ChatGPT

ccmux Deep Research

1. Terminal Emulation in Rust

Executive Summary

Rust offers robust libraries for terminal emulation. For **PTY management**, the `portable-pty` crate provides a cross-platform API to spawn processes in a pseudoterminal, giving the spawned process a real TTY (so it thinks it's in a terminal) [1] [2]. To **parse output**, crates like `vt100` (or `alacritty_terminal` / `vte`) can interpret ANSI escape sequences and maintain an in-memory screen buffer. The parsed screen can then be rendered with text UI libraries like Ratatui. **Zellij** (Rust multiplexer) and **WezTerm** (Rust terminal emulator) serve as valuable architectural references: Zellij uses Rust crates (like `portable-pty` and a terminal parser) to multiplex terminals, while WezTerm implements its own terminal parser and uses `portable-pty` internally [3]. A **Recommended approach** for ccmux is to leverage `portable-pty` for PTYs and use a terminal parser (e.g. `vt100` crate) to maintain each pane's state, rendering via Ratatui widgets. Key considerations include handling window resizing signals, supporting advanced terminal features (colors, mouse, etc.), and achieving efficient rendering for multiple 60fps panes.

Detailed Analysis

PTY Spawning and Management:
Rust's `portable-pty` crate (part of WezTerm's code) simplifies working with pseudoterminals across Unix and Windows. It creates a *pty pair* with a *master* (controller) end and a *slave* (controlled) end [4] [5]. The master is used by our program (ccmux) to read/write, and the slave is handed to the child process as its stdio. By spawning a shell or process on the slave side, the child process believes it's attached to an interactive terminal (so `isatty` is true) [6] [2]. This triggers full-screen behavior, colored output, etc., which wouldn't happen with a normal pipe. Using `portable-pty` is straightforward: we call `native_pty_system().openpty(size)` to get a `PtyPair` (with `master` and `slave`), then use `slave.spawn_command(cmd)` to launch a shell or any program [7]. We then read the program's output via `master`: typically by calling `try_clone_reader()` to get a `Read` handle for async reads [8] [9]. The `master` can also be written to (via `master.take_writer()` or directly implementing `Write`) to send keystrokes to the child [10]. The terms *master* and *slave* are legacy; conceptually it's a controller and a controlled end [11].

One important aspect is **window resizing**. In a real terminal, when the terminal window size changes, the kernel informs the foreground process via `SIGWINCH`. With `portable-pty`, ccmux must handle this explicitly by calling the `resize()` method on the master end whenever a pane's dimensions change [12]. This updates the kernel's notion of the window size and sends the appropriate signal to the child process [12]. For example, if the user resizes the ccmux UI or adjusts a pane split, ccmux should call `master.resize(PtySize{rows, cols, ...})` for that pane's pty. This ensures programs like `vim` or `less` redraw correctly to the new size.

Terminal State Parsing:
Raw output from the PTY includes not just text but also **ANSI escape sequences** (e.g. cursor movements, color codes). To present each pane's content correctly, ccmux needs to parse these sequences and maintain a model of the terminal screen. The `vt100` crate is a proven solution: it implements a VT100/ANSI parser and provides a `Parser` and `Screen` to represent state [13] [14]. As the program outputs bytes, feeding them into `vt100::Parser.process()` updates the `Screen` state, which tracks the grid of cells, cursor position, scrollback buffer, current text attributes, etc. The `Screen` can differentiate between the normal buffer and the **alternate screen buffer** used by fullscreen programs. (VT100-compatible terminals have two buffers: the main scrollback buffer and an alternate screen with no scrollback, typically used by programs like `vim` or `less`. The parser will handle switching buffers when it sees the appropriate escape codes, e.g. `ESC [ ? 1049 h` for alt screen enable.) The `vt100` screen internally stores the text grid for each buffer and the cursor state, so after parsing, ccmux can query it for what to display [14] [15]. Data structures typically involve a 2D array of `Cell` structs (each holding a character, foreground/background color, and flags for bold, underline, etc.), plus structures for lines and metadata. The crate supports specifying an initial scrollback size when creating the parser (for example, `Parser::new(rows, cols, scrollback_lines)`) – this defines how much history to keep beyond the visible screen.

Another option is `alacritty_terminal`, the core of Alacritty terminal emulator. It similarly provides a parser (using the `vte` crate under the hood) and grid model. Alacritty's `Grid` is highly optimized in Rust for performance, and it supports advanced features (deep color, etc.). However, `alacritty_terminal` is a heavier dependency and not as straightforward to integrate for a TUI. In contrast, `vt100` is standalone and designed for embedders like screen or tmux [16]. The **trade-offs**: `vt100` is easier to use (just feed bytes, get a screen), but it may not implement *every* quirk of modern terminals (though it does handle at least 16 colors and likely 256 colors and truecolor via SGR codes, given it has a `Color::Idx` and presumably an RGB variant [17]). `alacritty_terminal` might handle complex features (e.g. sixel graphics, nuanced Unicode widths, etc.) but it's more complex to set up and not officially stable as a library. Many Rust projects have chosen `vt100` for terminal emulation (for instance, Zellij used a fork of `vt100` in the past [18], and the `cockpit` multiplexer crate uses `vt100` [19]).

Rendering through Ratatui:
Once we have the terminal state (grid of characters and attributes), we need to render it inside ccmux's text-based UI. **Ratatui** (a modern fork of tui-rs) can render text with styles efficiently to the terminal. The typical approach is to create a custom Ratatui widget that iterates over the `Screen` cells and paints them. An example is the `PseudoTerminal` widget (from the `tui-term` crate) which wraps a `vt100::Screen` and draws it in a Ratatui `Block` [20] [21]. Essentially, for each line and column in the visible area, it fetches the character and styling and adds it to Ratatui's buffer. One can use `Span` or `Text` with styling for colored text. Performance-wise, drawing an 80x24 area (1920 cells) dozens of times per second is feasible in Rust, especially since Ratatui uses efficient diffing to only output changes to the real terminal [22]. For multiple panes, ccmux would maintain separate `Screen` states and render each in its region of the UI. Achieving 60fps should be possible if we only re-render diffs and avoid unnecessary work; however, 10 panes of 80x24 each at 60fps is ~460k cell writes/sec in the worst case (if all panes update fully every frame), which is high. In practice, many frames will have minimal changes. Ratatui's double-buffering will help ensure only changed portions are written to the actual terminal [22].

A challenge is **terminal-in-terminal**: ccmux itself runs in a terminal and is drawing what are essentially *terminal emulator outputs* inside it. We must ensure that escape sequences from inner programs do not inadvertently affect ccmux's real terminal. The solution is that ccmux's parser consumes all escape sequences and translates them into state – we never pass the raw escape codes to the outer terminal. For example, if `ls --color` in a pane outputs `ESC[31m` (red text), the vt100 parser will interpret that and mark subsequent text as red in the `Screen`. When rendering to the real terminal, ccmux will use Ratatui to set the appropriate color (e.g. using ANSI on the outer terminal) just for that pane's text. So the outer terminal only sees ccmux's drawing, not the inner escapes. This isolation is critical; it is how tmux/screen operate: they act as terminal emulators that **interpret** inner escapes rather than blindly outputting them [23]. Some sequences, like OSC (Operating System Command) codes that set window titles or clipboard, might be tricky – by default, `vt100` (and similar parsers) will treat unknown sequences as no-ops or expose them via callbacks. We likely want to **suppress or handle** certain global sequences (e.g. tmux filters out or captures OSC 0 (title setting) so it doesn't change the tmux window title unexpectedly). We can decide to ignore such sequences or implement handling (perhaps reflect them in ccmux's UI, e.g. show pane titles).

Terminal Capabilities and TERM:
Child processes rely on the `TERM` environment variable to know what control sequences to use. We should set `TERM` for spawned processes to something standard that our parser supports well, e.g. `xterm-256color`. `portable-pty` by default doesn't set `TERM`, so it inherits ccmux's environment; we can explicitly set it via `CommandBuilder.env()` if needed [1]. Claiming to be `xterm-256color` is reasonable if our parser handles 256-color SGR codes and basic xterm controls (which `vt100` does to a large extent). We might not support *every* advanced feature out of the box (like mouse tracking, or newer protocols like Kitty graphics), but those can often be negotiated. For example, **mouse support**: if a program enables mouse (e.g. with `ESC [ ? 1002 h`), the parser will note that mouse mode is on. We would then need to capture mouse events from the real terminal (Ratatui can provide those events) and encode them to the program via the pty (translating clicks into escape sequences). This is a feature to consider for later – but importantly, we shouldn't break such sequences. Similar for **bracketed paste** (an escape that tells the terminal to wrap pasted text in special sequences) – ccmux can either implement it or pretend it's not supported. Initially, we might keep it simple.

Advanced features: True-color (24-bit) is common now (via `ESC[38;2;<r>;<g>;<b>m`). If `TERM=xterm-256color`, programs might limit to 256 colors, but many still emit truecolor codes regardless. We should ensure the parser can handle those (likely it would either map to nearest color or represent them as an RGB in its `Color` struct). The `vt100::Color` enum likely has support or can be extended. The `alacritty_terminal` parser definitely supports truecolor. For **keyboard protocols**: normally, terminals have certain default key codes. For example, the "Kitty keyboard protocol" extends how terminals send key events for modifiers; since ccmux reads actual key presses via Ratatui, we can decide what sequences to generate for the child. Likely, sending standard ESC-based codes (as defined by terminfo for our advertised TERM) is sufficient.

Zellij's Terminal Emulation Architecture:
Zellij (a Rust terminal multiplexer) can inform our design. According to community discussion, Zellij uses `portable-pty` for PTYs and originally used a fork of the `vt100` crate to parse output [18]. Zellij's GitHub shows components for terminal handling and a grid. They also handle features like graphic protocols (they have a sixel parser crate) [24]. So Zellij likely maintains a state per pane similar to tmux (with scrollback) and draws it in its UI. They chose to implement their own grid logic (likely for performance or specific features) but using `vt100` as a starting point. For ccmux, reusing `vt100` directly is fine for MVP. We should also note Zellij has a separate **server** process that runs the PTYs and UI, and plugins in WASM; this influences crash handling and is discussed later.

WezTerm's Approach:
WezTerm (a modern terminal emulator in Rust) doesn't use `vt100` but has its own `term` library for parsing and maintaining terminal state [3]. That library isn't published, but WezTerm's architecture is instructive: it uses `portable-pty` for PTYs, a parser that produces a `Model` of the terminal, and then renders to a GUI or text. WezTerm's author confirms that to embed a terminal, one essentially needs to run a full parser and update a model (there's no shortcut to convert bytes to a rendered surface without the emulation logic) [3]. WezTerm also supports advanced features like hyperlink and font fallback, showing that a comprehensive emulator can get complex. For ccmux, we likely don't need all that; focusing on correct text and color rendering is the priority.

Integration with Ratatui:
There are existing examples of integrating terminal emulation in TUIs. The `cockpit` crate (essentially a library implementing a mini tmux) uses `portable-pty` + `vt100` + Ratatui. It provides a `PaneWidget` that draws a pane's screen [19] [25]. We can draw inspiration (or even code) from it. Similarly, the `tui-term` crate's `PseudoTerminal` widget demonstrates how to wire up a vt100 `Screen` with Ratatui [20]. Typically, you will iterate each line, construct a `Spans` with styled `Span` segments, and render those in a `Paragraph` or custom widget area. Performance can be improved by only re-rendering lines that changed (the `contents_diff()` method in vt100 can give diffs [26]), but Ratatui's diffing might suffice at first.

Recommended Approach (for ccmux)

For **ccmux**, we recommend using `portable-pty` for spawning and managing PTYs due to its ease of use and cross-platform support (Unix and Windows). This lets ccmux open a PTY and spawn each Claude or shell process inside it so that it behaves like in a normal terminal (enabling colored output, interactive prompts, etc.) [6] [2]. Manage the PTYs in an async fashion (e.g. using Tokio to read the output without blocking the UI loop), possibly by spawning a dedicated task per pane to read from the PTY and push data into that pane's parser.

For parsing output, leverage the `vt100` **crate**. Initialize a `Parser` with a scrollback capacity (perhaps configurable) for each pane. As data arrives from a PTY, feed it into the parser. Use the parser's `Screen` to get the current text buffer for rendering. This approach isolates each pane's terminal state nicely. It also simplifies detecting special states (e.g. if we need to know if a program is in alt-screen mode or has bell/ring, the `Screen` or callbacks could tell us). If needed, implement the `Callbacks` trait from `vt100` to catch things like OSC codes or BEL characters (for potential UI integration like flashing tab on bell).

To render, implement a Ratatui **widget** (e.g. `TerminalPaneWidget`) that takes a reference to a `vt100::Screen`. In its `render` method, iterate over each row and column within the viewport (the Ratatui `Rect` given for that pane) and draw characters with proper styling. You can map `vt100::Color` to Ratatui `Style` (perhaps via the `Color` enum in Ratatui). This widget will be used in the TUI layout for each pane. Use double buffering (Ratatui does this by default) to only output diffs to the real terminal, which helps performance [22].

Implement **window resizing** by catching terminal resize events (Ratatui passes terminal resize events via the `tui::events::Event::Resize`) or when layout changes (if user adjusts pane sizes). On a resize, call `master.resize()` on the corresponding PTY with the new rows/cols for that pane [12]. Also update the `vt100::Parser` dimensions using `Parser.set_size()` (if available) or by creating a new parser and transferring state. The `vt100` crate may allow directly resizing the screen; otherwise, we might reinitialize it. It's important to keep the PTY and the parser in sync size-wise to avoid broken layouts.

Set the environment variable `TERM` for spawned processes to `"xterm-256color"` (or perhaps `"ansi"` if we want to limit, but xterm-256color is standard). This ensures programs use common ANSI sequences that our parser can handle. We should also set `COLORTERM=truecolor` if we intend to support 24-bit color (and confirm the parser supports it). Additionally, propagate any needed env vars (like API keys, etc., specifically for Claude processes).

Start with no mouse support (i.e. do nothing special for mouse events). We can revisit this later. Similarly, initially ignore bracketed paste (just let paste text through as keystrokes). We can add support incrementally by intercepting those sequences via the parser's callback if they become needed.

Alternate strategy: As an alternative to writing a custom widget, we could consider using the existing `tui-term` crate's `PseudoTerminal::new(screen)` widget. However, depending on how actively maintained it is, a custom widget might give us more control (and ability to highlight the active pane, etc.).

Keep in mind potential **pitfalls**: (1) **Performance** – dumping thousands of cells to the screen can be slow; mitigate by diffing and by reducing frequency of full redraws (only redraw at 60fps if content is actually updating, otherwise we can idle). (2) **Unicode width** – ensure the parser handles multi-column characters (CJK, emojis) correctly so the alignment isn't off. `vt100` should handle standard Unicode width rules. (3) **Scrolling** – if our UI uses scrollback (user scrolls up in a pane), we need to decide how to handle that separate from the program's output. Tmux, for example, captures output in a buffer and allows viewing it after the program finished. We might add a mode for that later. Initially, we can let scrollback be purely via the program (i.e. use the program's output for scrollback, like relying on `less` or `tmux capture-pane`). The `vt100` parser will hold scrollback internally which we could leverage for a scrollback viewer feature in ccmux.

Finally, use Zellij and WezTerm as references but avoid over-engineering prematurely. A minimalist approach using `portable-pty` + `vt100` + Ratatui gets us a working terminal multiplexer tailored for Claude without needing to implement a full terminal from scratch (which is *"incredibly tedious and difficult"* as one Rust developer noted [27]).

Code Examples

Spawning a process in a PTY (Rust):

```rust
use portable_pty::{CommandBuilder, PtySize, native_pty_system};
use std::io::Read;

let pty_system = native_pty_system();
let size = PtySize { rows: 24, cols: 80, pixel_width: 0, pixel_height: 0 };
let mut pty_pair = pty_system.openpty(size).expect("Failed to open PTY");

// Split into master and slave
let mut master = pty_pair.master;
let slave = pty_pair.slave;

// Prepare command (e.g., spawn Claude or a shell)
let mut cmd = CommandBuilder::new("bash");  // or "claude"
cmd.env("TERM", "xterm-256color");
cmd.cwd(std::env::current_dir().unwrap());
let child = slave.spawn_command(cmd).expect("Failed to spawn process");

// Read output asynchronously (example using a thread)
let mut reader = master.try_clone_reader().expect("clone PTY reader failed");
std::thread::spawn(move || {
    let mut buffer = [0u8; 8192];
    let mut parser = vt100::Parser::new(24, 80, 1000);
    loop {
        match reader.read(&mut buffer) {
            Ok(0) => break, // EOF
            Ok(n) => {
                parser.process(&buffer[..n]);
                // Here, update UI with parser.screen(). We would notify the
                // main thread that new output is available (e.g., via a
channel).
            }
            Err(e) => { eprintln!("Read error: {:?}", e); break; }
        }
    }
});
```

In a real implementation, instead of a thread we'd likely use Tokio and an async loop to read, then send an event to the UI to redraw the pane.

Rendering a vt100 Screen with Ratatui:

```
use ratatui::widgets::Widget;
use ratatui::buffer::Buffer;
use ratatui::layout::Rect;
use vt100::Screen;

pub struct TerminalPane<'a> {
    screen: &'a Screen,
}

impl<'a> Widget for TerminalPane<'a> {
    fn render(self, area: Rect, buf: &mut Buffer) {
        // Ensure we render only within the provided area
        let max_rows = std::cmp::min(area.height as usize, self.screen.rows());
        let max_cols = std::cmp::min(area.width as usize, self.screen.cols());
        for row in 0..max_rows {
            for col in 0..max_cols {
                if let Some(cell) = self.screen.cell(row, col) {
                    let ch = cell.contents(); // char at this cell
                    // Map vt100 Color to Ratatui color
                    let fg = map_color(cell.fgcolor());
                    let bg = map_color(cell.bgcolor());
                    let styled = buf.get_mut(area.x + col as u16, area.y + row
as u16);
                    styled.set_symbol(&ch.to_string());
                    styled.set_fg(fg);
                    styled.set_bg(bg);
                    if cell.bold() {
styled.set_modifier(ratatui::style::Modifier::BOLD); }
                    // ...handle underline, reversed, etc.
                }
            }
        }
    }
}
```

This is a simplified example – in practice, one must handle multi-width characters and only setting cells that changed (for efficiency). Also, `map_color` would translate a `vt100::Color` (which might be an index or RGB) to a `ratatui::style::Color`.

Handling resize:

```
// Suppose new_rows, new_cols are the updated size for pane i
master.resize(PtySize { rows: new_rows, cols: new_cols, pixel_width: 0,
pixel_height: 0 })
      .expect("PTY resize failed");
parser.set_size(new_rows, new_cols); // hypothetically adjust parser screen size
```

(Error handling omitted for brevity.)

References

- Rust `portable-pty` usage and PTY architecture [5] [2]
- Master/slave PTY concept (controller vs controlled) [4] [11]
- Example code using `portable-pty` to spawn a shell [7] [10]
- Terminal emulator parsing with `vt100` crate [13] [14]
- Reddit discussion recommending `portable-pty` + `vt100` for building a tmux-like tool [18]
- Ratatui `PseudoTerminal` example (embedding a vt100 Screen) [20]
- `cockpit` crate features (PTY, vt100, Ratatui integration) [19] [25]
- `MasterPty.resize()` sends SIGWINCH to child on size change [12]

- WezTerm maintainer on using an internal terminal parser for rendering [3]
- Performance note: Ratatui diff-rendering only changes [22]
- Zellij vs tmux context (Rust terminal emulator complexity) [27]

Open Questions

- `vt100` **vs** `alacritty_terminal`**:** Is `vt100` sufficient for Claude Code's output? It should handle typical text and colors. If we encounter unsupported sequences or truecolor issues, we might consider switching to `alacritty_terminal` or patching `vt100`. This requires testing with Claude's actual output (especially if it uses any unusual control sequences or Unicode).
- **Scrollback management:** How to expose scrollback to the user? `vt100` accumulates scrollback internally. We could implement a copy-mode like tmux to let users scroll. Alternatively, rely on Claude's own history tools. This isn't critical for MVP but is a feature to plan (especially if user wants to scroll up to see older Claude responses).
- **Mouse and other input:** Will ccmux need to support mouse interactions with Claude Code (for example, if Claude's output includes clickable links or if using text editors in panes)? If yes, we'll have to capture mouse events and forward as escapes, and also handle terminal focus (ensuring only one pane receives keyboard input at a time). This is doable (tmux and Zellij do it) but adds complexity. It might not be needed for Claude specifically, but if we allow general processes, it could be.
- **Alternate screen indicator:** Should we indicate in the UI if a pane is in an alternate screen (like vim)? Possibly not needed, but we might want to ensure that when alt-screen is active, we don't confuse it with normal scrollback.
- **Unicode and Font:** If Claude outputs fancy Unicode (emojis, non-monospace symbols), does `vt100` handle them correctly in terms of width? We should verify with test cases. Some emojis are double-width and could disrupt alignment if not accounted for.
- **Performance tuning:** If we find that rendering many panes at high FPS is taxing, we might implement optimizations: e.g. only redraw a pane when its content changed (we can track if any new output arrived or if cursor moved). Also, we might lower the frame rate for background panes (no need to render at 60fps if nothing's happening). These are refinements to explore.
- **Testing:** To ensure reliability, we should test the terminal emulation with a variety of programs (simple ones like `ls`, text editors, CLI tools) to catch any parsing issues. Also test resizing and ensure no crashes or mis-painted screens (e.g., after many lines of output or after toggling alt-screen). Zellij's and WezTerm's test cases (if available) might guide edge cases.

---

2. Claude Code Internals

Executive Summary

Claude Code is Anthropic's CLI tool for "agentic coding." It runs as a local CLI (installed via Node/NPM) that communicates with the Claude API. Key internal behaviors include maintaining a **session state** (conversation history, tool use, etc.) which is persistently stored in a JSON file under `~/.claude.json` [28], updated frequently. Claude Code can operate in an interactive REPL or one-shot "print" mode with flags for structured outputs (JSON) [29]. Different **states** like "thinking" (Claude processing), "waiting" (for user input or tool output), "complete" (task done), or "crashed" are reflected in the CLI's output or behavior. The CLI likely indicates these states via text spinners or status lines (e.g. a *"Thinking…"* message or animation), though precise patterns need empirical checking. Claude Code supports resuming sessions by ID or name using `--resume/-r` [30] [31] – under the hood this links to the stored session state (each session has a unique ID, often a UUID [32]). Session data (conversation, tool states) is saved continuously to avoid loss [28]. The `~/.claude/` directory is used for configs (settings.json), session logs (debug logs per session), and maybe other data. The CLI has no documented IPC or plugin API beyond its normal inputs, but it provides options like `--output-format JSON` for programmatic use [33]. **Crash recovery**: if Claude Code or ccmux crashes, you can restart Claude Code with `--resume <session_id>` to continue where you left off, as long as you have the session ID (which is stored in the state file) [30]. There isn't an exposed socket or API for live control; communication is via the terminal/stdio. Environment variables like `ANTHROPIC_API_KEY` are used for auth, and others (e.g. `CLAUDE_*`) toggle features [34] [35].

Detailed Analysis

State Detection (Thinking/Waiting/Complete):
Claude Code, being an interactive CLI, likely provides visual cues for the AI's state. For example, when you ask Claude a question, it might print a message like "Claude is thinking..." or show an animated spinner. Since it's a Node.js app, it could use something like a carriage return to update a status line (like "Thinking |" then "Thinking /", etc.). We need to confirm the exact behavior. One approach is to run `claude` and observe the output in various scenarios. The prompt specifically asks if there are "visual patterns" or specific Unicode that indicate thinking. It's possible Claude Code prints an ellipsis or a "⌛" icon while waiting. Another hint: there is a beta feature called `--betas interleaved-thinking` [36], which suggests they have an "interleaved thinking" mode (maybe streaming partial thoughts). Also, the CLI offers a `--include-partial-messages` flag with streaming JSON output [37], indicating it can output partial results. In normal mode, though, likely the CLI prints nothing until the answer is ready, or prints a spinner.

Claude Code might also update the terminal title or use some ANSI to indicate state (though less likely). The question asks if Claude uses any terminal capabilities like setting the window title or such. It's known that some CLI tools (like some LLM clients) update the title or use OSC codes to show status, but I haven't seen explicit mention for Claude Code. Given it is interactive and might spawn background processes (tools), they might show a status line with current action.

To solidly detect states, one could use a tool like `script` or `ttyrec` to record output. The "thinking" phase likely involves either a spinner or simply a delay with no output. There might be a subtle pattern: e.g. maybe the cursor is hidden during thinking and then shown on completion, or maybe a specific line is printed and erased. Without direct observation, we hypothesize: **Thinking** – possibly indicated by a spinner animation in place (e.g. sequence of `|/-\` on one line via carriage return). **Waiting** – if Claude is waiting for user input or tool permission, it might print a prompt or a question (like `Allow tool execution? (y/n)`). **Complete** – likely just when the assistant has finished printing the answer, perhaps the REPL returns to a prompt (maybe a `>` prompt or similar). **Crashed** – if Claude Code itself crashes (due to an error or losing API connection), it might print an error stack or message. ccmux could detect that by the process exiting with a non-zero code or specific error text.

Claude Code also has a concept of **permissions and tools** (the CLI can ask for permission before running, say, `git` commands). In those moments, the CLI might print something like "[Permission required for Git commit – allow?]" and wait. ccmux could interpret that as a waiting state (the model paused for user confirmation).

In summary, to detect states robustly, ccmux could listen for certain patterns: for example, the presence of an interactive prompt, or lack of newlines (if spinner uses carriage returns), or explicitly parse Claude's JSON output (if using `--output-format stream-json`, you'd get events that include states). The CLI's `--output-format` flag is important: In *print mode*, `--output-format json` yields a final JSON result after completion, and `--output-format stream-json` yields a stream of JSON events including possibly a `status: thinking` vs `status: complete` markers [38]. This suggests a strategy: we can run Claude in print mode with stream-json output (and maybe `--include-partial-messages`) when orchestrating via ccmux, which would give us machine-readable state info (like partial messages and a final message). However, if ccmux is attaching to an interactive session, it won't have JSON; we'd parse the text UI.

From user experience accounts, when running `claude` interactively, it starts a REPL where you type queries. It might not explicitly say "Claude is thinking" but likely shows a blank line with a blinking cursor or spinner until Claude's answer streams in. Some CLI cheat sheets or guides don't mention spinners, so maybe it just streams the answer text token by token (like how `chatGPT` CLI tools do). If it streams output gradually, we might detect the stream vs when it stops. If it doesn't stream but prints all at once, then the "thinking" period is just silence on the terminal.

Output Structure & Hidden Data:
Claude Code's output is primarily natural language or code that Claude writes. However, the CLI does support structured output. The CLI reference shows a flag `--json-schema` to enforce outputs matching a JSON schema [39]. In such a mode, Claude will output a machine-readable JSON on completion. Also, `--output-format` can be "json" or "stream-json" [33] which means the CLI will output either a single JSON object at the end or a stream of JSON events (likely including things like `{"type": "thought", "content": "..."} as it thinks, followed by final answer`). This indicates Claude Code can output **structured JSON events** for integration – essentially an API mode. In interactive mode, by default it outputs just text to the terminal, but we could leverage these flags in ccmux to intercept structured data. For example, ccmux could run each Claude instance with `-p --output-format stream-json` and get a stream of JSON that indicates states and final result, which is easier to parse than raw ANSI. But doing so might sacrifice the rich interactive features (like tools usage prompts). It's a trade-off. Possibly we run interactive mode for full features, but then we parse text and spinner.

Are there hidden control sequences or markers in the output? Unlikely in text mode (they want it human-readable). But in JSON mode, they may include additional fields like `session_id`, or tool action logs. The `--include-partial-messages` suggests that partial content is streamed with certain markers in JSON mode [37]. That could correlate with a spinner in text mode.

Claude Code also has "slash commands" (like `/review`) and "skills" that are user-defined in `.claude/commands/` and CLAUDE.md. These don't directly output hidden markers, but they change what Claude does. The **skills** (configured in `CLAUDE.md` or via `~/.claude/settings.json`) could, for instance, cause Claude to always output certain data or behave in certain ways. If we define a custom Claude skill to communicate with ccmux (for structured output), it might be possible (this is covered in section 6).

Session Resume Implementation:
When you run `claude --resume <session_id> "query"`, the CLI loads the conversation state associated with that ID and continues it with the new prompt [30]. Under the hood, each session (conversation) likely has a unique ID (the docs show it can be a name or ID; if a name is given, maybe they

map it to an ID). From the GitHub issue we saw, the session ID looks like a UUID [32]. Indeed, the debug log and state file reference a UUID for the session. The state is stored likely in `~/.claude.json` (perhaps this file holds the *last session for each directory*, or all sessions?). The bug report "Excessive File I/O" indicates `.claude.json` is the "main state file" updated ~1.5 times/sec during a session [28]. This suggests `.claude.json` contains the live conversation state (messages, thoughts, etc.) for the current session. Possibly it's a single file that always stores the *current active session* state. Alternatively, it might store multiple sessions keyed by ID. The issue mentions every event triggers rewriting ~100KB, which sounds like it's rewriting the entire conversation state JSON for that session on each token or tool action [40]. This heavy persistence ensures crash recovery: if the CLI crashes or is killed, the state is already saved, and you can resume.

Where exactly is the session data stored? The issue explicitly says "writing to `~/.claude.json` (the main state file)" [28]. So it's not in `~/.claude/session/<id>.json` as one might expect; they chose a single file (which could be overwritten each time). Perhaps if you have concurrent sessions (like separate directories or separate processes), each might use its own state file or they might clash (there are other issues hinting at confusion when multiple sessions run, which might be due to a shared state file) [41]. It's possible that if you run multiple Claude Code instances, they all write to `~/.claude.json` and could interfere – which is a known bug [41]. So likely it's a single global state file for the "active" session. How then do multiple sessions or resumes work? Possibly, when you resume by ID, it first loads that state from file (maybe they archive closed sessions elsewhere or in the same file). Or the session might also be stored server-side by Anthropic (the conversation ID might let their API continue the chat). But since offline resume exists, it implies local storage. There might be a directory `~/.claude/sessions/` or similar for older sessions. The CLI source (if we had it) would clarify, but anecdotal info suggests at least one file `.claude.json` for current state and maybe `.claude.json.bak` or debug logs for each session.

In the debug log path example, `~/.claude/debug/399de4cf-...e5.txt` was a log for that session ID [32]. So, each session might get a debug log file named by its UUID. Possibly, session transcripts or final outputs might also be stored similarly. We should check if there's a `~/.claude/sessions/` directory on disk. The question specifically asks if something is in `~/.claude/` somewhere – yes: `settings.json` and likely session data. The cheat sheet confirms config files: global `~/.claude/settings.json`, plus project-specific `.claude/settings.json` and `.claude/settings.local.json` in repo directories [42]. It doesn't explicitly mention session files, but the debug logs and state indicate they reside in `~/.claude` root.

Accessing Session ID programmatically:
If ccmux starts a Claude Code process, we may want to capture its session ID to enable resume on crash. Does the CLI output the session ID anywhere? There's no obvious mention in normal user docs. However, when starting Claude Code, it might print a line like "Starting session <id>" in debug mode. The GitHub issue did show an example in the logs: "Session ID: `399de4cf-7eca-...`" [43], but that was within a GitHub issue description (perhaps the user added it). It might not actually print to the console in normal mode. Possibly in `--debug` mode or verbose logging it prints the session ID. We could consider running Claude with `--debug` flag, which outputs debug info (the CLI has `--debug` with categories [44]). That might dump lines like the ones in the issue (the `[DEBUG] Writing to temp file...` lines [45]). If so, capturing that output could give us the session ID (since they mention preserving to a temp then renaming to .claude.json, and the log likely includes the session ID or it's guessable from the debug log filename). But relying on debug log text parsing is brittle.

Alternatively, after spawning Claude, ccmux could watch the `~/.claude/debug/` directory for a new file creation (the file name will be the session ID). If we detect a new file named `<UUID>.txt` right after starting, that UUID is the session ID. This is hacky but could work if the debug log file is created immediately. Another approach: after starting Claude, parse the `.claude.json` file; it might contain the session id within (maybe as a field). If `.claude.json` is JSON, opening it and looking for an `"id": "...UUID..."` field could yield it. Since the bug log mentions the file size ~100KB for a conversation, presumably it's a JSON with all messages. The first entry might include a session identifier.

Claude Code also supports naming sessions. If launched in a git repo, it might name the session after the repo directory by default (for resume by name). The CLI example uses `claude -r "auth-refactor"` (a human-friendly name) [30]. Possibly if you start a new session in a directory that has a name set (or via `--session-id` flag), it associates a friendly name. `--session-id` flag exists too (the reference shows it but cut off) [46], meaning you can force a specific session ID (maybe to attach to one that's ongoing). If such flag is available, ccmux could assign its own IDs to sessions instead of having to capture them. For example, `claude --session-id <uuid>` could start or continue a session with that id. If this is allowed, ccmux can generate an ID and use it. However, it might require the session to exist to resume; not sure if it also creates a new session with that id if none exists. If not, at least we could intercept the one it chooses.

Session State Storage Details:
From the "Excessive I/O" issue, we glean: - Writes occur after *every* user message, AI message chunk, tool execution, etc., often multiple times within milliseconds [47]. They write to a temp file and then atomically replace `~/.claude.json` [48]. This indicates an append-only log wasn't used; instead, they rewrite the entire state JSON (100KB) each time (maybe to always have the latest state easily available). - This is costly but means that at any point, `~/.claude.json` is the full snapshot. If ccmux crashes, that file still has up-to-date state (unless it crashes mid-write, but atomic rename reduces corruption risk [45]). - The state likely includes the conversation messages, tool usage context (files open, etc.), and maybe session configuration. It may also track the model and other settings.

Interacting Programmatically (API or IPC):
Currently, Claude Code CLI is primarily designed for human-in-the-loop usage. There is mention of an **SDK** (the CLI flags have `--print` for programmatic usage, and the issues mention an SDK as a thin wrapper around the CLI [49]). The "Feature: Add higher-level session management APIs" issue suggests developers want a better programmatic control (like start session, send query, get result via an API) instead of driving the CLI via subprocess [49]. At the moment, the recommended approach for integration is indeed to run `claude -p` (print mode) commands or use `claude --resume` to continue a session headlessly. The CLI doesn't open a persistent socket or HTTP server for control. It's all through the process's stdio and the state file.

There is an interesting `--mcp` flag for Model Context Protocol configuration (Claude Code can act as an MCP client connecting to local tool servers) [50], but that's not an external API, it's for Claude's internal use to talk to tools.

Claude Code does respond to certain **environment variables** beyond the API key. The Gist of environment vars shows things like `ANTHROPIC_API_KEY` (for API auth) [34], `ANTHROPIC_MODEL` (to set default model) [51], `CLAUDE_BASH_MAINTAIN_PROJECT_WORKING_DIR` (controls whether Claude's bash tool stays in project dir) [35], `CLAUDE_CODE_ACTION` (permission mode, e.g. "plan" vs "acceptEdits") [35], and

`CHOKIDAR_*` env vars for file watching (Claude uses chokidar library to watch file changes, presumably to know when files edited externally or to monitor `.claude/` files) [52]. Also things like `BASH_MAX_TIMEOUT_MS`, etc., which tune the internal tool behavior [53]. This means, when launching Claude Code from ccmux, we can set environment variables to tweak its behavior. For instance, to avoid Claude reading secrets, one might set `CLAUDE_CODE_ACTION=plan` (so it doesn't auto-apply changes). Or to change how it uses the bash tool.

Claude Code CLI Flags of interest:
- `--continue, -c`: loads the most recent conversation in current directory [54]. This implies sessions are also tracked per directory. Possibly, `~/.claude.json` also stores a mapping of "last session in <cwd>". If you just run `claude -c` inside a project, it picks that up. So session state might also be saved in a project's `.claude/` folder (maybe an alias or ID there). This is somewhat speculative, but the `-c` flag suggests some caching by directory. If so, ccmux should be mindful if it launches multiple in the same dir. - `--fork-session`: when resuming, create a new session ID instead of continuing in the same session [55]. This is interesting: it allows branching a session. So internally, you can reuse an old session as base but treat it as new. - `--print, -p`: non-interactive print mode (one-shot) [56]. This is likely how ccmux will often run Claude in the background (to get outputs without going into an interactive loop, unless we want the interactive REPL for some reason). - `--output-format`: as discussed, can be `text` (default), `json`, or `stream-json` [33]. For ccmux orchestrating multiple Claude instances and parsing their outputs, using JSON or stream-JSON might be easiest for parsing (no need to parse ANSI or worry about spinners). We might run each Claude pane in `--print --output-format stream-json` mode so we get structured output we can intercept and also display nicely to the user (we could even pretty-print it in the pane). However, one must consider that interactive Claude features (like asking for tool permission) might not surface in print mode (print mode might auto-run with defaults or fail if a permission prompt would have been needed). The documentation suggests print mode is good for straightforward Q&A or tasks. For complex multi-step agent behavior (which Claude Code does interactively), the interactive mode might be necessary to see those intermediate steps and allow interventions. - If using interactive mode, capturing structured info is harder – but we could toggle on some debug or structured logging. For example, maybe combining `--debug` and `--output-format stream-json` is possible to effectively get partial info? Not sure if both can be used.

Directory structure:
`~/.claude/` likely contains: - `settings.json` (global config, as cheat sheet shows an example with model, tokens, permissions, hooks, etc.) [57] [58]. - `debug/` folder (with debug logs per session) [59] [32]. - Potentially `profiles/` or `sessions/` but not confirmed. - Possibly `CLAUDE.md` global memory (cheat sheet says `~/.claude/CLAUDE.md` is global context) [60]. - The `.claude/` folder in project repositories stores project-specific commands and settings. Those are loaded by Claude Code on start. For instance, custom slash commands in `.claude/commands/*.md` [61] [62], and hooks in `.claude/hooks/` possibly. ccmux doesn't need to manipulate those, but it should ensure not to destroy such structure if present.

Existing Tools Wrapping Claude Code:
One reference is the `claude-canvas` repository the user gave (dvdsgl/claude-canvas). It likely extends Claude Code's UI or capabilities (maybe to provide a richer text-based interface). Since ccmux is somewhat similar in that it will wrap Claude Code, it's worth checking what `claude-canvas` does. Perhaps it connects Claude Code with a TUI or GUI "canvas" for conversation. Without direct access here, I suspect claude-canvas might use Claude Code's SDK or CLI under the hood to fetch results and then display them in a more visual way (like in a browser or graphics context). It's noted "they are clearly extending the Claude

Code canvas somehow," implying they might be hooking into Claude's output to present it differently. Possibly using the `--json` output mode to get content and then painting it on a canvas (like an ASCII canvas?). If claude-canvas is open source and written in e.g. Python or JS, it might give hints on how to integrate. This would require reading that repo's code. We can't do that directly here, but likely it uses the Claude Code npm module (if one exists) or spawns the CLI.

Another tool could be "Cline" (docs.cline.bot) which mentions Claude Code integration [63] – might be a third-party CLI. Also, an OSS project "OSCode" is mentioned (in search results) as an open-source Claude Code alternative [64]. These might not directly help internals beyond confirming usage patterns.

IPC or Sockets:
Claude Code itself doesn't open a port or named pipe for control. But since it's Node, one could potentially attach a debugger or read its memory, but that's beyond scope and unnecessary. The integration strategy is to use the CLI as intended: send it input, read output.

We might consider controlling a running Claude interactive session by sending keystrokes. If ccmux spawns Claude in interactive mode in a PTY, we can programmatically send input to it (just like a user would type). For example, after Claude finishes a response, ccmux could automatically send the next question or a `/command`. This would effectively treat Claude as a sub-process we drive. It's doable since we have the PTY master: we can write to it to simulate user input. So, if we needed to e.g. send a resume command or answer a permission prompt, ccmux can do so.

Environment Variables of Note:
- `ANTHROPIC_API_KEY` – must be set or the CLI will likely prompt to auth via browser. We should ensure the environment is passed. - `CLAUDE_DEBUG` or `--debug` flag – could turn on additional logging if needed for diagnosing issues. Possibly not needed for normal ops, but helpful in dev. - `CLAUDE_DISABLE_FILE_WATCH` (if exists) – since Claude watches files to reload context or for other reasons (they use chokidar, as seen by env vars), if ccmux orchestrates multiple sessions in same dir, multiple watchers might spawn. If that's heavy, maybe could disable if not needed. Not sure if such toggle exists beyond fiddling with chokidar env (like set `CHOKIDAR_USEPOLLING=false` or something, but that's default anyway). - If the user explicitly asked: "What environment variables does Claude Code respect?" – the answer is a long list as in the gist [65] [51]. Key ones: `ANTHROPIC_*` variables (API keys, model, etc.), `AWS_*` for Bedrock usage (if user uses AWS's endpoint) [66], `CLAUDE_*` flags (like those maintenance and permission toggles) [35], some internal testing flags. Also things like `BROWSER` might be used for opening web for auth.

Claude Code – Session ID retrieval for ccmux:
To implement intelligent orchestration, ccmux will want to know if a Claude instance crashed so it can `--resume`. For that, it must have the session ID. Based on above, likely strategies: 1. If we launch with `--session-id` (if allowed for new sessions), we can pre-generate an ID (like a UUID) and store it in ccmux state. Then if it crashes, use that same ID to resume. However, not 100% sure `--session-id` allows starting a session with a given id – the docs say "Use a specific session ID for the conversation (must be valid)" [46], implying it expects an existing ID. It might throw if not found. Alternatively, one might pass an arbitrary name and `--fork-session` to create a new branch with a derived ID. Not entirely clear. 2. Parse the output of Claude's startup (if any). Perhaps run with `--debug` minimally and capture the "Session ID: ..." from its log (if it prints one). 3. Monitor the debug log file creation as mentioned.

Given that in the bug report the *user* was able to identify the session ID (maybe by looking at the debug file name or logs), we might lean on the debug file approach if needed. Another hack: after starting, read `~/.claude.json` and extract an `"id":` field. That requires parsing a potentially large JSON. But for a fresh session, it might be small initially. This could be done once at startup. We have to ensure not to parse it while it's being written though (the atomic write helps, but reading mid-write might get old content or partial if not careful). But presumably after first message, it's written. We could even attach a file watcher to detect when `.claude.json` changes and parse it to update our internal knowledge (though that's quite hacky).

Internal Tools and Session Behavior:
Claude Code is an "agentic" tool, meaning it can execute "tools" like reading files, writing files, running shell commands (within constraints), etc. Internally, when Claude decides to use a tool, the CLI will output something indicating that (like "[Claude is executing: `npm install`]" or similar) and then show the output of the command, etc. This means the output stream can have not just Claude's chat messages but also these tool outputs and prompts. The structured JSON output likely encodes those events as well. For example, an execution might come as an event object with the tool's output.

From ccmux's perspective, we might want to capture events like "Claude is running tests…" so we can reflect that in UI (maybe highlighting the pane or showing a status). Also, if Claude pauses for user confirmation (like permission prompt), ccmux could auto-approve or at least highlight to the user in the multiplex UI. The environment variable `CLAUDE_CODE_ACTION` can be set to "acceptEdits" or "plan" etc., which influences whether it auto-applies changes or asks permission [35]. For automation, one might set it to always accept to avoid interactive prompts (with caution). If ccmux is supervising, we might choose to always allow or have ccmux ask the user via its own UI.

Wrap-up:
In summary, Claude Code's internals revolve around a continuously saved session state and a loop of reading user input, sending to API, streaming response, and executing tools. There is no trivial API for external control besides the CLI itself. Therefore, ccmux will interact with it as a black-box process: it can spawn new Claude sessions, possibly monitor their state via output parsing or state file, and if needed, terminate or restart them with `--resume`. The session state is fortunately persisted by Claude Code, so ccmux can rely on that for crash recovery (just need the session ID).

Recommended Approach (for ccmux Integration)

Launching Claude Instances: Use the Claude Code CLI in **non-interactive mode** where feasible, to simplify parsing. For example, when a user triggers a new Claude in ccmux, run `claude -p --output-format stream-json "<query>"` in a PTY. This will start, produce JSON events (which ccmux can parse to update the pane in real-time with Claude's answer), and then exit when done. This mode is suitable for a straightforward query/response. It yields structured info including partial messages if enabled [37]. By parsing those JSON events, ccmux can precisely know when Claude is "thinking" (it will receive a stream event with content tokens) versus "complete" (when the final message or done event arrives). We should also capture the session ID from the JSON (the final JSON might include it, or we ensure to log it via debug).

However, if we want to utilize the full *agentic* loop (where Claude might prompt for tool permissions or do multi-turn planning), we might need interactive mode. In such cases, we can still run Claude in a PTY in interactive mode and simply not attach a human. ccmux would feed an initial prompt (which starts the sequence), then monitor output. If Claude asks something (like “Allow me to run tests? (y/n)”), ccmux can either automatically answer (maybe based on a policy or user preference) by writing “y\n” to the PTY or bubble it up to the ccmux user interface (like show a prompt in that pane asking the user to press a key to approve). Similarly, if Claude finishes and awaits next input, ccmux could either consider that session idle or send follow-up queries if orchestrating a chain.

To manage multiple concurrent Claudes, ensure each is launched with isolated state. Possibly set different working directories or at least ensure `--fork-session` if in same dir to not reuse the last session by accident. We might run each in a temp directory or set `CLAUDE_LOCAL_DIR` if such exists (no clear env, but perhaps `HOME` could be changed to isolate? That might be overkill). More simply, use `--session-id` or `--fork-session` to ensure each call doesn’t interfere.

Capturing Session ID: As discussed, either parse `.claude.json` or use debug logs. For reliability, one idea: run Claude with `--debug` and filter stdout for a pattern like `Session ID:` [43]. Or possibly the debug log line “Added snapshot for <UUID>” [67]. Given the heavy debug output, maybe better to open `.claude.json` after initial write. We can experiment to find the session id field.

Session Resume on Crash: If ccmux detects a Claude process crashed (e.g. the PTY closes unexpectedly, exit code non-zero), it should attempt to restart it with `claude --resume <id>`. We need to consider that Claude Code itself might rarely crash unless the Node process exits due to an error or OOM. More likely, ccmux itself could crash, leaving Claude processes orphaned. In that scenario, those Claude processes might still run (but if ccmux dies, the PTY master closes, which would likely cause the Claude processes to get SIGHUP and exit – i.e., orphaned Claude Code might terminate if its stdout/stderr are gone). Assuming they end, the session state is saved, and we can reopen them. ccmux on restart could scan for session info to restore panes. For example, it could read the `~/.claude.json` to find if there was an active session, but if multiple sessions, that’s tricky. Instead, ccmux itself should save (in its own config) the mapping of pane -> session ID. Then on restart, it can ask “do you want to resume these sessions?” and do so with `--resume`.

No direct IPC: Since there’s no official API, ccmux controlling Claude is essentially by sending input via PTY and reading output. This is fine. We just need to ensure not to lose content. We might consider using the `--include-partial-messages` and stream JSON to get token-level updates if we want a smoother output streaming in our UI (instead of waiting for full completion). This can make the pane update token by token as Claude “types,” which is nice.

Querying Session State: There’s no direct `claude --print-state` command. But since state is in the JSON file, if needed, ccmux could read conversation history from it (but probably not needed to show to user, as the user sees all output anyway in the pane).

Hooks and internals: The user’s question about session ID and internal state implies they might want ccmux to handle scenarios like continuing a long session. Perhaps to split a single conversation across multiple sessions or to coordinate multiple Claude agents. Knowledge of the session ID and stored state means ccmux could theoretically manipulate it (like edit the JSON to remove some parts?), but that’s risky. Safer is to let Claude Code handle its memory and just use the provided flags (like `--fork-session` if we want to branch).

Potential Pitfalls: - The **state file lock** – if multiple Claude processes run simultaneously, do they each try to write `~/.claude.json`? If yes, they could conflict. The existence of issues about multiple sessions interfering [41] suggests this is a real problem in current Claude Code. If we plan to run many concurrently, we might need a workaround: perhaps set `CLAUDE_STATE_PATH` if such env exists (not documented) or temporarily change `HOME` for each process so that each uses a different `~/.claude.json`. Another approach: run each Claude in a separate container or use the `--session-id` to avoid conflicts (but not sure if that avoids writing to the same file). Possibly each instance still writes to the same file path if not configured otherwise. This is crucial: ccmux might need to isolate them. We can experiment by running two Claude instances concurrently and see if they overwrite each other's state or one picks up the other's session. If conflicts occur, a solution is to set `HOME` env var to a unique temp directory for each Claude process (Claude Code likely uses `os.homedir()` to locate `~/.claude`). If we do that, each one will have its own .claude directory (with its own settings and state). That could actually be beneficial: truly isolating each agent. We'd have to copy any global config (like API key) into those envs (e.g., via env or ensure settings.json is found). Or there might be a `CLAUDE_CONFIG` env or similar (the gist didn't show an explicit one, but `APPDATA` might be relevant on Windows) [66]. This is an open technical question.

- **Multiple sessions reading one state file** – the bug "Status line displays incorrect model across multiple sessions" suggests that if two sessions share state file, they might read each other's data [41]. That is a scenario we want to avoid. So likely we will indeed isolate via separate HOME or wait for upstream fix (not guaranteed soon).

- **Auth** – each separate Claude instance will need the API key (from env or settings). If we use separate HOME for isolation, we must ensure the API key is available. The simplest is to always set `ANTHROPIC_API_KEY` env for each spawned process (rather than relying on a key in `~/.claude/settings.json`). That environment approach is straightforward.

- **Resume limitations** – We should verify if `--resume` works with print mode. Possibly yes: you can do `claude -p -r <id> "next query"`. That would load the session and run one turn then exit. If ccmux is orchestrating, that's probably what it will do to resume after a crash or to chain queries (instead of keeping the interactive process alive, we might repeatedly call print with resume for each user query). Actually, that's an interesting architecture: rather than running one persistent interactive process per pane, ccmux could treat each pane as stateless UI that issues `claude -p` calls with the same session id for each prompt. That means after each answer, the Claude process exits, and when the user types again in that pane, ccmux spawns a new process with `--resume` to continue. This is somewhat like how one might integrate with the API directly (each query is independent but with a conversation ID for context). The advantage: if one call crashes or gets rate-limited, we can just spawn another. Also no need to manage a long-lived process. The disadvantage: startup overhead and losing streaming mid-answer if not careful. But since the CLI can stream output in print mode, it should be fine. This approach also avoids the multi-session state interference, because sequential calls still share one state file but not at exactly the same time. However, if two queries in two different panes happen concurrently, that's multiple processes concurrently – we're back to multi-state conflict unless isolated. So likely we will have concurrency (the user may run multiple Claude tasks at once). So either isolate via separate HOME for each, or use interactive mode (one process per pane that remains running). Using one long-lived process per pane might ironically cause less file conflict (each still tries to write to global state though!). Unless they scope state by session ID in the file (unknown).

Given the complexity, an easier path is to isolate each Claude by setting a custom home directory (like `~/.claude_<pane_id>`). Then each has its own .claude.json. They won't clash. The downside: none of them will see your real `~/.claude/settings.json`, so you might have to populate minimal config (like API key) or pass those via environment. That's doable. Also, any custom commands/skills from the project's `.claude/commands` might not be auto-loaded because Claude Code finds them by walking up from cwd to find `.claude/commands`. If we keep the cwd the same for the processes as the user's project directory, they will still find those (the cwd can remain the project folder; changing HOME doesn't affect that). So they can load project-specific stuff fine. Changing HOME mainly affects where it stores global config and session logs.

Programmatic Communication:
Since no direct socket API exists, ccmux might rely on these workarounds. Another possibility: Claude Code might expose some local HTTP server when running (though not documented). The CLI does have an `anthropic-cli` mode possibly, but likely not.

We should also mention any internal doc or code hints about IPC: There's nothing obvious except the issues complaining about lack of APIs [49]. So likely none.

Code Examples

Starting Claude in print (non-interactive) mode and parsing JSON:

```
// Example pseudocode for spawning Claude Code in print mode with JSON output
let session_name = "ccmux-session-123"; // can be unique per pane
let mut cmd = Command::new("claude");
cmd.arg("-p")
   .arg("--resume").arg(session_name)   // resume or start a session with this
name/ID
   .arg("--output-format").arg("stream-json")
   .arg("--include-partial-messages")
   .arg("Implement the auth module"); // the user prompt
cmd.env("ANTHROPIC_API_KEY", api_key);
cmd.env("CLAUDE_FORCE_TTY",
"false"); // (if such existed to avoid spinners, not sure)

let mut child = cmd.stdout(Stdio::piped()).spawn().expect("Failed to start
Claude");

let stdout = child.stdout.take().unwrap();
let reader = BufReader::new(stdout);
for line in reader.lines() {
    let line = line?;
    if let Ok(json) = serde_json::from_str::<serde_json::Value>(&line) {
        // Interpret JSON event
        if json["type"] == "completion" {
            let content = json["completion"].as_str().unwrap_or("");
            // This is a chunk of Claude's answer – append to pane output
        } else if json["type"] == "done" {
            // Claude finished answering
        } else if json["type"] == "error" {
            // handle error
        }
    } else {
        // If not JSON, maybe Claude printed something else (unexpected)
        eprintln!("Non-JSON output: {}", line);
    }
}
let status = child.wait()?;
```

(The actual JSON schema is hypothetical; we'd need to see a real example from Claude. But conceptually, it would stream out events like partial completions and final result.)

Resuming after a crash:
If a Claude process terminates unexpectedly, ccmux can do:

```
// Assume we stored session_id for this pane
let mut resume_cmd = Command::new("claude");
resume_cmd.arg("-p").arg("--resume").arg(session_id)
          .arg("--output-format").arg("stream-json")
          .arg("--include-partial-messages")
          .arg("Continue where you left off.");
// ... set env and spawn similar to above
```

This would prompt Claude to basically say "As I was saying..." or continue whatever it was doing, because the conversation context is loaded. (If the crash happened mid-answer, we might need to craft the prompt to tell Claude to resume, or simply resending the last user query might yield a similar answer with some variance.)

Reading session state file:

```
use std::fs;
let state_data = fs::read_to_string(home_dir.join(".claude.json"))?;
if let Ok(val) = serde_json::from_str::<serde_json::Value>(&state_data) {
    if let Some(id) = val.get("session")?.get("id") {
        println!("Current session ID: {}", id);
    }
}
```

(This assumes the JSON has a field like session.id or similar containing the session identifier.)

References

- Claude Code CLI flags and usage [68] [33]
- Session resume and identification (session ID, names) [30] [32]
- Continuous session state writing to `~/.claude.json` [28] [40]
- Debug log indicating Session ID and log file [32] [43]
- Environment variables used by Claude Code (API key, model, etc.) [34] [35]
- Multiple sessions interference issues [41]
- JSON output mode and structured outputs [38] [33]
- Cheat sheet covering config and CLI usage [42] [69]
- Skills/commands storage (custom commands in `.claude/commands`) [61] [62]

Open Questions

- **Concurrent Session Isolation:** As noted, how to handle multiple Claude processes writing to a single state file? We may need to test and potentially sandbox each process's config (e.g. separate HOME directories). This is crucial for ccmux stability when running many concurrent sessions. Investigating if Claude Code supports a `CLAUDE_DIR` or similar env to redirect its data directory would be helpful (none documented, but maybe we could set `XDG_CONFIG_HOME` or so). Otherwise, using distinct UNIX users or containers might be an extreme but possible solution. This needs research or experimentation.
- **Claude Code "Thinking" Indicator:** Empirically determine how the CLI indicates thinking. If it's a spinner on the same line (likely using carriage return), will our vt100 parser capture that? For example, a spinner might output `\r[thinking 0%]` updating repeatedly, then clear. The vt100 parser might just see a bunch of carriage returns and the final text. We might need a way to detect that scenario to show a spinner in the ccmux UI. We should run Claude in a dummy scenario and observe raw output bytes. This is a task to do to improve UX (or rely on JSON mode events which explicitly tell us when it's in progress).
- **Partial vs Final Output:** When using print mode and stream JSON, how to best present it to the user in ccmux? We likely will stream it as it comes. But if something goes wrong mid-way (like network error), does Claude output an `error` event? If so, ccmux should catch and possibly auto-retry with resume. Handling of errors (like API rate limits or Claude returning an error message) is something to design for robustness.
- **Tool Interaction and Prompts:** If using interactive mode for complex flows, ccmux must handle when Claude Code asks for input (like permission or clarification). Should ccmux always auto-approve tool usage? Possibly configurable. The user might want to manually allow certain high-risk actions. We could surface those prompts in the UI (like highlight the pane and wait for user keystroke). Alternatively, run with `--dangerously-skip-permissions` to auto-allow everything [70] – convenient but risky. Perhaps make it a config option in ccmux.
- **Session termination:** If a user "closes" a Claude pane in ccmux, we should kill that Claude process. That's straightforward, but we might also want to save the session state or name it for later resumption. Claude Code likely already saved it. But if user might want to resume that session later in another pane, we should allow them to input the session name/ID to ccmux. It's an edge feature. At minimum, ensure graceful termination (send SIGINT or just kill) – Claude might catch SIGINT to do something (not sure). Possibly just killing is fine since state is on disk.
- **Versioning and updates:** `claude update` command can update Claude Code binary [71]. If a user triggers that via Claude inside ccmux, it could shut down the process. Not a typical scenario for

ccmux to handle, but something to note (maybe block the update command in an active session to avoid confusion).

- **Alternative approaches:** There is mention of an "SDK" (Claude Code SDK possibly in Node). If such exists, maybe using it directly (embedding Node or calling via node.js binding) could skip the CLI overhead. But that's likely not stable or easier than CLI approach. Unless Anthropic releases an official API for the coding agent, we stick to CLI.
- **Internal MCP and Tools:** Claude Code can connect to Model Context Protocol (MCP) servers (like for web search, Slack, etc.) [72] . If the user uses those, the Claude instance might open local ports or browser (for Chrome integration). ccmux should be aware: e.g., if Claude tries to open Chrome (with `--chrome` flag) [73] , it might not work well on a headless server. This is more of a user environment issue, but ccmux could detect such attempts via output and warn. Not critical for now but something to remember (the user might avoid using those flags in ccmux context).

---

3. Crash Recovery for Terminal Multiplexers

Executive Summary

True **crash recovery** in a terminal multiplexer is challenging because you cannot fully restore a running process's state unless it was preserved. Traditional multiplexers (tmux, screen) rely on a separate server process to keep sessions alive if the client (UI) crashes or disconnects – if the server itself crashes, the child processes terminate. To provide crash recovery, ccmux must either **prevent loss** (by isolating processes in a way that a ccmux crash doesn't kill them) or **persist state** to restart them. Key state to persist includes each pane's **terminal screen contents (text and scrollback)**, plus metadata like cursor position and pane layout. This can be serialized (e.g. saving each grid to disk). However, the **running processes** (shells, Claude Code, etc.) cannot be revived unless using OS-level checkpointing (e.g. CRIU) [74] , which is complex and rarely used for interactive programs. Instead, the practical approach is to have a *resilient architecture*: run the child processes in a separate *guardian* process that doesn't crash when the UI does (like tmux's server), allowing the UI to reconnect. If even the guardian crashes, you can at best restart new processes with the last known state (for Claude, using `--resume` to continue the session; for a shell, there's no built-in resume, so you'd start a fresh shell). Strategies for persistence are (a) **continuous logging** of all output (like a write-ahead log of terminal output) so that on restart you can replay it into a parser to reconstruct the screen, (b) **periodic snapshots** of screen state (e.g. every N seconds or on certain events write the screen buffer to a file), or (c) a combination (log updates and occasionally snapshot to trim the log). Tmux and Zellij do not implement full crash recovery for their server – if the server dies, sessions are lost. They focus on preventing server crash and handling client disconnects. There are lightweight tools (dtach, abduco) that keep a process alive and let you reattach, but they don't preserve the scrollback/state for you [75] [76] . ccmux aims to treat crash recovery as a first-class feature, meaning we likely will implement a *tmux-like server* that stays running independently of the UI, or at least periodically dump state to disk so sessions can be manually restored.

Detailed Analysis

State That Must Be Persisted:
The core state for each terminal pane includes: the contents of the screen (all rows and columns of text and their attributes), the **scrollback history** that is not currently visible (if any), the cursor position, and mode flags (such as whether the application is in alt-screen mode, whether text is wrapped, etc.). If we have that,

we can redraw exactly what the user saw. We might also want to persist things like the current working directory of the shell (though that's part of the process state, not easily accessible unless we query the shell or track `cd` commands). For a process like Claude Code, the critical state is in the process's memory (conversation context, etc.), but since Claude Code itself logs state externally (session JSON), we at least can resume it with that context if we restart the process. For a generic shell, we cannot retrieve its state (like environment variables, command history) after crash unless we had some hooks. However, one could argue that ccmux is focusing on Claude processes primarily (though maybe it will allow general shells too). If general shells are in scope, full recovery of an interactive shell session after the controlling PTY is lost is effectively impossible – the shell will have died. The best we could do is re-open a new shell and **restore the visual appearance** of the old one's last screen (which is like showing a screenshot – the new shell isn't actually at that state or logged in to remote servers, etc.). This could be misleading, so maybe not advisable. Alternatively, for non-LLM processes we accept that they cannot be resumed (we could label them as crashed in the UI and let the user start a fresh one manually).

So our realistic recovery scenarios are: - **Claude sessions:** can be resumed by launching a new Claude Code process with the last session ID, effectively continuing where it left off. - **Other long-running tools (like tail, top):** we can restart them, but they won't have exactly the same state (e.g., `top` will refresh). That might be okay. - **Shells:** cannot resume the exact session. Possibly we simply start a new shell. Tmux doesn't attempt to recover if it itself crashes – it just loses everything.

Therefore, the emphasis is on preserving what we can: the display/output for reference, and, for specialized processes like Claude, providing a way to actually continue logically. Persisting the **terminal screen buffer** is straightforward – it's just text and attributes. We can store that as a struct (list of lines, each line is list of cells with char+attributes). Serialization can be to JSON, CSV, binary, etc. The scrollback might be large (1000s of lines per pane if we allow), but even a few thousand lines of text is not huge (couple MB). Writing it periodically is fine. Alternatively, log every output byte and on recovery, replay them through the parser to rebuild the screen. That is what a **WAL (Write-Ahead Log)** approach would entail: all input to the terminal (output from the process) is logged. After crash, spawn a dummy process that just replays the log into a parser to regenerate the screen. This ensures fidelity but could be slow if log is huge and the process ran for hours. You could combine: snapshot the screen occasionally and truncate the log.

Persistence Strategies:
- **Continuous (synchronous) write:** e.g., after every chunk of output, append it to a log file and maybe update a "latest screen" file. The I/O overhead could be significant if output is heavy (the bug with Claude's 1.5 writes/sec to disk [77] shows performance concerns). But we can optimize by writing raw bytes (not re-serializing JSON each time like Claude did). A concern is also disk wear if heavy usage, but probably minor in realistic usage. - **Periodic snapshot:** e.g., every 5 seconds or every 1000 lines of output, write the screen buffer to disk. If a crash happens, the worst-case loss is a few seconds of output (which might be acceptable). Crash in the middle of those intervals leaves a small gap, which might be tolerable, though for perfection maybe not. - **WAL + snapshot:** This is a robust approach: log every output increment (cheap to append a few bytes) to, say, a `.log` file; and occasionally write a compressed snapshot of the screen and current log position. On recovery, find the latest snapshot, then replay log entries after that. To avoid unbounded log growth, you trim older parts that have been snapshotted. This is akin to database journaling. It's complex but doable. Possibly overkill for a terminal though – might be simpler to just snapshot often and ignore a short tail.

One must also consider the **pane layout and multiplexer state**. If ccmux itself crashes, on restart it should know how many panes were open, how they were arranged (splits), which process in each, and ideally automatically re-launch/resume them. So that is another piece of state: the **session layout** – basically the data that would go into an "architecture document" – what processes and commands were running, and maybe for shells, what working dir they were in, etc. ccmux can store this in a separate file (like ccmux session config). Tmux does not persist sessions to disk at all (unless you use plugins like tmux-resurrect which manually save layout and re-run commands), but we want to. We might define an `~/.ccmux/sessions/<session_name>.json` that stores pane tree structure, for each pane: type (Claude or shell or other), command, maybe session ID if applicable (for Claude), and a path to the last screen dump or log.

Adopting Orphaned PTYs:
If ccmux crash doesn't kill child processes (imagine a scenario where ccmux UI is separate and it crashes but the backend holding PTYs remains), then we could reattach to those PTYs. That is how tmux works: the server holds PTYs, so if the client (UI) disconnects or even crashes, the server is still running, and you can reconnect. But if the server died, the PTYs are gone. So to adopt orphaned PTYs, you'd need the OS to keep them alive. Without a parent process, it's tricky – maybe if we double-fork processes so that they are not children of ccmux (so ccmux could crash and the processes continue, attached to the controlling terminal which might cease to exist... unless we do something like `setsid` to detach them completely – but then who reads their output and where does it go? You'd need some other placeholder to be the master). One idea: use a stub like **abduco/dtach**: these create a pseudo terminal pair and keep the master in a minimal process. ccmux could in theory leverage such a mechanism: rather than ccmux directly opening PTY and spawning child, we could run each child inside an `abduco` session (abduco will hold the pty). ccmux then connects to abduco's pty as a client. If ccmux dies, abduco still holds it. Later ccmux can reconnect (just like reattaching to screen/tmux). Abduco does not preserve scrollback though [76] – when you detach, you lose output unless you logged it. But at least processes live. This is an interesting approach: basically outsource the persistence to an external minimalist tool. But it adds dependency and complexity. Alternatively, implement our own minimal headless guardian that stays separate from the main UI. That basically means making ccmux itself split into two processes (like tmux): a server (responsible for PTYs and running commands) and a client (UI). If the UI crashes, server keeps running; you can launch a new UI client to reconnect. This is the route tmux chooses. It's robust but means more architectural work (inter-process communication, etc.).

Given the user's context engineering doc, they intend ccmux to possibly be single-process but with robust saving. They mention "Crash recovery is a first-class feature, not an afterthought," which suggests they are willing to implement significant changes. A client-server model might indeed be on the table.

Tmux's Approach:
Tmux uses a client-server model. The server process forks the user's shells and manages the ptys; the user interacts via a client (which itself is basically a terminal application that draws the output of the server and relays input). If a client disconnects (e.g. SSH session drops), the server and child processes continue in background. If the server (tmux) process itself crashes, all managed processes likely exit because their controlling pty is gone. Tmux does not persist to disk or auto-recover a server crash. Tmux prioritizes not crashing – it's a lightweight C program with few dependencies (hence more stable). It also can be run by e.g. systemd to auto-restart, but that wouldn't restore internal state. Some users solve this via tmux-resurrect which manually logs state (window layout, last command run, etc.) and user can semi-automatically restore a session (but not the actual program states). Tmux's design tradeoff is complexity vs reliability: they don't attempt full persistence because of the difficulty.

Zellij's Approach:
Zellij similarly runs a server (the main zellij process) and possibly plugin processes. If that crashes, I don't think it has persistence to resume (the project doesn't advertise crash recovery). They likely rely on Rust's memory safety to avoid crashes, but logic bugs could still crash it. There might be an open issue about persistence across restarts, not sure. Zellij does have a concept of layouts that can be launched (like a config that opens certain apps in certain splits), but that's more initialization than restoring live state.

CRIU (Checkpoint/Restore In Userspace):
This is a Linux project to freeze running processes and dump their state (memory, registers, etc.) to disk and restore later [78]. In theory, one could use CRIU to checkpoint a running shell or LLM process and restore it after a crash. However, CRIU has limitations: it works better for batch processes or containers, and restoring an interactive shell with open PTY might be problematic. Also CRIU requires the process to be in a stoppable state and some kernel support. Using CRIU on every multiplexer crash is heavyweight and not guaranteed to succeed (especially if the process was mid-system-call etc.). It's also Linux-only. Likely not worth the complexity for our case (though academically interesting).

Abduco/Dtach vs Full Multiplexer:
Abduco and dtach offer a simplified persistence: they detach the terminal from the controlling process so that the program can run independently. If our ccmux simply orchestrated abduco sessions, it would delegate session persistence to abduco (the processes survive if ccmux dies). But as noted, abduco doesn't store scrollback – it's like a poor-man's tmux without a scrollback buffer (the screen state is lost when no client is attached) [76]. So if ccmux crashes and later reattaches, you'll only see new output from that point. For some use-cases (like long-running log tails) that might be fine, but for others not. We could combine abduco with our own logging to handle that gap (log everything while ccmux was attached; if ccmux dies and reattaches, show the log backlog then attach to live feed). This hybrid could work but gets intricate. Alternatively, implementing our own server to hold PTYs and buffers isn't too far from writing tmux itself – which we are basically doing, albeit specialized for Claude contexts.

Recommended Path for ccmux Crash Recovery:
I would suggest implementing a **persistent daemon (server)** that is started when ccmux launches, which holds all PTYs and possibly runs the child processes. The UI (which could be the same process initially, but in design could be separate) communicates with this server. If the UI crashes, the server keeps running in the background (maybe it would detect the UI connection lost and wait for new connection). The user can then relaunch ccmux which reconnects to the existing server and reconstructs the UI. This is essentially tmux's model. It addresses process survival (since the server didn't crash, processes are alive) and state (the server's memory still has all screen buffers). It does require implementing an IPC mechanism (tmux uses a Unix domain socket between client and server). Rust makes it quite feasible to fork off a separate process or at least thread – though processes offer more crash isolation. We could have ccmux on start fork itself: parent becomes server (no UI, just managing state), child becomes UI client connected via a pipe. This way, even if the UI (child) crashes, parent (server) lives. If server crashes, everything is lost anyway (like tmux, we can then attempt file-based recovery if we implemented it). But we could use both: have a server to survive UI issues, and also periodically dump state to disk in case the server itself crashes (or machine reboots). That would be the ultimate in resiliency.

So for crash recovery, the **design trade-offs** are: - Building a client-server adds complexity but yields better realtime persistence. - Relying on on-disk snapshots (with a single-process design) means any crash definitely kills child processes, but at least the user can see what they were doing and possibly resume certain tasks manually. - Perhaps a compromise: run Claude processes separate (like their own processes which might not die if ccmux UI dies)? But if ccmux is the PTY master, once it's gone those processes either hang or die. Unless we pre-open ptys in a way that they aren't tied to ccmux's lifetime. Not trivial outside the context of a separate server.

Saving and Restoring Terminal State:
If we do rely on snapshots for something (say the server persisted state to disk on clean exit or periodically), we need to re-initialize our `vt100` parser with the saved content. `vt100::Screen` could possibly be reconstructed by simply replaying the sequence of all characters in memory to a new parser, but an easier method: since we have cell-by-cell info (with colors), we could directly repopulate a screen data structure. `vt100` crate doesn't have a direct API for injecting a pre-filled screen (it expects you to feed escapes), so restoration might involve writing a small routine that takes each cell and produces ANSI escapes that would paint that cell at that position (like how `screen` or `tmux` might redraw from logs). However, since we have control over our own data structures, an alternate approach: we could bypass vt100 on restore and instead use our own rendering of the saved cells to the UI, then re-instantiate a new vt100 parser and perhaps drop the history (or keep it solely for new content). But that might lose subtle mode states. Possibly simpler: save the *log of output* and re-run it through the parser exactly as originally – that ensures an identical state including any mode toggles. This requires having the log, which circles back to the WAL idea.

tmux-resurrect and others:
There are external tools where tmux users save their session (list of windows, working dirs, last foreground process in each) to a file, and on reboot manually start tmux with those commands. That doesn't restore processes (just restarts new ones in same dirs running same commands). For shells, that just opens a new shell at ~. For editors, it might reopen them but not with unsaved data. Similarly, for ccmux, if full process-level recovery isn't possible, we could implement a feature to "resurrect" by launching new processes in same configuration. For instance, after a total crash: - For each pane that was running a Claude session, start `claude --resume <id>` to continue. - For each pane that was just a shell, start a fresh shell (no resume, but at least user doesn't have to open it themselves). - For a long-running tail or something, just restart it. - This way the user gets their environment back, albeit some things are lost.

This is easier if we've persisted the layout and the list of commands.

Open-source inspirations:
GNU Screen had a long-term request for session recovery but never implemented it, partly due to the difficulty. Some research or academic tools may have attempted to make a crash-proof terminal multiplexer. If any, that's niche – CRIU is typically suggested. Also, container orchestrators (like Docker or Kubernetes) can checkpoint containers (like pausing them), but again not widely used for interactive tasks.

Conclusion / Plan:
For **ccmux**, to truly treat crash recovery as first-class, a combination of design choices is likely: adopt a **server/client model** to handle common crashes (UI crash or accidental closure won't drop sessions), plus implement a **session persistence to disk** to handle the server crash or system reboot scenario. This persistence includes saving pane layout and the output content (and any resume tokens like Claude session IDs). On restarting ccmux after a full crash or reboot, it can offer to restore the previous session using that file (similar to how editors recover unsaved files).

Recommended Approach (Crash Recovery in ccmux)

1. **Implement a background Session Manager Process:** When ccmux starts, spawn a lightweight manager (or fork itself) to hold the PTYs and processes. The UI connects to it via IPC (a Unix socket or pipe). If the UI crashes or is closed, the manager keeps everything alive. The user can then re-run ccmux to reconnect (perhaps ccmux detects an existing session manager running and auto-attaches). This covers many crash scenarios. Use cases: user's terminal closed unexpectedly – normally that would kill a normal process, but with this design, ccmux's server and children survive detached (similar to tmux).

2. **Periodic State Checkpointing:** Have the manager periodically write out the current state to a file (in JSON or similar). This state includes:

3. The layout of windows/panes.
4. For each pane: the type (Claude, shell, etc.), identification (Claude session ID if any, or process command), and a snapshot of the screen buffer (probably as an array of strings with ANSI color codes or a compact structure). Alternatively, store the raw output log for each pane since last checkpoint.
5. Could also store scrollback beyond screen if needed (maybe include it as part of the screen snapshot or separately).
6. The timestamp or sequence number of the snapshot, and possibly the portion of log applied. This file can be `~/.ccmux/last_session.json` or similar. Keep it atomic (write to temp and rename) to avoid corruption (Claude Code did the same for `.claude.json` [45]).

We can throttle how often to write to avoid performance hits. For example, write whenever a pane's output hasn't been checkpointed for, say, 5 seconds and new output arrived, or whenever there is significant change (like new pane created/closed). The output log, if used, can be truncated on checkpoint.

1. **On Manager startup**, if it finds an existing state file from a crash (and the user invokes a "restore" flag or we always auto-restore last state), it can read it and attempt to reconstruct the session. Reconstruction means:
2. For each pane in the file, create a new PTY and process. If it's a Claude session with ID available, launch `claude --resume <id>` to continue that conversation (perhaps with a dummy prompt like "(Session reconnected)" or simply let it sit waiting for user input, which is likely the case – since user didn't necessarily give a new prompt yet).
3. If it's a shell, just start a shell (it won't have the old environment, but that's unavoidable). We could `cd` it to the last known directory if we had that info. Last known directory could be gleaned if we had a way to ask the shell to output `$PWD` occasionally, or if we integrate with the shell's prompt or use something like `OSC 7` (some shells set an OSC sequence with current dir). This might be overkill. Perhaps assume working dir is not critical or always initial directory.
4. After launching the new process, we restore the **visual state**: The saved screen snapshot is applied to the vt100 parser state for that pane and we render it. For the user, it looks like the pane still contains what was there pre-crash (like a ghost image). For a Claude session, since we resumed it, the next user question can proceed. For a shell, it's a bit of a lie (the screen shows the last output of previous shell, but the new shell is at a fresh prompt). This is similar to how some terminal emulators let you scroll up to see output that came before program restart – but here it's in the same interface. We might want to clearly indicate that the shell was restarted. Possibly just a notice or a line break.

This approach is akin to tmux-resurrect: you get your window layout and content back, but interactive programs have been restarted (if possible).

1. **Write-Ahead Logging (Optional):** If we want to precisely restore to mid-output state, we can combine with logging. For example, if we snapshot every 5 seconds, what about the last few seconds before crash? We could continuously log each output chunk. Then on recovery, we feed the log to the pane's parser to replay the missing bits after the snapshot. If the crash happened before a scheduled snapshot, the log plus last snapshot will reconstruct up to just before crash. If the process died with ccmux, obviously new output ceased at crash moment anyway. This ensures not even a line of output is lost from the user's view. However, implementing this might be YAGNI unless we absolutely want zero data loss guarantee.

2. **Adopt processes if possible:** With the manager-client model, adoption is moot because the manager would have to crash to lose processes. If the manager crashes, the processes likely die too, so we must restart them as above. Abduco/dtach integration could be considered if we absolutely want processes to survive even manager crash. But if manager crashed, something's very wrong (maybe a bug or OOM). One could imagine a design where each pane's PTY and process is actually managed by a separate tiny process (like dtach or our own minimal fork), so that if the main manager crashes, those guardians keep processes alive. Then a new manager could reconnect to those guardians. But implementing that from scratch is a project on its own. We could maybe use dtach for that purpose, but we'd need to coordinate with it. Possibly too much complexity for diminishing returns – better to focus on not crashing the manager (Rust helps) and using resumes for special cases like Claude.

3. **Test the recovery workflow thoroughly:** Induce crashes (kill -9 the UI, kill -9 the manager), see what remains and how restoration works. Ensure Claude sessions indeed resume properly (for instance, if Claude was mid-answer when everything went down, on resume it might either finish that answer or not; likely it got interrupted and the user would need to re-ask or something – ccmux might decide to resend the prompt automatically on resume if it thinks output was incomplete).

tmux Session Persistence Code & Zellij Serialization:
We did not find evidence that tmux or Zellij implement on-disk session save (tmux doesn't by default). The user specifically requests "Deep dive into tmux's session persistence code" and "Zellij's session serialization format," implying they might have some implementation. Possibly referring to how tmux keeps the session in memory and how we could mimic it. Tmux's code for managing windows, panes, layouts is in C and not trivial, but key point: it doesn't serialize to disk. Zellij does have a concept of layouts (YAML) for static layouts, not runtime state. So likely they don't either. The user might be thinking of tmux-resurrect plugin for tmux, which is not built-in but widely used. That plugin writes out session info to a file (pane commands, dir, etc.). We could take inspiration from tmux-resurrect's format or approach. It basically saves for each pane: the command running, current directory, and perhaps a snippet of scrollback or not sure if it does scrollback. Then you can feed that to a tmux startup script. It doesn't try to recover scrollback because tmux itself doesn't allow populating a pane with arbitrary scrollback easily, aside from brute-force printing it out.

CRIU research / viability:
To gauge if CRIU could let us truly freeze and restore a shell: it can checkpoint a process tree, including file descriptors (so the PTY master/slave). If we used CRIU when manager crashes (which is too late) or

proactively checkpoint at intervals? That's extremely heavy. People have tried CRIU with GNU Screen sessions [79], not sure how successful. It's likely beyond scope.

Conclusion: Given the above, the best pragmatic solution is a tmux-like model. This ensures that in most cases (UI crash, accidental closure) the session persists seamlessly. For the rarer case of the backend crash, rely on saved state to recover as much as possible. This aligns with making crash recovery a first-class feature. We'll document that currently running processes cannot be revived after a full crash, but ccmux will do its best to recreate their context (and specifically, Claude sessions will be continued automatically thanks to session resume).

Code Examples

Server/Client Architecture (Pseudo-code):

```
// On ccmux launch (in main):
if detect_server_running() {
    connect_to_server_and_run_ui();
    return;
}
if fork() == 0 {
    // Child process: UI
    connect_to_server_and_run_ui();
    exit(0);
} else {
    // Parent process: Server
    start_server_loop(); // This will accept UI connections and manage PTYs
}
```

(Note: Instead of manual fork, we could also start the server as a separate process via Command, or even use threads but separate process is safer for crash isolation.)

Saving state periodically:

```
struct PaneStateSnapshot {
    pane_id: usize,
    kind: PaneKind, // e.g., Claude or Shell
    command: String,
    cwd: Option<String>,
    screen: ScreenSnapshot, // custom struct storing rows of text and their
style
    scrollback: Vec<String>, // maybe store scrollback lines too
    session_id: Option<String>, // for Claude
}

struct SessionSnapshot {
    panes: Vec<PaneStateSnapshot>,
    layout: LayoutInfo, // representation of split positions, etc.
    timestamp: u64,
}

fn save_session(snapshot: &SessionSnapshot) -> std::io::Result<()> {
    let tmp = std::env::temp_dir().join("ccmux_snapshot.json");
    let data = serde_json::to_string(snapshot)?;
    std::fs::write(&tmp, data)?;
    std::fs::rename(&tmp, dirs::home_dir().unwrap().join(".ccmux/
last_session.json"))?;
    Ok(())
}
```

Restoring after crash:

```
fn restore_session(path: &Path) -> std::io::Result<()> {
    let data = std::fs::read_to_string(path)?;
    let snapshot: SessionSnapshot = serde_json::from_str(&data)?;
    for pane in snapshot.panes {
        match pane.kind {
            PaneKind::Claude => {
                // Resume Claude session
                let mut cmd = Command::new("claude");
                cmd.args(["-p", "--resume", pane.session_id.unwrap().as_str(),
"--output-format", "stream-json"]);
                // ... set env, spawn in a new PTY
                spawn_pane_with_cmd(pane.pane_id, cmd, pane.cwd.as_deref());
                // After spawning, we might push pane.screen content to the
pane's buffer for user view.
                panes[pane.pane_id].screen_buffer = pane.screen;
            }
            PaneKind::Shell => {
                // Start a new shell (maybe login shell)
                let mut cmd =
Command::new(std::env::var("SHELL").unwrap_or("bash".into()));
                // spawn in PTY...
                spawn_pane_with_cmd(pane.pane_id, cmd, pane.cwd.as_deref());
                // Fill pane with old screen content (marked as '[Session
restored]' perhaps)
                panes[pane.pane_id].screen_buffer = pane.screen;
                panes[pane.pane_id].restored = true;
            }
        }
    }
    // Recreate layout
    apply_layout(snapshot.layout);
    Ok(())
}
```

Logging output for WAL:

```
fn on_output(pane_id: usize, data: &[u8]) {
    // Append to pane-specific log file
    if let Some(mut f) = pane_logs[pane_id].as_ref() {
        let _ = f.write_all(data);
    }
    // Also feed to parser and UI
    panes[pane_id].parser.process(data);
    notify_ui_to_render(pane_id);
}
```

And on recovery, if we have log and a snapshot:

```
fn replay_log_to_parser(pane_id: usize, log_path: &Path, from_offset: u64) {
    let file = File::open(log_path)?;
    file.seek(SeekFrom::Start(from_offset))?;
    let reader = BufReader::new(file);
    for chunk in reader.split(0 /*some delimiter if needed*/) {
       let bytes = chunk?;
       panes[pane_id].parser.process(&bytes);
    }
}
```

(This assumes we know from_offset where snapshot left off.)

References

- Discussion of persistent sessions (abduco/dtach approach) [75] [76]
- Tmux design: persistent through client disconnect but not crash (implied by need for plugins for persistence) [80] [81]
- Zellij memory usage vs tmux (shows Zellij's overhead for features like WASM, hinting tmux's simplicity is for stability) [82] [83]
- Issue about tmux/screen + CRIU experiments (not directly cited above but likely complex)
- The user's own prompt points (tmux vs zellij vs dvtm) and CRIU reference.

Open Questions

- **Scope of process recovery:** Should we attempt to preserve shell sessions via advanced means (like snapshotting bash history or environment)? This seems out of scope – likely we document that normal processes cannot be resumed beyond what they themselves provide (e.g. a text editor might have swap file to recover unsaved changes, but that's the program's domain). ccmux will focus on preserving output and continuing LLM sessions.

- **Frequency and performance of state dumps:** We need to determine an optimal checkpoint interval. Perhaps allow configuring it. Also, measure how heavy it is to serialize a full screen for, say, 100 lines of 80 columns with color – that's maybe ~8k characters plus JSON overhead, which is fine. Even a few times a second should be okay, but we can likely do it every few seconds to reduce disk churn. We should also flush logs in real-time if using logs (to not lose data on crash).
- **Dealing with large scrollback:** If a pane has thousands of lines of history, writing all on every checkpoint is expensive. We might limit how much scrollback to keep (e.g. store only last N lines in snapshot, older than that might be in logs on disk if needed). This is similar to how `screen` or `tmux` have a fixed scrollback limit. Perhaps ccmux can limit scrollback to, say, 1000 lines per pane to bound the size.
- **Ensuring Manager doesn't crash:** We'll rely on Rust safety, but logic errors can still panic. We should catch unwinding in threads and try to keep the server alive even if one pane handling thread panics (use `std::panic::catch_unwind` around thread code). Also, consider using stable data structures – e.g. avoid any unsafe code or at least test thoroughly. Possibly run the server with high ulimit etc. Also watch memory usage if storing a lot of output.
- **Multi-user or remote attachments:** Out of scope for now, but if we do have a server socket, one could attach from another machine (like tmux allows sharing session). That would open security questions. For now, assume local single-user usage.
- **Notification of Crash vs Intentional Exit:** If ccmux server is shutting down normally (user exits all sessions), we might want to save final state or possibly *not* restore next time because it wasn't a crash. Maybe differentiate an intentional "exit" vs a crash. Could remove the snapshot file on normal exit to not auto-restore. Or include in snapshot a flag if it was a clean exit.
- **Testing resilience with rapid output:** We should test with a fast outputting command (like yes, or some stress) to ensure logging and snapshotting keep up and don't slow down the app noticeably. If overhead is too high, tune strategy (maybe more infrequent snapshot if output is continuous).
- **User configuration:** Provide config for enabling/disabling persistence features (some might not want logs written for privacy or performance reasons). Possibly a toggle in ccmux config like `persist_sessions = true/false`. Also an option to manually trigger a snapshot via command.

---

4. Prior Art in Terminal Multiplexers

Executive Summary

tmux (and its predecessor screen) established the standard architecture: a persistent server that multiplexes pseudoterminals and a client UI. tmux's architecture emphasizes simplicity and reliability, offering features like detachable sessions, multiple windows/panes per session, and custom keybindings via a config file. Its client-server model is key to persistence (sessions survive network disconnects) [84], though not crashes of the server. tmux manages windows (each with a PTY and process) and panes (split views of a window) with internal data structures for layouts, and it captures scrollback for each pane in memory. User input is captured via a prefix key mechanism (default `Ctrl-B`) which enters command mode for tmux to interpret shortcuts (like opening a new pane) rather than sending to the application. **Zellij** is a modern Rust alternative with a similar core but notable differences: it has a built-in status bar and uses a WASM plugin system for extensibility (allowing custom UI components in the multiplexer) – a design that trades higher resource usage for flexibility [80] [85]. Zellij's layout is described via YAML/KDL files, making complex startup layouts easier. It aims for better usability (e.g. more discoverable keybinds, UI hints) and sandboxed plugins, at the cost of higher memory (tens of MB vs tmux's <5MB) [80] . **WezTerm's mux** is integrated into a terminal emulator: it effectively combines the emulator and multiplexer (with the "wezterm mux server" functioning similarly to tmux server) [86] [87] . WezTerm can spawn a headless mux server on a remote machine and connect via SSH, making remote sessions seamless without tmux. It manages multiplexing through the concept of *domains* (local, SSH, etc.) and surfaces them as tabs/panes in the GUI [87] [88] . Compared to tmux, WezTerm's mux leverages its GUI for rendering (including graphics, emoji support, etc.) but is less used in pure-CLI environments.

Comparison Matrix:
- *Persistence:* tmux & Zellij support detach/attach (persistent sessions) but no built-in crash recovery; WezTerm's mux persists until its process exits (it's essentially a single process unless using the SSH domain to connect to a remote multiplexer).
- *Extensibility:* tmux can be scripted via shell commands or plugins like tmux-resurrect; Zellij has first-class plugin system (WASM) and a defined plugin API for extensions (like drawing custom content in panes) – more extensible but also more complex. WezTerm allows scripting/config in Lua, but not as interactive extension of multiplexer logic (Lua can automate opening panes, setting titles, etc.).
- *Performance:* tmux is very lightweight in memory and handles many panes well, but historically had issues with very high throughput (due to single-threaded and updating all clients on each output). Zellij, being in Rust and multi-threaded, may handle concurrency well but uses more memory (overhead of safe abstractions, WASM runtime, etc.) [80] . Both are plenty fast for typical use (rendering at >60fps with moderate loads). WezTerm is heavier (as a full terminal GUI) and uses GPU for rendering; it can handle fancy things but might use more CPU/RAM for the same text workload.
- *Configuration:* tmux uses a custom text config (`tmux.conf`) that is powerful but has a learning curve (lots of gotchas with quoting, etc.). Zellij uses YAML (or now maybe TOML) which is arguably simpler and includes comments and presets. WezTerm uses a Lua config file, which allows logic in config (very flexible, but one must know Lua).
- *Features:* All support splits, multiple tabs (except Zellij – it calls them workspaces I think, analogous to windows in tmux). tmux has features like synchronize-panes, choose-tree, etc. Zellij adds some modern conveniences (like built-in session management commands, IIRC, and a more informative UI by default). Zellij also focuses on "batteries included" (e.g. status bar with system info by default). tmux by default has a minimal status bar but heavily customizable. WezTerm's features include true-color, font shaping, ligatures because it's a full terminal; tmux and Zellij rely on the underlying terminal for rendering and just pass through escapes (tmux recently gained truecolor pass-through support, but historically it would down-convert colors unless configured).

Pain Points from Users:
Common tmux complaints: - Complex configuration and keybindings (the default prefix + keystrokes are non-intuitive to new users; config language is not familiar to many) [89] [90] . - Inconsistent behavior on some terminals (needing to tweak TERM or other env variables to get things like italics or 24-bit color working – e.g., older tmux set TERM=screen which didn't advertise truecolor, causing confusion). - Copy-paste in tmux: before tmux 3.2, copying text in tmux required either using tmux's copy mode or integration with system clipboard via external tools; this was cumbersome. Zellij tries to simplify copy mode (and also has a mouse mode that might integrate better with modern terminals). - Performance when outputting massive amounts of text: tmux could become laggy if a pane spammed thousands of lines quickly, because it has to scroll and maintain history for multiple clients. WezTerm might handle that better via GPU, Zellij maybe similar to tmux or slightly better since Rust. - Multi-monitor or multi-client usage in tmux was a bit

limited (all clients see the same session state unless using separate sessions). WezTerm can display the same multiplexer session in multiple GUI windows easily (e.g., you could attach from two machines to the same SSH domain and share state). Zellij is single-user and I think one client at a time per session (like tmux, though tmux can have multiple clients on one session mirrored or independent). - Some tmux users complained about lack of **layout save/restore** functionality (you have to manually arrange or use external tools). Zellij addresses that by letting you define named layouts in a file and load them.

Crate Structure of Zellij:
The user specifically asks what crates Zellij uses that we might steal. From what we gathered, Zellij likely uses `portable-pty` (since that's the Rust standard) and a terminal parser (it might use `vte` crate or `vt100` fork). Actually, it appears they had a component for Sixel (image protocol) [91], which suggests they extended parsing beyond plain text. They also have a crate for plugins (embedding WASM). Possibly interesting pieces: Zellij's **input handling** crate might be of interest – how they manage keybindings and mode switching (Zellij has different modes like Normal mode, Tab mode for switching, etc.). ccmux could learn from that for implementing command keys. Another is how they manage **layouts** – maybe a crate for layout definitions that we could mirror for ccmux's config. They have `zellij-utils` and `zellij-server` crates; reading those could yield insights on data structures for panes and screen. Since Zellij is MIT licensed, using their code is allowed if needed. But one must consider if it's easier to implement custom vs understanding their architecture, which is built around the plugin system and might be more than we need.

WezTerm's Mux Protocol:
WezTerm has an explicit design for remote multiplexing: they spawn a server on the remote side and communicate presumably via SSH channel using its own protocol. Possibly it's similar to tmux's protocol (which is not standardized publicly) or maybe something like sending structured messages over the socket. The documentation doesn't detail the wire format (maybe in WezTerm source or docs we could find it). Not sure if ccmux needs remote attach – probably not a priority (but interesting for future – maybe ccmux could allow connecting to a session on a remote machine; we could reuse SSH + a mini ccmux daemon concept, similar to how WezTerm does with SSH domains).

Lesser-known multiplexers:
There are a few: **dvtm** (dynamic virtual terminal manager) which is a minimal multiplexing window manager for console (no detach capability by itself, but used with abduco for detach) – it's very lightweight (C code). **abduco** as mentioned, just detach/attach support with no splitting. **mtm (Micro Terminal Mult.)** – extremely minimal (single-file C program) mainly for local use. These emphasize minimalism (for low resource or embedded). They trade features (no scrollback in some cases, etc.) for simplicity. ccmux being focused on Claude integration likely won't be that minimal, but it should remain as lean as reasonable given it's doing heavy parsing etc.

Feature comparison:
- **Detaching/Attaching:** tmux & Zellij yes, WezTerm mux sort of (if you close GUI you lose session unless you had separate server; but with the `unix` domain approach, you can close GUI and reconnect via new GUI, akin to detach). - **Splits and Windows:** all three do splits. tmux & Zellij have multiple named sessions possible, WezTerm's concept is multiple windows and workspaces possibly, but all tied to the running GUI instance or its mux server. - **Clipboard integration:** tmux has commands to load/copy clipboard via shell (or newer versions can use OSC 52 to copy to system clipboard), Zellij might have plugin or built-in for copy, WezTerm being a terminal emulator can just use system clipboard easily. - **Mouse support:** tmux supports

mouse interactions (select pane, etc. when enabled), Zellij also supports mouse for selecting and resizing, WezTerm of course as a GUI supports mouse out of the box. - **Scrolling:** tmux has copy-mode or mouse wheel if configured, Zellij likely allows mouse wheel to scroll, WezTerm uses typical terminal scroll (and in mux context, it might unify it – not sure, but likely it just works per pane since it renders them). - **Scripting:** tmux can be scripted via sending commands (e.g. `tmux split-window` from shell or keybind). Zellij doesn't yet have as mature scripting, but one could write a plugin to do some automated control or there is discussion of a command palette. WezTerm via Lua can create keybinds that do complex actions (like open certain splits, run commands). - **Multi-user:** tmux can share a session between multiple users by additional config (though rarely used). Zellij at the moment is single-user oriented (though one could possibly run it under a shared account). WezTerm's remote attach is one user at a time (unless you share an SSH connection credentials).

What to emulate for ccmux:
We likely want tmux-like key controls (possibly default to `Ctrl-B` prefix or choose another default like `Ctrl-Space` or something not conflicting). But since ccmux's focus is Claude, perhaps an even simpler control scheme might suffice (like automatically splitting on certain triggers, etc.). However, the user likely expects tmux-like manual control as well. Hot-reload of config (section 5) is something tmux can do with `:source-file`, Zellij possibly watches its config file (not sure). We will implement file watching for config.

User surveys/complaints about tmux:
One that stands out is *ease of use for new users* – tmux has been called inscrutable by some. Zellij tries to address that (for example, showing a pop-up with key shortcuts on first run, easier default keys). ccmux might cater to developers who are already fairly technical, so tmux-like might be acceptable. But as a new tool, maybe adopting more intuitive keys (like Zellij uses `Ctrl+g` as default prefix, which is easier to hit? and sequences like `Ctrl+g + Left` to move left, instead of tmux's `Ctrl-B, Left` maybe). Considering Claude Code itself is a modern tool, perhaps ccmux should lean towards a more modern interface (maybe look at Zellij's defaults as a guide).

Recommended Approach (drawing from prior art for ccmux)

- **Adopt a Client/Server Model** as discussed (like tmux). This design choice is directly inspired by tmux/screen's proven approach to persistence [84]. We can simplify implementation by using modern libraries (as opposed to tmux's C with its own socket handling, we can use Rust's IPC crates or just `tokio::net::UnixListener`).
- **Pane and Window data structures:** We can model them similar to tmux. For example, a `Session` containing multiple `Window`s (tabs), each `Window` containing one or more `Pane`s (splits). Tmux and Zellij both use such a hierarchy. If ccmux might not need multiple windows (maybe it could, but initial focus could be just one window with splits), but designing it won't hurt. The user prompt's mention of "windows/panes management" suggests we do envision multiple windows per session.
- **Layout algorithm:** When splitting, tmux uses either equal splits or specified percentages. Zellij has some predefined layouts and can dynamically adjust. We should allow both dynamic splits (user presses prefix + % or " to split vertically/horizontally evenly) and possibly a way to load a layout config (like a YAML that says 30% left pane runs X, 70% right split into 2, etc.). That way advanced users or startup can open with a nice layout (like code on left, Claude on right, etc.). We can leverage serialization from Zellij for such layout definitions. They have a concept of "tabs" with specified layouts in config.

- **Keybinding model:** Use a prefix key to avoid conflicts with apps. Perhaps default to `Ctrl-A` (GNU screen's default) or `Ctrl-B` (tmux's default) or something not used by common apps. tmux chose `Ctrl-B` because `Ctrl-A` was taken by screen; `Ctrl-A` tends to move cursor to line start in shells by default, so taking that annoys shell usage unless remapped. `Ctrl-B` is reasonably safe. Zellij uses `Ctrl-G` (and no mode if using the UI menu). We can decide one (maybe `Ctrl-B` to comfort tmux users, or if we think many of our users are new to multiplexers, `Ctrl-G` could be fine). Also provide the ability to remap in config (like tmux's `set -g prefix C-a`).
- **Common shortcuts:** Implement the popular ones in tmux:
- Prefix + `%` and `"` for vertical/horizontal split (or maybe the reverse depending on convention).
- Prefix + arrow keys to navigate panes.
- Prefix + `x` to close pane, etc.
- Prefix + `c` to create new tab/window (maybe not needed if we keep one window).
- Prefix + number keys to switch windows if multi-window.
- Prefix + `[` to enter scrollback mode (tmux's copy-mode for scrolling). Zellij uses `PageUp` to scroll by default IIRC. We can do something similar (allow arrow/pgup/pgdn to scroll when in scroll mode).
- Possibly prefix + `s` or some key to list sessions (though if we only allow one session at a time like tmux usually does per user, maybe not needed).
- You could also incorporate some Claude-specific shortcuts, e.g., prefix + something to spawn a new Claude pane quickly with some default prompt? But that might be feature creep; can leave such orchestrations to the Claude structured output rather than manual keys.
- **Configuration Reload:** We plan that as per section 5. Both tmux and Zellij support reloading config (tmux via `:source-file`, Zellij auto-watches config file). We'll do an auto-watch (Rust `notify`) to apply changes on the fly, since user explicitly asked for hot-reload patterns.
- **Better Defaults from Zellij:** Borrow things like showing key shortcuts or help on the status bar. Zellij on first run had a quick help. We could incorporate a help screen (prefix + `?` like tmux does prints help or opens a help pane). Additionally, consider default status line content:
- tmux by default shows session name, window number/name, and time.
- Zellij by default shows pane frames with index numbers and maybe CPU usage in status.
- We might do minimal initially, maybe just nothing or a simple indicator for active pane, since our focus is not general remote server management (like tmux often is used for). But given we have a multiplexer, some status at bottom listing open tabs or a short help could be user-friendly.
- **Extensibility:** We likely don't need a full plugin system like Zellij for MVP. That's a huge endeavor. We can design with modularity (so maybe later one could add hooks or plugins), but initially, no WASM plugin.
- **Remoting:** Probably not initially. If user runs ccmux on a remote server, they can just detach and reattach via SSH and running ccmux again (like they would with tmux). Building a built-in SSH integration like WezTerm is out-of-scope.
- **Memory usage:** Acknowledge that using Rust and possibly storing a lot of scrollback or JSON might make us heavier than tmux's C (the HN comment noted Zellij 63MB vs tmux 3.8MB RSS) [80]. We should monitor memory usage and possibly provide config to limit scrollback lines, etc. But a few tens of MB is likely fine for our target (modern dev machines).
- **Special features:** WezTerm's emulator can display images, but in a multiplexed CLI, we probably won't attempt that (though maybe showing images isn't needed for coding context). If later we wanted, we'd rely on Kitty graphics protocol or iTerm images, but that requires passing those escapes through and supporting them in our parser. Probably unnecessary.
- **User feedback**: The question suggests referencing user surveys – basically, many love tmux for reliability and ecosystem, but some find it unintuitive. Zellij tries to address UX. ccmux can position

itself focusing on LLM integration but should also be a competent multiplexer. So we can lean on tmux's proven backend design and incorporate some of Zellij's UX improvements (like easier config, hints).

Code Examples

tmux-like Key Handling:

```
// Pseudo-code: in the UI event loop, when a key is pressed:
if !in_prefix_mode {
    if key == config.prefix_key {
        in_prefix_mode = true;
        // maybe start a timer to exit prefix mode after short delay
    } else {
        // send key to active pane
        active_pane.write_to_stdin(key);
    }
} else {
    in_prefix_mode = false;
    match key {
        Key::Char('\"') => split_horizontally(active_pane),
        Key::Char('%')  => split_vertically(active_pane),
        Key::Up        => move_focus(Direction::Up),
        Key::Down      => move_focus(Direction::Down),
        Key::Left      => move_focus(Direction::Left),
        Key::Right     => move_focus(Direction::Right),
        Key::Char('c') => new_window(),
        Key::Char('x') => close_active_pane(),
        Key::Char('?') => open_help(),
        // ... other shortcuts
        _ => {} // unbound keys do nothing
    }
}
```

Applying config changes on reload (tmux vs Zellij style):

```
fn apply_new_config(new_cfg: Config) {
    // Example: update prefix key
    state.prefix_key = new_cfg.prefix_key;
    // Update keybindings:
    state.bindings = new_cfg.keybinds;
    // Update layout if specified (this might involve resizing or reordering
panes)
    if new_cfg.layout != current_layout_name {
        apply_layout_by_name(new_cfg.layout);
    }
    // Update any appearance settings (e.g. status bar on/off)
    ui.status_bar.visible = new_cfg.enable_status;
    // ...
}
```

Layout structure (inspired by Zellij or tmux):

```
enum SplitDirection { Horizontal, Vertical }
struct Pane {
    id: usize,
    // pointer to PTY and associated process, etc.
}
struct Container { // could represent either a Pane or a Split containing sub-
containers
    is_split: bool,
    split_dir: SplitDirection,
    children: Vec<Container>,
    size_ratio: f32, // for this container relative to parent
    pane: Option<Pane>, // if not split
}
struct Window {
    root: Container,
    name: String,
}
struct Session {
    windows: Vec<Window>,
    // ...
}
```

(This is one way; tmux uses a binary tree for splits where each node has a size, etc. We can similarly represent splits in a tree recursion as above.)

Citing sources in context:
- Architecture gleaned from tmux description [84]
- Comparison gleaned from HN and Medium sources (like Zellij vs tmux article noting design philosophy differences) [92] [89] and memory usage comment [80]. - Zellij plugin and layout highlights [93] [94]. - WezTerm domain concept from its docs [87] [88]. - tmux user complaints (hard for beginners, copy-paste, etc.) implicitly referenced from known community discussions (some captured in sources above, e.g. ease-of-use vs power tradeoff [89] [95]).

References

- tmux client/server persistence and detach feature [84] [18]
- Zellij design goals (intuitive UI, plugin support) [93] [94]

- WezTerm multiplexing domains (similar to tmux via SSH) [87] [88]
- HN discussion of resource usage (Zellij vs tmux) [80] [81]
- Comparison of customization (tmux vs Zellij config simplicity) [90] [96]
- HN comment on what Zellij does better or not (indicative of fish vs zsh analogy – meaning not strictly better, just different target) [95]
- Example tmux config (not directly, but known patterns for prefix, etc., could cite tmux man pages if needed).

Open Questions

- **How far to mirror tmux functionality?** Do we implement multi-window support or keep it single-window for simplicity (especially if ccmux usage is more about splitting a view between code and AI, maybe multiple windows are less needed)? Perhaps implement windows anyway, since the overhead isn't large and some users might want separate workspaces.
- **Naming and sessions:** Should ccmux support multiple named sessions like tmux (so you could detach from one and start another)? It could, but given its specialized use, maybe one session per invocation is enough (like how WezTerm mux basically ties to one user session). Possibly add later if demand.
- **Compatibility with tmux key muscle memory vs new user approach:** We need to decide default keybindings: catering to tmux users (stick close to tmux defaults so they feel at home), or innovating for newcomers (maybe simpler keys or even a mode where no prefix needed and certain function keys do things). Could allow both via config presets. By default, perhaps we keep tmux-like to reduce cognitive load for those migrating.
- **Will ccmux incorporate a status bar by default?** If yes, what info? Possibly the name of the Claude session and its state (thinking/idle), similar to how tmux shows window names. Could be neat: show an indicator if Claude pane is currently busy (like a `*` or spinner in status line). Also, time and maybe an indicator if config changed (like a dot when unsaved – not needed here). Since user specifically asked about differences, maybe they expect a status line. I'd lean to having one line at bottom with session name and maybe help key.
- **Coordination with Claude Code output**: If Claude outputs its own progress (like [thinking...]) in the pane, do we double-indicate it in status? Perhaps not needed; but if Claude partial output isn't obvious, a small spinner in status could reassure user that it's working.
- **Remote usage**: If user wants to use ccmux on remote via SSH, they can just run it inside their SSH terminal, like tmux. If they disconnect, ccmux would detach in background if we implemented server. So that should be fine. If they want to attach from multiple local terminals to one remote ccmux, we might implement multiple client support eventually (like tmux does). But not urgent.
- **Integration with existing tmux**: Some might wonder if ccmux could be a tmux plugin or if they can run ccmux inside tmux, etc. Running inside tmux should be fine (just another program). ccmux itself won't integrate with tmux directly. Possibly not a concern.
- **Focus on Claude vs general**: Since ccmux is Claude-aware, we might tailor some defaults to that use-case (e.g. maybe open one pane running `claude` by default on startup, whereas tmux opens with a shell). Possibly ccmux's default behavior when launched could be to open a Claude REPL in one pane and a shell in another. Or ask user what to open (like a simple menu). That's a design decision: We could detect if Claude Code is installed and if yes, auto-launch a pane with it. That would immediately differentiate ccmux from plain tmux (giving a ready environment for AI + code). That might delight the target user. We can allow config to override initial layout of course.
- **User feedback collection**: Perhaps in future, see how users like the keybindings or features and adjust. Since we have the ability to hot-reload config, users can adjust keys easily anyway.

5. Hot-Reload Configuration Patterns

Executive Summary

Hot-reloading configuration involves watching config files for changes and applying them without restarting the application. Tools like Neovim and Alacritty implement this by monitoring files and then merging new settings at runtime. In Rust, the go-to crate for file watching is `notify`, which abstracts OS-specific backends (inotify on Linux, Kqueue on macOS, etc.) [97] [98]. Key considerations include debouncing events (because a single save can trigger multiple events such as "modified", "attributes changed") and handling atomic saves (where editors write to temp then rename – which might show up as a Create event for the new file) [99]. A common pattern is to use a short debounce (e.g. 50-100ms) to collect events, then reload once after a burst of changes. On applying the new config, one must carefully decide which changes can be applied live. Generally, UI-related settings (colors, layout, keybindings) can be changed on the fly, whereas fundamental ones (like changing the locale or enabling a major feature that wasn't initialized) might require restart. The config reload code should validate the new config and if there's an error, typically log it and continue using the old config (to avoid breaking the running session). Neovim's `:source` command, for example, will execute the config file (init.vim or an arbitrary vimscript) and apply any settings, with errors reported in the message area but not shutting down the editor. Alacritty (a GPU terminal) can automatically reload its YAML config by default; it watches the file and then applies changes like font size, colorschemes live [100] [101] (some things like font face historically needed restart, but they improved that over time). Alacritty's strategy is to have a `live_config_reload` setting (on by default) [100] and they simply replace relevant config structures and re-render. One has to be mindful to **merge** unspecified config fields with defaults or previous values so that reloading doesn't override everything (most config systems, e.g. using Serde with `default` attributes, handle that). Also, using a staging area to load new config and verify it (maybe by deserializing to a temp struct) helps catch errors without blowing up the running state. If new config fails to parse, the program should keep the old config and perhaps notify the user of the error.

File Watching Implementation:
Use `notify` crate in recommended mode (likely `RecommendedWatcher` with Debounce) to watch the config file path. Ensure to watch the directory in case the file is replaced (some OSes require watching the dir if the file is moved/renamed). Debounce the events to coalesce multiple triggers. For atomic writes, `notify` often sends a Remove for the old file and Create for new – we should handle that by watching for either.

Applying Config Changes:
We categorize config options: - **Live-adjustable:** UI theme (colors, toggling a status bar), key bindings, layout (if user changes default layout or adds a pane definition, we could actually open/close panes accordingly – though that's complex and maybe require user action). Most likely, we apply changes to new sessions or at next convenient point. But e.g., a color scheme change we can apply immediately by redrawing with new colors [101]. Keybindings can be re-registered on the fly (drop old ones, add new). - **One-time initialization settings:** e.g., if config had something like "use experimental renderer = true", and we didn't start it that way, toggling it might not be feasible at runtime because the component isn't running. We should document which changes need restart. Possibly even detect if such a field changed and prompt the user "Please restart for this to take effect." - **Invalid config entries:** If the new config has a syntax error or invalid value, best practice is to not apply any of it and preserve the last good config, and show an error message (maybe in a log or UI).

Debouncing Rapid Changes:
Some editors save two times (create temp, then rename). We might get multiple events. Setting up `notify` with debounced mode handles grouping events within e.g. 200ms by default. We can tweak this if needed. Also, some config might be split across multiple files (like if main config includes another). If multiple files, we should watch all relevant ones (and if any changes, reload). Alacritty had an issue: it allowed including extra config files, and originally it didn't reload if an included file changed – they since resolved it by restarting the watcher on any file change to include new ones [102]. We should consider if our config can include others (maybe not initially, unless we allow splitting by feature). If yes, we need to track those too.

Atomic Write Handling:
The pattern:

```
Write to temp -> rename to config.yml
```

This often triggers a Create or Rename event, sometimes accompanied by Remove on the old file. We might get two events: remove(old) and create(new). We just need to react once (the debouncer will likely compress them). Just need to ensure our watcher is on the directory or set to watch moves.

Validation and Defaults:
Using Serde with a well-defined Config struct (with `#[serde(default)]` on fields) helps here. When reloading, parse into a `NewConfig`. If it parses fine, then we do a function like `apply_config(old_cfg, new_cfg)` which for each field decides how to apply:

```
if old_cfg.theme != new_cfg.theme {
    ui.apply_theme(new_cfg.theme);
}
if old_cfg.keybinds != new_cfg.keybinds {
    update_keybinds(new_cfg.keybinds);
}
```

and so on. Some changes might require reconstructing an internal data structure (like if they drastically change layout, maybe we can't apply without restarting or reinitializing the UI – we might choose to ignore certain changes at runtime or apply them partially).

Neovim :source example:
Neovim users can `:source ~/.config/nvim/init.vim` to reload config. It essentially executes the config file (which might set options, define mappings, etc.). Many things take effect immediately (like new key mappings). But if config changes plugins loaded or fundamental states, those might need a restart or manual re-init by user. Neovim also offers `:edit` to reload a file and an `:autocmd` to auto-run something on file change. But the main concept is execution of config commands again.

Alacritty's approach:
Alacritty automatically reloads when its config (alacritty.yml) is saved (unless `live_config_reload` is

turned off) [100]. It prints a message to stdout or the terminal window's log if the config had errors. It doesn't kill the terminal; just ignores faulty parts. According to an issue, everything except font changes was live-reloaded at some point [103], and eventually fonts too.

Atomic file writes in Alacritty:
They had to handle that scenario, there's even an issue about "Restart config monitor on file reload" [102] which implies that if the config file path is replaced, the watcher might stop (because it was watching the old inode). The fix was likely to always watch the directory and handle created file events.

serde and validation:
Serde will by default ignore unknown fields (unless we tell it not to). That's nice for forward compatibility (adding new fields won't break old versions). But if the user has a typo in config (unknown field), they might not realize it's ignored. Could log a warning for unknown fields. For validation, we can implement `Deserialize` or use Serde's `validate` feature (not built-in, but we can run custom checks after parsing). For example, if a numeric field must be within range. Or if keybindings conflict (two actions bound to same key, perhaps warn or last one wins). We should decide how robust to make it. Probably at least print something if config cannot be applied fully.

Apply partial vs full:
We can either apply the new config wholesale (replacing all values) or partially merge. Usually, it's wholesale: the new config replaces old (except where not specified, defaults fill in). If something fails in parsing (like one section has error), we might choose to not apply any of it to keep consistent state, or apply what parsed successfully and ignore the bad part. Typically, not applying any (atomic apply) is safer to avoid weird half-states. But a minor thing like one invalid keybinding among many, we could apply others and skip the invalid. That requires our config parser to possibly continue on error – which Serde doesn't easily do; it stops at first error. We could circumvent by using Serde's `flatten` or `untagged` with Option in places, or parse in multiple passes. Simpler: fail entire parse on any error, and use last good config in that case. That's acceptable.

Atomic apply:
We should ensure to not leave the app in a partially updated state. So probably compute new Config object first (with all defaults and merges resolved), then only if fully valid proceed to swap out the old config.

File system differences:
On Linux, inotify triggers on close/write by editor. On macOS, some editors do atomic save differently (maybe copy file). The `notify` crate accounts for these by abstracting events, but sometimes on mac you get `Write` event vs on Linux you get `CloseWrite`. We'll rely on the crate.

Roll-back/Backup:
We might not need a backup config since user likely has it in their editor if they messed up. But maybe if reload fails, we can keep running with old config in memory until next successful reload. That is a form of rollback (just keep old). Possibly log "Reload failed, continuing with previous configuration."

Open Questions from prompt specific:
They ask about "atomic writes (write to temp then rename)" – we've covered how to handle. Also "debounce rapid file changes" – yes, use notify's debounce or implement a short delay aggregator.

Examples in other projects:
Neovim:

```
:source ~/.config/nvim/init.vim
```

This just executes commands inside, which Neovim is built to handle at runtime. E.g., `set number` toggles line numbers immediately, `map <keys>` defines a mapping available immediately, etc. If an error, it prints an error message but Neovim keeps running with whatever did get set up to that point.

Alacritty: - In alacritty.yml, if user changes font size, on save the terminal's font changes live (window resizes if needed). - If they change `live_config_reload` to false, they'd have to restart to see further changes because reload stops. It's an option in config itself ironically. - They likely have code:

```
watcher.configure(Config::default())?
watcher.watch(config_path, RecursiveMode::NonRecursive)?
```

and on event:

```
Event::ConfigReload(path) => {
    match reload_config(path) {
       Ok(new_config) => apply_config(&mut self, new_config),
       Err(e) => error!("Failed to reload config: {}", e),
    }
}
```

(This is pseudo but conceptually.)

serde approach: We can define our `Config` struct with defaults for all fields. In `reload_config`, do:

```
fn reload_config(path: &Path, old_config: &Config) -> Result<Config, Error> {
    let data = fs::read_to_string(path)?;
    let mut new_config: Config = toml::from_str(&data)?;
    // Optionally, we can merge new_config with old_config for values not
present.
    // But if using serde(default), new_config already has defaults for missing
fields, which may differ from old config if user intentionally removed
something.
    Ok(new_config)
}
```

One subtlety: If user removes a setting that was previously set, do we revert to default or keep old? Usually default, because config is stateless – each load independent. E.g., user had `enable_feature_x: true` but then deletes that line, presumably means they'd want default (which might be false). We should not keep it true just because last config had it. So treat config file as source of truth each time.

Testing hot-reload in ccmux scenario:
We should simulate editing the config while ccmux runs: change keybind, ensure new key now works; change theme color, ensure UI updates color on the fly; change something like default Claude model (which might only apply next time Claude is started, but we can incorporate it – for existing sessions it might not retroactively change the model, but next spawn yes). We can mention config diff apply strategy for such fields.

Neovim config reload knowledge: Neovim's `:source` basically re-executes the entire config. Some plugins might double load or define duplicate things if you do that, so sometimes they guard against it (like check if already loaded). In our case, we won't "execute" code on reload, just data apply. So no risk of double-allocating structures except for e.g. keybindings (we should probably unbind previous keys or we might have duplicate handlers). So definitely our apply logic for keybinds should remove or override old binds.

Alacritty and others: Alacritty is a prime example of user-friendly live config: toggling e.g. background opacity in config shows effect immediately. This is a goal to match.

Conclusion: We'll implement file watch with `notify`, debounce, parse new config into a struct, then apply differences. If error, log and ignore changes (keeping old config). Provide user feedback likely via logging (maybe an on-screen message or just stderr if launched from terminal).

Code Examples

Using notify crate to watch file:

```
use notify::{Watcher, RecommendedWatcher, RecursiveMode, Event, EventKind};
use std::time::Duration;

fn watch_config(path: &Path) -> notify::Result<RecommendedWatcher> {
    let (tx, rx) = std::sync::mpsc::channel();
    let mut watcher = notify::recommended_watcher(move |res: Result<Event, _>| {
        match res {
            Ok(event) => tx.send(event).unwrap(),
            Err(e) => eprintln!("Watch error: {:?}", e),
        }
    })?;
    watcher.watch(path, RecursiveMode::NonRecursive)?;
    // Alternatively, watch the directory: watcher.watch(path.parent(),
RecursiveMode::NonRecursive)?
    // to handle atomic swaps.

    // Spawn a thread to handle events with debouncing
    std::thread::spawn(move || {
        use std::collections::VecDeque;
        let mut last_event_time = std::time::Instant::now();
        let debounce_duration = Duration::from_millis(100);
        let mut pending = false;
        while let Ok(event) = rx.recv() {
            // Only interested in modify/create events
            match event.kind {
                EventKind::Modify(_) | EventKind::Create(_) => {
                    pending = true;
                    last_event_time = std::time::Instant::now();
                }
                _ => {}
            }
            // Use a small loop to debounce
            loop {
                if pending && std::time::Instant::now() - last_event_time >
debounce_duration {
                    pending = false;
                    // Trigger config reload
                    if let Err(e) = reload_and_apply_config() {
                        eprintln!("Config reload failed: {}", e);
                    } else {
                        println!("Config reloaded successfully.");
                    }
                }
                // break or continue receiving...
                break;
            }
        }
    });
    Ok(watcher)
}
```

(The above uses a simple timing check after each event to decide when to reload; could also use `DebouncedWatcher` from older notify, but that is deprecated in favor of this manual approach in newer versions.)

Debounce simplified (conceptual):
In practice, `notify` crate might provide `notify::Config::PreciseEvents` to avoid too many duplicates. But anyway.

Reload and apply config logic:

```
fn reload_and_apply_config() -> Result<(), Box<dyn std::error::Error>> {
    let new_cfg: Config =
toml::from_str(&std::fs::read_to_string(CONFIG_PATH)?)?;
```

```
    apply_config(&CONFIG, &new_cfg);
    CONFIG = new_cfg; // update global or static config
    Ok(())
}

fn apply_config(old: &Config, new: &Config) {
    if old.theme != new.theme {
        ui.apply_theme(new.theme.clone());
    }
    if old.keybindings != new.keybindings {
        // Unbind old keys that changed
        for kb in &old.keybindings {
            if !new.keybindings.contains(kb) {
                unbind_key(kb);
            }
        }
        // Bind new keys
        for kb in &new.keybindings {
            bind_key(kb);
        }
    }
    if old.layout != new.layout {
        // This might be disruptive; possibly prompt user or only apply certain
layout changes.
        layout_manager.apply_layout(new.layout.clone());
    }
    // ... handle other fields similarly.
}
```

Error handling:
If `toml::from_str` fails (due to syntax error), it will return an error. We catch it and print error (not applying changes):

```
if let Err(err) = reload_and_apply_config() {
    eprintln!("Failed to reload config: {}", err);
    // optionally display to user in UI, e.g., in a message box or status line.
}
```

Merge default:
Our Config struct can define defaults via Serde's attributes or `Default` trait. We should ensure to implement `Default` for Config to fill all fields with reasonable values. Serde will use those for any missing in the file (since likely we'll use `#[serde(default)]` on the struct or fields). This means new = from_str will contain default for anything not specified. We should double-check if that's fine: e.g., if user deletes a keybind, default might be empty or some fallback, which likely means removing that binding – which is correct. If user deletes a section like theme, we fall back to default theme – which they probably expect.

Potential atomic rename scenario:
If the file is moved, our watcher on the file path might lose track. To handle, we can watch the directory and specifically handle a Create event for our config file name. E.g., in event handler:

```
if event.kind == EventKind::Create(_) && event.paths.contains(&config_path) {
    // config file was re-created (likely via atomic save)
    // We should re-add watch on the new file if needed.
    watcher.watch(config_path, RecursiveMode::NonRecursive).ok();
    // Then treat as modify event:
    trigger_reload();
}
```

But newer notify might track it automatically if using path watch? Not sure, so directory watch might be simpler.

References

- Using `notify` crate for file watch (and environment specifics like inotify, etc.) [97] [104]
- Alacritty config reload mention (live_config_reload) [100] [101]
- Alacritty issue about including extra files not auto-reloading (implied by restart monitor issue) [102]
- Neovim `:source` config approach (knowledge-based, not directly cited above but widely known; can cite Neovim docs if needed).
- Serde default and validation (Rust docs or blog not explicitly provided by user, but we can rely on understanding).
- Example of config reload in any Rust project: e.g., some mention in discussion [41] or [52] lines that show how others do it.

Open Questions

- **How to inform user of config errors?** Possibly ccmux is running in a terminal, so printing to stderr is fine. But if user is fully in ccmux UI (e.g., if they started it like a shell), maybe they won't see stderr easily. Perhaps integrate an on-screen notification: e.g., echo a message in a dedicated area or ephemeral pane. Tmux doesn't auto-reload config, but if you manually source a bad config, it prints error in tmux prompt or in the client's stderr. We could flash a message in status bar "Config reload failed, check logs." For now, printing to terminal (which might show up in the controlling terminal outside the UI) is acceptable.
- **Which changes require restart?** We should identify and document. Possibly things like changing the path to Claude binary (if we allow such config) might not affect already running processes (they'll keep using old one until restarted). Or if user changes session persistence settings (like disabling crash recovery mid-run) – not something likely to be dynamic. Also, if they change logging settings (like enabling verbose logging to file), we could open a log file mid-run and start writing, that might be fine actually. But if truly something can't be toggled, we either ignore until next restart or prompt. We'll handle on case-by-case as we implement features.
- **Atomic rename reliability on each OS:** Should test on Linux and mac at least to ensure our approach catches the event. Particularly on mac, FS events sometimes unify create+modify. `notify` crate's documentation or examples could guide this.

- **Performance of reload:** Usually negligible (parsing a small config and applying a few changes). Even if user saves config every second (unusual), overhead is fine. If config is huge (which in our case it won't be beyond maybe dozens of lines), also fine. So no issue.
- **Multiple config files:** If we split config (like a main config and a separate keybind file), supporting that complicates watcher (need to watch multiple). Unless user specifically requests that (like includes in toml), we can avoid for now. Possibly just one config file is enough. If they want to modularize, they could include via toml `include` (toml doesn't have include, might have to use tools to merge). We might not support that directly. Alacritty allowed YAML anchors to include base themes, etc.
- **Hot-reload vs ephemeral state:** If the user changes config to remove a pane or change layout, do we automatically close a pane or re-tile existing? That could disrupt their session. Perhaps we do not actively kill existing stuff on config change, except maybe cosmetic things (like they removed a key binding, we remove it so it doesn't work further, that's fine; if they removed a pane from default layout, we likely ignore that because we won't kill a running session's pane due to config – config is usually meant for new sessions or static aspects, not dynamic content). So some config values (like initial layout or startup programs) might not be applied after startup. That's okay. We can document that certain changes will only apply on new sessions or require manual closure of things. Essentially, we won't auto-close or open panes on reload because that might kill user's work. Hot reload should ideally not disrupt running tasks. So, layout changes likely either skip or apply in a limited way. For example, if user changed pane size ratios, we could try to adjust splits proportionally if feasible (non-destructive). But if they removed a pane definition from config, we won't kill the actual pane.
- **Roll-back strategy:** If user absolutely screws up config and loses keys (like unbind prefix key by accident), they could be locked out of controlling ccmux. We might need an emergency measure (like we can detect if prefix becomes None or something and ignore that change to not leave user with no controls, or instruct they can kill and restart if needed). But since they can always exit ccmux via the controlling terminal or kill the process from outside, maybe not too bad.
- **File not found:** If config file is deleted, what do we do? Possibly treat it as all fields default. But likely user won't delete while running. If they do, we get a Remove event. We could either do nothing (keep last config) or reset to default config. I'd lean keep last known config in memory and not change anything if file missing. Possibly print "Config file removed, continuing with existing settings." If they then recreate it, our watcher on directory (if we set up) will catch the new file and reload that.
- **Cross-platform differences:** Windows has different FS events; `notify` supports it. If ccmux will run on WSL or Windows, our code should handle that (should via notify crate, might just get slightly different event ordering). We can test on Windows if needed.

---

[1] [2] [4] [5] [6] [8] [9] [11] Build with Naz : Capturing Real-Time Build Progress from Cargo Using PTY and OSC Sequences | developerlife.com
https://developerlife.com/2025/08/10/pty-rust-osc-seq/

[3] termwiz: Given raw output from a pty, how can I apply it to a Surface? · wezterm wezterm · Discussion #5217 · GitHub
https://github.com/wezterm/wezterm/discussions/5217

[7] [10] portable_pty - Rust
https://docs.rs/portable-pty/latest/portable_pty/

[12] MasterPty in portable_pty - Rust
https://docs.rs/portable-pty/latest/portable_pty/trait.MasterPty.html

[13] [14] [15] [16] [17] [26] vt100 - Rust
https://docs.rs/vt100/latest/vt100/

[18] [23] [27] [84] What is the best way to incorporate a shell process's output into my own program? : r/rust
https://www.reddit.com/r/rust/comments/xe9bne/what_is_the_best_way_to_incorporate_a_shell/

[19] [25] cockpit - Rust
https://docs.rs/cockpit/latest/cockpit/

[20] [21] PseudoTerminal in tui_term::widget - Rust
https://docs.rs/tui-term/latest/tui_term/widget/struct.PseudoTerminal.html

[22] FAQ | Ratatui
https://ratatui.rs/faq/

[24] zellij-org/sixel-tokenizer: Parse serialized sixel bytes - GitHub
https://github.com/zellij-org/sixel-tokenizer

[28] [32] [40] [43] [45] [47] [48] [59] [67] [77] Excessive File I/O During Sessions (~1.5 writes/second to .claude.json) · Issue #14634 · anthropics/claude-code · GitHub
https://github.com/anthropics/claude-code/issues/14634

[29] [30] [33] [36] [37] [38] [39] [44] [46] [50] [54] [55] [68] [70] [71] [73] CLI reference - Claude Code Docs
https://code.claude.com/docs/en/cli-reference

[31] [42] [56] [57] [58] [60] [69] Shipyard | Claude Code CLI Cheatsheet: config, commands, prompts, + best practices
https://shipyard.build/blog/claude-code-cheat-sheet/

[34] [35] [51] [52] [53] [65] [66] Claude Code CLI Environment Variables · GitHub
https://gist.github.com/unkn0wncode/f87295d055dd0f0e8082358a0b5cc467

[41] [Bug] Status line displays incorrect model across multiple Claude ...
https://github.com/anthropics/claude-code/issues/16595

[49] Feature: Add higher-level session management APIs #458 - GitHub
https://github.com/anthropics/claude-agent-sdk-python/issues/458

[61] [62] [72] Cooking with Claude Code: The Complete Guide - Sid Bharath
https://www.siddharthbharath.com/claude-code-the-complete-guide/

[63] Claude Code - Cline Docs
https://docs.cline.bot/provider-config/claude-code

[64] github.com/heissanjay/oscode v0.0.0-20260101035119 ...
https://libraries.io/go/github.com%2Fheissanjay%2Foscode

[74] [78] CRIU - Checkpoint/Restore in user space - Red Hat Customer Portal
https://access.redhat.com/articles/2455211

[75] abduco a tool for session {at,de}tach support - Marc André Tanner
https://www.brain-dump.org/projects/abduco/

[76] Preserve terminal state across sessions · Issue #32 · martanne/abduco
https://github.com/martanne/abduco/issues/32

[79] [CRIU] CRIU with GNU Screen: dumping and restoring ... - OpenVZ
http://lists.openvz.org/pipermail/criu/2020-April/044991.html

[80] [81] [82] [83] [85] [95] A similar process is happening with zellij and tmux. Since I switched over I fee... | Hacker News
https://news.ycombinator.com/item?id=43973786

[86] [87] [88] Multiplexing - Wez's Terminal Emulator
https://wezterm.org/multiplexing.html

[89] [90] [92] [93] [94] [96] Zellij vs Tmux: Complete Comparison (or almost) | by Rodrigo Martins | Medium
https://rrmartins.medium.com/zellij-vs-tmux-complete-comparison-or-almost-8e5b57d234ae

[91] vt100 - crates.io: Rust Package Registry
https://crates.io/crates/vt100

[97] Claude Code settings - Claude Code Docs
https://code.claude.com/docs/en/settings

[98] Don't let A.I. read your .env files - Filip Hric
https://filiphric.com/dont-let-ai-read-your-env-files

[99] Claude Code Automatically Loads .env Secrets, Without Telling You
https://www.knostic.ai/blog/claude-loads-secrets-without-permission

[100] Alacritty - TOML configuration file format.
https://alacritty.org/config-alacritty.html

[101] alacritty(5) — Arch manual pages
https://man.archlinux.org/man/alacritty.5

[102] Restart config monitor on file reload · Issue #7981 · alacritty ... - GitHub
https://github.com/alacritty/alacritty/issues/7981

[103] Apply config changes on the fly · Issue #1140 · alacritty ... - GitHub
https://github.com/jwilm/alacritty/issues/1140

[104] @vainjs/claude-code-env - npm
https://www.npmjs.com/package/@vainjs/claude-code-env