ccstatusline: a tiny tool on a busy code path

Key takeaways

1. Anything on the hot path must be fast, bounded, and self-cleaning.
2. You can’t prevent every hang — give each run a hard budget, then kill it.
3. Hold the leash from outside: an external killer outlives its parent.

127 lines, and two ways they broke

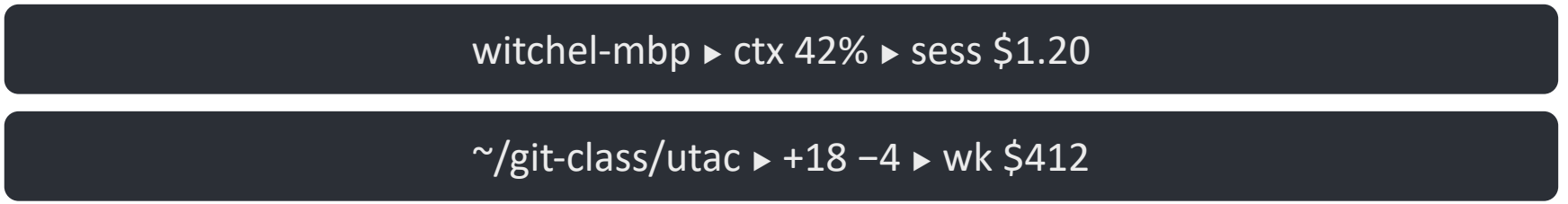
Why a trivial tool needed two rounds of hardening

- A two-line status bar at the bottom of every Claude Code prompt — the whole project is 127 lines: a 65-line config plus a 62-line shell wrapper.
- Those lines run on every prompt, and they failed there twice: once burning a CPU core forever, once hanging forever.
- The arc: what the tool is, what running constantly demands, then the two failures — each one buys a rule about hot-path code.

What a status line is

First principles, before the bug stories make sense

line 1 — situational: which machine, how much context is left, session cost



line 2 — workspace: current directory, git changes, this week's spend

- A strip at the bottom of the window, redrawn on every prompt
  - Same idea as the status bar in vim, emacs, or VS Code — just for the chat window
  - Claude Code runs a small script each time it draws a prompt; the script prints these two lines
- Which means it runs A LOT — so it must be fast and never hang
  - Hundreds to thousands of times per active hour: roughly every Enter, and every few seconds while streaming
  - It must finish in well under a second, and a hung status line freezes all of Claude Code

What ‘hot path’ means

The vocabulary for code that runs constantly

- Hot path = code that runs many times per second
  - Every keystroke, every prompt, every UI refresh — the inner loop of an interactive tool
  - Opposite of a cold path: code that runs once at startup, or once per command
  - The status-line script is on the hot path of Claude Code
- Why it matters: small flaws compound enormously
  - 1 second of slowness × 1000 invocations = 16 minutes of waiting per day
  - 1 MB of leaked memory × 1000 invocations = 1 GB gone before lunch
  - A bug that orphans one process per call = dozens of zombie processes by tomorrow
- The rule of thumb
  - Anything on the hot path must be FAST, BOUNDED, and CLEAN UP AFTER ITSELF
  - ‘It probably won’t hang’ isn’t good enough — at 1000 calls, ‘probably’ happens daily
  - This deck is two stories about the status line failing exactly that test

The config file — settings.json

What goes in the two-line bar, and how it’s styled

```
{
  "version": 3,
  "lines": [
    [
      { "type": "custom-command", "commandPath": "short-hostname", "timeout": 1000 },
      { "type": "context-percentage-usable" }, { "type": "session-usage" }
    ],
    [
      { "type": "current-working-dir" }, { "type": "git-changes" }, { "type":
"weekly-usage" }
    ]
  ],
  "flexMode": "full-minus-40", "compactThreshold": 60,
  "powerline": { "enabled": true, "separators": ["\u25B6"], "autoAlign": true }
}
```

Dev story #1 — why is the laptop hot?

What the user observes
An idle laptop with the fan roaring. Task Manager: five node.exe processes, each pinning a core at 50–100% CPU — ~74 CPU-hours burned in 36 hours of wall time.

The pattern
Each node.exe is a ghost of a Claude Code session that already exited. Every session leaves one more behind; they mount until you kill them by hand or reboot.

The suspect
The node.exe is ccstatusline — the status-line renderer. It is supposed to live for milliseconds per prompt, not forever.

- ccstatusline is a one-shot: Claude Code pipes it session JSON on stdin, it prints two lines, it exits. These renders started and never finished — and nothing cleaned them up.
- The first hypotheses were all wrong (accidental TUI mode? a hung usage fetch?). The experiment that cracked it: orphan the same node + JS from cmd.exe → 0.8% CPU; from MSYS2 bash → 97%. The shell is the culprit.
- That leaves two failures to explain: why does the render never finish, and why does nothing kill it? The next slide answers both.

source: dotfiles/HISTORY.md @ 2026-05-13; commit f1bbcfd

Why the ghost neither dies nor finishes

- Why nothing kills it: node is orphaned
  - bash can't truly `exec` a Windows program — it stays in the tree as node's waiting parent
  - When CC exits, Windows kills it with no signal cascade, and no init exists to reap the leftovers
- Why it never finishes: the read that can't fail and can't succeed
  - node reads stdin until EOF; on Unix a dead writer means instant EOF and a clean exit
  - The MSYS2 pipe instead returns '0 bytes, no error' forever — node retries instantly: a pinned core
- The fix: drain stdin in the shell (`$(cat)`) — node reads a clean copy, hits real EOF, exits.

Dev story #2 — the ghost comes back

Four hours later
A CPU survey after the fix landed finds two fresh ghosts: one blocked on stdin for 8 hours at 0% CPU, one back in the spin loop at 77%.

The real lesson
Draining the pipe removed one way to hang — not hanging itself. You cannot enumerate every way a render might fail to finish.

The one-line fix
`timeout -k 1 2 node …` — every render gets a 2-second budget and is killed when it expires. Stop preventing hangs; bound them.

- Why not have the wrapper kill its own child? Windows terminates the wrapper hard — SIGKILL-style, so its cleanup code never runs. A dying process can’t be its own executioner.
- `timeout` works because it is an independent process with its own countdown: when the wrapper dies, timeout is orphaned too — and still kills node when the 2 seconds expire.
- The budget is safe: a healthy render finishes in under 500 ms, so 2 s clips nothing real. Worst case is 3 retries × 3 s = 9 seconds of blank bar — never a freeze.

source: dotfiles/HISTORY.md @ 2026-05-13; commit b772ff1

The wrapper — 62 lines of hot-path armor

scripts/ccstatusline-cmd: both fixes, plus two more defenses

macOS / Linux: the easy path
One line: `exec ccstatusline`. The POSIX side never misbehaved, so the wrapper stays out of the way.

Windows (MSYS2): the defensive path
Every defense the two stories bought: drain the pipe, bound the render, retry on crash, fail to a blank bar.

- Each line of defense exists because something actually happened
  - Drain stdin first (`$(cat)`) — story #1: node never reads the pipe that lies about EOF
  - Run node under `timeout -k 1 2` — story #2: every render has a hard budget, enforced from outside
  - Retry up to 3× on crash — a flaky native crash killed ~25% of renders; retrying drops that to ~1.6%
  - Exit 0 even on total failure — a blank bar beats an error message where the status line should be

Key takeaways

1. Hot-path code must be fast, bounded, and self-cleaning. At 1,000 runs a day, ‘probably won’t hang’ means it hangs daily.
2. You can’t enumerate the ways code can hang, so don’t try to prevent them all — give every run a hard time budget and kill it on overrun.
3. Hold the leash from outside: an independent killer (`timeout`) keeps counting after its parent dies; in-process cleanup dies with the process.