bytes of a torn journal tail. So we keep both, and everything between.

Layer	The question it answers	Command
Unit	Do the codecs, chain hashes, journal replay, torn tails, payload-store faults, and the WAL capture spike behave at the byte level?	`zig build test`
Single process	Does one real node survive restart, rebuild a deleted or stale image, expire sessions, roll snapshots, and isolate failed SQL?	`zig build test-single`
Crash points	When a real process dies at each write-pipeline failpoint, does recovery ever invent a false success?	`zig build test-crash`
Cluster	Do three real processes elect, fail over, catch up, transfer snapshots, and survive a total restart? The full script follows below.	`zig build test-cluster`
Roles	Do the witness, standby, and read replica obey their read restrictions and bounded-freshness refusals?	`zig build test-roles`
Gateway	Does authenticated RPC pass end to end through a process that holds no Paxos or SQLite state?	`zig build test-gateway`
Adverse network	Does the cluster stay correct under real TCP loss, duplication, reordering, seven-byte fragmentation, and delayed durable sync?	`zig build test-fault-network`

Layer	The question it answers	Command
CLI contract	Do exit codes, JSON shapes, session flags, the scripted shell, locking, loopback-only development PSK, startup/leader logs, enrollment/revocation, and single-seed mTLS redirects match the reference?	`zig build test-cli`
C ABI	Does every exported function work, including locking and replay after reopen?	`zig build test-cabi`
Fuzz	Do seeded random bytes break the decoders, the journal files, or a whole node under random SQL, crashes, and a rebuild-convergence oracle?	`zig build fuzz`
Soak	Does sustained mixed load, with restarts and snapshots, ever drift from a live row-count model?	`zig build soak`
Benchmark	What do writes, reads, recovery, and rebuild cost in ReleaseFast on this machine?	`zig build benchmark`
Cluster benchmark	What do replicated writes and reads cost across three real server processes, in test-only plaintext, development PSK, and mTLS transport modes?	`zig build bench-cluster`

`zig build test-cluster -Dcluster-runs=N` repeats the whole cluster scenario for flake hunting. One hundred consecutive runs are an explicitly deferred CI gate, not a result this release claims. The rqlite and dqlite comparison harnesses are scripts under `zaxonlite/benchmarks/`, not build steps. They appear in the measured section below.

17.2 The mandatory cluster scenario

The cluster suite is a controller that spawns three real `zaxon serve` processes on loopback ports. It drives them only through the public RPC protocol. There is no test back door into node internals. If the public surface cannot observe a behavior, the test may not rely on it either. Every wait is a deadline on an observable status field. On failure the controller dumps all three statuses and log tails, because a distributed test that hides its state teaches nothing.

The scenario strings together every failure that chapters 6 and 7 promised to survive, in one run:

Step	Actor	Event and reason
1	Controller	Waits for an election, then creates the schema.
2	Controller	Submits writes 1–100 through all three endpoints. Half travel inside a session, half without.
3	Oracle	A linearizable count must see exactly 100 rows.
4	Controller	Stops one follower, writes 101–150 on the surviving quorum, restarts the follower, and waits for catch-up.
5	Oracle	Chain and content digests must be identical on all three members.
6	Controller	Arms the leader's `before_client_reply` failpoint and submits a session write. The leader dies after quorum choice, before it can reply.

Step	Actor	Event and reason
7	Controller	Retries the same session sequence at the new leader.
8	Oracle	The retried write must be applied exactly once.
9	Controller	Rolls the epoch while another member is stopped. The member must rejoin through a snapshot transfer.
10	Controller	Deletes one member's `current.db`. The member must rebuild it from snapshot plus committed suffix.
11	Controller	Kills all three processes without ceremony, then restarts all of them.
12	Oracle	153 rows, the failpoint row exactly once, clean integrity, and identical digests on every member.

Step 6 is the heart of the script. It manufactures the exact ambiguity that sessions exist for: the write was decided, the client never heard so. The retry in step 7 must not apply twice, and step 12 checks that it did not, even after every process has since been killed.

17.3 Persistence assertions, mapped

Chapter 8's crash matrix and the product plan's persistence assertions each name an enforcement point and an oracle. This table is the map between them.

Assertion	Enforced and checked by
Chain equality across members.	The cluster scenario compares chain and content digests on all three members after every recovery arc.
Contiguous monotone apply.	Commit accounting inside the node permits no gap and no reorder in applied slots.
Acknowledged writes survive single-node loss and total restart.	The cluster scenario kills a follower, then the leader, then every process, and counts every acknowledged row afterward.
At-most-once session sequences.	The single-process, CLI, and C ABI suites, plus the cluster failpoint arc in steps 6 to 8 above.
Payload durable before any vote references it.	Transport gating holds value-bearing frames until the payload is stored; the payload-store fault tests cover corruption and loss.
Journal sync before any message leaves.	The ordering is structural: the journal fsync runs before the outbox drains.
Refusal on missing durable state.	The corrupt-journal and missing-payload tests require the node to refuse to serve or vote.
Torn tail truncated, interior corruption fatal.	The journal unit tests and the seeded journal fuzzer discriminate the two cases.
Quorum loss blocks acknowledgement.	The two-members-down drill must refuse writes and fenced reads.

17.4 The three-node transport benchmark

`zig build bench-cluster -- [--sync os|full] <plaintext|psk|tls> [writes] [reads]` runs the harness in `zaxonlite/src/cluster_bench.zig`. It spawns three ReleaseFast `zaxon serve` processes on loopback ports in the named transport and sync mode, waits for a leader,

then drives sequential replicated writes followed by `leader` and `linearizable` reads over one persistent client connection to the leader — the shape of an embedded client, not a pipelined load generator. Every response is verified before it counts; a request the node declines (for example after a leadership move) is retried through the configured endpoint list, and the retry stays in that operation's latency sample. It reports ops/s with p50, p95, p99, and max latency per workload, plus each server process's RSS and CPU delta across the run. The defaults are 1000 writes and 2000 reads. In `tls` mode the harness generates a throwaway CA and per-node `zaxon-node-<id>` certificates with the `openssl` CLI, so the three processes exercise the real mutual-TLS peer and client paths.

The build step always passes `--record benchmarks/results/transport-latest.json`, so every run replaces its mode-and-sync row in that file, and the table below is read from it when this book compiles — the same contract as the rqlite dashboard: the compiled book always shows the last recorded runs, never a number copied into prose. The current write row uses a fixed 256-byte value.

Mode	Sync	Replicated writes	`leader` reads	`linearizable` reads	Node RSS
`plaintext`	`full`	29 ops/s · p50 33310 µs	21925 ops/s · p50 39 µs	12341 ops/s · p50 77 µs	6 MiB
`plaintext`	`os`	350 ops/s · p50 2253 µs	25932 ops/s · p50 31 µs	12730 ops/s · p50 76 µs	6 MiB
`psk`	`full`	28 ops/s · p50 34317 µs	25153 ops/s · p50 31 µs	12298 ops/s · p50 80 µs	6 MiB
`psk`	`os`	348 ops/s · p50 2259 µs	26552 ops/s · p50 31 µs	12045 ops/s · p50 79 µs	6 MiB
`tls`	`full`	63 ops/s · p50 15944 µs	22446 ops/s · p50 34 µs	13078 ops/s · p50 75 µs	22 MiB
`tls`	`os`	609 ops/s · p50 1584 µs	15989 ops/s · p50 72 µs	9311 ops/s · p50 102 µs	11 MiB

3-node loopback cluster, ReleaseFast, sequential single client; indicative numbers from the machine that last ran `zig build bench-cluster – –record ...`. 1000 writes and 2000 reads per configuration; RSS is the largest server process after the workload.

Two lines dominate this table. Across transport modes the write row barely moves, because a sequential replicated write is dominated by quorum fsyncs and the mTLS cost disappears inside them; the read rows show encryption's real price on this host — a few microseconds at p50 and a throughput cost under about 16% — plus roughly 5–6 MiB of additional RSS per node for OpenSSL state. Across **sync** modes the write row moves by more than an order of magnitude: `full` issues one `F_FULLFSYNC` per commit point on macOS — the payload install flushes to the drive and the journal sync is the single drive-cache barrier that makes both power-loss durable — while `os` trusts `fsync(2)` and the drive cache. The write latency under `full` is therefore two barriers overlap: an ordered stream queues `payload_data` immediately before the phase-two accept, the receiver installs it before stepping the accept, and the leader holds its node mutex until its own vote barrier completes. Thus neither a volatile local vote nor an unstored payload can count, while the two devices do useful work concurrently. Reads never touch the journal, so their rows are indifferent to the sync mode. Sustained `full`-mode writes can also stall the leader's tick loop long enough to move leadership — visible as occasional large maxima — which is exactly the behavior an operator should expect from a workload that saturates a full-flush storage budget.

17.5 Measured against rqlite

Benchmarks are the easiest place for a book to lie, so this dashboard states its scope before it shows a number.

TIMED
Client-visible operations against three live voters

CHECKED
Row counts, exact payloads, integrity on every node

EXCLUDED
Real networks, concurrent clients, tuned deployments

Every number in the two tables below is read from recorded result files under `zaxonlite/benchmarks/results/` when the book is built. We never copy a measured number into prose. If prose and table ever disagreed, the table would be right.

17.5.1 One-row durable writes

The first comparison is deliberately narrow. Each system runs alone as three voters on loopback with durable node directories. One client issues single-row autocommit inserts, sequentially, over one persistent connection, with the same payload size for both systems. Warmup writes are excluded from timing. rqlite's queued-write mode stays off, because it acknowledges before durability and that would be a different contract.

This is a product-level comparison, not a consensus microbenchmark. Each system is timed through its own front door. The Zaxonlite driver speaks the framed binary RPC from chapter 12 and follows `not_leader` redirects until it holds a connection to the leader. The rqlite driver speaks HTTP to a node, and that node forwards each write to its leader. Writes reach the leader in both systems, but the path there includes each product's parsing, scheduling, fsync policy, and storage layout. After timing, the driver verifies the data: a strong count and an exact-payload predicate must both equal the number of acknowledged writes. Zaxonlite answers that barrier at level `linearizable`; rqlite answers it at its `strong` level.

Current Zaxonlite transport run plus the recorded rqlite v10.2.7 baseline from 2026-07-20T18:50:40.276035+00:00.

System	writes/s	p50 ms	p95 ms	p99 ms	max ms
Zaxonlite	63	15.94	18.68	24.07	42.03
rqlite	44.5	21.95	28.41	33.69	241.2

1,000 single-row durable autocommit writes, 256 payload bytes each, three loopback voters and one sequential persistent connection. The runs are separate executions on the same host, so treat this as a reproducible baseline rather than a paired statistical trial.

Read the columns in order. The p50 column is the typical cost of one acknowledged durable write through the full product pipeline, and it is the most portable column. The p95 and p99 columns show how wide the tail is. The max column records the single worst write of the run; one scheduler stall can own it, so never quote it alone. The gap between the rows is a product gap. Front ends, durability scheduling, and storage layouts differ by design, and the table cannot attribute its gap to any single one of them. It is not evidence about Paxos versus Raft, and it is not evidence about languages.

The recorded baseline pins the sync contract: both systems flush the drive's cache on every acknowledged write — Zaxonlite's default `full` mode issues `F_FULLFSYNC` on macOS, exactly as Go's file sync does for rqlite. Group fsync already consolidates Zaxonlite's per-write flushes to one barrier per node per commit point (the journal sync; payload installs ride it — see chapter 6). The gap that remains is ordering: Protocol v6 retains the v5 barrier overlap. Only phase-two accept requests are released before the leader barrier; promises, accepted replies, recovered values, commit delivery, and client replies remain behind durable evidence. A commit-only local marker is derived from an already durable accepting quorum and is reconstructed through phase one after a crash instead of forcing a second full barrier. The table combines the current Zaxonlite mTLS/full row with the pinned rqlite v10.2.7 baseline and labels them as separate executions on the same host. A development `--sync os` run on the same machine reverses the ranking at roughly a tenfold lower write latency, but at the price of power-loss durability; chapter 13 states when that trade is acceptable, and chapter 6 states why a consensus voter must not make it silently.

17.5.2 Failure and recovery under a realistic workload

The second comparison stops treating the database as a write loop. It is a deterministic order-processing simulation. Each system runs alone as three voters seeded with 1,000 customers and 500 products, while four concurrent clients spread across all endpoints. The traffic is 70% reads and 30% orders. Reads cover point inventory, customer history, and a sales dashboard. An order is one SQLite transaction whose trigger creates an order line, decrements stock, and records a ledger entry, so every write exercises real relational work.

Two contract details matter. First, every write carries a unique operation ID and uses `insert or ignore`, so a client can safely retry an ambiguous response after a process dies. Second, measured reads use each product's quorum-fence `linearizable` level. rqlite's `strong` mode is reserved for the final verification barrier, because rqlite's read-consistency guide describes that mode as a testing tool which writes through Raft, and recommends `linearizable` for production.

> **Predict before reading on**
>
> The run kills a follower in one phase and the leader in another. Which death costs more throughput, and why? Decide before the table.

After 100 untimed warmup operations, the controller runs three 400-operation phases. Phase one is healthy three-voter traffic. Phase two sends `SIGKILL` to one follower, continues on the quorum, then restarts the follower and waits until its local SQLite copy matches. Phase three sends `SIGKILL` to the leader, begins traffic immediately so the retries ride through the election, then restarts the old leader and waits for its catch-up. Finally the controller kills all three processes, restarts their original identities and data directories, and checks every local copy.

Order-processing phase	Zaxonlite	rqlite
Healthy cluster, mixed reads and writes (ops/s)	1783.4	177.4
One follower down (ops/s)	1073.2	222.3
Leader killed, after failover (ops/s)	387.2	94
Leader crash to first success (ms)	590.6	2190.3
Restarted follower catch-up (ms)	111.5	949.3
Restarted leader catch-up (ms)	337.7	547.9
Full cluster restart to service (ms)	446.2	1626.5

Recorded 2026-07-20 UTC. Both systems finished with 372 orders, identical rows on every node, and clean integrity checks.

Walk the rows top to bottom. The healthy row is each product's mixed throughput ceiling on this host. The follower-down row shows what a quorum can still do with a member missing. The leader-killed row is lower for both systems, and that answers the prediction: a dead follower removes capacity, but a dead leader stalls every client until an election finishes, so its phase carries the retry and election cost inside its throughput. The crash-to-first-success row isolates that outage: it is the gap a client actually experiences between the kill and the next acknowledged operation. The two catch-up rows measure how long a restarted member needs before its local copy matches the cluster again. The full-restart row is the cold path from three dead processes back to service.

The correctness line under the table is the reason the throughput rows are worth reading at all. Both systems finished with the same order count on every node, and the table's footer carries that count. Order lines, ledger entries, inventory, and revenue matched on all three nodes in both systems. No product ever showed negative stock, and every SQLite integrity check came back clean. Zaxonlite additionally passed its chain and payload-store verification. A throughput table without those checks would be noise.

17.5.3 What these numbers are not

> **Read the exclusions before quoting anything**
> Everything above ran on one development host, over loopback, in one recorded run per table, with one client in the write benchmark and four in the simulation, against default configurations and an unreleased `zaxon` build. These numbers are observations of that run. They are not portable claims, not service-latency predictions, and not verdicts about languages or consensus algorithms. Real networks, concurrent client fleets, and tuned deployments are all excluded.

Two omissions are deliberate. The initial rqlite bootstrap time is not reported, because the manual join path sat out its default three-second retry interval; charging a timer as if it were work would be unfair. And the dqlite comparison is deferred entirely. Its harness is committed at `benchmarks/compare-dqlite-3node.sh`, but dqlite is supported on Linux, so no dqlite number appears anywhere in this book until the script has run on a Linux host.

To reproduce, run `benchmarks/compare-rqlite-3node.sh` and `benchmarks/compare-rqlite-realworld-3node.py`. Each writes JSON under `benchmarks/results/`, and rebuilding the book re-renders the dashboard from your run.

17.6 What this release does not verify

Honesty about the suites requires the same honesty about their edges. The engineering headroom tracked in the product plan: pipelined and batched writes under one chosen value, which the descriptor already anticipates with `transaction_count`; group fsync; random fault schedules longer than the deterministic adverse run; automatic voter replacement; and sharding. The 10,000-crash, 100-consecutive-run, and 1-GiB recovery gates are explicitly deferred. The checked large recovery fixture is 1 MiB.

> **Exercise 17.1.** Run `benchmarks/compare-rqlite-3node.sh` on your own machine. Before it finishes, predict which columns of your table will differ most from the recorded one and which will differ least. Then explain why p50 travels between machines better than max does.
> Hint: Medians summarize the whole run; the max column is one event. fsync cost and scheduler noise vary far more across machines than the shape of the pipeline does.

Chapter 18 finishes the job. It leaves suites behind and traces each user-visible guarantee, one row at a time, to the function that enforces it and the oracle that checks it, and it names the guarantees that still have no oracle at all.

18 Conformance: guarantee to evidence

Learning contract

By the end of this chapter you should be able to trace any user-visible guarantee to the function that enforces it, the automated oracle that checks it, and the normative clause that states it, to name the guarantees that still lack an oracle, and to apply the one-change rule when a guarantee moves.

Every guarantee in this book stands on three legs. A normative clause states it. A piece of code enforces it. An automated oracle checks it. Remove any leg and the guarantee wobbles: a clause without code is a wish, code without a clause is an accident, and both without an oracle are a claim. This chapter is the Zaxonlite analogue of the core library's Lamport conformance table. Each row names one guarantee and its three legs. Line numbers drift, so the rows anchor on function names, test names, and scenario steps instead.

Chapter 17 described the suites as layers. Here each row names the specific evidence inside them. Unit tests pin codecs and single-file invariants. `integration_test.zig` drives one real node through restart and recovery. `cluster_test.zig` drives three real `zaxon serve` processes over the public RPC surface with no back door. The role, fault, and gateway suites cover the remaining topologies, and `fuzz.zig` attacks decoders, journal files, and whole-node schedules. In the clause column, "Format §n" names a numbered section of `docs/zds/records/0004-zaxonlite-format.typ`, and "plan" names a section of `docs/zds/records/0002-zaxonlite-product-plan.typ`. Where no automated oracle exists, the row says so directly. An unchecked guarantee is a claim, not evidence, and we would rather you know which rows are which.

18.1 Durability and the write path

Guarantee	Evidence
An acknowledged write is durable and decided. Only a phase-two accept request may precede the sender's local barrier; it carries no claim that the sender's own vote is durable.	**Enforced.** In `protocol.zig`, `preDurableMessages` exposes only accept requests. In `node.zig`, every promise and vote record is appended before the required `journal.sync`; the host mutex prevents an early reply from re-entering the leader until its own vote is durable. Promise evidence, accepted replies, commits, and client replies remain behind `confirmWritesDurable`. In `server.zig`, `runWrite` replies only after the slot is chosen and applied. Every sync routes through `durability.zig`, whose default `full` mode issues `F_FULLFSYNC` on macOS so the flush reaches stable media, not just the drive's volatile cache; the development-only `os` mode keeps plain `fsync` there. **Checked.** The protocol test pins the narrow pre-barrier message class. The `crash_test.zig` cases `after_accept_sync` and the legacy-named `after_commit_sync_before_apply` (now chosen-before-apply) require a recovered count of exactly 1. The cluster scenario's final step asserts every acknowledged write is present exactly once, 153 rows after total restart. The crash campaigns simulate process death, under which the two sync modes are identical; the power-loss durability of

Guarantee	Evidence
	`full` rests on the platform's `F_FULLFSYNC` contract and has no automated oracle. **Clause.** Format §5; plan "Ordering rules", steps 2 to 4; chapter 6's sync policy.
Payload bytes are verified and durable before any value-bearing Promise, Accept, or Commit enters the Paxos transition. Normal Phase 2 avoids an extra storage-ACK round trip without weakening that receiver-side invariant.	**Enforced.** `server.zig` `drainOutbox` queues `payload_data` immediately before the dependent envelope on the same ordered stream. `onPayloadData` verifies and durably stores the object before the reader can consume the following envelope. A separately arriving or fault-reordered envelope is parked by `holdEnvelope` and released only after storage; this fallback also covers Phase-1 recovery. `payload_stored` caches readiness for subsequent sends, and `wire.zig` `envelopePayloadHash` covers promise, accept, and commit alike. **Checked.** The `wire.zig` test "envelope payload hash extraction"; `fault_cluster_test.zig` under loss, duplication, reordering, fragmentation, and delayed durable sync, ending in equal counts; the `payload_store.zig` test "payload store detects corruption and missing objects". **Clause.** Format §3, the receiver-side payload-storage invariant.
A session sequence executes exactly once. An ambiguous retry returns the recorded result and never re-executes.	**Enforced.** `node.zig` `checkSession` and `execIdempotent` replay the last sequence and answer `ResultExpired`, `SequenceGap`, and `UnknownSession` without executing SQL. The session row rides inside the captured transaction. **Checked.** `integration_test.zig` "idempotent sessions execute a sequence exactly once"; the cluster failpoint arc, where the leader dies after quorum choice and the retry at the new leader must apply exactly once. **Clause.** Plan "User-visible guarantees", the timed-out write; format §7.
Session storage stays bounded by activity, not by history.	**Enforced.** `node.zig` `expireSessions`, using the replicated `write_seq` retention window. **Checked.** `integration_test.zig` "idle sessions expire after the retention window". **Clause.** Plan "Bounded client sessions and retry semantics".

18.2 Reads

Guarantee	Evidence
The default read is linearizable. It includes every write acknowledged before it began, across leader changes.	**Enforced.** `server.zig` `opQuery` defaults the level to `linearizable`. `awaitReadFence` confirms the exact ballot with a distinct-member read quorum before querying applied state: `onFenceRequest` answers from the durable promise and `onFenceAck` counts members.

Guarantee	Evidence
	Checked. `cluster_test.zig linearizableCount` after every phase, at 100, 150, and 153 rows. Fence failure paths return `timeout` or `retry`, never stale data. No differential run against the committed barrier exists; see the gaps below. **Clause.** Format §7, the fresh exact-ballot quorum fence; plan "Linearizable read arguments".
Stale reads are opt-in and bounded. `freshness_ms` is legal only at level `any`, and a learner past its bound refuses.	**Enforced.** `server.zig opQuery` rejects `freshness_ms` outside level `any`. On learners it refuses when leader contact exceeds the bound, or when `applied_slot` lags `observed_leader_decided`, which `onLearnerHeartbeat` feeds. **Checked.** `role_cluster_test.zig expectReplica`, where a 2000 ms bound answers, and `expectStale`, where a 100 ms bound returns `"error":"stale"` after the voters stop. **Clause.** Format §7, explicit stale consistency; format §3, `learner_heartbeat`.
A witness serves no SQLite reads.	**Enforced.** `node.zig query` returns `error.RoleCannotRead` when the role's `capabilities().serves_reads` is false, per `roles.zig`. The server maps it to `forbidden`. **Checked.** `role_cluster_test.zig expectCannotRead` expects `"error":"forbidden"` from the witness endpoint. **Clause.** Plan "Node roles and scaling law".

18.3 Crash recovery and storage

Guarantee	Evidence
Recovery is snapshot plus committed-suffix replay. The materialized image is never accepted as newer evidence than Paxos state.	**Enforced.** `node.zig rebuildMaterializedImage` deletes `current.db`, the WAL, and the SHM, copies the verified snapshot generation, then replays the contiguous committed suffix with chain validation. The image is never an input to recovery. **Checked.** `integration_test.zig` "journal is authoritative: materialized image rebuilds from scratch", "a stale materialized image converges to the journal state", "recovery discards a corrupt materialized image even with an empty suffix", and "snapshot seals the epoch and recovery uses snapshot plus suffix"; the cluster step "delete a follower image; node rebuilds from snapshot plus suffix". **Clause.** Format §6; plan "Restart sequence".
Only a torn final record is truncated. Interior journal corruption is fatal, and the node refuses to vote.	**Enforced.** `journal.zig replay` and `parseRecord`: a record that fails validation is recoverable only when `recordTouchesEof`. Any interior magic, version, sequence, or CRC failure is `CorruptJournal`. **Checked.** The `journal.zig` tests "journal truncates a torn tail but keeps the durable prefix" and "journal rejects interior corruption"; `integration_test.zig` "torn journal tail is

Guarantee	Evidence
	truncated and the node reopens"; `fuzz.zig fuzzJournal` with seeded file damage. **Clause.** Format §5, which truncates only an incomplete final record.
No crash point yields a false success. Once the accept or commit prefix is synced, recovery completes the value.	**Enforced.** `failpoint.zig` hooks each pipeline boundary, from `before_payload_sync` through `before_client_reply`, marking the contract the write path must honor. **Checked.** The `crash_test.zig` five-failpoint matrix with per-case recovered-count bounds; `fuzz.zig fuzzNode` with random SQL, crashes, and rebuild convergence. Not every chapter-6 crash row is automated in both roles yet; see the gaps below. **Clause.** Plan "Successful-write durability theorem"; the chapter 6 crash matrix.
A node never serves or votes with missing durable payload bytes.	**Enforced.** `node.zig rebuildMaterializedImage` fails with `PayloadMissing` for a committed descriptor without bytes; `server.zig onLearnerCommit` verifies the store before `learnChosen`; `payload_store.zig verify` re-hashes on demand. **Checked.** The `payload_store.zig` corruption and missing-object tests; the `fuzz.zig` journal and store damage schedules. **Clause.** Plan "Restart sequence", step 4; format §5.

18.4 Identity, membership, and transport

Guarantee	Evidence
A data directory is pinned to one node ID, database, and role. Reuse under another identity is refused.	**Enforced.** `node.zig loadOrCreateIdentity` raises `NodeIdMismatch`, `NodeRoleMismatch`, and `DatabaseMismatch`; `server.zig` startup rejects zero and duplicate registry IDs. **Checked.** `integration_test.zig` "a data directory cannot silently change learner role". The duplicate-ID and zero-ID startup refusals have no automated test today. **Clause.** Format §6: a node ID cannot be reused, and a role mismatch is fatal.
A learner accepts a chosen slot only from a configured voter. Enforcement is layered in the core and in the server.	**Enforced.** The core `src/protocol.zig learnChosen` raises `NotLearner` and membership-checks the certifying sender. The server guard in `zaxonlite/src/server.zig onLearnerCommit` requires the sender's configured role to have `capabilities().votes`; heartbeats pass the same guard in `onLearnerHeartbeat`. **Checked.** `role_cluster_test.zig expectRogueStandbyCommitRejected`: a well-formed `learner_commit`, authenticated as the standby,

Guarantee	Evidence
	is rejected by the read replica. Also the `wire.zig` test "learner commit round trips and rejects invalid certificates". **Clause.** Format §3: a configured voter certifies one chosen slot.
No silent wire-version downgrade. Wrong protocol versions and replayed or tampered optional-PSK frames are refused. Production TCP uses mTLS; the explicit PSK-only development mode is numeric-loopback-only.	**Enforced.** `wire.zig Hello.decode` accepts exactly version 6 and raises `UnsupportedProtocolVersion` otherwise; `server.zig` refuses TCP storage listeners without TLS unless `--dev-psk` is paired with an owner-only secret and all addresses are numeric loopback; `transport_auth.zig Session.readFrame` enforces the exact next sequence with `ReplayDetected` and MAC equality with `AuthenticationFailed`. **Checked.** The `wire.zig` test "hello rejects a future protocol version"; the `transport_auth.zig` test "authenticated session rejects replay and tampering"; the cluster step "reject a client with the wrong transport secret". **Clause.** Format §1 to §3: no negotiation down, exact-major wire. The identity and application-owned authorization boundaries are documented in chapters 7, 12, and 13 and in the security remediation plan.
Every production storage TCP connection is mutual TLS 1.3: both sides present certificates chaining to the cluster CA, and a peer connection's certificate common name must match the node id claimed in its hello, on the accept side and the dial side alike.	**Enforced.** `tls.zig Context.initCommon` sets a TLS 1.3 minimum and verifies with `SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT` in both directions. `server.zig serveConnection` compares an accepted peer hello's node id against the certificate's `zaxon-node-<id>` common name, and the peer dial loop applies the same comparison to the dialed member. A client certificate must chain to the CA; the client additionally refuses a server certificate whose common name is not the canonical `zaxon-node-<positive-id>` form. **Checked.** The `cli_test.zig` mutual-TLS section generates a CA, node, client, and foreign-CA certificate with the `openssl` CLI, then proves RPC round trips over mTLS, refusal of a plaintext client, and refusal of a foreign-CA certificate that carries the right name. `tls.zig` unit tests pin the common-name format and credential refusal. **Clause.** Security remediation plan, SEC-001: per-node transport identity.
A `not_leader` redirect cannot silently change node identity: PSK-only clients stay within their seeds, while mTLS redirects pin the advertised node ID to the target certificate.	**Enforced.** `client.zig callClusterWithTransport` follows unmatched hints only with mTLS, copies and validates the numeric address, and requires the target to present `zaxon-node-<advertised-id>`. PSK-only clients fall back to round-robin over the caller's endpoint list. **Checked.** The CLI suite starts three mTLS voters and proves a leader-only write sent to one follower reaches a leader outside the seed list. Client unit coverage proves the same

Guarantee	Evidence
	hint remains seed-only without mTLS. No automated oracle sends an adversarial hint today; see the gaps below. **Clause.** Security remediation plan, SEC-011: validated client redirects.
One serialized writer per database. Concurrent endpoints never create multi-leader writes.	**Enforced.** The `server.zig` `runWrite` `writer_busy` gate allows one replicated write at a time; `node.zig` `WriteInFlight` and the exclusive directory lock, `tryLock(.exclusive)` raising `error.NodeLocked`, guard the node; non-leaders answer `not_leader`. **Checked.** `integration_test.zig` "a second process cannot open a locked node directory"; the cluster scenario submits writes through all three endpoints and counts every row exactly once. **Clause.** Format §7: one serialized writer per database.
Voter and campaigner bounds hold: at most nine voters, at least one voter, at least one campaigning data voter, and learners never enter the voter set.	**Enforced.** `server.zig` startup validation emits "too many Paxos voters (maximum is 9)", "at least one Paxos voter is required", and "at least one data voter must be able to campaign"; `roles.zig` `capabilities` classifies roles; the core `Membership.init` rejects non-intersecting quorums. **Checked.** The `roles.zig` test "roles keep learners out of the Paxos voter set"; `role_cluster_test.zig` elects only among campaigners. The nine-voter and zero-campaigner refusals themselves have no automated test today. **Clause.** Format §7; plan "Node roles and scaling law".

18.5 Known verification gaps

Chapter 17 listed the suites. These are the holes the suites leave, stated so that this chapter cannot be read as a completeness claim:

- The PSK mode proves only shared-secret possession and is no longer a production TCP mode. Mutual TLS 1.3 supplies per-node identity and confidentiality (SEC-001); mTLS redirect targets are pinned to their advertised node ID while PSK-only clients stay within supplied seeds (SEC-011). A reloaded node-ID denylist tears down active peer links and client connections carrying the same canonical node credential. The CLI test issues a bound one-time token through an authenticated mTLS client, generates and redeems a CSR, checks owner-only installation, uses the new identity for mTLS status, and rejects replay of a copied bundle. Initial CA and issuer credentials remain operator-provisioned. Gateways deliberately pass TLS through to a storage backend and cap raw connections at 128. No automated oracle sends an adversarial redirect hint yet. Zaxonlite intentionally has one application principal rather than database-user/RPC roles.
- Public SQL is now screened by the narrow authorizer in `guard.zig`, which protects the outer transaction, the attached-file boundary, capture-critical pragmas such as `wal_autocheckpoint`, and `__zaxon_*` metadata, with the capture contract re-verified before every payload extraction. The guard's decision table and contract check have unit tests, and a build test asserts `SQLITE_OMIT_LOAD_EXTENSION`. It remains an invariant guard for a trusted application, not a multi-tenant SQL sandbox, and no fuzzing targets it yet.

- Transferred snapshots now carry the canonical existing stop sign, and the receiver requires a matching voter read quorum before validating and activating the image. Static membership is intentional; membership-changing snapshot transfer remains unsupported.
- Connection admission is globally and per-peer capped, handshakes and idle sockets have deadlines, transfers are bounded at 4 GiB, and remote queries have SQL-text, row, result-byte, and SQLite VM-step budgets. Several aggregate recovery-size campaigns and streaming JSON results remain release evidence work. Existing fuzzing is not a full denial-of-service oracle.
- The chapter-6 crash matrix is not fully automated in both the one-node and three-node roles. Completing it is a release blocker.
- The 10,000-crash, 100-consecutive-cluster-run, and 1-GiB recovery gates are explicitly deferred. The checked large fixture is 1 MiB.
- The committed `read_barrier` command is the slow reference read, but no current suite issues it as a differential oracle against the fence path. Fence evidence today is the cluster's linearizable counts plus the safety argument in chapter 6.
- The core Paxos library is model-checked in `specs/Paxos.tla`, but the Zaxonlite layering has no machine-checked specification. The journal, payload store, capture and apply, and session table rest on the plan's proof section plus the oracles above.
- The startup configuration refusals for the nine-voter cap, zero campaigners, and duplicate node IDs are enforced but untested by automation.
- Fault schedules beyond the deterministic adverse run, meaning long random interleavings of loss, reordering, and crashes, remain future work.

When a guarantee changes, edit the normative clause, the code, the oracle, and this table in one reviewable change. That is the same rule the core library applies to its handlers, simulator oracles, and TLA actions, and it is the discipline that keeps this chapter true.

Teach it back. Close the book by teaching its central habit. Pick any row of this chapter and explain it to a colleague three times: once as the promise a user relies on, once as the code path that enforces it, and once as the test that would fail if the code path broke. Then name one row whose oracle is still missing, and describe the test you would write to close it. When you can do that for a guarantee you did not build, you have what this book set out to give you.