Zaxon Working Group
Request for Discussion and Implementation Record

ZDS 0005
Engineering Discussion

A Rich Interactive Shell for the zaxon CLI

STATE
discussion

CREATED
2026-07-21

LAST UPDATED
2026-07-21

INTENDED STATUS
Open for Discussion

AUTHORS
Zaxon Contributors <team@zaxon.local>

DISCUSSION

Plans the rich zaxon interactive shell on the libvaxis terminal library: grapheme-aware cursor line editing, ctrl+r incremental history search, comptime-driven dot-command dispatch and SQL keyword highlighting, width-aware colored tables with expanded and paged views, first-class Windows support, and the extraction of all CLI presentation code out of main.zig.

LABELS
cli, product, engineering

Status of This Memo

This document is an internal Zaxon discussion record authored in Typst and tracked in git. It intentionally follows RFC-style structure so the design scope, rationale, trade-offs, and operational constraints remain explicit. Documents that still use the placeholder number XXXXX are provisional drafts. Numbered ZDS documents are part of the project record.

Contents

1. Abstract 2
2. Introduction 2
3. Terminology and Scope 3
4. Problem Statement 3
5. Goals and Non-Goals 4
   5.1. Goals 4
   5.2. Non-Goals 4
6. Design Overview 4
7. Detailed Design 5
   7.1. The Terminal Layer: libvaxis 5
   7.2. Dependency Policy 6
   7.3. The Line Editor 6
   7.4. SQL Keyword Highlighting 7
   7.5. History and `ctrl+r` Search 7

7.6. Result Presentation 8
7.7. Dot Commands as a Comptime Registry 8
7.8. Module Layout and Context Threading 9
7.9. Signals and Failure Behavior 10
7.10. Style-Guide Compliance 10
7.11. Testing 10
8. Delivery Plan 11
9. Security Considerations 11
10. Operational Considerations 12
11. Alternatives Considered 12
12. Open Questions 12
13. References 13

1. Abstract

The `zaxon sql` shell is currently a bare loop: it prints `zaxon>` , reads a whole line from buffered stdin, and dispatches it. There is no line editing, no command history, no reverse search, and the result tables are unaligned header-width ASCII. Operators who live in `psql` or the `sqlite3` shell notice the gap immediately.

This discussion plans a rich interactive client for `zaxon`: cursor line editing with arrow-key history navigation, `ctrl+r` incremental history search, multi-line statement entry with continuation prompts, SQL keyword highlighting in the input buffer, an expanded set of dot commands, and width-aware colored table rendering with an expanded record view and a pager for large results — on macOS, Linux, BSDs, and Windows alike.

The terminal layer is the libvaxis library, a pure-Zig, MIT-licensed TUI library that tracks Zig 0.16 and handles raw mode, input decoding, grapheme clusters, and platform differences. Everything above it — the line editor, history, highlighter, renderer, and REPL — is Zaxonlite code organized as a `cli/` module family of pure, deterministic state machines, using idiomatic Zig `comptime` facilities for command dispatch, help generation, and keyword tables. The first draft of this record proposed hand-rolling the terminal layer from the standard library alone; review rejected that as misdirected effort, and the reasoning is preserved in Alternatives Considered.

2. Introduction

Zaxonlite ships one binary, `zaxon`, that is both the server (`zaxon serve`) and the operator client. The client side matters: it is how an operator inspects a cluster, runs ad-hoc SQL, and rehearses recovery. Today the interactive experience lags far behind the engineering underneath it.

The current state, from the source:

- `zaxonlite/src/main.zig` is roughly 1,760 lines and contains everything user-facing: usage text, flag parsing, config precedence, both interactive shells (`shell` for embedded mode, `remoteShell` for client mode), every per-command handler, two nearly identical table renderers, and the JSON helpers.
- Both shells read input with `takeDelimiter('\n')` on a buffered `std.Io.Reader`. A typo at the start of a long statement can only be fixed by retyping the line; there is no way to recall the previous statement.
- The package contains no terminal handling at all: no raw mode, no TTY detection, no ANSI escape output, no color, and no column-width alignment (table rules are sized to the header text, not the data).

- The execution plumbing, by contrast, is already well factored: `client.ClusterConnection` handles endpoints, PSK/mTLS transports, leader redirects, and reconnection; `embedded.Embedded` wraps a local in-process node behind the same request shape. Both return the same JSON `columns`/`rows` result form.

That last point is the opening this plan exploits: a rich shell needs no new protocol or storage work. It is purely a terminal input and presentation project, layered on execution backends that already exist. What this revision changes is **how** the terminal problem is solved: by adopting a vetted library for the layer that is genuinely hard and easy to get subtly wrong, and spending Zaxonlite's effort on the parts that are actually ours — editing semantics, history, SQL awareness, and presentation.

3. Terminology and Scope

- **libvaxis**: a pure-Zig terminal library (MIT license) providing raw-mode management, decoded input events, grapheme-cluster segmentation and display width, color and underline styling, and platform terminal setup including the Windows console
- **grapheme cluster**: the user-perceived character unit (an emoji with modifiers, a combining sequence); cursor movement and width math must operate on clusters, not bytes or codepoints
- **line editor**: the state machine that owns the input buffer, cursor position, and repaint computation for the current prompt
- **key event**: a decoded input token delivered by libvaxis (a keypress with text and modifiers), replacing hand-rolled escape-sequence parsing
- **history**: the bounded, ordered list of previously entered statements, navigable with the cursor keys and searchable with `ctrl+r`
- **dot command**: a shell-local command such as `.tables` interpreted by the shell rather than sent as SQL
- **backend**: the execution target of the shell, either `embedded.Embedded` (local database directory) or `client.ClusterConnection` (remote cluster)

In scope: the interactive `zaxon sql` experience in both embedded and client mode on all supported platforms, the extraction of shell and rendering code from `main.zig`, the CLI dependency policy, and the non-interactive fallback used when stdin or stdout is not a terminal. Out of scope: any change to the wire protocol, the request/response JSON schema, the `--json` automation output, server behavior, and the replication core.

4. Problem Statement

Four concrete gaps, in the order an operator meets them:

1. **Input.** No cursor movement, no in-line correction, no kill/yank editing shortcuts, and overlong lines fail with `error.StreamTooLong`.
2. **History.** Nothing typed survives even to the next prompt. There is no up-arrow recall, no `ctrl+r` search, and no history across sessions — painful when rehearsing an operational runbook where the same statements are repeated with small edits.
3. **Presentation.** Result tables do not align columns to the data, do not distinguish `NULL` visually, use no color even on capable terminals, and a thousand-row result scrolls the terminal without a pager. Wide rows wrap into unreadable output, and the input line is monochrome text with no structure.
4. **Structure.** All of this lives in `main.zig`, with the table renderer duplicated for the embedded and remote paths. Growing the shell in place would make an already monolithic file worse and none of it would be testable without a terminal attached.

5. Goals and Non-Goals

5.1. Goals

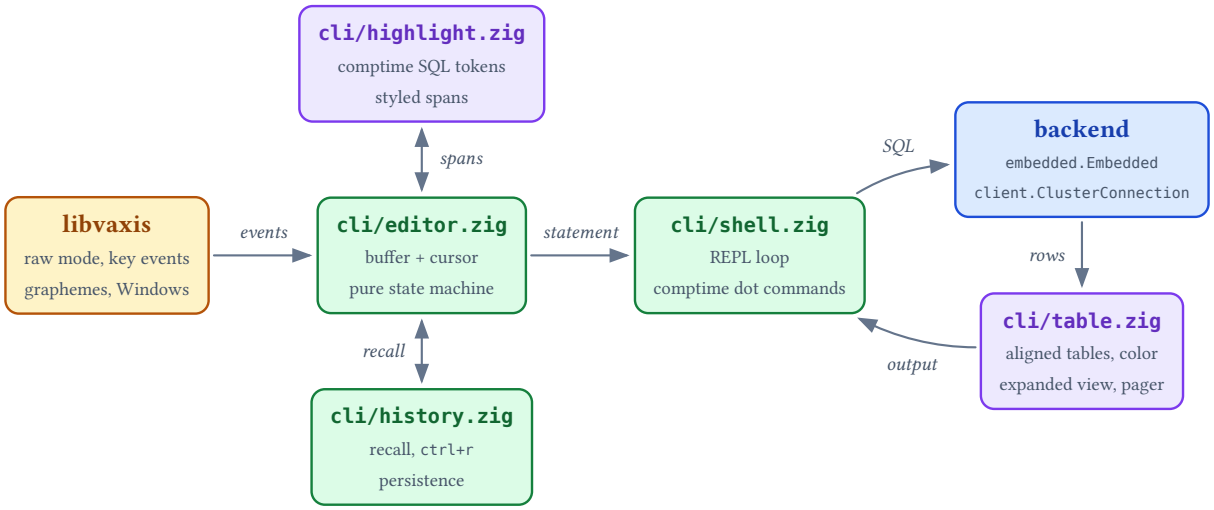
- Line editing comparable to `psql`/`sqlite3` built on GNU-readline-style keys: cursor movement, `ctrl+a`/`ctrl+e`, word movement, `ctrl+w`, `ctrl+u`/`ctrl+k`, `ctrl+l` — correct on grapheme clusters and wide characters, not just ASCII.
- Command history navigable with the up/down arrows, edited recall, `ctrl+r` reverse incremental search, and opt-in persistence across sessions.
- Multi-line SQL entry with a continuation prompt, terminated by `;`.
- SQL keyword highlighting in the input buffer as the operator types, driven by a `comptime`-built keyword table.
- Beautiful, correct tables: data-width alignment, Unicode box drawing on capable terminals, colored headers and dimmed `NULL`s, an expanded per-record mode for wide rows, and a pager for tall results.
- First-class platform support: macOS, Linux, BSDs, **and Windows** get the same rich shell through libvaxis's platform terminal layer.
- Idiomatic modern Zig throughout: explicit `std.process.Init` context threading, tagged unions with exhaustive `switch`, `comptime` registries for dot commands and keywords (`std.StaticStringMap`, `inline for`), and explicit error sets — no `anyerror` in the CLI's public seams.
- Modular code under `zaxonlite/src/cli/`, with the editing, history, highlighting, and rendering state machines pure and deterministic so they are unit-testable without a terminal, per the project style guide.
- Strict fallback: when stdin or stdout is not a TTY, or `TERM=dumb`, the shell behaves exactly like today's plain line reader, so scripts, pipes, and `cli_test.zig` keep working unchanged.

5.2. Non-Goals

- No full-screen curses-style application (no persistent panes, mouse support, or windowing). The alternate screen is used only by the result pager. libvaxis's `vxfw` widget framework is likewise out of scope; the shell uses the low-level API only.
- No SQL autocompletion in the first delivery; the highlighter's tokenizer is designed so completion can build on it in a follow-up discussion.
- No change to `--json`, `exec`, `query`, or any non-interactive command output: automation contracts stay byte-identical.
- No new dependencies in the consensus core or the replication path. The dependency policy change in this record is scoped to the CLI presentation layer alone.

6. Design Overview

The shell is a small pipeline of single-purpose modules. libvaxis owns the terminal: raw mode, event decoding, grapheme width, color, and platform setup. Decoded key events mutate a pure line-editor state; completed statements flow through the existing backends; results are rendered by one shared presentation layer.



Three properties anchor the design:

1. **The library owns the terminal; we own the semantics.** libvaxis solves the problems that burn hand-rolled editors — ambiguous escape sequences and their timeouts, the Kitty keyboard protocol with legacy fallback, bracketed paste, grapheme segmentation and display width, capability detection by terminal query rather than terminfo, and Windows console setup. Zaxonlite writes none of that.
2. **The editor, history, and highlighter are pure.** They never touch a file descriptor; they consume events or text and return values, and the shell loop owns all I/O. This keeps the interesting behavior testable in `zig build test` with no TTY.
3. **One renderer, two backends, comptime registries.** `cli/table.zig` renders the `columns`/`rows` shape both backends already produce, deleting the duplicated `writeTable`/`renderRemoteTable` pair. Dot commands and SQL keywords live in `comptime` tables, so dispatch is compiler-optimized, `.help` is generated rather than maintained, and a command cannot exist without documentation.

7. Detailed Design

7.1. The Terminal Layer: libvaxis

libvaxis (`github.com/rockorager/libvaxis`) is adopted as the CLI's terminal layer. The facts that justify it:

- Pure Zig, MIT license, and its main branch tracks Zig 0.16.0 — the same toolchain this repository pins.
- It does not use terminfo; capabilities (truecolor, Unicode width handling, synchronized output) are detected by querying the terminal itself, which matches real emulator behavior far better than a capability database.
- Input arrives as decoded key events with text and modifiers, via the Kitty keyboard protocol where available and legacy parsing elsewhere, with bracketed paste as an event rather than a byte-stream surprise.
- Grapheme clusters and display width are handled by its vendored Unicode machinery (the `uucode` package), so cursor movement over an emoji or a combining sequence is correct without Zaxonlite shipping Unicode tables.
- Windows is a supported platform: the library performs console setup so the same event loop and styled output work in Windows Terminal, removing the first draft's "POSIX only" carve-out entirely.

The shell uses the low-level `Vaxis/Tty` API: an event loop delivering `Key` and `Resize` events, and styled cell output for the prompt, tables, and pager. The `vxfw` widget framework and its `TextInput` are deliberately not used: the widget is single-line oriented, and the shell needs multi-line statements, history integration, and SQL highlighting — semantics that belong in Zaxonlite's own pure editor over library primitives.

`cli/term.zig` remains as a thin seam wrapping libvaxis initialization, capability decisions (`NO_COLOR`, `--no-color`, `TERM=dumb`), and guaranteed teardown: terminal restoration runs under `defer` on every exit path and is registered with the panic handler so a crashing shell never leaves the operator's terminal in raw mode. In safe builds the shell asserts the terminal was restored before process exit. The seam also keeps every other `cli/` module free of a direct libvaxis import, which bounds the blast radius if the dependency ever has to be replaced.

On entry the shell probes stdin and stdout; the rich path requires both to be TTYs and `TERM` not `dumb`. Otherwise the shell runs the plain loop — the current `takeDelimiter('\n')` behavior, byte for byte — so pipes, redirection, CI, and `cli_test.zig` observe no change.

7.2. Dependency Policy

Zaxonlite's "no dependencies" posture was never absolute — the package already pins the SQLite amalgamation. The precise rule this record establishes:

- The consensus library (`src/`) and the replication path remain dependency-free.
- The CLI presentation layer may take a vetted, pure-Zig, permissively licensed dependency when the alternative is re-implementing a hard, well-solved problem. libvaxis qualifies; a transitively heavy or C-linking package would not.
- Dependencies are declared in `zaxonlite/build.zig.zon` and pinned by content hash to an exact commit, upgraded deliberately and reviewed like any other change. Because libvaxis's tagged releases lag its main branch (main tracks Zig 0.16; the last tag predates it), the pin is a commit hash, and each Zig toolchain bump revisits it.

7.3. The Line Editor

`cli/editor.zig` is the pure line-editor state machine: input buffer, cursor position on a grapheme boundary, and a `feed` function:

```
pub const Action = union(enum) {
    none,
    redraw,
    submit: []const u8,
    cancel,
    eof,
};

pub fn feed(editor: *Editor, key: vaxis.Key) Action { ... }
```

`feed` is a single flat, exhaustive switch — one editing action per arm, in keeping with the style guide's dispatch rule — implementing insertion, backspace/delete, `ctrl+a`, `ctrl+e`, word movement (`alt+b`/`alt+f`), `ctrl+w`, `ctrl+u`, `ctrl+k`, and `ctrl+l`. Paste events insert their text atomically. The buffer is bounded at 64 KiB, matching today's input limit; cursor arithmetic delegates grapheme iteration and width to the terminal layer's Unicode machinery rather than re-deriving it. The editor never prints; it returns an `Action` and exposes what the shell needs to repaint.

Repaint renders the prompt and the highlighted buffer through styled segments, repositioning the cursor by display width — with synchronized output, repainting the line per keystroke is flicker-free and simple to reason about.

Multi-line entry follows `psql`: a statement is submitted when a complete line ends with `;` (outside string literals and comments, using the highlighter's tokenizer states rather than a second scanner) or when the line is a dot command. Until then the shell re-prompts with a continuation prompt:

```
zaxon> select id, body
  ...> from notes
  ...> where id > 10;
```

History stores the full multi-line statement as one entry, with newlines preserved, so recall reproduces the whole statement for editing.

7.4. SQL Keyword Highlighting

`cli/highlight.zig` is a small single-pass SQL tokenizer that classifies the input buffer into spans: keyword, identifier, string, number, comment, operator, and dot-command. The classifier returns spans; the shell maps span kinds to styles, so the tokenizer stays pure and golden-testable as text in, spans out.

The keyword set is a `comptime`-constructed `std.StaticStringMap` over the SQLite keyword list with case-insensitive lookup — built once at compile time, no table maintenance at runtime, and no allocation. The tokenizer is a flat state machine with no recursion and one bounded loop over the buffer; tokenizing even a maximal 64 KiB statement per keystroke is microseconds, so there is no incremental-parse complexity.

The same tokenizer states drive the multi-line continuation decision (a `;` inside a string does not submit) and are the foundation a future completion feature would build on.

7.5. History and `ctrl+r` Search

History is a bounded list (default 1,000 entries) owned by the shell's allocator. Navigation follows readline semantics: pressing up saves the in-progress line, walks backward through entries, and editing a recalled entry edits a copy — the stored history is immutable. Consecutive duplicates are stored once. Entries beginning with a space are not stored, providing the standard escape hatch for sensitive statements.

`ctrl+r` enters reverse incremental search, a sub-mode of the editor:

```
(reverse-i-search)`upd': update members set weight = 2 where node = 3;
```

Each typed rune narrows the search backward from the current position; `ctrl+r` again jumps to the next older match; `enter` accepts the match into the editor; `ESC` or `ctrl+g` cancels back to the untouched line. The search is a plain backward substring scan — at 1,000 bounded entries there is no need for an index.

Persistence is opt-in by default in embedded mode and follows the data directory: `<data>/.zaxon_history`, written with owner-only permissions, loaded on shell start, appended on clean exit. Client mode uses `$ZAXON_HISTORY` if set. A `--no-history` flag and a `.history off` dot command disable persistence entirely; the security section discusses why.

7.6. Result Presentation

The renderer measures every cell by display width, sizes columns to their data, right-aligns numeric columns, and draws with Unicode box characters on capable terminals, falling back to today's ASCII forms otherwise:

```
zaxon> select id, author, body from notes where id < 3;
┌────┬────────┬──────────────────┐
│ id │ author │ body             │
├────┼────────┼──────────────────┤
│  1 │ vik    │ first note       │
│  2 │ NULL   │ replicated note  │
└────┴────────┴──────────────────┘
(2 rows, 1.2 ms)
```

Headers render bold, `NULL` renders dimmed, and the row-count trailer includes elapsed time when `.timer on` is set. Three additional modes are selectable with `.mode`:

- `expanded`: one block per record, `column | value` pairs — the `psql \x` answer to wide rows. `.mode auto` switches to expanded automatically when a table row would exceed the terminal width.
- `json` and `csv`: machine-readable dumps of the same result, useful for copy-paste without leaving the shell.

Results taller than the terminal open in a minimal pager on the alternate screen: arrows and page keys scroll, `q` returns to the prompt, and teardown follows the same guaranteed-restore discipline as the shell itself. The pager renders from the already-materialized result set — `QueryResult` rows are arena-owned and bounded by the existing response limits — so it needs no cursor or streaming protocol.

Every rendered cell is sanitized: C0 control bytes and `ESC` are replaced with visible escapes before printing, in every mode including `csv`. Database contents are untrusted input to the terminal, and this rule is what prevents a hostile row from injecting escape sequences into an operator's session.

7.7. Dot Commands as a Comptime Registry

Dot commands are declared once, in a `comptime` table, with dispatch and documentation derived from it:

```
const DotCommand = struct {
    name: []const u8,
    help: []const u8,
    run: *const fn (shell: *Shell, args: []const u8) Shell.Error!void,
};

const dot_commands = [_]DotCommand{
    .{ .name = "tables", .help = "List user tables.", .run = runTables },
    .{ .name = "schema", .help = "Print CREATE statements.", .run = runSchema },
    // ...
};
```

Lookup goes through a `std.StaticStringMap` built from the table at compile time; `.help` output is generated by an `inline for` over the same table, so a command cannot be added without documentation, and the compiler rejects a registry entry whose handler has the wrong signature. `run` returns the shell's explicit error set — not `anyerror` — so every failure a handler can produce is visible at the seam, per the style guide's error discipline.

The command set grows from three to a `psql`-inspired working set:

Command	Behavior
`.help`	List dot commands (generated) and key bindings.
`.tables`	List user tables (both modes; via a catalog read).
`.schema [name]`	Print `CREATE` statements for a table or all tables.
`.status`	Node or cluster status, as today.
`.members`	Cluster membership (client mode).
`.mode table|expanded|auto|json|csv`	Select the result display mode.
`.timer on|off`	Toggle elapsed-time display.
`.history [off|clear]`	Show, disable, or clear history.
`.quit` / `.exit`	Leave the shell (also `ctrl+d` on an empty line).

Unknown dot commands print a one-line hint listing `.help`, in the diagnostic voice used elsewhere in the CLI.

7.8. Module Layout and Context Threading

New files under `zaxonlite/src/cli/`, each with a single responsibility:

Module	Responsibility
`cli/term.zig`	The libvaxis seam: initialization, TTY probing, capability and color decisions, guaranteed teardown with panic-handler registration. The only module that imports libvaxis directly (the editor sees `vaxis.Key` through this seam's re-export).
`cli/editor.zig`	The pure line-editor state machine described above: buffer, cursor, `feed(key) Action`, repaint data, multi-line accumulation.
`cli/highlight.zig`	The pure SQL tokenizer and comptime keyword table; spans for styling and statement-termination decisions.
`cli/history.zig`	Bounded history with readline navigation semantics, `ctrl+r` incremental search as its own sub-state machine, and optional file persistence.
`cli/table.zig`	The single result renderer: width measurement, alignment, box drawing, styling, sanitization, expanded display, and the pager threshold.
`cli/shell.zig`	The REPL: prompt state, event loop, dot-command registry and dispatch, statement routing to the backend, timing display, and the non-TTY fallback loop. Replaces both `shell` and `remoteShell` in `main.zig`.

The shell receives its world explicitly, in the idiom `main` already uses: `std.process.Init` supplies the allocator and `Io`, and the shell packs its dependencies into one context passed down, never reached for globally:

```
pub const Shell = struct {
    gpa: std.mem.Allocator,
    io: std.Io,
    term: *Term,
    backend: Backend,
    history: History,
```

```
    mode: DisplayMode,
    // ...
};
```

`Backend` is a small tagged union over `*embedded.Embedded` and `*client.ClusterConnection` exposing `execute`, `queryStatus`, and `describe`; both variants exist today and need no behavioral change. `main.zig` keeps argument parsing, config precedence, and command dispatch, and shrinks by everything listed above. This explicit threading is what makes the shell constructible in tests with a fake terminal seam and an in-memory backend.

7.9. Signals and Failure Behavior

`ctrl+c` at the prompt cancels the current line (printing `^C` and a fresh prompt) rather than killing the shell; during a running remote statement it abandons the wait and returns to the prompt, leaving the statement to complete or fail server-side — the shell must communicate this honestly ("canceled locally; the statement may still apply"). `ctrl+d` on an empty line exits cleanly, flushing history. Backend errors keep the existing Elm-style diagnostic blocks from `diagnostic.zig`; the shell renders them and continues rather than exiting, in contrast to one-shot `exec`.

Terminal restoration is the one hard invariant of the terminal layer: every exit path — clean exit, error return, signal-driven unwinding, panic — restores the terminal before the process ends. In safe builds the shell asserts the restore ran.

7.10. Style-Guide Compliance

The CLI is not the consensus core, so the zero-allocation rule does not bind here — the shell uses the GPA it is given. The rules that carry over, and how the design meets them:

- **Bounded everything**: input buffer 64 KiB, history 1,000 entries, pager bounded by the materialized result. Every loop bounds on a capacity or a validated length; no recursion in the editor, tokenizer, or dispatch.
- **Flat, explicit control flow**: key dispatch is one exhaustive `switch` over a tagged union; dot-command and keyword lookup are comptime-built static maps; no callbacks beyond the registry's typed function pointers, and no hidden state machines.
- **Comptime for structure, not cleverness**: `comptime` builds tables and enforces that commands carry documentation and correctly typed handlers. It is not used for code generation that would obscure control flow.
- **Errors versus assertions**: closed terminals, failed backends, and unwritable history files are operating errors with diagnostics; a cursor off a grapheme boundary or an unrestored terminal at exit is an invariant violation and asserts in safe builds. Public seams use explicit error sets, never `anyerror`.
- **Function and line limits**: each editing action, tokenizer state, renderer stage, and dot command is its own function at or below 70 lines.
- **Naming**: names come from the domain — `Key`, `Action`, `Span`, `history`, `expanded` — with no abbreviations beyond established terms.

7.11. Testing

The pure core makes the interesting behavior testable without a terminal:

- **Editor tests**: scripted `vaxis.Key` sequences against `feed`, asserting buffer, cursor, and returned actions — covering word movement, kill/yank, grapheme-boundary movement over emoji and combining sequences, paste insertion, and the multi-line accumulator.
- **Highlighter tests**: text in, expected spans out — keywords in mixed case, nested quotes, comments containing semicolons, and hostile input that must never panic the tokenizer.

- **History tests**: navigation with a preserved in-progress line, duplicate suppression, space-prefix skip, and `ctrl+r` narrowing and cancellation.
- **Renderer golden tests**: fixed result sets rendered in each mode and width, compared to expected strings, including sanitization of hostile cells.
- **Registry tests**: a comptime assertion that every dot command has non-empty help, plus dispatch tests through the static map.
- **Integration**: `cli_test.zig` gains cases that drive `zaxon sql` through a pipe and assert the non-TTY fallback is byte-identical to today's behavior, plus history-file permission checks.

All of it runs under `zig build test` deterministically; nothing spawns a PTY. libvaxis carries its own test suite; Zaxonlite does not re-test the library, it tests the seam. Manual verification on real terminals (macOS Terminal, iTerm2, a Linux console, Windows Terminal, and `TERM=dumb`) is a release-gate checklist item rather than an automated test.

8. Delivery Plan

Five milestones, each landing green with `zig build fmt` and `zig build test`, each usable on its own:

1. **M1 — Extraction.** Create `cli/` and move both shells and both table renderers out of `main.zig` behind the `Backend` seam, with no behavior change and no new dependency yet. `cli_test.zig` passes unmodified; this is the refactor gate.
2. **M2 — Terminal layer and editor.** Add the pinned libvaxis dependency, `term.zig`, and `editor.zig`; raw-mode editing with cursor movement, the non-TTY fallback, and in-memory history with arrow navigation. Verified on macOS, Linux, and Windows Terminal.
3. **M3 — Search and persistence.** `ctrl+r` incremental search, the history file with owner-only permissions, `--no-history`, multi-line statements with the continuation prompt.
4. **M4 — Highlighting and presentation.** `highlight.zig` with the comptime keyword table; the unified renderer: alignment, box drawing, color, `NULL` styling, `.mode` including expanded and auto, sanitization, `.timer`.
5. **M5 — Pager and dot commands.** The alternate-screen pager, the comptime dot-command registry with generated `.help`, `.schema`, `.history`, and the cross-platform manual terminal checklist.

9. Security Considerations

- **Supply chain.** The CLI gains its first external Zig dependency. Controls: libvaxis is pinned by content hash in `build.zig.zon` to a reviewed commit; upgrades are deliberate changes reviewed like code; the dependency is confined behind `cli/term.zig` and never linked into the consensus library, server, or replication path — a compromised terminal library could affect an operator's session, not the cluster's safety argument.
- **History files store SQL.** Operators paste secrets into statements (tokens, credentials, key material in inserts). The history file is written with owner-only permissions, lives inside the data directory whose protection is already an operator responsibility, is disabled by `--no-history` or `.history off`, and honors the leading-space convention for individual statements. The book's operations chapter must document all four.
- **Terminal escape injection.** Query results are untrusted bytes rendered to a terminal. Unsanitized `ESC` sequences in a cell can retitle windows, move the cursor, or (on some emulators) worse. The renderer's mandatory sanitization of C0 bytes and `ESC` in every cell — in every mode, including `csv` — is the control, independent of any styling the terminal layer provides.

- **Raw-mode residue.** A shell that dies without restoring the terminal leaves the operator's console unusable, which in an incident is an availability problem. Restoration is `defer`- and panic-handler-guaranteed on every path and asserted in safe builds.
- **No new attack surface on the wire.** The shell adds no protocol messages, no server code, and no parsing of untrusted input beyond what `query` already returns; transport security (PSK, mTLS, Unix sockets) is unchanged from ZDS 0003.

10. Operational Considerations

- Scripted use is contractually unchanged: non-TTY invocation, `--json`, `exec`, and `query` produce byte-identical output, so existing automation and the conformance suite are unaffected.
- `NO_COLOR`, `--no-color`, and `TERM=dumb` all force plain output; operators on serial consoles or logging terminals lose nothing.
- Each Zig toolchain upgrade now has a companion step: revisit the libvaxis pin. Because the library tracks Zig closely, this is expected to be a same-week bump rather than a blocker, but it belongs on the upgrade checklist next to the SQLite amalgamation.
- The book's CLI chapter (`docs/zaxonlite/book/02_cli.typ`) must be updated at M2 and M4 to describe the shell, and the operations chapter gains the history-file guidance.

11. Alternatives Considered

- **Hand-roll the terminal layer from the standard library (this record's first draft).** Rejected in review. A from-scratch raw-mode driver, escape decoder, and editor must solve ambiguous escape timeouts, grapheme clusters and display width, emulator quirks, and a separate Windows console path — months of terminal-artifact debugging spent rebuilding a solved problem instead of building database features, and a standing maintenance tax afterward. The stdlib-only constraint also made Windows a second-class citizen and left no Unicode tables to do width math with.
- **Link GNU readline or libedit.** Instant editing parity, but a C dependency with GPL implications (readline) and platform variance, and it solves only line editing — none of the rendering. Rejected.
- **Vendor linenoise or a Zig port.** Smaller than readline but still single-line oriented, ASCII-era, and without rendering support; it would cover the least interesting third of the problem. Rejected.
- **Build on libvaxis's `vxfw` framework and `TextInput` widget.** Closer, but the framework targets full-screen widget apps; the shell needs an inline scrolling REPL, multi-line statements, history, and SQL-aware styling that `TextInput` does not model. The low-level API gives the primitives without the framework's control inversion. Rejected for now; re-examinable if `vxfw` matures toward inline apps.
- **Full-screen TUI application.** A persistent-pane database browser is a different product from a shell; it would obsolete none of this work and can be a later discussion layered on the same modules. Deferred.
- **Grow the shell inside `main.zig`.** The zero-refactor path, rejected because it makes the largest file larger, keeps the renderer duplicated, and leaves the new machinery untestable without a TTY.

12. Open Questions

- Pin strategy: track libvaxis main by commit hash (current proposal, since tags lag the Zig 0.16 port) or vendor a snapshot into the repository the way the SQLite amalgamation is vendored? Proposed: hash pin first; revisit vendoring if upstream cadence becomes a problem.

- Should the pager threshold be "taller than the terminal" or opt-in via `.pager on`? Proposed default: automatic, matching `psql`.
- Should client-mode history default to a per-user file (`~/.zaxon_history`) rather than requiring `$ZAXON_HISTORY`? Persisting cluster statements outside the data directory has a different exposure profile; the draft proposes opt-in until discussed.
- Is `ctrl+c` during a remote wait allowed to send a best-effort cancel request in a later protocol revision, rather than only abandoning locally? That would touch the frozen protocol surface and needs ZDS 0004's upgrade rules.
- Does `.schema` in client mode read the schema through the ordinary read path at `linearizable`, or should it use `leader` level to avoid surprising a lagging learner? Proposed: the ordinary read path with the session's current level.

13. References

- libvaxis — `https://github.com/rockorager/libvaxis` — the adopted terminal layer: pure Zig, MIT, Zig 0.16 on main, no terminfo, Kitty keyboard protocol, grapheme-aware, Windows support
- ZDS 0002: Zaxonlite: Product and Delivery Plan — the CLI as the operator surface of the embedded cluster
- ZDS 0003: Zaxonlite Security and Trust Plan — trust boundaries the shell must not weaken
- ZDS 0004: Zaxonlite Format and Compatibility Contract — the frozen protocol this design leaves untouched
- `zaxonlite/src/main.zig`, `client.zig`, `embedded.zig` — the current shells and the execution backends this plan reuses
- `docs/book/04_style.typ` — the style guide the module and `comptime` design follows
- GNU Readline and the `psql`, `sqlite3`, `pgcli`/`mycli` shells — the interaction vocabulary this design adopts