READ TIME: ABOUT 30 MINUTES IN FULL. THE ABSTRACT AND THE BOXED FINDINGS TAKE 2.

Plain gh Is Hard to Beat: What It Costs an AI Agent to Read GitHub Through gh, gh-axi, GitHub's MCP Server and a Drop-in Replacement

Hud Wahab, Multimode AI

26 November 2026

Draft for review. Not yet peer reviewed.

ABSTRACT

AI coding agents read GitHub either through a command-line tool or through a tool server that speaks the Model Context Protocol. Some GitHub tools are now designed specially for agents, on the claim that they cost fewer tokens than the tools people use. We tested that claim. We compared plain `gh`, GitHub's own command-line tool; `gh-axi`, the reference GitHub tool of the AXI design principles; GitHub's MCP server; and `gh-tai`, an unreleased drop-in replacement for `gh` that we built. Agents answered two separate sets of 42 read-only questions about public repositories. Each set was written and sealed before its first use, and a script graded every answer against GitHub's own data. We ran four Claude models with two repeats each: 2,688 runs in all. The two agent-specific tools cost more than plain `gh`. `gh-axi` used 1.5 to 2.5 times plain `gh`'s input tokens and GitHub's MCP server 2.1 to 2.5 times, and neither answered more questions correctly. `gh-tai` matched plain `gh` on correct answers on every model and on tokens on three of the four; on Claude Opus 5.5 it cost 7% to 9% more on both sets. Across tools, cost followed the number of round trips between the agent and the tool far more than the size of each output. A re-run of AXI's own published benchmark found that `gh-axi` used more tokens than plain `gh` on 14 of 17 tasks. Our main finding is a negative one for tool builders: plain `gh` is already very good for agents.

Key findings in plain words

1. The `gh` command that GitHub already ships is cheap for AI agents. On simple questions it gets the answer in the fewest steps possible, and agents never had to look up how to use it.
2. The two GitHub tools built specially for agents cost about twice as much as plain `gh` in our tests, and did not answer more questions correctly.
3. Our own replacement for `gh` tied with it. It did not beat it. On one of the four models it cost a little more, on both question sets.
4. Cost comes from going back and forth. Every extra step re-sends the whole conversation to the model, about 15,000 tokens per step here. Making each tool output shorter saves far less than avoiding one step.
5. If you build a tool for agents, test it against the plain tool first, on questions you have never seen, graded by a script. The plain tool is the hard rival.

1. Introduction

An AI agent, in this paper, is a language model running in a loop. It reads a task, runs a command, reads what the command printed, and repeats until it can answer. Developers now hand agents everyday work on GitHub: check the latest release, find who opened an issue, count the pull requests merged last year. Every one of those steps costs tokens. A token is the unit a language model reads and is billed by, roughly four characters of English text. More tokens means a higher bill and a slower answer.

There are two common ways for an agent to reach GitHub. The first is a command-line tool, or CLI: a program run by typing a command in a terminal, such as `gh issue list`. The second is the Model Context Protocol, or MCP: a standard way to offer an agent a set of named tools, each described by a schema the model reads before it calls the tool. GitHub ships both, the `gh` CLI and an MCP server.

In March 2026 Kun Chen published AXI, a set of ten design principles for command-line tools meant for agents, together with `gh-axi`, a GitHub tool built on them, and a 425-run benchmark [1, 2]. The AXI site states the result as "100% task success at the lowest cost" [1]. The idea is attractive: if a tool is shaped for how agents read, agents should need fewer tokens than with tools shaped for people.

We set out to build a tool that would beat both `gh` and `gh-axi`. Before measuring, we fixed a rule for keeping it: the new tool had to beat both rivals by more than chance, on more than one model. The tool, `gh-tai`, is a drop-in replacement for `gh`. It did not meet the rule. It tied with plain `gh`. Along the way we found that the agent-specific tools, `gh-axi` and GitHub's MCP server, cost about twice what plain `gh` costs.

This paper reports that result and the method behind it. We describe a measurement design for agent tools in which the questions are sealed before use and every answer is graded by a script. We report results on two independent question sets across four models. We report a re-run of AXI's own benchmark. And we explain, with our data and outside studies, why plain `gh` is hard to beat: the cost of an agent's work is set mainly by how many times it goes back and forth with a tool, and on most everyday questions plain `gh` already goes back and forth the minimum number of times.

2. Background

2.1 Plain gh

`gh` is GitHub's official command-line tool [5]. It covers issues, pull requests, releases, workflow runs, repositories and search, and it has a general `gh api` command that can call any part of GitHub's API, the interface programs use to ask GitHub for data. It can print JSON with `--json` and filter it with `--jq`. It was written for people, but it is old and widely used, so current models have seen a great deal of it in their training data. One known weakness matters for agents: list commands stop at a default row limit (30 rows for most lists) without saying the list was cut. An agent that counts the rows it got can report a page size as a total. We saw this in our re-run of AXI's benchmark (Section 4.6).

2.2 AXI and gh-axi

AXI ("Agent eXperience Interface") is Kun Chen's set of ten principles for agent-facing command-line tools, released under the MIT licence with a reference GitHub tool, `gh-axi` [1, 2, 3]. According to the AXI study, `gh-axi` uses the same subcommands as `gh` but prints a compact structured format, adds pre-computed summaries such as counts and check results, suggests next commands, and reveals long content step by step [2].

The published GitHub study, dated 21 March 2026, ran five conditions on 17 tasks against one repository, with five repeats each: 425 runs [2]. The agent and the grader were both Claude Sonnet 4.6; the grader was a language model acting as a judge. It reported `gh-axi` at 100% success and $0.050 per task, plain `gh` at 86% and $0.054, and three MCP setups at 82% to 87% and $0.101 to $0.148 [2]. It reported 46K average input tokens for `gh-axi` against 47K for plain `gh`, and 137K to 176K for the MCP setups [2]. A re-run of the `gh-axi` condition by an outside contributor in July 2026, posted as an issue on the AXI repository, scored 71 of 85 (83.5%), partly because three grading hints had gone stale as the repository changed [4].

We used `gh-axi` version 0.1.43, as shipped. We did not tell any agent to read any tool's help.

2.3 GitHub's MCP server

GitHub runs a remote MCP server [6]. At the time of our runs its default tool sets offered 49 tools. None of them reads GitHub Actions workflow runs. Claude Code, the agent program we used, does not load every MCP tool description at the start by default. It loads them through a built-in search tool, ToolSearch: the agent searches for the tools it needs, and only their descriptions enter the conversation. The AXI study ran the MCP server both ways [2]; we used the default, ToolSearch.

2.4 gh-tai

`gh-tai` is a tool we built and have not released. It accepts `gh`'s own commands and flags, so an agent types exactly what it would type after `gh`. For a set of common reads (listing and viewing issues, pull requests, workflow runs and releases; pull request checks; labels;

repository details; search; and read-only `api` calls) it answers itself, calling GitHub directly. Its lists state the true total from GitHub, and it can count and page on the agent's behalf. Any other read is handed to `gh` unchanged, with one line on the error stream saying so. It refuses every write. It builds on AXI's MIT-licensed toolkit.

Between our two question sets we made three small fixes to `gh-tai`, each aimed at a cost seen on the first set: a plain one-line "not found" message with no suggestion to try another command; answering `gh`'s date flags on searches itself instead of handing them to `gh`; and accepting every JSON field `gh` accepts.

3. Method

3.1 Questions

We used two question sets of 42 questions each. Each set has seven questions of each of six kinds: simple lookups, single fields, questions about a related object (such as a milestone or a branch), counts and rankings, "for each of the last N" questions, and questions whose answer is an error or an empty result (such as an issue that does not exist). Every question is a read. Each set covers seven public repositories that no earlier test of ours had used. Neither set uses AXI's benchmark repository, and the second set avoids the first set's repositories.

Both sets are held out. A held-out set is a set of test questions kept away from the people building the tool, so the tool cannot be tuned to them. Each set was written by a separate agent session that did not read `gh-tai`'s code, help text or past results, and did not run it. Each set was then sealed: its files were fingerprinted with a cryptographic hash and made read-only, and the hash was checked before the first run, so any later edit would show. Each set was used once. The second set was written and sealed before the fixes in Section 2.4 were made.

Every question ends with the same instruction: give the answer on a final line that starts with `ANSWER:`. Questions about things that may not exist offer `not found` as an answer. An agent that cannot answer is told to write `ANSWER: unknown`.

A script, never a language model, grades each answer against GitHub's own data, read through GitHub's API (the interface programs use to ask GitHub for data). Because live repositories change, the script reads the true answer twice: just before the agent starts, and again at grading. An answer passes if it matches either reading, and for counts any value between the two readings passes. Each run gets one of three grades: correct, confidently wrong (an answer that does not match), or no answer (including `unknown`).

3.2 Tools and instructions

Each run gave the agent one way to reach GitHub, which we call an arm. The three command-line arms received the same instruction text, with only the tool's name changed:

```
# Tools

You have `<TOOL>` installed. Use `<TOOL>` for all GitHub operations; do not use
any other
GitHub tool or call the GitHub API another way.

Examples:
- `<TOOL> issue list --repo owner/repo`
- `<TOOL> pr view 123 --repo owner/repo`
- `<TOOL> release list --repo owner/repo`
```

The MCP arm received the same first lines, naming "the `github` MCP server", and no examples, because the examples are shell commands. No instruction mentions help. In the MCP arm, `gh`, `gh-axi` and `gh-tai` were removed from the shell's search path, so any shell route to GitHub would fail and be recorded as a rule break. Every `gh-tai` call passed through a small logging wrapper that recorded which commands `gh-tai` answered itself and which it handed to `gh`.

3.3 Agent program and models

All runs used Claude Code 2.1.331 in non-interactive mode, with the version pinned and checked on every run. We switched off everything that could leak in from the machine: user settings, memory, slash commands, saved sessions and any MCP server other than the arm's own. Each run started in a fresh, empty workspace with the instruction text in it. The agent had Claude Code's standard built-in tools (shell, file read and edit, search, sub-tasks

and web access); the MCP arm also had ToolSearch. A run was stopped after five minutes. We used four models, pinned by exact ID: `claude-sonnet-5-5`, `claude-haiku-5-5`, `claude-opus-5-5` and `claude-fable-5-2`. Other versions: `gh` 2.86.0 and `gh-axi` 0.1.43. Every run logged in with a subscription account rather than a pay-per-use key.

3.4 Run order

Each question ran twice per arm and model, in two blocks. Sonnet and Haiku ran together in one batch; Opus and Fable each ran in a batch of their own. Within a block the questions were shuffled, and each question's cells (every arm on every model in the batch) ran back to back in a shuffled order, so changes in GitHub's data or speed during a batch fell on all arms alike. Batches ran one after another, not in parallel, because parallel runs would raise GitHub search traffic and hit the search-heavy arms with rate limits. Each set took 1,344 runs (42 questions, four arms, four models, two repeats). The first set ran on 25 November 2026 and the second on 26 November 2026.

3.5 Measures

Our main cost measure is total input tokens per run. Each time the agent takes a step, the whole conversation so far is sent to the model again, and all of it counts as input. Part of that input is usually cached: the provider has seen it recently and can reuse it, and bills it at a discount. We count cached input in full, at the same weight as uncached input, because how much of a run is cached depends on timing and on what else is running, not on the tool. This makes our token counts larger than a cache-discounted bill would be. Every arm is measured the same way.

A round trip is one cycle in which the model reads everything so far, decides, and runs one or more commands. The final answer counts as the last round. A question answered with a single command therefore takes two round trips. We also counted runs in which the agent read a tool's help (any command containing `--help`, `help` or `-h`, each checked by hand), and confidently wrong answers separately from "no answer", because a confident wrong answer does more harm.

3.6 Statistics

For success rates we report Wilson 95% ranges [14]. A Wilson range is an interval around a measured success rate that, with this many runs, very likely contains the true rate. It behaves sensibly near 0% and 100%, where simpler formulas give nonsense.

To compare two arms we used a paired sign test [15]. For each question we took the mean over its two repeats for each arm and asked which arm was cheaper. Gaps within 2% counted as ties; this band was fixed before these runs and has been used in all our earlier rounds. The sign test ignores ties and asks how often a split at least as lopsided as the one we saw would happen if the two arms were really equal. That probability is the p value. A small p (below 0.05 by convention) means the split is unlikely to be luck. A large p means the data cannot tell the arms apart; it does not prove they are equal. We tested correct answers the same way, counting only questions where one arm passed more often than the other.

Our keep rule, fixed before the runs, was that `gh-tai` must beat both plain `gh` and `gh-axi` by more than chance, on more than one model.

3.7 Re-running AXI's benchmark

Separately, on 23 November 2026, we re-ran AXI's published GitHub benchmark with its own 17 tasks, condition texts, MCP wiring and language-model judge, unchanged. We used AXI's later grading hints, which the AXI project itself updated so that answers are graded by process rather than by values from March. Agent and judge were Claude Sonnet 4.6, as in the original study, under Claude Code 2.1.330 with the same leak controls as above. Our usage allowance covered one full round of all 17 tasks and a second round of 9: 156 runs, one or two per task and arm, instead of the planned five. A second run the same day, on the same 17 tasks with five repeats, compared plain `gh` and `gh-axi` again. Both runs also included an earlier version of our own tool; it differs from `gh-tai`, so we leave its numbers out.

4. Results

All 2,688 runs on the two sealed sets finished. There were no stalls, crashes or retries. On each set GitHub briefly failed or refused some of the grader's own lookups; Section 6 describes how we handled it. Every run used the pinned agent program and the requested model.

4.1 The second set: correct answers and tokens

The second set is our headline result, because it ran after the fixes on questions nobody building `gh-tai` had seen. Table 1 gives correct answers and Table 2 gives tokens.

Table 1. Correct answers on the second set, out of 84 runs per cell (42 questions, two repeats), with Wilson 95% ranges. In Tables 1, 2, 4 and 5, bold marks the best value in each row: the most correct answers, or the fewest tokens or round trips.

Model	plain gh	gh-axi	GitHub MCP	gh-tai
Sonnet 5.5	77 (84-96%)	74 (79-93%)	67 (70-87%)	**78** (85-97%)
Haiku 5.5	**83** (94-100%)	78 (85-97%)	71 (75-91%)	80 (88-98%)
Opus 5.5	**84** (96-100%)	**84** (96-100%)	79 (87-97%)	**84** (96-100%)
Fable 5.2	**84** (96-100%)	83 (94-100%)	79 (87-97%)	**84** (96-100%)

Question by question, no difference in correct answers between `gh-tai` and any rival reached $p < 0.05$ on any model. The lowest p was 0.11 (Sonnet, `gh-tai` against MCP).

Table 2. Mean total input tokens per run on the second set (cached and uncached counted together), and each tool's cost as a multiple of plain gh.

Model	plain gh	gh-axi	GitHub MCP	gh-tai
Sonnet 5.5	**24.6K**	42.3K (1.72x)	57.1K (2.32x)	25.5K (1.04x)
Haiku 5.5	**31.7K**	68.4K (2.16x)	79.1K (2.50x)	33.1K (1.04x)
Opus 5.5	**26.4K**	50.4K (1.91x)	57.6K (2.18x)	28.8K (1.09x)
Fable 5.2	**26.8K**	39.9K (1.49x)	65.3K (2.44x)	27.0K (1.01x)

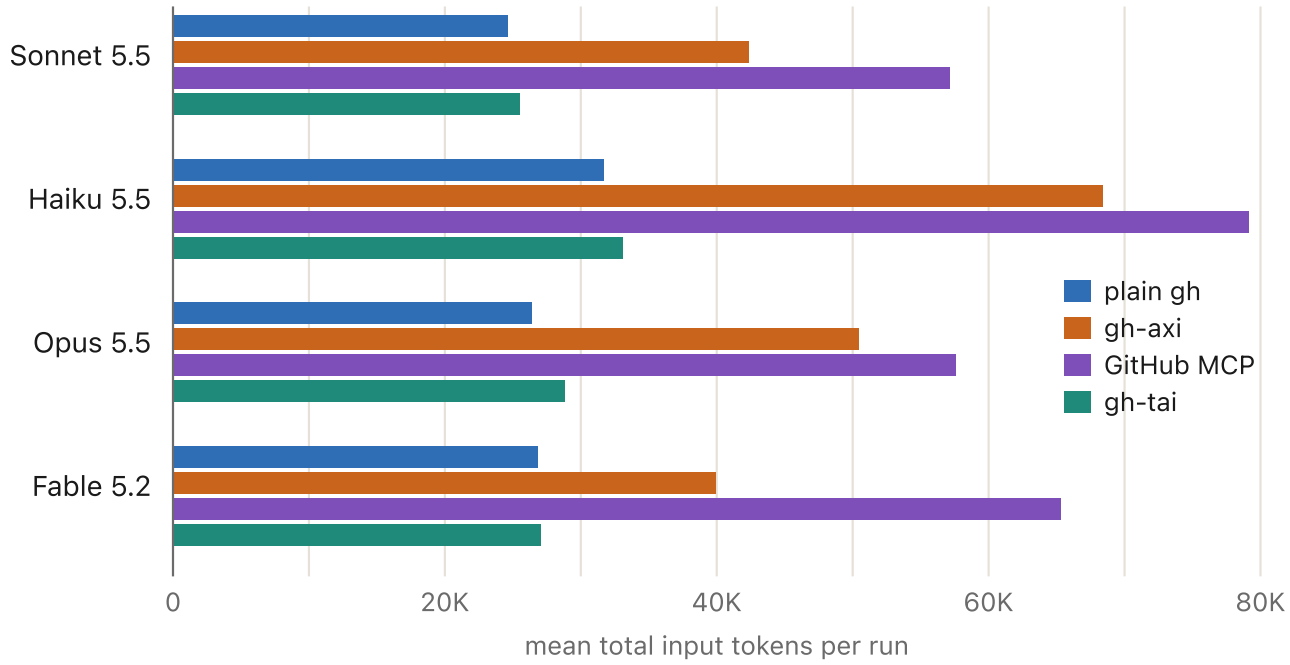

Figure 1. Mean total input tokens per run on the second set, by model and tool (bars in the legend's order within each model). The same data as Table 2. Plain `gh` and `gh-tai` sit together at the short end on every model; `gh-axi` and the MCP server are roughly twice as long.

On every model, plain `gh` and `gh-tai` were the two cheapest arms and were within 1% to 9% of each other. `gh-axi` used 1.49 to 2.16 times plain `gh`'s tokens and the MCP server 2.18 to 2.50 times. Neither answered more questions correctly than plain `gh` on any model (Table 1). Question by question, `gh-tai` was cheaper than `gh-axi` on 25 to 31 of the 42 questions, depending on the model, and cheaper than the MCP server on 39 to 42 (all $p < 0.0001$). We did not run the paired test between plain `gh` and the other two tools on these sets, but the means in Table 2 put plain `gh` where `gh-tai` is.

4.2 gh-tai against plain gh, question by question

Table 3 shows the paired comparison that decides our keep rule, on both sets.

Table 3. gh-tai against plain gh on tokens, question by question. "Cheaper" and "dearer" count questions where gh-tai's mean was more than 2% below or above plain gh's; the rest are ties. Ratio is gh-tai's mean tokens over plain gh's. Bold marks a split unlikely to be chance (p below 0.05) and the side it favours.

Model	Set	cheaper	dearer	tie	p	ratio
Sonnet 5.5	second	6	7	29	1.00	1.04
	first	9	6	27	0.61	0.99
Haiku 5.5	second	8	12	22	0.50	1.05
	first	6	13	23	0.17	1.04
Opus 5.5	second	5	14	23	0.06	1.09
	first	2	**11**	29	**0.02**	1.07
Fable 5.2	second	3	3	36	1.00	1.01
	first	1	4	37	0.38	1.04
All four, pooled	second	22	36	110	0.09	
	first	18	**34**	116	**0.04**	

The pooled rows treat each model's answer to a question as its own pair (168 pairs). Read it as a summary, not as a fifth test.

On Sonnet, Haiku and Fable the two tools cannot be told apart. On Opus, `gh-tai` cost more on both sets: 7% more on the first (p = 0.02) and 9% more on the second (p = 0.06). Taken alone, the second result could be chance. Taken together with the first, we read it as a small real loss on Opus. Pooled over the four models, `gh-tai` leans slightly dearer on both sets. On no model and on neither set is `gh-tai` cheaper than plain `gh` by more than chance. The keep rule is not met.

Most questions were ties for a simple reason. On the 14 lookup and single-field questions, all four models took exactly two round trips with both tools, in 56 of 56 question-model pairs: one command, then the answer. There is no step left to save. On the other four kinds, `gh-tai` saved half a round trip or more on 22 question-model pairs and cost that much on 27. No kind of question showed a reliable saving.

4.3 The first set

The first set ran before the fixes, on different repositories. Table 4 shows that it tells the same story.

Table 4. First set: correct answers out of 84 and mean total input tokens per run.

Model	plain gh	gh-axi	GitHub MCP	gh-tai
Sonnet 5.5	74, 25.9K	**75**, 48.4K	62, 57.9K	**75**, **25.8K**
Haiku 5.5	81, **30.9K**	74, 78.3K	68, 66.7K	**82**, 32.0K
Opus 5.5	**84**, **26.7K**	83, 51.8K	74, 57.0K	81, 28.6K
Fable 5.2	**84**, **25.6K**	**84**, 40.9K	77, 60.5K	**84**, 26.6K

On the first set `gh-axi` used 1.60 to 2.53 times plain `gh`'s tokens and the MCP server 2.13 to 2.36 times. Across both sets, then, `gh-axi` cost 1.5 to 2.5 times plain `gh` and the MCP server 2.1 to 2.5 times. On the first set `gh-tai` had 3 fewer correct answers than plain `gh` on Opus. On two of those questions GitHub itself gives two different numbers depending on where one looks (a milestone's closed-issue counter says 51 while its issue listing returns 50), and `gh-tai` printed exactly what `gh` printed.

The fixes did what they were meant to do. On the first set, `gh-tai`'s "not found" message suggested another command, and Opus and Fable followed the suggestion, spending an extra round trip that plain `gh`'s blunter error avoided. On Fable the "repository does not exist" question cost `gh-tai` 52% more than plain `gh`. On the second set, Fable's question of the same kind cost `gh-tai` 34% less (24.6K against 37.3K tokens). Those questions were a small share of the set, so the totals barely moved.

4.4 Round trips and help

Table 5. Mean round trips per run, and runs (of 84) in which the agent read the tool's help. Second set first, first set in brackets.

Model	plain gh	gh-axi	GitHub MCP	gh-tai
Sonnet 5.5	**2.25** (2.37); help 0 (0)	3.55 (3.99); help 36 (33)	3.96 (3.87)	2.31 (2.33); help 2 (0)
Haiku 5.5	**2.62** (2.56); help 0 (0)	4.83 (5.37); help 32 (44)	4.42 (3.92)	2.69 (2.60); help 1 (1)
Opus 5.5	**2.37** (2.40); help 0 (0)	4.07 (4.29); help 66 (72)	3.94 (3.90)	2.54 (2.56); help 8 (1)
Fable 5.2	**2.15** (2.07); help 0 (0)	3.02 (3.12); help 29 (34)	3.71 (3.76)	2.17 (2.14); help 0 (0)

The MCP arm has no help command; its nearest equivalent, ToolSearch, was called about once per run.

Agents never read plain `gh`'s help, in any of its 672 runs across both sets. They read `gh-axi`'s help in 29 to 72 of 84 runs per model, even though nothing told them to. They rarely read `gh-tai`'s (13 of 672 runs), because it takes the commands they already know. The round-trip counts follow the token counts closely: on the second set the cheap arms averaged 2.1 to 2.7 round trips and the dear arms 3.0 to 4.8. Section 5 returns to this.

`gh-tai` answered most commands itself. It handed between 1 and 10 runs in 84 per model to `gh` on the second set, mostly for commands it does not answer, and those runs cost more on average. Its own extra flags for counting and date ranges appeared in 6 of 481 `gh-tai` commands: agents type `gh`'s language.

4.5 Failures

Most misses were shared by every command-line tool. On the second set, all of plain `gh`'s 7 Sonnet misses and all of `gh-tai`'s 5 Sonnet "no answer" results were questions about something that does not exist, where the agent wrote `unknown` although `not found` was offered. Haiku showed the same habit. It affects every tool about equally.

The MCP server has no tools for workflow runs, and it gave no answer to the GitHub Actions question in any of its 16 runs of that question across both sets. It also missed a release-asset question in every Opus and Fable run of the second set. Agents in the MCP arm also

hit GitHub's rate limit in 8 runs of the second set; no plain `gh` or `gh-tai` run did. Confidently wrong answers were rare for plain `gh` (0 on the second set, 2 on the first) and most common for the MCP server (8 on the second set, 18 on the first).

The second set exposed two small gaps in `gh-tai` as a drop-in. It reads long lists through GitHub search, which stops at 1,000 results, so it refused a request for 5,000 merged pull requests that plain `gh` returned in full (2,418 rows); this cost extra steps in 4 of 336 `gh-tai` runs. It also prints a few lines that `gh` does not, which broke a `2>&1 | jq` pipe in 2 runs. Both touched 6 of 336 runs and would not turn the tie into a win if fixed.

4.6 AXI's benchmark, re-run

Table 6 sets AXI's published numbers beside our re-run on the same tasks.

Table 6. AXI's GitHub benchmark: published (March 2026, five repeats) and our re-run (November 2026, one or two repeats per task). Success, mean total input tokens per run, and mean agent turns as Claude Code counts them. Agent and judge: Claude Sonnet 4.6. Bold marks the best value in each column.

Tool	Published: solved	tokens	Re-run: solved	95% range	tokens	turns
plain gh	73/85	47.1K	22/26	66-94%	**43.4K**	**3.65**
gh-axi	**85/85**	**46.5K**	**25/26**	81-99%	55.2K	4.54
MCP, all tools loaded	74/85	175.8K	22/26	66-94%	86.7K	3.88
MCP, ToolSearch	70/85	153.6K	19/25	57-89%	65.5K	4.58
MCP, code mode	71/85	137.4K	20/26	58-89%	109.8K	4.69

Published values are computed from AXI's published per-run results file; AXI's study page rounds them to 47K and 46K for plain gh and gh-axi [2]. One MCP ToolSearch run is excluded because the judge returned no verdict.

Paired by task, `gh-axi` used more tokens than plain `gh` on 14 of the 17 tasks and fewer on 3 ($p = 0.013$). The study's claim was the opposite [2]. The same-day run with five repeats agreed: on AXI's tasks, plain `gh` averaged 44.7K tokens and `gh-axi` 54.1K, over 85 runs each. `gh-axi` solved 25 of 26 runs against plain `gh`'s 22 ($p = 0.35$). Of AXI's 12 headline claims, 4 reproduced, 4 reproduced in part and 4 did not, and none failed only because GitHub's data had changed.

Two things changed between March and November that explain much of the gap. The published March runs had a browser-automation MCP server with 29 tools, and two other connectors, loaded in every arm: a leak from the machine the study ran on. In the MCP arm that loaded every tool up front, all 29 browser tool descriptions sat in the context of every request, which roughly doubled its token count. Our runs switched these off, which is part of why our MCP numbers are about half of March's. And the agent program moved from Claude Code 2.1.77 to 2.1.330. Today's agent uses plain `gh`'s `--json` and `--limit` flags well, and the March advantage of `gh-axi` over plain `gh` is gone.

Plain `gh` still has one real weakness here. It failed every run of two counting tasks (how many labels a repository has, and how many issues are open), mostly by reporting a page of 100 or 30 rows as the total. `gh-axi` prints a total, and solved those tasks more often. This is the case AXI's principles were written for, and it is real. It did not show up as a cost or correctness gap on our sealed sets, where plain `gh` answered the count questions well.

5. Why plain gh is hard to beat: round trips cost more than characters

Every time an agent takes a step, the model re-reads the whole conversation: the system prompt, the tool descriptions, the task, and every command and output so far. In an earlier run with an earlier version of the same agent program, the first request alone carried a median of about 14,800 tokens of system prompt and tool descriptions before any work began. Each later step re-sends all of that plus everything since. The number of steps therefore sets the cost far more than the length of any one output.

Our data show this directly. Figure 2 plots each arm's mean round trips against its mean tokens, for every model on both sets. For the three command-line tools, a straight line through those 24 averages fits closely:

$$\textit{tokens per run} \approx 15{,}200 \times \textit{round trips} - 8{,}500 \quad (R^2 = 0.97)$$

Each extra round trip went with about 15,200 more input tokens. An earlier fit of ours, over 465 individual runs of a similar benchmark with Claude Sonnet 4.6, found about 17,800 tokens per round trip and about 930 tokens per 1,000 characters of tool output (output is re-read on every later step; Section 6 notes how that run differs). By that fit, avoiding one

round trip is worth as much as cutting about 19,000 characters of output. In that run, the median run of our earlier tool printed only about 1,400 characters of tool output in all. A tool that halves its output saves little; a tool that saves one step saves a lot.

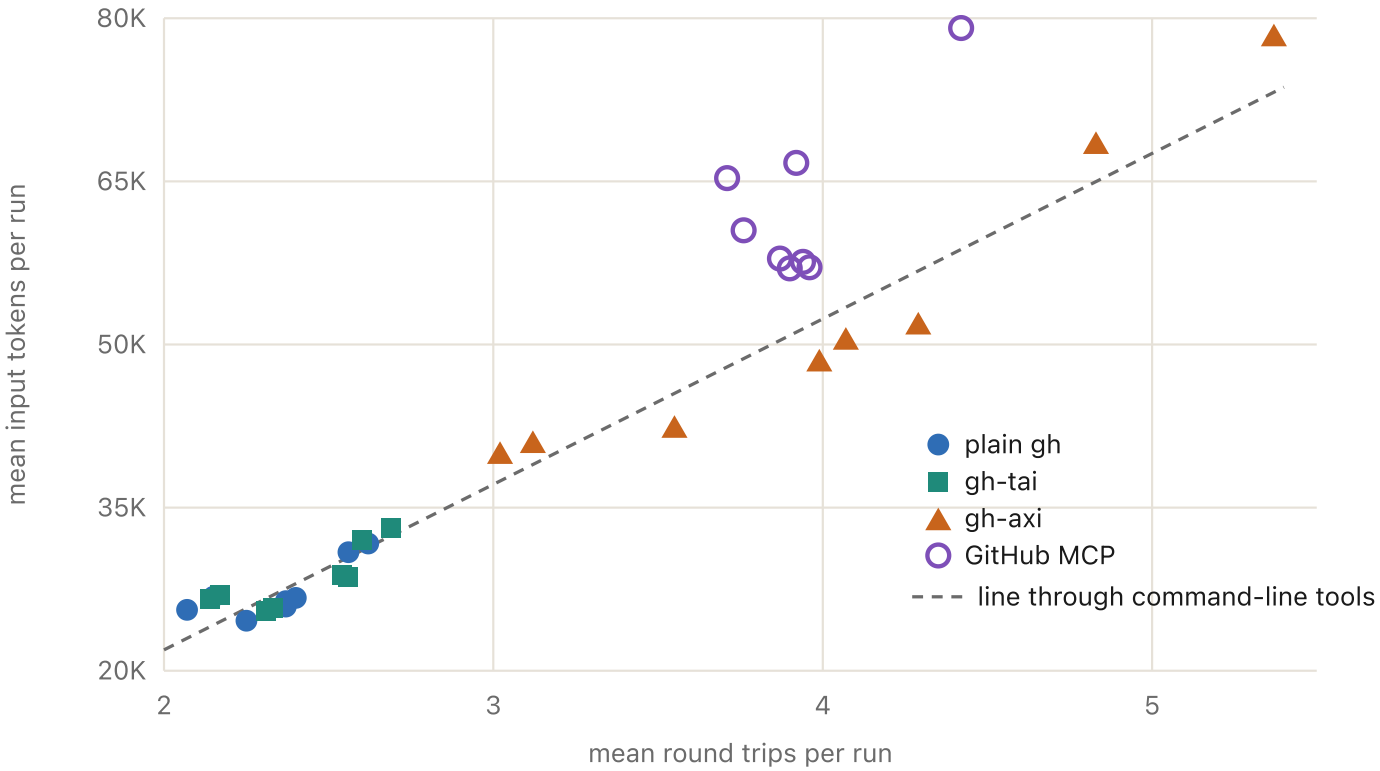

Figure 2. Each point is one tool on one model on one question set (tokens from Tables 2 and 4, round trips from Table 5): its mean round trips per run against its mean total input tokens per run. For the three command-line tools the points fall close to one line (dashed), about 15,200 tokens per round trip. The MCP points (hollow) sit above the line because the agent also loads tool descriptions through ToolSearch. This is a fit across averages, so it shows association, not cause.

Seen this way, the results make sense. On the 14 simplest questions, plain `gh` needs exactly two round trips, the minimum: one command and then the answer. No tool can do better. On harder questions plain `gh` still averages under three round trips, because the agent already knows its commands and flags from training and composes them in one shell line with `--json`, `--jq` or `gh api`. It never stopped to read help. An agent-specific tool starts at a disadvantage on unfamiliar ground: the agent does not know its commands, so it reads help, and each help read is often a round trip of its own. That is much of the extra cost we saw for `gh-axi`, which agents stopped to learn in 29 to 72 of 84 runs.

The same mechanism explains where `gh-tai` lost ground despite speaking `gh`'s language. A helpful-looking suggestion in an error message invited one more step. A refusal where `gh` would have answered (the 1,000-result search cap) forced a detour. Extra lines in the output broke a pipe and needed a retry. Each cost a round trip, and a round trip is worth more than anything a few saved characters can buy back. Kun Chen's study observed the same mechanism for MCP tool search: "Each extra turn re-sends the entire growing context, so the savings from a smaller initial context are consumed by the accumulation across additional turns" [2].

Outside studies agree. An analysis of 600 public coding-agent runs found that the number of turns explained 84% of the variation in the log of run cost, against 4% for mean output size [8]. A controlled study of token-reduction add-ons for Claude Code found that the strongest one cut tool-output tokens by 38.4% and raised the billed cost by 6.8%, because the agent took extra steps to recover what was cut [9]. Production traces from GitHub Copilot show a nearly one-to-one coupling between model calls and tool executions [10], so each tool call is about one round trip in real use as well. A comparison of the `gh` CLI and GitHub's MCP server across seven agent programs found failures equally common on both, but 12.9% of MCP spending bought no finished work against 2.2% for the CLI [11]. And a study of enterprise agent tasks found a plain shell beat purpose-built typed tools while using 19% to 72% fewer tokens [12]. The broader point is older: the SWE-agent work showed that the design of the interface an agent uses changes how well the agent works [13].

For a tool to beat plain `gh`, then, it has to save round trips on questions where plain `gh` needs several. Shorter output is not enough, and a familiar command grammar only brings a tool level with `gh`. On our sealed sets we found no kind of question where any tool reliably saved steps over plain `gh`.

6. Threats to validity

We built `gh-tai`, so we have an interest in the outcome. We reduced the risk with sealed questions written by a session that had not seen the tool, script grading, a keep rule fixed in advance, and by reporting the result against our own tool. The question sets were

nonetheless written by agent sessions in our own setup, and their mix of question kinds is one writer's guess at what developers ask.

Every question was a read on a public repository. We measured no writes, no private repositories and no long investigations of the kind where an agent works through many objects over many steps. The results say nothing about savings on writes.

We used one agent program, Claude Code, and one model family, Claude. Other agent programs and other models may favour different tools. The re-run of AXI's benchmark used an older model, Sonnet 4.6, and AXI's language-model judge rather than script grading, and its budget allowed only one or two runs per task, so its per-task results are noisy.

The MCP arm used GitHub's default tool sets, loaded through ToolSearch, as a developer would get them out of the box. A configuration with more tool sets, such as the one for GitHub Actions, would answer questions this one could not. Loading all tools up front would change its cost. MCP is not always worse; this configuration was worse on these questions. Agents in the MCP arm were also the only ones to hit GitHub's rate limit on the second set (8 runs), and shared that with `gh-axi` on the first (3 runs each), which may have added steps.

`gh-axi` was used as shipped, and the instruction text said nothing about reading help. A prompt that taught the agent `gh-axi`'s commands in advance might lower its cost, but would also add tokens to every request, and no arm received such help.

The instruction text for `gh-tai` is 20 characters longer than for `gh` (the name is 4 characters longer and appears 5 times), re-read on every step, about 5 to 10 tokens per step. When every difference counts, with no 2% band, plain `gh` comes out cheaper on every model (p between less than 0.0001 and 0.008), and this small difference in instruction length explains much of that. The 2% band, fixed before these runs, absorbs differences of that size.

With 42 questions and two repeats, the sign test can miss a small real difference of a few percent. Our tie result means we could not detect a difference, not that the tools are identical.

Live data can change during a run. Reading the truth twice handled this. On the first set GitHub's rate limit emptied both truth readings for one run, which was then graded wrong by mistake; we re-graded it from its saved answer with the same grader, and it was correct. On the second set GitHub returned a server error to one truth reading before a run; that run was graded against the reading taken at grading time and was correct. No other grading faults occurred.

Our token measure counts cached input in full, so it overstates what a cache-discounted bill would show; one study estimates flat input pricing overstates loop cost by about seven times under near-ideal caching [8]. We compare arms with each other, not with a bill, and every arm is measured the same way. Our earlier 465-run fit (Section 5) comes from a previous run of ours on a similar benchmark with an older agent version; we cite it for its per-character estimate, and our own data in Figure 2 give the per-round-trip figure.

7. Conclusion

We asked whether GitHub tools built for AI agents are cheaper for agents than the tool built for people. On two sealed, script-graded sets of 42 read-only questions, across four Claude models, they were not. `gh-axi` cost 1.5 to 2.5 times plain `gh`'s input tokens and GitHub's MCP server 2.1 to 2.5 times, with no gain in correct answers. Our own drop-in replacement matched plain `gh` on correct answers, matched it on tokens on three models, and cost a little more on the fourth. It did not meet the rule we set for keeping it. A re-run of AXI's own benchmark found plain `gh` cheaper than `gh-axi` there too.

The reason is round trips. Each step re-sends the whole conversation, so the number of steps sets the cost, and plain `gh` already takes the fewest steps on most everyday questions because models know it well. A tool that wants to beat it has to remove steps on multi-step questions, and has to do so without adding help reads or detours that each cost a step of their own.

For developers the practical advice is simple. If your agent reads GitHub, plain `gh` is a strong default. If you build a tool for agents, measure it against the plain tool first, on questions you have never seen, graded by a script, and count total input tokens and round trips. The plain tool is the rival that is hard to beat.

We have not yet tested writes, long multi-step investigations, other agent programs or other model families. Those are where a purpose-built tool might still earn its place.

Availability and acknowledgement

`gh-tai` is unreleased. The two question sets remain sealed. For every run we kept the full agent transcript, every tool call and the grade. We thank Kun Chen for publishing the AXI benchmark with its tasks, conditions, grader and every trajectory, which made an exact re-run possible.

References

1. K. Chen. "AXI: Agent eXperience Interface." Project site, 2026. https://axi.md/ (read 26 November 2026).
2. K. Chen. "AXI Benchmark Study: Agent Interface Comparison (GitHub)." Study dated 21 March 2026, in the AXI repository, MIT licence. https://github.com/kunchenguid/axi/blob/main/bench-github/published-results/STUDY.md
3. K. Chen. gh-axi, the reference GitHub tool for AXI. https://github.com/kunchenguid/gh-axi (version 0.1.43 used).
4. "Three grading hints pin point-in-time facts." Issue 117 in the AXI repository, re-run dated 26 July 2026. https://github.com/kunchenguid/axi/issues/117
5. GitHub CLI (gh). https://cli.github.com/ (version 2.86.0 used).
6. GitHub MCP server. https://github.com/github/github-mcp-server (remote endpoint `https://api.githubcopilot.com/mcp/`, default tool sets, 49 tools at the time of our runs).
7. Model Context Protocol specification. https://modelcontextprotocol.io/
8. "TurnCost: Turns, Not Observations: Cache-Aware Loop Engineering for Coding Agents." arXiv:2610.15093, October 2026. https://arxiv.org/abs/2610.15093
9. "Token Reduction Is Not Cost Reduction." arXiv:2607.12161, July 2026. https://arxiv.org/abs/2607.12161
10. "Agentic Coding in the Wild: Characterizing GitHub Copilot Traces at Production Scale." arXiv:2608.00101, July 2026. https://arxiv.org/abs/2608.00101

11. "The Scaffolding Matters More Than the Interface: A Controlled Comparison of MCP and CLI Tool Use Across Seven Agent Scaffoldings, Five Language Models, and One Software Task." arXiv:2608.08654, August 2026. https://arxiv.org/abs/2608.08654
12. "Is Bash All You Need? An Empirical Study of Tool Interfaces for Enterprise Digital Worker Agents." arXiv:2609.11999, September 2026. https://arxiv.org/abs/2609.11999
13. "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering." arXiv:2405.15793, May 2024. https://arxiv.org/abs/2405.15793
14. "Probable inference, the law of succession, and statistical inference." Journal of the American Statistical Association 22 (1927): 209-212. The source of the Wilson score interval.
15. "The statistical sign test." Journal of the American Statistical Association 41 (1946): 557-566.