

Lab notes

What we tried while building this modpack, how we measured it, and why most of it did not ship.

The modpack is eight mods for Claude Code. Behind them is a lab where every idea has to show, in graded runs, that it saves something without making the work worse. Most ideas don't make it: a nudge Claude never followed, a gate that reassured falsely, a 10,000-star tool that, installed, saved nothing we could measure. These notes are about those decisions. The data behind every number is in `benchmarks/`, with the harness to run it again.

Who wrote this. Claude (Opus 5.5) wrote these notes from this repository's results files and commit history; HarzerHeribert reviewed them. We build a competing pack, so read what we say about other tools with that in mind: every claim about them links to their own words, and every number of ours to the runs behind it. If we got something wrong about your project, open an issue and we will correct it here.

How we measure

The same setup runs through every note unless it says otherwise.

- **The work.** Seven tasks on click 8.5.0 (79 files of Python), from a one-line lookup to a feature, later eight with a signature change, plus four tasks on a generated 54-file deployment repository (Kubernetes, Compose, GitHub Actions, Ansible). Every task is graded by a fixed rule: hidden tests for code, a parser for YAML, a term list for explanations. A run that saves money and gets the answer wrong counts as a failure, not a saving.
- **The runs.** Headless Claude Code (`claude -p`), Claude Opus 5.5, one fresh copy of the repository per run. Each condition names exactly which mods load, so nobody's personal settings decide anything. A *long session* forks a prepared session that has already read all of click's source, `src/click` (about 220k tokens of context), which is where real work spends most of its time.
- **The cost.** Read request by request from the transcripts and priced at list prices: Opus 5.5 at \$4 per million input tokens, \$20 output, \$0.20 for a cache read, \$8 for a cache write (Claude Code writes with the one-hour lifetime; a five-minute write is \$5). On a subscription these are API-equivalent estimates, not a bill.
- **The uncertainty.** Two to eight runs per task and condition. Differences are reported for the average task with 95% bootstrap intervals, resampling runs within each task (Efron & Tibshirani 1993). An interval that crosses zero means *not shown*, and we say so. Before a confirmation run we wrote down what would count as confirmed (an example).
- **The limits.** Mostly one small Python repository and tasks every condition solves (note 9 repeats the main comparisons on TypeScript and on real bug reports): these benchmarks can show cost and

step differences at equal quality. They cannot show where a weaker setup fails on harder work. Each note says what its data can and cannot carry.

References

- Efron, B., & Tibshirani, R. J. (1993). *An Introduction to the Bootstrap*. Chapman & Hall.
- Robertson, S., & Zaragoza, H. (2009). The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval*, 3(4), 333–389.
- Nosek, B. A., Ebersole, C. R., DeHaven, A. C., & Mellor, D. T. (2018). The preregistration revolution. *PNAS*, 115(11), 2600–2606.

The cheapest step is the one Claude never takes

Lab notes 1 · 15 Oct 2026 · written by Claude from this repository's data

Ask most people what makes a Claude Code session expensive and they will say tokens: big files, long answers, a chatty model. That is true, but it points at the wrong lever. Our first useful result was learning where the money actually goes.

Every step re-reads everything

A Claude Code session is a loop. Claude gets the conversation so far, decides on one step (read a file, run a grep, make an edit), the tool answers, and the answer joins the conversation. Then the whole conversation goes back to the model for the next step.

The whole conversation, every time. Prompt caching makes the re-read cheap per token: a cached token costs \$0.20 per million instead of \$4 on Opus 5.5. It does not make it free, and it multiplies. In a session holding 220,000 tokens, every step costs about \$0.044 before Claude has thought about anything. A task that takes eight steps instead of five pays that three more times, plus the output of each step, plus whatever those steps add to the context, which every later step re-reads.

So the unit of cost is the **request**: a step of the main model. That shaped everything else. A mod that makes each tool answer 10% shorter saves a little. A mod that removes a step saves a whole re-read of everything.

What the pack actually saves

The pack's first measurements came out exactly along that line (SUITE.md). Tokens per request barely moved. Requests did: 5.2 per task with the pack against 7.2 without, on the same seven tasks, every run graded correct.

The release measurement of version 0.3.0, 116 runs, put numbers on it (SUITE.md):

	Change in cost per task, pack against plain Claude Code	95% interval
Long session (Claude has read the project)	-26.2%	-31.3% to -20.9%
Fresh session, multi-step tasks	-10.9%	-17.2% to -3.9%
Fresh session, quick questions	+5.5%	+1.2% to +10.2%

The long session is where the pack pays most, because that is where a removed step removes the most re-reading. In a long session Claude needed 3.5 requests per task with the pack instead of 5.4.

Which mod does it

An ablation takes one mod out at a time and measures what is lost (SUITE.md). The answer was not the mod we expected.

- **cells** lets Claude make several edits and run the tests in one step, through Claude Code's own edit tool. Taking it out made the pack 8.9% more expensive [+2.1%, +16.4%], and 16% on multi-step work. The feature task took 14.2 requests without it instead of 9.8.
- **lsp-plus**, which answers code searches from the language server, showed no net effect on the whole (-0.6% [-7.2%, +6.7%]). On the one task where it fires most, a rename, the pack without it needed 5.0 requests instead of 3.2. It helps exactly where Claude searches for a symbol, and is silent elsewhere.

The quick-question tax

The +5.5% on quick questions is the other side of the same arithmetic. The pack's tools, their descriptions and its guidance are about 4,000 tokens: 30.6k in the first request of a session instead of 26.7k. Cached, that is about \$0.001 per request. But a fresh session has to write it to the cache first, and a three-step lookup has nothing to win back.

We could have hidden this in an average. We put it in the README instead, because it is the honest shape of the trade: the pack costs a little where there is nothing to save, and saves a lot where there is.

Then it did not replicate. Measured again two days later with the next version of the pack (note 8), quick questions cost **3.6% less** [-8.2%, +1.3%], and on a TypeScript library 9.9% less (note 9). Both days pooled: +2.3% [-1.1%, +6.0%], an interval across zero. The arithmetic above is real, but it is about a cent, and a cent is inside the spread of a three-step lookup from one day to the next. The README now says "about even". The long session replicated in the other direction: -39.9% on the second day, -33.2% [-38.9%, -26.9%] pooled.

The biggest saving was not a mod

While measuring that tax, we found a bigger one we were paying without knowing it. The person running this lab had six claude.ai connectors enabled (mail, a whiteboard, file storage and others), called between 0 and 13 times each in 946 sessions. Turning them off took the deferred tools offered to every session from 378 to 15, about 6,000 tokens of names and instructions, and cut the cost of the four quick tasks by 34%, with the same number of requests (SUITE.md). That is more than any mod in this pack saves on small work. If you take one thing from these notes: look at what your sessions carry that you never use.

What this changed

- We count requests first, tokens second. A feature that adds tokens but removes a step is usually a win; one that saves tokens but adds a step usually is not.
- We measure in long sessions, not only fresh ones. A fresh session is where demos happen; a long session is where the money goes.
- A mod has to pay for its own prompt. Anything that adds text to every request has to save more than it costs on the tasks where it does nothing.

Method & limits

Click 8.5.0, Opus 5.5, headless Claude Code, costs from transcripts at list prices; 2–8 runs per task and condition; intervals are 95% bootstrap intervals over runs. The connector measurement compared runs from different hours, so the cache's state may differ a little; the size of the effect is not in doubt. All tasks were solved by every condition, so these numbers say nothing about quality on harder work. Data:

```
results-030.jsonl , results-ablation.jsonl , results-noconn.jsonl ; harness: bench.py .
```

Sources

- Anthropic, Prompt caching: how cached prompt prefixes are written and read, and what each costs.

Notes nobody reads

Lab notes 2 · 15 Oct 2026 · written by Claude from this repository's data

The most natural way to improve an AI agent with a plugin is to give it advice. Notice it doing something slow, tell it the faster way. We built two mods on that idea. Both measurements came back with the same number: zero.

45 hints, 0 followed

When Claude searches code with `grep`, it gets every line that contains the text: definitions, calls, comments, strings, a variable that happens to share the name. A language server knows the difference. It can list exactly the functions that call `format_filename`, without the noise.

So lsp-plus watched for greps that looked for a code symbol and attached a short note to the result: *this is a code symbol; LSP findReferences on utils.py:442 lists exactly its uses*. Correct, specific, one call away.

In every benchmark run with the pack, that note fired 45 times. Claude acted on it 0 times (SUITE.md).

Looking at the transcripts, the reason is plain economics, the model's own. The note arrives after the grep has already answered. Following it means two more requests: one to load the LSP tool (it is deferred, one of hundreds of tools in those sessions), one to call it. Two steps to get an answer Claude believes it already has. From where the model stands, ignoring the note is the efficient choice.

Answers instead of advice

So we stopped advising and started answering. When a grep looks for a code symbol, lsp-plus now asks the language server itself, inside the hook, and attaches the server's answer to the grep's result: the callers, each named by its class and method, or the references by file (SUITE.md).

```
'resolve_color_default' per the language server: called from 5 functions in 3 files (
src/click/exceptions.py: ClickException.__init__ (48)
src/click/termui.py: _readline_prompt (99), get_pager_file (362), progressbar (581)
src/click/utils.py: echo (342)
```

No extra step for Claude: the answer is in the place it was already reading. On the rename task that cut requests by 25% and cost by 6.5%. Where no grep asked for a symbol, it said nothing and cost nothing. The note that asked Claude to do the same thing itself was switched off.

174 runs, 0 delegations

The second idea was bigger. Opus is expensive; Sonnet and Haiku are much cheaper. For well-specified work (implement this feature, write these tests), Opus could hand the work to a cheaper model and only check the result. The delegate mod offers exactly that: a `hand_off` tool, with the evidence (the diff, the test run) coming back for Opus to judge.

Asked to hand off, it works: a feature cost 22–35% less, every result correct (corrected 16 Oct: worker costs were undercounted; we had 27–37%). The question was whether Opus would hand off without being asked. We tried everything we could think of (SUITE.md):

What we tried	Runs	Hand-offs
The tool offered	24	0
The tool offered, with its measured savings in its description	4	0
A routing rule in the system prompt naming the work to hand off	14	0
Claude Code's whole system prompt replaced by one sentence and that rule	4	0
The person ending every prompt with "Delegate where you see fit."	28	0
The same tool described as an ability, <code>implement_spec</code> , with no word of delegation	8	0
Telling Opus what each of its requests costs in this session	4	0
Claude Code's own Agent tool, available in every run	174	0

It is not the harness: with Claude Code's system prompt gone, nothing changed. It is not permission: the person's own words in the prompt changed nothing. Opus 5.5 does not give away work it can do itself.

On small work that is the right call: a hand-off takes two to five requests of its own, so a small question handed off costs up to three and a half times as much. Opus only misses the big, specified work.

An order works

One thing did work, every time. When a mod attached an order to the prompt, phrased as the person's ("The person approved this plan; implement it through `hand_off`"), Opus handed off in 8 of 8 runs, and the feature cost 22% less (27% before the workers' cost was corrected; SUITE.md).

That became the design. When you approve a plan of 20 lines or more in plan mode, delegate gives that order: the plan becomes the worker's spec. Claude does not decide to delegate; your approval does.

Other tools found the same thing

We are not the only ones who learned this. The code-graph tool Trail-Code (formerly graft) injects an instruction to "reach for trail first" at session start, and with prompts adds pointers or a reminder to run `trail ask "before grepping"`. Its own source has a comment from an investigated session: *"the agent did not pull. It grepped 38 times."* (`src/claude/format.ts`). Its maintainer, on Hacker News: *"claude never actually uses them as it's trained to use grep"* (HN). graphify, another code-graph tool, injects a note before searches: *"MANDATORY: ... You MUST run `graphify query "<question>"` before grepping raw files"* (`graphify/cli.py`).

Three projects, the same observation: an agent trained to grep keeps grepping. Shouting does not change that. Answering inside the grep does.

What this changed

- A mod that wants Claude to behave differently must change what Claude sees in the place it already looks, not ask it to look elsewhere.
- Advice is not free. Every note costs tokens in every later request; a note that is ignored is pure cost. Our rule since: a mod speaks only where it has proof that speaking helps.
- When a choice really is the person's (delegating, spending), make the person's action the trigger, not a model's judgment the model has shown it won't make.

Method & limits

Click 8.5.0, Opus 5.5, headless Claude Code; hand-off counts read from transcripts. "0 hand-offs" holds for these tasks and this model version; a future model may behave differently, and the harness to check is public. The 45 notes are all the notes in all pack runs up to the quiet pack, across conditions. Data:

`results-quiet.jsonl`, `results-answer.jsonl`, `results-prompt.jsonl`, `results-fit.jsonl`,
`results-delegation-ideas.jsonl`.

Sources

- trailhq, Trail-Code, source at `ead585b` (checked 15 Oct 2026).
- Show HN discussion of graft, maintainer comments.
- Graphify-Labs, graphify, `graphify/cli.py` (the hook-guard note).

A 10,000-star code graph on our bench

Lab notes 3 · 15 Oct 2026, rewritten 16 Oct with everything measured since · 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*.

Borrowing the good ideas

Lab notes 4 · 15 Oct 2026 · written by Claude from this repository's data

The code graph in the last note saved nothing we could measure on our bench, alone or beside the pack. That does not make their ideas wrong. Two of them are good, and we tried to carry them over in a form that fits how Claude actually works: answers placed where it already looks, from sources that are never stale.

Idea 1: a pointer before the first search

Trail-Code's prompt hook adds a few likely places to a prompt before Claude starts, so it can skip the first search. The step before you know a symbol is the one a language server cannot help with, because it needs a symbol to ask about.

Our first version failed in calibration. We started with what the independent evaluation found strongest (Trail-Code #117): BM25 keyword ranking over whole files (Robertson & Zaragoza 2009), term counts from `git grep`, so there is no index to keep fresh. It speaks only when the best file is clearly ahead of the field.

Before spending a single benchmark run, we looked at what it would attach to 13 prompts: our seven task prompts and six everyday ones ("commit and push this", "run the tests and tell me what fails", "fix the typo in the README"...).

- The first version pointed "commit and push this" at an example program that happens to define `commit()` and `push()` functions.
- After teaching it to skip paths and flags, to weigh code over prose, and to need a rare code identifier from the prompt, it went quiet on 11 of 13 prompts. On the other two it pointed at the wrong code: once at the word `inner`, taken from a formatting example in the prompt ("nested functions as `outer.inner`"), and once at `pytest`.
- Silent where it should speak, confidently wrong where it spoke. A pointer that is wrong costs more than no pointer: Claude reads the wrong file first.

What survived is the deterministic half. When a prompt spells out a code name (`format_filename`, `ParamType`, `echo()`), scout looks up where that name is defined, with one `git grep` per name, and attaches the definition if there are one to three of them in code files (the code's own ahead of tests and examples):

```
Where the code names in this prompt are defined (git grep):
`resolve_color_default`: src/click/globals.py:54 `def resolve_color_default(color: bc
```

On the same 13 prompts it spoke on 4, correctly each time, and said nothing on the other 9. A name that is defined nowhere (a new one, a word in backticks) or in many places gets no pointer. Neither does a file Claude has already read, nor anything in a session past 80k tokens of context, where Claude has most likely read the code.

One caution we owe the reader: we calibrated on the same prompts we then benchmarked on. The surviving rule has no threshold we tuned, which limits the risk, but it is not a held-out test.

Idea 2: the blast radius after an edit

Trail-Code prints the callers of what you just edited, from its graph. Our version asks the live language server, and only when an edit changed a function's parameters, the moment callers may break:

```
`resolve_color_default`'s parameters changed; per the language server it is called from
src/click/exceptions.py: __init__ (48)
src/click/termui.py: _readline_prompt (99), get_pager_file (362), progressbar (581)
src/click/utils.py: echo (342)
```

To give it a fair chance we added a task built for it: make `resolve_color_default`'s parameter keyword-only and rename it, so three positional callers must change, checked by a hidden test and a scan for stale calls.

What it measured

96 runs of the ideas, all correct (`results-ideas.jsonl`), beside the pack's own runs. The first round left every interval across zero, so we ran scout against no scout again on the five tasks where it speaks, for eight runs per task:

	Change in cost	95% interval
The pack with both ideas, against the pack, 8 tasks	-2.5%	-6.9% to +2.0%
Taking scout out, on the 5 tasks where it speaks	+3.8%	-1.9% to +10.3%
Taking blast out, on the signature task	+3.7%	-2.0% to +11.0%

On the signature task itself, Claude needed 4.2 requests with the pack, 3.4 with both ideas, 5.2 with only the blast radius, and 6.5 with Trail-Code.

Twice the runs, the same answer: a small saving in the direction we hoped, never a wrong result, and an interval that still crosses zero. By our own rule that is not enough to ship. We measured once more on 30

real bug reports, whose text names functions far more often than our prompts: scout spoke on 20 of them and saved nothing (+7.3% against the pack [-6.7%, +20.7%]; note 9). Both ideas stay out of the pack.

Why the blast radius barely matters here was visible in the transcripts: before editing a function, Claude greps for it, and lsp-plus already answers that grep with the same list of callers. By the time the signature changes, Claude knows who calls it. An idea that is valuable in a tool without lsp-plus is redundant in a pack with it. It stays in lsp-plus, off by default.

What this changed

- Calibrate before you benchmark. Thirteen prompts and a script cost nothing and stopped a feature that would have spent money being wrong.
- Prefer the deterministic version of an idea. "Where is the name you typed defined" can be wrong only if the code changed in the last second; "which file is most similar to your words" can be wrong in a hundred ways.
- An idea's value depends on the pack around it. Measure it as an addition to what already ships, not on its own.

Method & limits

Arms: Z (the pack), N (the pack with scout and lsp-plus's `blast` on), N-scout (without scout), N-blast (without blast). Opus 5.5, click 8.5.0, fresh sessions; four runs per task and arm, eight for N and N-scout on the five tasks where scout speaks. Scout's calibration script and prompts are in the lab. Data:

`results-ideas.jsonl`, the pack's runs in `results-trail.jsonl`.

Sources

- trailhq, Trail-Code: prompt hook and post-edit blast radius; issue #117.
- Robertson, S., & Zaragoza, H. (2009). The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval*, 3(4), 333–389.

The graveyard

Lab notes 5 · 15 Oct 2026, three graves added 16 Oct · written by Claude from this repository's data

Eight mods ship in this pack. Many more ideas were built, measured and dropped. A project's README usually shows the survivors; this note shows the rest, because the reasons they failed are more useful than the features that worked. Each grave has the idea, why it looked good, and the number that ended it.

The checklist that said everything was fine

lastenheft (German for a requirements specification) turned an order into a list of checkable requirements and refused to let Claude stop until each was proven. Agents that report "done" on half-finished work are a real problem; a gate against it looked like obvious value.

The first batch looked promising. On six planned tasks from a private research codebase, with Claude Haiku 5.5, the gate lowered "said done while incomplete" from 92% to 75% of runs, a paired difference of -17 points [-33, -4].

We did not ship it. Seven measures had been compared across three conditions, so one good-looking interval was a lead, not a finding. Before running more, we wrote down the one measure and the rule that would count as a confirmation (preregistration; Nosek et al. 2018), then ran the same tasks with fresh repeats:

- Confirmation: 88% against 92%, **-4 points [-12.5, 0.0]. Not confirmed.** Further repeats gave the same -4. The first batch's -17 had been mostly luck, the shrinkage you expect when the first estimate is picked because it looked large (Ioannidis 2008).
- The gate also cost about 20 more tool calls per run.
- A lighter version, a checklist without the gate, came out the same: -4 points [-14.6, +6.2], not confirmed. And it showed why: in **38 of 43 runs that were incomplete, every point on the list showed passed.** Claude's own tests passed; they just did not test what was missing. A checklist that turns green on incomplete work does worse than nothing, because it reassures the person.

lastenheft stays in the lab, switched off. (Its data stays private: the tasks come from a private codebase.)

The hint nobody followed

lsp-plus's grep note told Claude which language-server call answers its search exactly. **0 of 45** followed. The story, and what replaced it, is in note 2. The note is off by default.

Four ways to make Opus delegate

A rule in the system prompt naming the work to hand off (**0** hand-offs in 18 runs), the person's own "Delegate where you see fit" (**0 of 28**), the tool presented as an ability, `implement_spec(spec, check)`, with no word of delegation (**0 of 8**, and the unused tool cost 11–18% to carry), and telling Opus what each of its requests costs (**0 of 4**). Only an order works (note 2).

Handing off the small stuff

If handing off a feature saves a quarter to a third, why not hand off lookups? We let delegate take read-only work. Told to hand off a lookup, Opus turned it into an edit task (the worker wrote the answer to a file, the check printed the file) and the runs cost **17–40% more**, leaving stray files in the repository (corrected 16 Oct: worker costs were undercounted; we had 13–33%). A hand-off takes two to five requests of its own; a lookup takes one. Reverted.

The projection that was wrong

An early analysis projected that hand-offs save more the fuller the session, **–53%** for the feature at 200k tokens, by assuming each step re-reads more. We measured it in a 233k-token session: the feature saved **35%** (37% before the workers' cost was corrected), about what it saved fresh, and a small question handed off cost up to **3.6 times** as much (the lookup 2.5 times) (SUITE.md). The projection assumed Opus needs as many steps in a session that knows the code as in a fresh one. It needs far fewer.

Moving text before the cache boundary

cells' guidance is the same text in every session. In theory, putting it in a system-prompt section shared across sessions means writing it to the cache once and reading it ever after. Measured: **+1.0% [–5%, +8%]**, no saving; the first request wrote no less. Reverted.

Keyword pointers

BM25 ranking to point Claude at files before it searches. It failed calibration before a single benchmark run: silent on 11 of 13 prompts and wrong on the other 2. The version that survived is deterministic (note 4).

The log squeezer

squeeze replaced a long test log with its failures (file, line, message) and kept the full log on disk. Locally it is excellent: all 18 failures of an 18-failure run kept, 85–99% of the text gone, in about a millisecond. On a task with 18 failing tests it fired **0 times in 4 runs**, because Opus piped every test run through `tail` or `grep` itself, 24 times across the 8 runs. We tried Claude Haiku 5.5, expecting a smaller model to dump

whole logs: it piped through `tail` too (44 times in 8 runs), and squeeze fired 0 times again. A tool for a habit neither model has.

Compression that forgets

ScaleDown's Compress keeps the passages that look relevant to a question. Asked five factual questions about click from a compressed copy, a model answered 4 of 5 when each question had its own compression, and **0–1 of 5** when one compression had to serve all five; one answer was confidently wrong (exit code 1 instead of 2) because the line that said 2 was gone. In a coding session the next question is never known in advance (note 7).

Windows after a search

The first versions of **seek** attached to a search's result the lines around its hits, so Claude would not need a request of its own to read them. **seek 0.1** fired in **1 of 36 runs**: Opus's searches sit behind a `cd`, in chains and in pipes, and it took only a plain `grep`. **seek 0.2** read any search's output, fired, saved 7.9% of the requests, and cost **+2.1%** fresh. Priced with what the windows left in the context, re-read by every later request, **+6.3%** [+1.4, +11.2]. Replayed on real sessions, a window held Claude's next read in 0.5 to 12% of searches. What replaced it is an order at the prompt (note 11).

An order in the middle of a search

pathfinder waited for the third consecutive search in a large context, where long chains of looking around spend their money, and then told Claude to hand the rest of the search to a cheaper read worker. The order came in 6 of 12 runs; Claude handed off **once**. Cost **-3.4%** [-11.3, +5.1], not shown. The same kind of order beside the person's prompt is followed every time; inside Claude's own search it is not (note 11).

A code graph we did not adopt

Trail-Code, a code graph with about 10,000 GitHub stars, is not ours. We measured it because its own benchmark reports it about a third cheaper. Installed into Claude Code the way its README says, it saved nothing we could measure: alone, **-1.8%** [-5.9, +2.7] against plain Claude Code over 64 runs a side. Added to the pack it cost **+12.8%** fresh and **+32.1%** in a long session (+4.0% without its Stop hook, which ended a turn with a "team learning" in place of the answer); against the pack 0.6.0, **+19.0%** [+13.1, +25.1]. Its own best case, the bundle its benchmark puts in front of every task, rebuilt in Claude Code: **+3.2%** [-7.3, +15.0]. Two of its ideas were borrowed and measured on their own (note 4). The whole story, and what we do not claim, is in note 3.

The mod nobody measured

turn-savers (lighter thinking on routine steps, the code at a failure attached to the failing run) was built and never measured. So it does not ship. A mod without a measurement is a guess with a version number.

What the graveyard teaches

- **The first good number is usually too good.** Write down what would confirm it, then run it again.
- **A check that passes is not a check that checks.** Agents' own tests share their blind spots.
- **Models have habits.** They grep, they tail logs, they keep work they can do. A feature that fights a habit loses; one that answers inside it wins.
- **Count what stays.** A step saved by adding text to the conversation is paid for again by every later request.
- **Measure what gets installed.** A tool's own harness measures its mechanism; a person gets the install.
- **Removing is a result.** Every grave here made the pack cheaper or simpler.

Method & limits

Delegate bench results: `benchmarks/delegate-bench/` and `SUITE.md`. The lastenheft runs used Claude Haiku 5.5 on tasks from a private codebase; the preregistration texts and analysis stay in the lab with them. squeeze: `results-squeeze.jsonl`. seek's windows: `results-seek.jsonl`, `results-seek2.jsonl`, priced with `carry.py`. pathfinder: the PFD runs in `results-seek03-docs.jsonl`. ScaleDown measurements: note 7.

Sources

- Nosek, B. A., Ebersole, C. R., DeHaven, A. C., & Mellor, D. T. (2018). The preregistration revolution. *PNAS*, 115(11), 2600–2606.
- Ioannidis, J. P. A. (2008). Why most discovered true associations are inflated. *Epidemiology*, 19(5), 640–648.

One server per extension

Lab notes 6 · 15 Oct 2026 · written by Claude from this repository's data

lsp-plus answers Claude's code searches from a language server. That only works where a server answers. The obvious next question, asked by someone who works with infrastructure all day: what about Terraform, Ansible, Kubernetes YAML? This note is about adding languages, and how often "there is a language server for that" turned out to be the start of the work, not the end.

The rule that shapes everything

Claude Code picks a language server by file extension, and only by extension: no file patterns, no first-line detection. Each extension gets exactly one server; when two plugins claim the same one, the first registered wins (plugin docs). For most languages that is fine. For `.yaml` it is a problem, because Kubernetes manifests, Compose files, GitHub Actions, Ansible playbooks and Helm values all share it.

Five languages, five surprises

We gave lsp-plus Go, Kotlin and Ruby (whose servers it could install but whose files its grep answer ignored), and declared servers for shell scripts and Terraform. Then we tested each against a real server, in a real Claude Code session, on a small repository where we knew the right answer.

- **Go** worked at once. gopls even found callers through method receivers (`func (c *Calculator)`
`Total`).
- **Shell** worked after one change. bash-language-server has no call hierarchy, so a search for a function got no answer. lsp-plus now falls back to the server's references when there is no call hierarchy.
- **Ruby** crashed with exit code 78 before answering anything. ruby-lsp refuses to start in a project with a `Gemfile` but no `Gemfile.lock`. Most apps commit the lock file; many gems do not. Now in the setup guide.
- **Terraform** answered symbol searches but found no references. Our own LSP client got the same, so it was the server, not Claude Code. Its log said why: no provider schema for `hashicorp/aws`. terraform-ls only sees references inside blocks whose schema it knows, and the Homebrew build of terraform-ls ships without the provider schemas HashiCorp's own build embeds (0 schema entries in the binary against 6). Installed from HashiCorp's tap, every `var.region` was found.
- **Kotlin** we could not test: JetBrains' Homebrew formula pointed at a download that returned 404 that day. It is covered by tests only.

Terraform also needed lsp-plus to understand its names. A grep for `var.region` or `aws_s3_bucket.logs` searches for an address, not an identifier. lsp-plus now maps an address to the block that declares it, asks the server in each module (directory) where the grep found it, and answers with the references:

```
'var.region' per the language server: 4 references in code, in 3 files (code only: st
main.tf: 2
outputs.tf: 2, 6
providers.tf: 2
```

Ansible and Kubernetes: no references to give

For YAML the research came first, and it was sobering (sources):

- The **Ansible language server** offers hover, completion, definition and diagnostics, but no references and no symbols; its "definition" jumps from a module name to the module's Python source. A pull request adding references for handlers, variables and roles was closed unmerged. And it decides a file is Ansible by path patterns and content, which Claude Code cannot express.
- **yaml-language-server** with Kubernetes schemas validates structure (a misspelled field, a string where a number belongs) but has no references: its "definition" only jumps from a YAML alias to its anchor in the same file.
- No language server we found resolves the names that actually connect a Kubernetes deployment: a Service's selector to the pods it selects, a `configMapRef` to the ConfigMap, a RoleBinding to its ServiceAccount.

So the question became: one YAML server for everything is forced on us anyway; could it also check the names?

yaml-plus

yaml-plus is one server for every `.yaml` file. It runs yaml-language-server underneath for syntax and schemas, choosing the schema from each file's content (Kubernetes by `apiVersion` and `kind`, Ansible by structure). It skips Helm templates, which are not YAML until rendered. On top it indexes the repository's names and checks the references between them:

- **Kubernetes:** ConfigMaps, ServiceAccounts and volume claims a pod uses; Services and the workloads they select; Ingress backends, autoscaler targets, role bindings; kustomize paths.
- **Compose:** `depends_on`, networks, volumes, configs, secrets.
- **GitHub Actions:** `needs`, `needs.X.outputs`, `steps.X`, inputs, local actions.
- **Ansible:** handlers and `notify`, roles, included task files.

It also answers go-to-definition and find-references for those names.

Its rule is to say nothing it is not sure of. It never flags a Secret (they are often created outside the repository), never a name from a remote base, never a variable. Run over about 30 local repositories and 6

public ones (Argo CD, kube-prometheus, the Kubernetes examples, awesome-compose, an Ansible book, VS Code), it raised one warning in 5,041 name uses: a PersistentVolumeClaim whose manifest exists only inside a README. It indexes 4,000 manifests in about 0.7 s.

One thing we did not expect. In a live session, yaml-plus's warning reached Claude after an edit:

```
ConfigMap "cart-config" (namespace shop) is not declared in this repository (did you mean "cart-config"?)
```

Reading the same broken file delivered nothing. Claude Code syncs a file to its language server when it is edited or asked about, not when it is read. A checker that only speaks after edits cannot help Claude *find* a broken reference, only avoid making one. yaml-plus therefore gained hooks: a read of a YAML file gets that file's name problems, and a grep for a name that hits YAML files gets where it is declared and used, the way lsp-plus answers code searches.

What it measured

We generated a 54-file deployment repository (ten services across Kubernetes, Compose, two GitHub Actions workflows and six Ansible roles) and four tasks:

- a ConfigMap rename across 11 references;
- six planted broken references, one in each kind of file and two more in Kubernetes;
- "which objects refer to this ServiceAccount";
- a Compose change.

Every task was graded by parsing the YAML, never by the server under test.

Model	yaml-plus added to the pack	95% interval	Runs	Correct
Opus 5.5	+2.0% cost	-1.3% to +5.3%	64	all
Haiku 5.5	about the same, at roughly one cent a run		32	all

yaml-plus spoke 56 times in Opus's runs. Opus found all six broken references and completed the 11-reference rename without it. We expected Haiku, a weaker model, to miss what the checker catches; it missed nothing either.

Fifty-four files are few enough for any model to read whole, and a checker cannot save what a model never misses. The question yaml-plus exists for is different: do names resolve correctly every time, and faster, in a repository too big to read?

So we made one. The same generator with 300 more services comes to 954 YAML files, and we hid ten broken references in it: the six from before and four among the generated services (a misspelled ConfigMap, a ServiceAccount spelled with the letter O for a zero, a Service whose selector matches no pods, an autoscaler aimed at a Deployment that does not exist).

Model	Without yaml-plus	With yaml-plus
Opus 5.5	10 of 10 in every run, \$0.61, 10.2 requests	10 of 10 in every run, \$0.70, 10.0 requests
Haiku 5.5	10 of 10 in every run, \$0.03	10 of 10 in three runs, 8 in one, \$0.03

Four runs per cell. Opus did not read 954 files. It listed them, read one of each kind, and then **wrote its own reference checker**: about 50 lines of Python with PyYAML, which it ran over the whole repository, and confirmed the result with `kubectl kustomize`. In one step, it built for this task the tool we had built for every task.

That is the honest end of this experiment. Within the time we could give it, we did not find the repository where yaml-plus earns its place: small ones are read whole, large ones get a checker written on the spot. The server costs no tokens until it speaks, only memory (about 200 MB for 4,000 manifests, plus yaml-language-server), so it does no harm to keep. But nothing here says it helps, and by our rule that keeps it out of the pack. A real infrastructure repository with irregular names, Helm charts and kustomize overlays, too tangled for a 50-line script, is the test it would still need.

Method & limits

Language checks: one headless Claude Code session per language on a small repository with known answers, the hook's note read from the transcript; terraform-ls also queried directly with a minimal LSP client. yaml-plus: 44 server tests and 13 hook tests, and the false-positive sweep described above, run locally. The YAML benchmark's generator and grader are in `yaml_repo.py`; runs in `results-yaml.jsonl`, `results-yaml-b.jsonl` and `results-yaml-haiku.jsonl`; the 954-file repository in `results-yaml-big.jsonl` and `results-yaml-big-haiku.jsonl`.

Sources

- Claude Code, Plugin components: LSP servers.
- HashiCorp, terraform-ls, `docs/schema.md` (provider schemas bundled at compile time in HashiCorp's released binaries).
- Red Hat, ansible-language-server, and the closed pull request #2647.
- Red Hat, yaml-language-server.
- Ruby LSP; bash-language-server; gopls.

Small models as plumbing

Lab notes 7 · 15 Oct 2026 · written by Claude from this repository's data

If every token in Claude's context is paid for again on every step, an obvious trick is to shrink what enters it. A long test log, a big diff, a sprawling grep result: hand it to a small, fast model that compresses or summarizes it, and give Claude the short version. We tested that with ScaleDown, an API with four modes (Compress, Summarize, Extract, Classify) and latencies under a second for small inputs.

The price question settled itself. The person running this lab paid €2.50 for 50 million tokens, about \$0.054 per million: a quarter of what Claude charges just to *re-read* a cached token. Every call we made cost under 1% of what the same text would cost in Claude's context. So price is not the question. Whether the short version keeps what Claude needs is.

Compress: the right passages for the wrong question

Compress keeps the passages it judges relevant to a question and drops the rest. On click's source it kept 1–5% of the text, in 1.6–3.6 s. We asked five factual questions about click, with answers checked in the source, from the original text and from compressed copies:

Text given to the model	20k-token context	60k-token context
Original	5 of 5	5 of 5
Compressed once per question	4 of 5	4 of 5
Compressed once for all five questions	0 of 5	1 of 5

The miss in the per-question case was the same both times: the line of `confirm()` that shows `[Y/n]` had been cut. In the shared case one answer was confidently wrong: an error class "inherits exit code 1", because the line setting `exit_code = 2` was gone while its docstring stayed.

An agent's next question is never known when its context is built. Compression that keeps what matters for a question is compression that drops what matters for the next one. Prompt-compression research gets its best ratios by making compression question-aware (Jiang et al. 2024); the other side of that is a result that fits the one question it was made for.

Summarize: confident, and wrong about the important part

On a test log with three failures, Summarize named the three failing tests every time and **never** kept a file and line number. On a log with 18 failures, four of five summaries were just the counts line, every failure gone.

The worst case was a report from a worker model (the kind delegate hands back). It truthfully listed a diff and a test run with three failures, and then said, falsely, *"the new tests pass"*. A plain summary repeated the false claim **3 of 3 times**, and once did not mention the failure at all. Told what to keep ("failing tests, exit codes, claims that contradict the check"), it kept the failures and flagged the false claim every time. Which raises the obvious question: if you have to tell it what matters, why not extract what matters yourself?

Extract: right, and slower than a parser

Extract with a schema (failing test, file, line, message) got the test logs right: all failures, all locations, every time. It took 4–10 s, and one call failed after 77 s. The local parser we had already written for the same job gets the same logs fully right in about a millisecond, with nothing leaving the machine. (On a large diff, Summarize recovered at most 41% of the changed functions and Extract at most 26%, where reading the diff's own hunk headers gets 80%.)

Classify: the one job it did perfectly

Classify sorts a text into labels you give it. We gave it the 8 benchmark prompts and 17 everyday ones and seven labels: lookup, explain, small edit, rename, feature, tests, other. Asked one prompt at a time: **25 of 25** right, median 0.42 s, \$0.00003 a call. (In one batch call of all 25 it got one wrong, though the documentation says calibration is the same as for single calls.)

That is exactly the missing piece from note 2: Opus never hands work to a cheaper model unless told, and an order from a mod is followed every time. A classifier that recognizes "implement this feature" or "write tests for this" in 0.4 s could give that order for exactly the work where a hand-off pays (22–35% less on features and tests), and stay silent on small questions, where a hand-off costs up to 3.6 times as much.

So we built it. **route** is a small mod: when you send a prompt, it asks Classify what kind of work it is, with the same seven labels and rubrics that scored 25 of 25, and waits up to three seconds. If the answer is feature or test work with a score of at least 0.7, it attaches delegate's order beside the prompt: *the person's setting for feature work is: implement it through the hand_off tool*. Anything else, or no answer in time, adds nothing.

On our eight click tasks, four runs each, all 32 correct (`results-route.jsonl`):

	The order given	Hand-offs	Cost, the pack → the pack with route
The feature	4 of 4	4	\$0.61 → \$0.45 (9.5 → 5.2 main requests)
The coverage task	4 of 4	4	\$0.67 → \$0.41 (8.5 → 4.2)
The six other tasks	0 of 24	0	about the same

Every routing decision was right: the order on all eight feature and test prompts, on none of the other 24. On the two tasks it routed, the pack with route cost **33.8% less** than the pack alone [−39.4%, −27.4%]; on the other six, +4.3% [−0.1%, +9.0%], within the runs' spread. Over all eight, **−14.2%** against the pack [−17.9%, −10.1%] and **−19.1% against plain Claude Code** [−22.5%, −15.3%].

Corrected 16 Oct: worker costs were undercounted. A worker's transcript records its output as each response began; with it counted in full, from Claude Code's own totals, the routed tasks went from −39.6% to −33.8% and all eight from −17.0% to −14.2% (CORRECTION-1016.md).

This closes the loop from note 2. Opus never delegates on its own and always follows an order; a 0.4-second classifier decides when an order is worth giving. What it costs: a ScaleDown key, about 0.4 s (up to 3 s) before each prompt starts, \$0.00003 a prompt, and the prompt's text leaving the machine. So it can only ever be opt-in.

What it adds up to

Use	Verdict	Why
Compress Claude's context	No	Loses facts the next question needs
Summarize logs or worker reports	No	Drops locations; repeats false claims unless told what to keep
Extract failures from logs	Not needed	A local parser is as right, thousands of times faster, and private
Classify prompts for routing	Yes, opt-in	25 of 25, 0.4 s; routing feature and test work to a cheaper worker: −14% over the pack

Small models make good plumbing where the job is narrow and checkable: a label from a fixed set, on which a stronger model then acts. They make poor editors of what a stronger model will reason over, because they cannot know what it will need.

Privacy is part of the cost: every call sends the text to a processor in the US. ScaleDown's data processing terms, as provided with the account, promise zero retention and no training on customer data. We sent only public code (click) and text we generated, and any mod that used it would have to be off by default and say so.

Method & limits

151 API calls, 1.8 million billed tokens in total, on click's source and on logs, diffs and greps generated from it; answers scored against ground truth worked out from the code. Fidelity of compressed context judged by Claude Haiku 5.5 answering from the text alone. Five runs per case unless noted; Classify on 25

prompts, one run each plus repeats. The key ran out of credits midway through the first round; the second round ran on new credits. Scripts and results stay in the lab with the API client.

Sources

- ScaleDown, documentation, model overview and Classify reference; price and data terms as provided with the account.
- Jiang, H., Wu, Q., Luo, X., Li, D., Lin, C.-Y., Yang, Y., & Qiu, L. (2024). LongLLMLingua: Accelerating and Enhancing LLMs in Long Context Scenarios via Prompt Compression. *ACL 2024* (Vol. 1), 1658–1677.

Our plain runs were not plain

Lab notes 8 · 15 Oct 2026 · written by Claude from this repository's data

Every comparison in these notes rests on one thing: knowing exactly what loaded in each run. On 15 October we found out that for a day and a half, in one condition, we had not known it. This note is about how that happened, how close it came to becoming a finding, and what we changed.

The good news that wasn't

The bench names what loads in each condition. Our mods are listed in an environment variable; "plain Claude Code" is the same list, empty. That had worked for hundreds of runs.

On the afternoon of the 15th we re-ran plain Claude Code on the click tasks, to compare the pack against today's version of Claude Code rather than the one from two days earlier. The result was striking: plain cost the same as the pack (+2.1% for the pack [-2.4%, +7.0%]). The feature task, which plain had needed 14.8 requests for, now took 9.2.

There was a ready story: Claude Code had caught up. A new version, smarter defaults, the pack's advantage gone. It would have made a good note.

Before writing it, we looked at what plain was doing differently, request by request. In the feature runs, plain Claude Code called `mcp__cells__run_cell` five times. That tool exists only in our cells mod.

What had happened

On 14 October at 00:33, the person running this lab installed the released modpack from its marketplace, to use it day to day. From then on, every headless run that read that person's settings loaded the installed pack as well.

- **In the pack's conditions** this did no harm. Where the lab's copy of a mod and the installed copy share a name, only the lab's copy runs. We checked this rather than assumed it: lsp-plus answered each grep exactly once, before the install and after.
- **In plain Claude Code's condition**, nothing replaced the installed mods. "Plain" was the pack, version 0.4.0. The +2.1% was the pack measured against itself.

The opposite mistake, the same day

Our other harness, for SWE-bench, avoids this problem by leaving the person's settings out entirely. That also left out the one thing every condition is meant to have: the official Python language server plugin, which the person's settings switch on. In the first 45 SWE attempts, lsp-plus had no server to ask and

could never answer. We found it by asking the LSP tool directly under the harness's settings: *"No LSP server is configured for .py files."*

A clean environment can remove what a condition needs, just as a dirty one can add what it must not have.

What we changed

- Every run of the click bench now switches the person's installed plugins off explicitly, whatever the condition, and every SWE run switches the official Python language server on explicitly.
- We checked the fix the way we caught the fault, by what loaded. A plain run shows no mod tools at all, and needed 14 requests for the feature, as plain did before the install.
- The contaminated runs are kept, labelled for what they were (`results-v-1015-installedpack.jsonl`): the installed pack 0.4.0 against the lab's pack, the same thing measured twice, a few percent apart. The 45 SWE attempts without a language server are kept apart too.

With plain Claude Code really plain, today's numbers are back where the release measured them, and in a long session larger:

The pack against plain Claude Code, 15 Oct	Change in cost	95% interval
Fresh session, 8 tasks	-5.7%	-9.6% to -1.6%
Long session, 8 tasks (warm runs)	-37.4%	-40.9% to -33.9%

"Warm runs" is the other thing a long session can hide. One of the 46 long runs started with 16k tokens in the cache instead of 220k, because its prepared session had been idle for more than an hour, and paid \$1.91 to write the cache again for a five-request task. We report the warm runs and say so; the bench records each run's first cache read so that this can be checked.

What we keep from it

- **Name what loads, then check what loaded.** The transcripts say which tools each run had. A five-line check would have caught this on the first run.
- **Distrust good news about the baseline as much as good news about yourself.** "The competition caught up" was as tempting a story as "our tool saves 40%", and it was wrong for a mundane reason.
- **A setting has two sides.** Removing the person's settings removed a plugin the conditions needed. Every exclusion needs the same check as every inclusion.

Method & limits

Click 8.5.0, Opus 5.5, headless Claude Code 2.1.299; plain condition re-run with the installed plugins switched off (four runs per task fresh, two in a long session prepared without any mods). Pack runs from the same day (`results-trail.jsonl`). Data: `results-v-1015.jsonl` ; the bench's switch is `neutral()` in `bench.py` .

Beyond click

Lab notes 9 · 15 Oct 2026 · written by Claude from this repository's data

Every number in the first eight notes comes mostly from one library: click, 79 files of Python, tasks we wrote ourselves. A fair reader's first objection is the obvious one. Maybe the pack is tuned to click. Maybe the code-graph tools would look better on code they were built for. So we ran the main comparisons again, on code we did not choose and tasks we did not write.

Three new grounds

- **A TypeScript library.** chrono, a natural-language date parser (267 TypeScript source files, 753 tests, jest), at release v2.10.3. Eight tasks of the same kinds as click's (trace the callers, rename, explain, look up, list subclasses, a feature to a spec, raise coverage, change a signature), each graded by hidden tests or fixed answers. Every grader was checked both ways: 0 on the untouched code, 1.0 on a reference solution.
- **Real bug reports.** 30 tasks from SWE-bench Verified (Jimenez et al. 2024; OpenAI 2024): issues filed against django, sympy and other Python projects, each stated as the original issue text, graded by the project's own hidden tests. One attempt per task and condition, all four conditions of a task running at the same time.
- **Deployment configuration.** The YAML repository from note 6, small and large.

The pack on TypeScript

64 runs, Opus 5.5, four per task and condition (`results-ts.jsonl`):

	Change in cost, the pack against plain Claude Code	95% interval
All eight tasks	-7.6%	-12.1% to -3.1%
Multi-step tasks	-6.3%	-12.5% to +0.8%
Quick questions	-9.9%	-13.7% to -5.9%

The pack took fewer requests on all eight tasks: the feature in 10.5 instead of 13.2, the rename in 4.2 instead of 6.0. Quality was the same; each side had one partial answer (plain missed one term of an explanation once, the pack one class of a list once).

One difference from click stands out. On click's quick questions the pack cost 5.5% more on one day and 3.6% less on another, about even (note 1). On TypeScript it saved 9.9% on them. Questions about a typed

codebase lean on symbol lookups more, and lsp-plus answers those from the TypeScript server inside the search.

The pack on real bugs

120 attempts, Opus 5.5 at its highest effort, as the harness runs SWE-bench (`swe-bench/2026-10-15-swe`):

Condition	Bugs fixed	Cost of all 30	Requests per bug	Time per bug
Plain Claude Code	29	\$15.01	12.9	213 s
The pack	30	\$13.17	9.9	144 s
The pack with scout and blast	29	\$14.13	10.2	150 s
The pack with Trail-Code	29	\$14.59	11.5	162 s

With one attempt per bug, cost intervals are wide: the pack against plain -12.3% [-32.5%, +11.9%]. What carries the result is consistency rather than size: on real issues in projects it had never seen, the pack again needed about a quarter fewer requests and finished in two thirds of the time. The one bug only the pack fixed (sympy #22080) is most likely chance at this sample size.

Trail-Code, measured where it reports its own SWE-bench results, again cost more than the pack alone: +10.8% [0.0%, +25.5%]. Scout, the definition pointers from note 4, spoke on 20 of the 30 issues, which name functions and classes far more often than our tasks; it still saved nothing (+7.3% against the pack [-6.7%, +20.7%]). That settles scout: it stays out.

Deployment configuration

Note 6 has the details. On 54 files and on 954 files, Opus and Haiku found every planted broken reference with or without yaml-plus; on the large repository Opus wrote its own reference checker in one step.

What holds, and what does not

- **The pack's saving holds** on a second language, on real bug reports and on code it was not built around: fewer requests at equal quality, everywhere we looked. Its size varies (-6% to -12% fresh, -37% in a long session on click), and with single attempts per bug the SWE interval still crosses zero.
- **Trail-Code cost more than the pack alone on every ground** we measured: click fresh, click long, and SWE-bench.

- **The ideas we borrowed did not survive replication.** Scout's small saving on click did not appear on real issues. yaml-plus found nothing a model did not find itself.
- **The router (route) has not been replicated yet.** Its -14% (-17% before worker costs were corrected on 16 Oct) is from click alone; bug reports are mostly neither features nor tests, so SWE-bench is the wrong ground for it, and a set of real feature requests is the right one.

Method & limits

TypeScript: chrono-node v2.10.3, tasks and graders in `ts_repo.py` ; runs with the person's installed plugins switched off and the official TypeScript language server on. SWE-bench: 30 validated SWE-bench Verified tasks, the harness in `pilot/swe.py` of the lab (each condition loads frozen copies of the mods; the official Python language server switched on in every condition, see note 8); one attempt per task and condition, so per-task differences are noise and only totals carry weight. All costs at list prices from transcripts.

Sources

- Jimenez, C. E., et al. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *ICLR 2024*.
- OpenAI (2024). Introducing SWE-bench Verified.
- chrono, natural-language date parser, release v2.10.3.

Two requests

Lab notes 10 · 15 Oct 2026 · written by Claude from this repository's data

A hand-off to a cheaper worker saves money because the worker does the building. But the main model still pays for every request it makes around the hand-off, and in a long session each of those re-reads the whole conversation. This note is about getting those requests down to the two that cannot go: handing the work over, and answering.

Claude does not hand off on its own

Measured again with this release's delegate: in 16 runs on the feature and test tasks of click and of a TypeScript library, with delegate loaded and nobody asking, Claude handed off **0 times**. Before that, 0 of 174 (note 2). In the person's own sessions since the pack was installed, Claude called `hand_off` 70 times; 66 came after an order or after the person had asked for it in one of their last six prompts, and 4 did not. And the Claude writing this note, in a long session full of fully specified build work, did not hand off once until the person asked why not.

An order is followed every time; an invitation almost never. So everything below is about what happens after an order: delegate's own, when a plan is approved, or route's, when a classifier says a prompt is feature or test work (note 7).

Where the requests went

We took the 16 routed runs of the feature and the coverage task and listed every request of the main model. On average: **1.6 requests looking around** before the hand-off, the hand-off itself, and **2.5 after it**, the final answer included.

The requests after were not blind re-runs. They were questions the evidence had left open:

- "The run reports **31,000 deselected**, so I'll confirm the new tests actually ran" (4 of 16 runs).
- "Run the new test file **on its own**": the check had run the whole suite.
- "Pyright flagged new lines, I'll check whether that matters."
- And real review: "one assertion is always true, I'll make it a real one". That request is worth paying for.

delegate already ran the check command itself and showed the diff. What it did not do was answer those questions.

A report in a fixed form

The idea came from the person running this lab: let the worker answer in a fixed form that a program can check. The worker now ends its report with a block:

```
VERIFY
tests: PYTHONPATH=src python -m pytest -q tests/test_duration.py
open: none
```

delegate runs that command beside the check, counts both outputs, and puts one line first in its result:

Verified: the check passed (2020 passed); the new tests passed on their own (27 passed); nothing left open. If the diff is what you asked for, report to the person now: nothing is left to run.

Or `Not verified:`, with each reason: a check that failed, new tests that did not run (0 selected is not a pass), test files changed without a command that ran them alone, a worker that stopped on an error, or what the worker itself left open. The parsers are strict on purpose: an output they do not recognise is "not counted", never "passed".

The order changed with it: make the project's whole test suite the check, and on a Verified result with the right diff, report without running anything again.

Handing off before reading the code

With the verified report, the requests after a hand-off fell to one. But in the same runs a second thing showed: **allowed to look around first, Claude kept the work in 5 of 16 runs**. It read the code, judged the job small enough, and did it itself in 6 to 9 requests.

So the order now says to hand off before reading the code: the worker reads it, and the task is written from the request as given. Claude followed that **24 of 24 times**.

What it measured

All 52 runs below were correct.

On click, one batch:

Feature and coverage	Route as released	+ verified report	+ hand off before reading
Handed off	6 of 8	5 of 8	8 of 8
Main-model requests	5.9	5.0	2.6
Cost per job	\$0.53	\$0.47	\$0.38

The last column against the first: cost -28% [-42%, -10%], main-model requests -55% [-68%, -36%], time -12% (not shown).

Corrected 16 Oct: worker costs were undercounted. A worker's transcript records its output as each response began; counted in full, from Claude Code's own totals, every cost with a worker in it rose. Here the three columns were \$0.50, \$0.44 and \$0.34, and the saving -33%; below, the feature with Sonnet was \$0.43 (-57%) and the tests \$0.27 (-40%) (CORRECTION-1016.md).

On TypeScript, against the pack alone in the same batch:

	Pack alone	Route, Sonnet worker	Route, Haiku worker
Writing tests: cost	\$0.45	\$0.29 (-36%)	\$0.24 (-46%)
Writing tests: main-model requests	6.2	2.0	2.0
A feature: cost	\$0.99	\$0.48 (-51%)	\$0.44 (-55%)
A feature: main-model requests	11.0	2.0	2.8
A feature: time	214 s	125 s (-42%)	178 s

Two requests is the floor: one to hand over, one to answer. Writing tests reached it everywhere; the feature reached it with Sonnet as the worker.

Haiku or Sonnet. On writing tests Haiku was cheaper (-16%). On the feature it needed almost twice the steps (12 against 7), cost about the same (8% less, within the runs' spread: [-24%, +8%]), took longer, and twice left points open that Claude then had to settle. One of them was worth it: the task's example contradicted its own rule about week 53, and Haiku said so. Build work stays with Sonnet, for the steps, the time and the open points, not for the cost.

A classifier that hesitated

The first TypeScript feature runs never handed off. route had asked ScaleDown what the prompt was, and it said "feature" with a score of **0.66**, under route's threshold of 0.7: a long, detailed specification read less like "a feature" than a one-line request does. TypeSafe's Jev, asked the same, said "feature" with a

probability of **1.0**. On our 25 calibration prompts both were right 50 times out of 50; Jev answered in 0.25 s against 1.15 s. route now takes Jev when its key is set.

What this does not show

- Four runs per task and condition. The direction is the same on every task; the sizes have wide ranges.
- **Time improved less than cost** (−12% on click, −42% on the TypeScript feature): the worker now does the reading, in its own requests.
- **Handing off before reading removes Claude's own look at the code.** Every prompt in our benchmarks is a complete specification. On a vague request ("make the export faster"), a task written without reading the code may be a worse task. We have not tested that. It is the next thing route should ask its classifier: is this request fully specified?

Method & limits

Arms RT (route as in 0.5.0), RTV (with delegate 0.3.0's verified report), RTD (and the order to hand off before reading), RTDH (RTD with Haiku as every worker), Z (the pack alone), all with the person's installed plugins switched off (note 8). Each comparison ran in one batch, so times compare too. Opus 5.5, click 8.5.0 and chrono 2.10.3. Data: `results-route-verify.jsonl`, `results-ts-route.jsonl`, `results-ts-route-jev.jsonl`, `results-unmasked.jsonl`; the classifiers in `classifiers/jev-vs-scaledown.json`.

Sources

- TypeSafe AI, Jev documentation and privacy policy.
- ScaleDown, Classify reference.

What looking around leaves behind

Lab notes 11 · 16 Oct 2026 · written by Claude from this repository's data

Before Claude changes code, it looks around: a search, a read, another search. Each step is a model request, and each request re-reads the whole conversation (note 1). What we had not priced is what a step leaves behind: a search's output and the lines it read stay in the conversation, and every later request reads them again until a compaction removes them. This note prices that, shows how it sank the first two versions of a mod, and follows the version that ships, seek, to a rule about where an instruction has to ride.

A token's second life

In the author's own sessions (nearly a thousand, counted only in aggregate), a token added to the conversation was re-read by a median of **143 later requests** before a compaction removed it (interactive sessions; 22 in headless `claude -p` ones). On Opus 5.5 a token costs \$8 per million to write to the cache once and \$0.20 per million each time a request reads it again: 143 re-reads add \$28.60. **About \$37 per million over its life, more than four times what writing it cost.** Over their life, the results of looking around came to about 11% of all the cost of the sessions before the pack was installed (15% in headless ones).

Our benchmark never saw this. A run ends when its task is done, so nothing it added is read again. We added a column, **carry**: the tokens a run added to the context $\times$ 140 later requests $\times$ the cache-read price, beside the run's own cost (`carry.py`). With carry, a run costs roughly what the same task costs inside a session that goes on.

Windows after a search

The first seek came from the same sessions. The most common predictable step after a search was opening a file it had just listed, mostly to read the lines around a hit. So seek attached them to the search's result: a narrow search got the lines around its hits (7 before, 30 after), a broad one its files ranked and the top file's lines, an empty one up to three wider searches.

- **seek 0.1** took only a plain `grep`. Opus's searches are rarely plain: about 100 of 110 sat behind a `cd`, in a chain or in a pipe. It fired in **1 of 36 runs**.
- **seek 0.2** read the hits from any read-only command line's output. Now it fired, and saved requests: 4.9 $\rightarrow$ 4.5 per task in a fresh session (−7.9%). The cost went the other way: **+2.1%** [−1.4, +5.5] fresh; **−4.5%** [−7.4, −1.6] in a long session, where a saved request is dearer.
- **With carry**: **+6.3%** [+1.4, +11.2] fresh, and **−2.2%** [−5.2, +0.7] in the long session, no longer shown. The windows saved requests by adding text, and the text outlived the requests it saved.

Replayed against what Claude actually did next in the mined sessions, a window held Claude's next read in 0.5 to 12% of searches, depending on its size. At every size, its tokens cost more over their life than the requests it saved. The windows are retired (note 5).

An order in the middle of a search

The same sessions showed where the money sits: from the third consecutive request of looking around on, chains held 15% of requests and 10 to 16% of the cost, mostly in contexts over 400k tokens.

pathfinder watched for exactly that, and at the third such request in a large context told Claude to hand the rest of the search to a cheaper read worker through delegate.

Its plumbing worked; Claude did not follow. In five trial runs it took the order 0 times (in four of them the search was already over). In the measured batch, in a long session that had read click's docs and tests but none of its source, the order came in 6 of 12 runs and Claude handed off **once**. Cost **-3.4%** [-11.3, +5.1], not shown. Into the graveyard.

An order with the prompt

seek 0.3 turned the idea around: say it before the search starts, beside the person's prompt, in the person's name. The text names the cost and asks for lean steps:

The person's setting for looking around in this session (about 356k tokens of context so far): every response you make re-reads all of it, and whatever you read stays in it ... So look around in few, lean steps: put the searches and reads you can already name into one command; when a search shows where to look, read those places in the same step; read only the definitions or lines you need ..., not whole files or long ranges; and do not read again what this session already holds unless it changed.

It comes with the first prompt of a session, and again after a compaction or 150k more tokens of context; in between it stays in the context. (The batches below ran with an earlier wording that called every session "long", a fresh one too; the shipped one gives only the size so far. On the eight TypeScript jobs the two wordings came out the same: +1.4% [-4.6, +7.9] between them.) Against the pack alone, in the same batch:

Where	Runs	Cost	With carry
Fresh session: four click tasks and one TypeScript explanation	20 each	-11.9% [-17.2, -5.9]	-18.0% [-23.9, -11.5]
Long session that had read click's docs and tests, not its source	24 each	-10.2% [-17.6, -2.2]	-13.0% [-19.0, -6.8]
Long session that had read all of the source	24 each	-15.9% [-22.5, -8.6]	-17.7% [-24.6, -10.4]
Fresh session, TypeScript (chrono), eight tasks	24 each	-16.4% [-21.8, -10.5]	-23.1% [-28.6, -17.1]

- **Fewer requests, and less left behind.** On TypeScript, 6.3 → 5.3 requests per task and 18.6k → 13.2k tokens added to the context; fresh on click, 14.3k → 10.9k tokens. So carry makes the order look better, not worse: the opposite of the windows.
- **The bigger the job, the more it saved.** The TypeScript feature took 8.3 requests instead of 11.3; writing tests cost 29% less, the rename 26%. The smallest lookups gained nothing and sometimes took a step more (3 requests instead of 2 for the lookup in the docs session).
- **Quality held.** Every run got full marks but two: in the fresh batch, two runs of the TypeScript explanation scored 0.83, one with the order and one without, that task's usual spread.

The map, measured and left off

seek 0.3 had a second part: a code search's result named the definitions that hold its hits, with their lines, in no more tokens than a saved request is worth at the current context. It is the window idea at its leanest. Fresh on click it changed nothing (+1.2%, not shown); on TypeScript -4.5% [-9.9, +1.2]; on top of the order in the docs session -7.9% [-15.1, -0.5], and with carry -5.5% [-11.7, +1.0], not shown. It ships switched off (`map`).

Where an instruction has to ride

The orders and hints this lab has given, side by side:

What	Where it rode	Followed
delegate: build the approved plan through a hand-off	the person's approval of a plan	8 of 8
route: hand this feature work to the worker	the person's prompt	24 of 24
seek: look around in few, lean steps	the person's prompt	44 of 44 runs
pathfinder: hand the rest of this search off	Claude's own search, mid-way	1 of 6
lsp-plus: this LSP call answers your grep (note 2)	a tool's result	0 of 45

An order in the person's name that arrives with the prompt, before Claude has a plan, becomes part of the plan. One that arrives inside Claude's own search, attached to a tool's output, meets a plan already in motion, and Claude finishes what it started. Why is our reading; the counts are not. So seek speaks at the prompt and nowhere else: short, in the person's name, with the reason.

A subagent's task

Two of the pack's instructions ride where a subagent cannot see them. cells' chaining guidance is a section of Claude Code's system prompt, and a subagent Claude starts with the Agent tool has a system prompt of its own. seek's setting rides beside the person's prompts, and a subagent never sees those either: it sees the task Claude wrote for it. In the author's sessions none of 18 subagents had the guidance, and general-purpose subagents ran about a third of the Opus requests.

So both mods now add a line to a subagent's task as it starts (Claude Code's `agent . spawn` hook), under one heading, in the person's name:

The person's settings for this work:

- Fewer round-trips: every response you make re-reads the whole conversation, so a step that needs no judgment should not cost a response of its own. ...
- Lean looking around: whatever you read stays in your context ...

Not for a fork, which has the parent's context and both already; not for a teammate, a resumed subagent or delegate's worker, whose prompt is tuned for its job. Measured with every task handed to one general-purpose subagent, the pack as 0.6.0 ships with `subagents` off and on, 5 tasks × 4 runs per arm, Opus 5.5:

	Off	On	Change
Run cost, main and subagent	\$0.729	\$0.600	-17.6% [-21.0, -14.0]
The two features and the tests	\$0.850	\$0.664	-21.8% [-26.0, -17.4]
The two explanations	\$0.547	\$0.505	-7.8% [-12.3, -2.6]
Subagent requests	10.0	6.9	-31.0% [-36.8, -24.7]
Bash calls chained	88%	99%	

- **The lines arrived.** In 20 of 20 subagents with the option on, in none with it off.
- **Fewer, fuller steps, not more cells.** The subagents put more into each Bash call; run_cell was used less, not more (2.3 → 0.6 calls per subagent).
- **Quality held.** 37 of 40 runs got full marks; three TypeScript explanations scored 0.83, two of them without the settings.
- **What it turned up.** A subagent's transcript records each response's output as the response began, a few tokens, not what it ended with. Our bench had priced every delegate worker that way. These runs are priced in full from Claude Code's own count; the earlier numbers with a worker in them are corrected (CORRECTION-1016.md).

A subagent's task is the only text of the person's that reaches it, so that is where the settings ride.

What this does not show

- **Carry is a model.** It assumes 140 later requests, near the interactive median; headless sessions re-read a median of 22. Without carry, the order still saved 10 to 16%.
- **Three to six runs per task and arm.** The four measurements agree in direction; the sizes have wide ranges.
- **Bench sessions, not real ones.** Every prompt is a complete specification. The long sessions are forks of a prepared session; in the one that had read all of the source, runs whose fork had gone cold were left out (`--warm`).
- **The README's pack-against-plain numbers were measured without seek.** seek's are against the pack, in the same batches.
- **The subagents were ordered.** Every subagent run was told to hand the whole task to one subagent. How often Claude starts one by itself, and for what, decides what the settings are worth in a real session.

Method & limits

Arms, all with the person's installed plugins switched off (note 8): Z, the pack alone; ZKO and ZKM, with seek's order or its map (at every context size); PD, the pack in a session that had read click's `docs/` and `tests/` but nothing under `src/`, and there KDO, KDM and KD (order, map, both) and PFD (pathfinder); PL and KL, the pack without and with seek in a session that had read all of `src/click` (about 220k tokens). seek 0.1 and 0.2 ran as ZK against Z, and 0.2 as KL against PL. SA-off and SA-on: the pack as 0.6.0 ships, every task handed to one general-purpose subagent, cells' and seek's `subagents` off or on; click's feature and explanation, chrono's feature, tests and explanation; checked and summed by `subagent_report.py`. Opus 5.5, click 8.5.0, chrono 2.10.3. Carry, for example: `python3 -I carry.py Z ZKO results-seek03-fresh.jsonl`; 95% bootstrap intervals over runs within tasks, whose ends move by a few tenths of a point with the arms resampled beside. The mined numbers are aggregates; the sessions stay private. Data: `results-seek03-fresh.jsonl`, `results-seek03-docs.jsonl` (pathfinder too), `results-seek03-long.jsonl`, `results-seek03-ts.jsonl`; subagents: `results-subagent.jsonl`; seek 0.1 and 0.2: `results-seek.jsonl`, `results-seek2.jsonl`, `results-seek-long.jsonl`.