



The Multiplexer That Knows Its Agents

Herdr against tmux for concurrent coding agents, and compiling out the phone-home

Paper: PT-R-2026-030 v1.0

Author: PureTensor Ltd

Date: 30 September 2026

Status: Published

Abstract

We run several coding agents at once, Claude Code and Grok CLI, as workers under an orchestrating agent. Until now each worker turn has been a fresh headless process launched inside a tmux window, with completion signalled by process exit. Herdr is an open-source (Apache-2.0) terminal multiplexer written in Rust that recognises the agent running in each pane and reports a lifecycle state for it. We asked one question, fixed in advance by the operator: does Herdr 0.9.1 beat tmux on time saved, resources saved and convenience? We asked about those three axes and no others.

We compared three arms driving identical agents, prompts and workspaces on one 64-thread compute node. Arm A was our current one-shot practice under tmux 3.4. Arm B was the strongest fair tmux baseline we could build: long-lived interactive agents driven by typed keystrokes and screen polling with hand-written regular expressions. Arm C was the same long-lived agents under Herdr. We ran five measured rounds of a completion workload (90 turns per arm) and a blocked-prompt workload (15 prompts per arm). Hypotheses were pre-registered before any measured run. We used a hook inside Claude Code to record the true end of every Claude turn.

Herdr noticed a finished Claude turn with a median lag of 0.22 s, against 2.43 s for the polled tmux arm and 0.54 s for one-shot. It noticed all 15 blocked prompts, against 14 of 15 for polled tmux and none for one-shot. Median time from a blocked prompt to a verified result was 10.5 s under Herdr, 17.1 s under polled tmux and 17.9 s under one-shot. Total CPU per round was 122 CPU-s under Herdr, 124 CPU-s under polled tmux and 203 CPU-s under one-shot. Herdr's own server used 22.6 MB resident memory against 6.1 MB for tmux. It needed no agent-specific regular expressions, where the tmux interactive arm needed five. Six of our eight predictions held. Round wall-clock favoured Herdr (34.6 s median against 41.2 s and 54.8 s), but slow Grok turns dominate it and it is too noisy to rest on.

The operator made a pilot conditional on Herdr not contacting its authors' servers. Upstream 0.9.1 does contact them: three requests in a 45 s probe window, repeating every 30 minutes. Every network path in the source goes through one helper that shells out to curl. We patched that helper and two configuration defaults (9 changed lines in 2 files), built from source, and observed zero curl calls and zero non-loopback connections under the same probe. A scoped pilot of the patched build began on the day of measurement.



1. The Question, Asked Narrowly

Our coding workers run under an orchestrating agent that hands out tasks, waits for them to finish and decides what happens next. The waiting is where the multiplexer earns or loses its place. In our current pattern each turn is a new headless process (`claude -p`, `grok -p`), follow-up turns reuse the previous session through the CLI's `continue` flag, and the orchestrator learns that a turn has ended because the process exits and `tmux wait-for` fires. This is simple and reliable for single-shot work. It has two known costs. Every turn pays for a CLI start. A headless process also has no way to stop and ask for permission, so it cannot tell the orchestrator it is waiting for anything.

Herdr's authors describe it as "tmux for coding agents". It detects 22 agent CLIs out of the box, reports each pane's agent as idle, working, blocked, done or unknown, and exposes this through a CLI and a socket API (`herdr agent prompt --wait`, `herdr agent list`, `herdr agent wait`). As public context, the project is in the Y Combinator Fall 2026 batch and announced a \$6 million seed round led by Bessemer Venture Partners on 8 September 2026. That has no bearing on the measurements. We note it because a funded project with a version number below 1.0 tends to change quickly, which bears on how long any of these results stay true.

The operator fixed the question before we started: time saved, resources saved, convenience, and no other axis. We did not score features, ergonomics beyond orchestration, or anything Herdr offers outside agent supervision.

2. Three Ways to Drive an Agent

All measurements ran on one 64-thread workstation-class compute node. Versions were Herdr 0.9.1, `tmux` 3.4, Claude Code 2.1.283 driving Claude Sonnet 5, and Grok CLI 1.0.42 driving Grok 4.7 Fast at high effort. All three arms drove the same agents with the same prompts in the same workspaces.

Arm A is our current practice. Each turn is a new headless process, and completion is the process exit.

Arm B keeps one long-lived interactive agent per `tmux` window. The orchestrator types prompts with `send-keys` and infers state by capturing the pane every 1 s and matching the screen against regular expressions written for each agent. A turn counts as finished only after the screen has held still across two polls. An 8 s safety timer prevents the arm from declaring "done" before a turn has visibly started. We built this arm to be the strongest fair `tmux` baseline, because comparing Herdr only against one-shot would confound the multiplexer with the change from one-shot to persistent sessions. Arm B needed five agent-specific expressions: Claude working, Claude blocked, Claude ready, Grok working and Grok ready. Writing them required screen captures, and one detail cost us time. Claude Code's prompt glyph is followed by a non-breaking space (U+00A0), not an ordinary space, and this silently broke our first "ready" expression.

Arm C keeps one long-lived interactive agent per Herdr workspace and dispatches each turn with `herdr agent prompt --wait`, which returns when Herdr's own state tracking says the turn has ended.

3. Workloads, Ground Truth and Accounting

The completion workload (W1) ran six concurrent workers, three Claude and three Grok. Each worker did three dependent turns: create a module with `add` and `mul`; add `pytest` tests and run them; add `sub` with a test and run it. That is 90 turns per arm over five rounds. We verified



correctness from the files on disk: the tests pass and sub exists. Permissions were bypassed for this workload.

The blocked workload (W2) ran three concurrent Claude workers in the default permission mode. Each received one prompt requiring a shell command that needs approval, giving 15 prompts per arm. The command writes a random nonce, fresh for each run, into a file, and success means the file holds that nonce. We did not start with a nonce. The first smoke run asked for "print 6*7" and checked for "42". In two of three headless runs the output contained "42" without any command having run, because the model had computed the answer itself. The check would have scored those as successes, so we replaced it before measurement.

For the true end of each Claude turn we used Claude Code's Stop hook, which fires when the agent finishes a turn. We had it write a timestamp every time. Detection lag is the time at which the orchestrator noticed the turn had ended, minus the Stop timestamp. Grok CLI gave us no equivalent, so lag is measured for Claude only.

Resource accounting took more care than we expected. We ran each arm's multiplexer server in its own systemd user slice (a kernel accounting group), so that CPU and memory cover the server and every agent it spawned. tmux 3.4 moves each new pane into its own transient scope. Those scopes escaped our first accounting unit, and the smoke run reported 0.4 CPU-seconds for a full round of six agents. Slice counters keep the usage of scopes that have already exited, which is why we moved to slices. We also recorded multiplexer server resident memory (RSS), CPU over a 30 s idle window with all persistent agents alive, peak anonymous memory (we used this in preference to the kernel's peak figure, which also counts page cache), and orchestrator CPU including its children.

We rotated arm order across rounds (ABC, BCA, CAB and so on) so that drift in API latency would fall on all arms roughly equally. Every arm was launched from a clean systemd unit. The multiplexer servers had been inheriting an environment marker from the launching agent session that disables Claude Code transcripts, and without transcripts the continue flag stops working. Two smaller harness details surfaced in the smoke run and were fixed before measurement: Herdr's CLI reports errors as JSON on stderr, and herdr server runs in the foreground rather than daemonising.

4. What We Predicted

We registered hypotheses at 15:56:00Z, before any measured run. H1a predicted Herdr's round wall-clock 15 to 40 % faster than one-shot, and H1b predicted it within plus or minus 10 % of polled tmux. H2a predicted a median Claude detection lag under Herdr of at most 1.0 s, H2b a lag of 1.5 to 3.0 s under polled tmux, and H2c a lag of at most 0.5 s under one-shot. H3 predicted that one-shot cannot notice a block at all, that polled tmux notices only through a Claude-specific expression, and that Herdr's time to notice would be at most polled tmux's. H4 predicted that agents dominate CPU and memory, that Herdr's server RSS exceeds tmux's, that one-shot uses the most CPU, and that polled tmux's orchestrator CPU exceeds Herdr's. H5 predicted zero agent-specific expressions for Herdr against at least four for polled tmux, and less orchestration code for Herdr than for polled tmux.

We filed one amendment at 16:07:39Z, after the smoke run and before the measured runs. It introduced the nonce check for W2 and per-arm slices, and raised the round count from three to five because the smoke run showed high variance in Grok tail latency. The five measured rounds ran between 16:07 and 16:36 UTC.

5. Results: Time



All three arms were fully correct on W1: 30 of 30 worker results passed in each, and no arm ever declared a turn done before it was. The differences lie in how quickly the orchestrator learned what had happened.

Metric (median unless marked)	A one-shot	B tmux interactive	C Herdr
Claude done-detection lag, median	0.54 s	2.43 s	0.22 s
Claude done-detection lag, p90	0.67 s	3.17 s	0.37 s
Claude turn, orchestrator-observed	8.3 s	9.0 s	6.6 s
Claude dispatch to Stop	7.77 s	6.64 s	6.35 s
Grok turn, orchestrator-observed	8.9 s	10.0 s	9.4 s
W1 round wall-clock, median	54.8 s	41.2 s	34.6 s
W1 round wall-clock, range	27.5 to 74.2 s	35.2 to 71.3 s	34.3 to 53.5 s

Herdr's detection lag is 91 % below polled tmux's and 59 % below one-shot's. The p90 figures (the value below which nine in ten observations fall) rank the arms in the same order, so the medians are not hiding a bad tail. Polled tmux's lag of about two and a half seconds follows from its design. A 1 s poll plus a two-poll stability gate cannot report sooner than the screen has visibly settled.

The Claude turn as the orchestrator saw it was 27 % shorter under Herdr than under polled tmux and 20 % shorter than under one-shot. The dispatch-to-Stop row separates two causes. It measures the agent's own work, including CLI start-up where there is one. One-shot's 7.77 s against Herdr's 6.35 s puts the cost of a fresh CLI start at about 1.4 s per Claude turn. Polled tmux has no start-up cost, and its dispatch-to-Stop of 6.64 s is close to Herdr's. Its disadvantage in observed turn time is therefore mostly detection lag.

Grok gives a less tidy picture. Herdr was 6 % faster than polled tmux on Grok turns but 6 % slower than one-shot. We have no ground-truth timestamp for Grok, so we cannot say how much of that difference is detection and how much is the agent. We report it as measured: for Grok, the one-shot arm was faster.

Round wall-clock favoured Herdr, 16 % faster than polled tmux and 37 % faster than one-shot, and Herdr's range was the narrowest of the three. The per-round figures explain why we treat this as supporting evidence only.

Round	A one-shot	B tmux interactive	C Herdr
1	35.9 s	41.2 s	34.3 s
2	27.5 s	35.2 s	34.4 s
3	58.8 s	46.2 s	53.5 s
4	74.2 s	36.2 s	34.6 s
5	54.8 s	71.3 s	34.9 s

A round ends when its slowest worker finishes, and that is nearly always a Grok turn. Single Grok turns reached 59.8 s under one-shot, 50.2 s under polled tmux and 31.3 s under Herdr. One slow Grok response can move a round by tens of seconds, and nothing in any multiplexer controls that. In round 2 one-shot was the fastest arm of all. Four of Herdr's five rounds were close together, which is encouraging, but five rounds cannot tell us whether that clustering is a property of Herdr or of when its rounds happened to fall.



6. Results: Blocked Prompts

The largest differences appeared in the blocked workload.

Metric	A one-shot	B tmux interactive	C Herdr
Blocked prompts noticed	0/15	14/15	15/15
Time to notice (when noticed)	never	10.0 s	9.6 s
Blocked prompt to verified result, median	17.9 s	17.1 s	10.5 s
Correct in the end	15/15 (after rerun)	14/15	15/15

One-shot never noticed a block because none ever became visible. In default permission mode the headless CLI denies the tool silently and exits. The orchestrator sees a normal exit and a wrong result, and the only recovery is to rerun with permissions bypassed. That is how one-shot reaches 15 of 15 in the end, and it is why the one-shot figure includes a second full run. It also means the approval step, the purpose of the default permission mode, never took place.

Polled tmux noticed 14 of 15 blocks and Herdr 15 of 15. The time to notice is effectively the same (9.6 s against 10.0 s, inside noise). The two arms separate after approval. Herdr's median from blocked prompt to verified result was 38 % below polled tmux's and 41 % below one-shot's. Its spread was also narrower: 6.9 to 13.5 s, against 6.0 to 24.1 s for polled tmux.

Polled tmux is slow here because of how its poller works. After approval, the command and Claude's reply finish within one 1 s poll interval, so the poller never observes the "working" state. It then waits for the 8 s safety timer, which exists to stop it declaring "done" before a turn has started. That timer is what gives polled tmux its clean record of no premature completions in W1, and the same timer costs it here. We could shorten it, but any shorter value would make premature "done" calls more likely, and we did not measure that trade.

Polled tmux's single miss was a screen that its expressions classified as idle when the agent was blocked. The screen was not captured. Our best explanation is an approval dialog for something other than a shell command, whose wording the blocked expression did not cover. We cannot confirm it.

7. Results: Resources

Metric	A one-shot	B tmux interactive	C Herdr
Total CPU per W1 round (slice)	203 CPU-s	124 CPU-s	122 CPU-s
Peak anonymous memory, all agents	4982 MB	4977 MB	4960 MB
Multiplexer server RSS	5.2 MB	6.1 MB	22.6 MB
Idle 30 s, server CPU	0	0	0.07 CPU-s
Idle 30 s, whole slice CPU	0.87 CPU-s	1.16 CPU-s	1.65 CPU-s
Orchestrator CPU per round	0.30 s	0.85 s	0.29 s

The agents dominate. Peak anonymous memory is within a few tens of megabytes across arms, against totals near five gigabytes. The CPU saving belongs to persistent sessions rather than to Herdr: both persistent arms used about 40 % less CPU per round than one-shot, and Herdr against polled tmux is a tie. Starting a fresh CLI for every turn is what costs one-shot its extra CPU.



Herdr costs more to keep running. Its server holds 16.5 MB more resident memory than tmux in the interactive arm, and with six idle agents the whole slice under Herdr used 0.49 CPU-s more per 30 s than under polled tmux, about 1.6 % of one core. On a 64-thread node we do not expect to notice either figure, but they are real and continuous. On the orchestrator's side, the relationship reverses. Polled tmux spends 0.85 s of orchestrator CPU per round capturing and matching screens, and Herdr spends 0.29 s, 66 % less. In effect Herdr does inside its server the state-tracking work that polled tmux pushes onto the orchestrator.

8. Results: Convenience

We tried to make convenience measurable rather than a matter of taste.

Metric	A one-shot	B tmux interactive	C Herdr
Orchestration code for the arm	26 lines	63 lines	47 lines
Agent-specific regexes	0	5	0
"Which workers need me?" query	not possible	18 ms (hand-built classifier)	2.5 ms (built-in JSON)

One-shot has the least code because it does the least. It cannot see a block and has no persistent state to ask about. Among the persistent arms, Herdr needed 25 % less code than polled tmux and no agent-specific expressions against five. The line counts understate the gap. The five expressions each took screen captures to discover, and one failed without any error because of a non-breaking space. Each is tied to the current screen output of one version of one agent CLI. Any upstream change to a prompt glyph or a status line can silently break them, and a new agent CLI needs a new set. Herdr moves that maintenance to its authors, who currently cover 22 agents. That is a real transfer of dependency, and it is not free: if Herdr's detection for an agent breaks, we would be waiting on someone else's release.

The question "which workers need me?" is the one an operator asks most often. Under one-shot it cannot be answered. Under polled tmux it takes 18 ms through a classifier we wrote. Under Herdr it takes 2.5 ms and returns structured JSON.

9. Predictions Scored

Hypothesis	Prediction	Observed	Result
H1a	C 15 to 40 % faster than A (round)	37 % faster, noisy	pass
H1b	C within ± 10 % of B (round)	16 % faster	fail
H2a	C Claude lag median ≤ 1.0 s	0.22 s	pass
H2b	B lag 1.5 to 3.0 s	2.43 s	pass
H2c	A lag ≤ 0.5 s	0.54 s	fail
H3	A blind; B via Claude regex; C notice $\leq$ B	0/15, 14/15, 15/15; 9.6 s vs 10.0 s	pass
H4	Agents dominate; Herdr RSS > tmux; A most CPU; B orchestrator CPU > C	all four parts	pass
H5	C 0 regexes vs B ≥ 4 ; C less code than B	0 vs 5; 47 vs 63 lines	pass



Six of eight held. H1b failed in Herdr's favour, and it rests on round wall-clock, the noisiest measure we took. H2c missed by a small margin: one-shot's lag was 0.54 s against a predicted ceiling of 0.5 s. H3 passes on the counts; its timing clause passes only in the sense that 9.6 s is not more than 10.0 s, and we regard that difference as inside noise.

10. The Phone-Home and How We Removed It

The operator's condition for a pilot was that Herdr must not send signals home. We probed upstream 0.9.1 with default configuration and no config file. A logging curl shim sat first in the search path, and we traced process execution and socket connections. Within milliseconds of server start Herdr requested `https://herdr.dev/agent-detection/index.toml` and `https://herdr.dev/latest.json`, and herdr update requested `latest.json` again: three calls in the 45 s window. The background checks repeat every 30 minutes. The requests carry no account identifier, but they reveal our address, our timing and our usage cadence. Our own benchmark servers, run on upstream defaults, made these calls at every server start, about 15 starts in all.

An audit of the 0.9.1 source tag found no HTTP, TLS or telemetry crate in the dependency lock file, and no code that opens an internet socket. Every network path shells out to curl through one helper function, used at four call sites. These are the background version check (which fetches `latest.json`, `preview.json` and a package-manager formula URL), the agent-detection manifest update, the remote-attach version check and release download, and herdr update. Plugin installs from GitHub run only on an explicit user command. Two config keys, `version_check` and `manifest_check`, switch the background checks off. Both default to on.

We mirrored the source to our own git forge and patched the helper to run false instead of curl. Every caller therefore takes its existing failure path, and we added no new code paths. We also flipped both config defaults to off. The patch is 9 changed lines in 2 files. The build uses the pinned Rust toolchain 1.96.1 and Zig 0.16.0 for the vendored terminal-emulation library; the Zig step fetches hash-pinned dependency archives at build time only. A release build takes 1 min 38 s.

Probe (default config, 45 s window)	Upstream 0.9.1	Sovereign build
curl calls	3	0
curl executions	3	0
Non-loopback connects	0 (the shim refused each request)	0
herdr update	requested latest.json	ran the stub, failed closed

The sovereign probe included a workspace, a pane command and herdr update, under the identical harness. We also hit a harness bug along the way. Unix socket paths are capped at 108 bytes, and a long probe label pushed Herdr's client socket path over the limit. The server exited with a path-length error that at first looked like a crash in our build.

For the pilot we added two further layers. The config file sets both checks off explicitly, and a start-up guard refuses to launch any binary whose SHA-256 hash is not pinned, or any config with a check enabled. In negative tests the guard refused the upstream binary and refused a config with `version_check` on.

11. What These Numbers Will Not Carry

The sample is small. There were five measured rounds, 90 W1 turns and 15 W2 prompts per arm, all on one host, with two agent CLIs and small, synthetic coding tasks. We ran every arm once per round. We have no second host, no second day and no confidence intervals. The



rotation of arm order spreads API latency drift across arms, but with five rounds it cannot remove it.

Round wall-clock is noisy and dominated by the slowest Grok turn, and it does not carry a claim about Herdr. Detection lag is measured for Claude only, because only Claude gave us a ground-truth end-of-turn timestamp. The Grok comparison therefore cannot separate detection from agent time, and there one-shot was the faster arm. W2 is Claude-only because Grok never hit an approval prompt under our command allowlist. Polled tmux's one missed block was not captured, so its cause is an inference. Its slowness after approval depends on our 8 s safety timer. A different timer would give a different figure and a different risk of premature completion, and we did not measure that trade.

The benchmark ran the upstream binary with update checks on. Those calls run in separate curl processes and should not affect agent timing, but we did not re-run the timing workloads on the sovereign build. Herdr 0.9.1 is pre-1.0, and its detection of each agent depends on screen behaviour that the agent's authors can change.

These claims survive. Persistent sessions under either multiplexer use about 40 % less CPU than one-shot and avoid about 1.4 s of CLI start per Claude turn. Herdr detects the end of a Claude turn faster than screen polling does (0.22 s against 2.43 s median, with the p90 in the same order). One-shot cannot see a permission block at all. Herdr turns a blocked prompt into a verified result faster than our best polled baseline, largely because it does not need a safety timer. It needs no agent-specific expressions and less orchestration code than the polled baseline. It costs 16.5 MB of server memory and about 1.6 % of one core at idle over tmux. Our patched build made no outbound connections under our probe. These claims do not survive: that Herdr shortens whole rounds by any particular percentage; that it is faster for Grok; that it notices blocks sooner than polling; and anything about larger tasks, other agents or other hosts.

12. What We Changed

We started a pilot of the sovereign build on 27 September 2026. It covers persistent, multi-turn or approval-gated coding-agent workers, and an operator view of which workers need attention. Single-turn, permission-bypassed jobs stay on the one-shot pattern. In a live smoke test a persistent Claude worker answered 5050 (the sum of 1 to 100), then, in the same session with its context kept, 500500 (the sum of 1 to 1000). A persistent Grok worker answered 3628800 (the product of 1 to 10). We verified both files by running them. The pilot's review date is 11 October 2026. The next measurement should repeat the timing workloads on the sovereign build and cover blocked prompts for more than one agent.

PureTensor operates a sovereign AI infrastructure fleet (on-premises GPU compute, storage, and serving) run day-to-day by autonomous agents under human direction. This paper is PT-R-2026-030. All measurements were taken on 27 September 2026: pre-registration at 15:56:00Z, amendment at 16:07:39Z, five measured rounds between 16:07 and 16:36 UTC, and the sovereignty audit, sovereign build, egress probes and pilot start between about 18:10 and 18:46 UTC. Raw artefacts, including per-turn timestamps, slice counters and probe logs, are retained.