

A 10,000-star code graph on our bench

Lab note 3 15 Oct 2026, rewritten 16 Oct with everything measured since 12 min read
written by Claude from this repository's data

A friend asked a fair question: why build our own code tools when projects like Trail-Code (about 10,000 GitHub stars) put a whole codebase into a graph that Claude can query, and report large savings doing it? Someone they knew had cut their Claude usage sharply with a tool like it.

The first version of this note measured one thing: what Trail-Code adds to our pack. A reader could object, fairly, that we never tested Trail-Code without the pack, nor whether the two get in each other's way, nor Trail-Code at its best. We have measured all three since. This is the whole story.

What we wanted to know

One question: **what does a person get when they install Trail-Code into Claude Code the way its README says?** That is how we measure our own mods too: the same graded tasks with and without, several runs of each, an interval around every difference, and every run's record published.

That is a narrower question than "is Trail-Code good?". Its own headline numbers come from a harness of its own. Further down we say where they come from, and why they do not carry over to an install.

How it is meant to be used

Its README's quick start is two commands, `npm install -g @trailhq/trail-code` and `trail init`, and then: *"That is the whole setup. `trail init` asks which of your coding agents to wire up, builds the graft map from your code, creates `.trail/` for the repo's learnings, and adds a statusline and hooks to Claude Code."* It also registers an MCP server. In Claude Code that gives:

- **A graph of the code**, the "graft map", built with tree-sitter and no model: click's 79 files gave 2,028 nodes and 3,221 edges in about a second. It checks the working tree before every query and rebuilds what changed, so it is never stale (`src/graph/refresh.ts`). That is why we picked it over graphify, whose Claude Code setup asks the model to update the graph after changing code.
- **Hooks**: at session start an orientation ("reach for trail first"); with each prompt, likely places in the code, or a reminder to run `trail ask` before grepping; after an edit, what depends on the file; and once a session, after a turn of real work, a request to save a "team learning" (its Stop hook).
- **Tools**: six MCP tools, and a command line, `trail`, with `ask`, `grep`, `callers`, `skeleton`, `check` and more.

What we measured

Four measurements, each answering the objection the one before it left open. click 8.5.0, eight tasks from a one-line lookup to a feature, graded by hidden tests or fixed rules, Opus 5.5 and fresh sessions with four runs per task and arm unless said.

Added to our pack

The first question was the one a pack user has. On 15 Oct, against the pack alone:

Trail-Code added to the pack	Cost	95% interval	Runs
As installed, fresh session	+12.8%	+7.5% to +18.3%	64
As installed, long session	+32.1%	+21.9% to +43.6%	28
Without its Stop hook, fresh session	+4.0%	+0.1% to +8.0%	64

The Stop hook did the most damage. On the explanation task, in all four runs, Claude answered, then the hook asked it to save a team learning, and its final message was about the learning file. In a headless run the final message is the result; graded on all of Claude's messages, all four were right. In the long session, where Claude had read every file already, the orientation and pointers repeated what it knew, and every later request re-read them. On 30 SWE-bench Verified issues it cost +10.8% [0.0%, +25.5%] on top of the pack (note 9).

That left the obvious objection: lsp-plus already answers Claude's symbol searches, so Trail-Code may have had nothing left to add.

Alone, and both together

So on 16 Oct we ran all four combinations in one batch: plain Claude Code, Trail-Code alone, the pack alone without seek, and both. Trail-Code ran without its Stop hook: as installed, in a smoke run, the hook replaced the answer again (score 0.17, the final message about a saved learning).

Against plain Claude Code	Cost	95% interval
Trail-Code alone	-4.6%	-9.3% to +0.1%
The pack, without seek	-5.6%	-10.8% to -0.3%
Both	-2.8%	-8.5% to +3.0%

128 runs, all full marks. Alone, Trail-Code came out about where the pack of that day did, its interval just touching zero. Together they did not add up: Trail-Code on top of the pack, +3.0% [-2.7, +9.1]. Both work

on the same steps, the first searches, and the second one adds text more than it saves.

Alone, against today's pack

A -4.6% that touches zero needs a second batch before anyone believes it. We ran Trail-Code alone again, beside plain Claude Code and the pack as 0.6.0 ships, with seek:

Against plain Claude Code	Cost	95% interval	With carry
The pack 0.6.0	-15.1%	-19.8% to -9.7%	-20.4%
Trail-Code alone	+1.1%	-5.4% to +7.7%	+0.1%

96 runs, all full marks. Carry adds what each run left in the context, re-read by 140 later requests (note 11). Trail-Code alone against the pack 0.6.0: **+19.0%** [+13.1, +25.1], with carry +25.7%.

Pooled over both batches, 64 runs a side, Trail-Code alone against plain Claude Code: **-1.8%** [-5.9, +2.7]. No saving we can measure. The first batch's -4.6% was chance: the same two arms on the same tasks read -4.6% in one batch and +1.1% in the next.

Its best case

Trail-Code's own benchmark has a variant it calls push: `trail ask --source` puts a bundle of the likely code in front of the task. The install does not do that; its prompt hook adds pointers when it finds a strong match, and otherwise a reminder. So we built push into plain Claude Code: the bundle, exactly as `trail ask --source` prints it (about 2,000 tokens), in front of the prompt, on Claude Sonnet 5.5, as their benchmark ran on Sonnet, for the explanation and the feature, two runs each.

+3.2% [-7.3, +15.0], the same number of requests as plain Claude Code on both tasks (5.5 and 3.5), all 8 runs full marks. The bundle asks Claude to end its reply with a tally, and every answer did: " *trail saved ~57,792 tokens this turn*". That figure is the bundle (2,086 tokens) set against reading all eight files it drew from whole (59,878 tokens): "97%". Claude reads no eight files whole for these tasks; it greps and reads a few lines.

Eight runs is few, and the interval is wide. They show no saving; they cannot rule out a small one.

How Claude used it

From the 64 transcripts of the four-way batch with Trail-Code loaded:

- **Its MCP tools: 0 calls in 64 runs.** Claude Code keeps an MCP server's tools behind its tool search until Claude asks for them. Claude never asked.

- **Its command line, through Bash: in 24 of 32 runs without the pack (32 calls), and 17 of 32 with it (20 calls).** `trail grep` 21 times, `trail ask` 9, `callers` 8, `check` 7, `skeleton` 6, and once `trail note`, a learning Claude saved without being asked.
- **Its prompt hook found "no strong match" for 56 of 64 prompts**, and then asked Claude to run `trail ask "<your task>" --source` before grepping. For the other 8, all on the signature task, it named starting points.

So Claude did use Trail-Code, mostly as a second grep, and needed about as many requests as without it: 6.3 and 6.8 per task, against 7.0 and 6.8 plain, in the two batches.

Where its published numbers come from

Trail-Code's README (version 0.22.1) reports three sets of numbers:

Its claim	How it was measured
32% lower cost, 46% fewer tool calls, 60% less time, the same correctness (93%)	162 runs of its own agent loop on Claude Sonnet 5, push against cold, an Opus 4.8 judge, two repositories
SWE-bench Verified: 33 of 50 solved, against 27	Claude Code on Sonnet 5; cost, tokens and calls "over the instances both arms resolved"
PocketBase, 15 tasks: \$13.91 → \$11.02 (~21%), 14% faster	Claude Code headless on Claude Opus, wired with <code>graft init</code> (its earlier name), "with an empty MCP config"

Why these do not carry over to an install:

- **The headline is its own loop with the bundle injected.** The README says so: *"The sweep above is our harness measuring our mechanism."* The install does not inject the bundle, and when we injected it ourselves, it saved nothing.
- **Single runs.** The SWE-bench comparison reports one result per arm: 33 against 27 of 50 solved is within what chance gives (Fisher's exact test, $p \approx 0.31$). Cost "over the instances both arms resolved" leaves out every instance either arm failed. On our bench the same comparison moved from -4.6% to +1.1% between two batches of 32 runs a side; one run per task moves further.
- **One question carries much of PocketBase.** Of the \$2.89 gap, \$1.35 is one question, *"how does auth work across OAuth2 providers"*, at \$2.19 against \$0.84. Without it, the rest is about 13% cheaper.

Their tasks in our harness: PocketBase

Trail-Code's README lists its PocketBase tasks: ten developer questions and five merged pull requests. We ran them the way it did, once each, Opus, headless Claude Code: the questions word for word at PocketBase v0.40.5 (the README names no commit), the pull requests re-implemented from each one's base commit, from a request written from its title and description. Go's language server ran in both arms. The answer keys were fixed before any run, four to eight terms per question; a pull request scores the share of the maintainers' non-test files it touched.

We did not run Trail-Code here. This is plain Claude Code against our pack 0.6.0:

PocketBase, 15 tasks	Plain Claude Code	The pack 0.6.0	Change
Total cost	\$9.98	\$7.88	-21.0%
Requests per task	12.8	9.4	
The average task			-18.1% [-26.7, -8.3]
Ten questions			-17.8%
Five pull requests			-18.5%

- **The interval is over tasks.** With one run per task it resamples the 15 tasks, so it is wide. -21.0% is also the figure Trail-Code reports for itself on these tasks; we do not compare the two. The PocketBase, the Claude Code and the Opus differ, and both are single runs.
- **Fewer requests on 10 of 15 tasks**, as many on 4, more on 1.
- **All five pull requests touched the maintainers' files, in both arms**, and every change built as well as its base commit did (one old base does not build with today's Go, in either arm).
- **One pack run scored 0.** On q10 ("Where are the tests for the record CRUD API..."), lsp-plus answered one of Claude's greps by asking gopls, which was still loading PocketBase. The grep waited past Claude Code's 120 s and was moved to the background. Claude answered in full; then the background command's end arrived, Claude replied to that notice, and that reply became its last message, the one the grader reads. Plain Claude Code scored 0.88 there. The run took 735 s, so the pack's total time was the longer one, 2,173 s against 1,804 s (without q10, 1,438 s against 1,746 s). Fixed in lsp-plus 0.4.2: every language-server question after a grep or an edit now gets 8 s, and past that the tool's result goes back without the answer.
- **Two questions fell short in both arms:** q4 (0.71 plain, 0.57 with the pack) and q5 (0.75 both). Their answers left out terms of the key.

What we do not claim

- **Not that Trail-Code's numbers are wrong.** We did not rerun its harness. Measured its way, in its own loop with the bundle injected, they may hold.
- **Not that it saves nothing anywhere.** Its maintainer describes its job as the step before Claude knows a symbol: "where does auth happen". Its best case is a large repository Claude does not know, an orientation question, Sonnet, and the bundle injected. click has 79 files, our tasks name what they are about, and all but eight of our runs were Opus. It may help there; we did not test it.
- **Nothing about team learnings across sessions,** half of what Trail-Code is for. Every bench run starts fresh.
- **Nothing about Trail-Code on PocketBase.** We ran the pack there, not Trail-Code.

What we do claim: installed into Claude Code the way its README says, on our tasks, Trail-Code did not make Claude's work measurably cheaper. Alone it came to -1.8% $[-5.9, +2.7]$ over 64 runs a side; beside our pack it cost more; its own best case, rebuilt in Claude Code, saved nothing either; and in headless runs its Stop hook replaced Claude's answer. Every run is published.

What we learned

- **A bundle can help only if it arrives with the task.** In Trail-Code's own harness it does. In the install it does not: the prompt hook found nothing for 56 of 64 prompts, and the tools stayed behind tool search. Where we put the bundle with the task ourselves, on click, it still saved nothing, because Claude's own first search was already cheap. Where an instruction rides decides whether it counts (note 11); whether it then saves anything is a second question.
- **A plugin must be measured in the harness it ships for.** A person installs it into Claude Code, with its tool search, its hooks and Claude's habit of grepping. A number from another loop says what the mechanism can do, not what the install will.
- **One batch is not a result.** Trail-Code alone read -4.6% in one batch and $+1.1\%$ in the next. With one run per task, a single question can carry half a saving.
- **Measuring someone else's tool found a bug in ours.** A cold language server on a large Go project cost the pack a whole answer. lsp-plus now waits 8 s at most.

Method & limits

Arms: V, plain Claude Code; Z, the pack without seek and route, as committed on the day; Z6, the pack as 0.6.0 ships; T, the pack with Trail-Code as `trail init` wires a repository; TG, T without Trail-Code's Stop hook; VT and VTG, plain Claude Code with Trail-Code as installed (one smoke run) and without its Stop hook; VP, plain Claude Code with the `trail ask --source` bundle in front of the prompt, on Sonnet 5.5; PL and TL, Z and T in a prepared 220k-token session. Trail-Code 0.22.1 from npm, wired with `trail init`

`--no-agents --no-global --no-statusline` (Claude Code only, nothing outside the repository, no status line), its MCP server started from the installed version, its telemetry off; its hooks at repository level only, so they ran once (#497). Plain runs had the lab machine's installed plugins switched off (note 8); the plain runs of 16 Oct were checked for any trace of the pack and have none.

Costs from the transcripts at list prices; 95% bootstrap intervals over runs within tasks, for example `python3 -I carry.py V VTG,Z,TG results-trail2x2.jsonl` (the ends move by a few tenths with the arms resampled beside); for PocketBase over its 15 tasks. The four explanation runs of T score 0, 0, 0.33 and 0 graded on the final message, as the bench grades; on all of Claude's messages, all four score 1.0. The counts of Trail-Code's calls come from the transcripts, which stay in the lab. On click, per-task differences are within the runs' spread; the totals' intervals are not.

Data: `results-trail.jsonl` (Z, T, TG, PL, TL), `results-trail2x2.jsonl` (V, VTG, Z, TG) with its smoke run `results-trail2x2-smoke.jsonl` (VT), `results-trail-vs-060.jsonl` (V, VTG, Z6), `results-trail-push.jsonl` (V, VP), `results-pb.jsonl` and `results-pb-smoke.jsonl` (V, Z6); the PocketBase tasks, keys and grading in `pb_repo.py`; setup in `trail_init()`, `trail_bundle()` and `TRAIL_ARMS` in `bench.py`.

Sources

- trailhq, Trail-Code (formerly graft): the README of `@trailhq/trail-code` 0.22.1 (quick start, benchmark, SWE-bench Verified, tested on popular repos), source at `ead585b`, issue #497; checked 15 and 16 Oct 2026.
- Graphify-Labs, graphify, issues #2216 and #3718 on stale graphs; checked 15 Oct 2026.
- Show HN: graft: the discussion of its SWE-bench sample, and its maintainer's replies.
- PocketBase v0.40.5 and pull requests #6744, #6947, #6690, #2726, #3192.
- Jimenez, C. E., et al. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *ICLR 2024*.