Claude Code with a Language Server: Higher Cost, No Clear Gain in C/C++ Bug Fixes

Comparing Claude Code with and without the clangd code navigation tool on 255 issues from 40 software projects.

Pratyush Saini, CTO, ByteAsk Research pratyush@byteask.ai
Anirudha Kulkarni, CEO, ByteAsk Research anirudha@byteask.ai

Code and sample tasks: github.com/ByteAsk/lsp-claude-code-cpp

Abstract

AI coding assistants must find the right parts of a program before they can fix a bug. A language server is a tool that helps them find where a piece of code is defined or used. We tested whether adding this tool to Claude Code would improve bug fixes or reduce cost. We compared Claude Code with and without the clangd language server on 255 issues from 40 projects written in C and C++. We prepared the tool and instructed Claude Code to use it. Each version made 359 repair attempts: 270 succeeded with the tool, compared with 275 without it. This difference was too small to establish an improvement. Across all code search and repair tasks, the setup with the tool cost 11% more. Its search results were shorter, but Claude Code read more code and made more requests to the AI model. The tests also had a limitation: for 126 issues, the model could identify all the locations needing changes from the problem description alone, without seeing the project files. For the tasks and instructions we tested, adding the language server increased cost without a clear improvement in bug fixing. Different tools, instructions, or tasks may produce different results.

1 Introduction

Navigation in C and C++ depends on the build. Whether `book.apply(msg)` calls one overload or another, a template specialisation or a function generated by a macro is decided by the include paths, macros and language options of the build's compile commands, which a language server such as clangd reads from a compilation database [20] and exposes through the Language Server Protocol [21]. Coding agents have started to use language servers: Claude Code ships code-intelligence plugins [2], and MCP servers such as Serena wrap them for other agents [24]. Published comparisons report large savings: 2–34× less tool output in scripted workflows [7], and 31% less tool output with 3% lower cost for Opus 4.8 (14% for Sonnet 4.6) on three reference queries [31].

For a team that maintains a large C++ system the question is practical: will the plugin make the agent cheaper or better? We answer it with a paired experiment on 255 C/C++ issues from 40 repositories, from the most used C++ libraries to small projects, in three tiers that differ in what the agent may do: list the functions the fix changes, edit without building, and repair end to end. PLAIN is Claude Code 2.1.312 with Claude Opus 5.5 as shipped; LSP adds a configured clangd workflow.

Left optional, the agent rarely called the server (5% of runs). Required to use it, the agent called it alongside grep, mostly to look names up (89% of calls; our instruction prescribes a first name lookup, so the mix reflects agent and policy together), consumed more file content and made more model calls, and outcomes did not improve. On constructed requests scored against clangd-derived callers by file and function name, the plain agent found 96.6% of the callers by reading the calls; the enforced workflow queried references in 93% of runs, and its task-averaged recall differed by +0.6 points (95% CI 0.0 to 1.7) at ×1.11 (1.07 to 1.15) the cost (Section 5). Four lessons follow. *Price the work after the answer*: re-reads were 94% of the extra tokens and 28% of the estimated extra API cost, and holding the observed runs fixed, zero-pricing every LSP result removes only 18% of the premium (Section 6). *Availability is not use*: optional, the server was rarely called; required, it served as a name index on natural issues, whether or not their names were ambiguous (Sections 3 and 5). *The model's reading is the baseline*: where a references

query returns the scored answer, grep and reading recovered nearly all of it, so a semantic tool competes with the model's own reading, not with grep (Section 5). *Measure the baseline and repeatability*: the issue alone completes 126 of the 255 navigation tasks, and only 30 of 359 repair pairs showed any outcome variation over the available runs (Section 7).

What to do with this

1. **Do not expect an LSP plugin to make the agent cheaper.** Our configured workflow shortened search results but raised API cost by 11% (95% CI 8% to 15%) and resolved no more issues.
2. **Price a tool by everything it adds to a run.** Re-reads were 94% of the plugin's extra tokens and 28% of its estimated extra API cost; the rest paid for new file content, context and output. Holding the runs fixed, zero-pricing every LSP result removes only 18% of the premium.
3. **Check tool discovery and returned results.** With tool search on, the tool was never invoked in 57 development runs; exposed but optional, it was called in 5% of navigation runs; required, 15% of its calls returned an error or no result.
4. **Screen issues for leaked fixes, and repeat runs.** 13 of 73 post-cutoff issues, 9 of 22 QuantLib candidates and 6–8 of 129 Multi-SWE-bench C++ issues contain the fix code; between two runs, 8% of tasks changed outcome.
5. **Pilot on your own code, on tasks that can tell tools apart.** From the issue alone the model finds 63% of fmt's fix sites but 30% in projects with fewer than 1,000 stars, and only 30 of our 359 repair pairs showed any outcome variation. Measure on your own code, with repeats, including tasks where names are ambiguous.
6. **Audit the grader before the agent.** Pin images by digest; run the reference fix and an empty change through the grader.

2 Study design

Arms. PLAIN is Claude Code 2.1.312 with `claude-opus-5-5` at automatic effort, as shipped, with default tool search. LSP is the same agent with a configured clangd workflow: clangd 21.1.8 with the configured build directory, compilation database and background index built into the image; a wrapper that opens one project file at start-up so the index loads before the first query; Anthropic's `clangd-lsp` plugin with tool search disabled and the ten other deferred tools removed, which leaves the first request 586 tokens longer than PLAIN's on average; and a 15-line instruction to find code symbols with the server, read code where it points, and use grep only for text that is not a symbol or when the server cannot answer (Appendix D). The plugin also reports clangd diagnostics, which LSP received in 24% of runs, including edit-only runs, where they are the only compiler feedback either arm gets. The comparison therefore estimates the effect of deploying this workflow; it does not isolate any one of its parts. Both arms share the model, task text, base images, CPU allocation (three cores per run) and graders.

Tiers. *Navigation* is read-only: the agent lists at most ten `path:function` lines. Its targets are the sites changed by the audited reference fix, which other valid fixes need not share. Site recall is matched target sites over target sites, pooled over tasks, with each task's denominator capped at ten; a task is complete when it finds every target site or reaches the cap. Matching uses the file and the function name and ignores namespaces, template arguments and parameters. A strict matcher that requires exact paths and agreeing qualified names changes one site in one of 742 agent answers. In *edit only* a pre-tool-use hook denies build, compile and test commands in both arms. *End to end* allows building and testing. A graded attempt is resolved when every fail-to-pass test passes and every pass-to-pass test still passes. Navigation covers all 255 issues; edit only and end to end cover the 194 and 165 issues whose graders we could validate.

Tasks. Issues come from four sources (Table 1). The *tuning set* was used to build the LSP arm. The *held-out set* was opened, for evaluation only, after the arm was frozen; one environment repair followed (four images whose index came up empty were rebuilt before they ran; no instruction or tool setting changed). Every held-out, Multi-SWE-bench and post-cutoff LSP run started only after an LSP probe on its image returned symbols. *Multi-SWE-bench C++* is the benchmark's full C++ subset [36]. The *post-cutoff* set consists of bugs fixed between 18 August and 22 October 2026, after the model's documented

Table 1: Task sources and the tasks analysed per arm. A repair task is gradable when its reference fix passes the hidden tests and an unchanged tree fails them.

Source	Nav.	Edit	E2E	Grader	Notes
Tuning set	30	30	30	task tests	15 projects; used to build the LSP arm
Held-out set	29	29	–	task tests	opened after the arm was frozen; 1 failed the image leak check, 5 leaked fixes
Multi-SWE-bench C++	129	122	122	official tests	7 not graded: fix code in the test patch (3), reference fix fails (4)
Post-cutoff bugs	67	13	13	task tests	20 projects; 13 + 9 issues with fix code excluded; repair only where tests could be validated

June 2026 training cutoff [3]; a later fix date does not guarantee that the issue or the surrounding code was unseen. After the first round we added graders for C projects and cpp-httplib and post-cutoff QuantLib bugs from the same window, applying the leak screen below before any run and the same grader validation; candidates whose tests check behaviour the issue does not ask for were dropped on review. Our selection rule excludes an issue that contains the fix code, such as a diff, the new condition or the exact replacement line; we applied it when building the tuning set. On 4 November 2026, after the run-1 results existed and a reviewer found a post-cutoff issue we had missed, we screened every issue retrospectively: two passes of a model judge that saw only the issue text and the reference fix. We excluded the post-cutoff and held-out tasks flagged in either pass (13 and 5) from both arms and all repeats, after checking each flag against its fix, and kept the flagged Multi-SWE-bench issues so that the benchmark stays as published. This is a retrospective eligibility correction. Table 5 reports the as-run and corrected cohorts side by side: the pooled repair rates move by less than one point and the cost ratio is unchanged at reported precision, but some small cells move more.

Repository groups. For results we group the 40 repositories by how widely they are used: nlohmann/json, fmt, simdjson and Catch2 (each over 20,000 GitHub stars, most of Multi-SWE-bench C++) on their own; 23 other projects with at least 1,000 stars (such as libuv, googletest, spdlog, QuantLib and CLI11); and 13 with fewer. Stars measure use, not size or training exposure; the groups cut across the sources.

Grading and integrity. Agent images were scanned for reference-fix and hidden-test content, git history past the base commit and extra refs; flagged images were admitted only after review (the 12 admitted as build output show only ignored files in git status). Hidden tests run in a separate container after the agent stops. The navigation gold was derived from the reference fix and audited site by site by a model reviewer that did not see the agents' answers. Multi-SWE-bench attempts were graded with the official tests in the official images, pinned by digest, with our own patch preprocessing and result classification (Section 8). A scan of the 1,996 logs of the main study and its repeats, including the Sonnet runs, found no web-tool call, no edit-only run whose tool output shows a configure, build or test step (the hook refused 19 attempts), and 5 network-like shell commands, all edits of documentation links; the 288 navigation runs added later (Sections 3 and 5) contain no web-tool call, edit or network-like command, and no change to the tree. Token counts come from an API proxy; within the scanned main-study runs, Claude Code's own usage agrees within 0.5% in every run but one, which started a subagent whose calls Claude Code leaves out of its total. We price uncached input, five-minute cache writes, cache reads and output at \$4, \$5, \$0.20 and \$20 per million tokens, and reprice the first request's cache-read tokens at the write price so that a run's cost does not depend on whether an earlier run warmed the cache; priced as recorded, the pooled cost ratio is $\times$1.15 instead of $\times$1.11. Sonnet runs use Sonnet's list prices, half of these. Costs exclude container compute and index construction. Four model-assisted reviews of the graders are in the supplement.

Analysis. Runs are paired by issue and tier. The primary comparisons are the pooled resolve rate and the pooled cost ratio; source, tier and group results are descriptive. Pooled means weight each task–tier pair equally, and all confidence intervals (CIs) are 95%. Within a task set and tier we report paired outcomes, two-sided exact McNemar tests on the tasks where the arms disagree, Newcombe's score interval for the

paired difference in resolve rate [22], and paired bootstrap intervals (10,000 resamples) for recall and for ratios of means. An issue appears in up to three tiers, so pooled results resample whole issues and the pooled test flips the arm labels of all of an issue's attempts together. Sensitivities weight tasks or repositories equally and resample repositories. Numbers in parentheses after an estimate are its 95% confidence interval.

3 Getting the agent to call the language server

With the plugin installed and tool search on, the agent made no LSP call in 57 development runs (19 of an earlier study, 18 of a paired comparison and 20 whose instruction made loading the tool the first step). Claude Code lists deferred tools by name and loads a tool's schema only when the model asks for it [4]; the model never asked. Disabling tool search exposed the tool but also loaded ten other deferred tools (first request 17,274 to 28,281 tokens); removing them left 542.

"Prefer LSP over grep" produced no LSP calls in its pilot, and "your first action is LSP" produced one, followed by grep. To measure optional use properly we ran a *permissive* arm: the final arm's image, tool exposure, index, diagnostics and settings, with the rule replaced by a neutral note that the server exists and may be used, "grep, or both, as you see fit" (Appendix D). On a seeded random sample of 30 navigation tasks outside the held-out set, run twice, it called the server in 5% of 60 runs, never for references, and used grep in 100%; its cost was ×1.05 (1.01 to 1.08) that of the study's Plain runs on the same tasks. The final rule against grepping for code symbols produced sustained use: the agent called the server in 96% of 614 runs (2.2 calls per run) and used grep in 70% (2.0 per run, against 3.3 without the plugin; these counts include the text searches the rule allows). It used the server as a symbol index: 89% of its 1,342 calls looked a name up (`workspaceSymbol`), 81% even after dropping each run's first, prescribed call, and 9% listed references, while definition and implementation lookups, hover and call-hierarchy queries together made up 1%. 85% of calls returned a nonempty result; 6% found no symbol (often a name the code does not contain) and 5% passed a directory where the tool needs a file.

4 Accuracy and cost

Table 2 reports the results by repository group, pooled over sources and tiers; Table 4 in the appendix gives every source and tier. In no group does the plugin find more fix sites or resolve more issues by a margin its interval or test supports. On the four big libraries recall is lower and repair counts are lower or tied; in projects with fewer than 1,000 stars it resolves 46 against 42 of 49 attempts ($p = 0.13$). Pooled, Plain resolves 275 and LSP 270 of 359 attempts on 194 issues ($p = 0.41$; interval of the difference −4.2 to 1.1 points). Resampling repositories instead of issues gives intervals of −3.0 to 3.8 points for the repair difference and 1.07 to 1.14 for the cost ratio; weighting each repository equally gives +5.2 points (−0.9 to 13.5), carried by the small projects, and weighting each task equally changes navigation recall by +0.7 points (−1.5 to 2.8). End to end alone, the tier closest to deployment, LSP resolves 124 against 126 of 165 at ×1.07 the cost (1.01 to 1.13). By source, the largest gaps are Multi-SWE-bench edit only, 85 against 80 of 122 for Plain ($p = 0.12$), and held-out edit only, 26 against 22 of 29 for LSP ($p = 0.12$). Table 5 gives the same comparison as run, with the flagged Multi-SWE-bench issues removed too, and without the tuning set: LSP never resolves more.

We repeated all 122 Multi-SWE-bench edit-only tasks in both arms. The second run resolved 83 (Plain) and 84 (LSP), against 85 and 80 in the first; averaged over the two runs, resolve rates are 68.9% and 67.2%, a difference of −1.6 points (−4.5 to 1.2; $p = 0.38$, $n = 122$), and 8% of tasks changed outcome between runs of the same arm. LSP again used ×1.41 the tokens. With Claude Sonnet 5.5, cost also rises without a detectable repair gain: Plain resolves 62 and LSP 63 of 122 tasks ($p = 1.00$; interval of the difference −4.2 to 5.8 points), LSP calls the server in 99% of runs, and it costs ×1.15 (1.09 to 1.22) as much (tokens ×1.27). A second navigation run on 59 graded post-cutoff tasks gives 49.4% (Plain) and 46.3% (LSP) recall over both runs, a difference of −3.1 points (−6.2 to −0.3); LSP found fewer sites on 9 tasks and more on 2 (sign test $p = 0.07$).

The plugin raises mean cost in every group (Figure 1), by ×1.08 to ×1.14, and in every source and tier. Pooled over all 614 paired attempts (255 issues), tokens rise ×1.19 (1.11 to 1.27) and cost ×1.11 (1.08 to

Table 2: Results by repository group (run 1, all sources and tiers). Navigation: site recall, matched over required sites; Δ: LSP minus PLAIN in points with a paired bootstrap interval. Repairs: edit-only and end-to-end attempts resolved, with an issue-level permutation p. Ratios: LSP over PLAIN, ratio of means over every attempt in the group, issues resampled. Issue only: site recall (%) from the issue text alone, no repository or tools, three tries pooled. Groups: Section 2; source and tier detail: Table 4.

		Navigation: site recall (%)		Repairs resolved			LSP / Plain (95% CI)		Issue
Repository group	Tasks	Plain / LSP	Δ (95% CI)	Attempts	Plain / LSP	p	Tokens	Cost	only
nlohmann/json	65	62.3 / 57.8	−4.4 (−16.3 to 5.9)	114	87 / 83	0.36	1.23 (1.12–1.35)	1.14 (1.07–1.22)	52
fmt	50	72.7 / 71.7	−1.0 (−5.9 to 4.2)	92	67 / 64	0.25	1.22 (1.11–1.35)	1.13 (1.08–1.17)	63
simdjson	27	53.0 / 51.2	−1.8 (−14.0 to 7.9)	34	25 / 23	1.00	1.21 (1.04–1.38)	1.09 (1.01–1.16)	45
Catch2	13	44.6 / 42.9	−1.8 (−7.0 to 0.0)	18	6 / 6	1.00	1.22 (0.97–1.52)	1.10 (1.04–1.17)	46
Other projects, ≥1k stars (23 repos)	63	64.4 / 62.3	−2.1 (−6.1 to 1.7)	52	48 / 48	1.00	1.22 (1.08–1.35)	1.12 (1.06–1.17)	61
Other projects, <1k stars (13 repos)	37	75.7 / 77.4	+1.7 (0.0 to 4.5)	49	42 / 46	0.13	1.10 (0.94–1.37)	1.08 (0.99–1.18)	30
All	255	62.8 / 60.8	−1.9	359	275 / 270	0.41	1.19 (1.11–1.27)	1.11 (1.08–1.15)	50

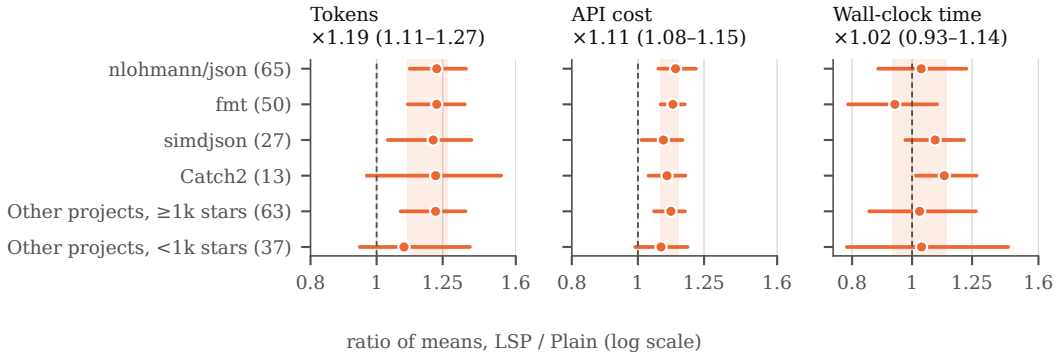


Figure 1: Ratios of mean tokens, API cost and wall-clock time (LSP over PLAIN) per repository group, on logarithmic axes: dots are estimates and lines 95% intervals (issues resampled); titles and shading give the pooled ratio and its interval. Tasks per group in parentheses.

1.15); the LSP run uses more tokens than its PLAIN twin in 74% of pairs. Navigation alone represents each issue once: tokens ×1.29 (1.20 to 1.39), cost ×1.14 (1.11 to 1.18) and wall-clock time ×1.24 (1.18 to 1.31). Over all tiers the time ratio is ×1.02 (0.93 to 1.14). In dollars the gap is small, $0.27 against $0.30 per attempt, and we measured no gain in return. These results do not show that compiler-semantic navigation is unnecessary; they show that, on these issue-resolution tasks, Claude Code did not turn access to it into better outcomes or lower cost.

5 When names are ambiguous

A name search returns occurrences without establishing which function each call invokes. We compare the workflows on existing issues and on constructed requests containing same-name functions and decoys.

Ambiguous names in the existing tasks. For an exploratory analysis of the existing runs we defined a task's *grep fan-out* as the median, over its required gold functions, of the number of source files that contain the function's unqualified name (the script reads no outcomes). In the top tertile of the 201 graded navigation tasks (more than 9.7 files; 67 tasks) PLAIN found 61% of the sites and LSP 59%, a difference of −1.4 points (−10.6 to 4.9), and they resolved 78 and 72 of 109 repairs. We detected no association between fan-out and the per-task recall difference (Spearman $\rho = 0.10$, $p = 0.17$). The enforced workflow averaged 1.8 LSP calls per run in the top stratum and 1.9 in the lowest, with reference queries in 7% and 13% of runs.

A positive control. We then built navigation tasks whose answer is a references query: "add a parameter to this one overload of `f` and update its callers". On the newest snapshot of each repository, with the study's images and indexes, a miner fixed before any run (Appendix C) kept a name only if at least two distinct functions declare it and every textual occurrence of it in library code is a location that clangd's references

Table 3: Overload change requests (29 tasks after the correction, two runs per arm). Gold callers come from clangd's references and are matched by file and unqualified function name. Recall pools scored callers across runs; precision and completion average per-run scores. Refs: share of runs with a `findReferences` or `incomingCalls` call. Differences in the text: arm minus PLAIN, mean over tasks of each task's run average, tasks resampled (the permissive arm's task means equal PLAIN's on every task, so its recall interval has zero width; not an equivalence test).

Arm	Recall	Precision	Complete	Uses LSP	Refs	Grep	Cost/run
PLAIN	97%	89%	91%	–	–	100%	$0.129
LSP (enforced)	97%	89%	93%	100%	93%	45%	$0.144
Permissive	97%	90%	91%	7%	5%	100%	$0.133

assign to one of them. A target has two to eight caller functions and at least two *decoys*: functions that call another function of the same name. The gold is the target's callers, keyed like every navigation site; listing the target itself is neither credited nor penalised. Recall counts callers found and precision, which listed decoys lower, counts answer lines that are callers. This gave 29 tasks in 12 repositories with 73 callers (median 5 decoys and 14 textual occurrences of the name per task). Each of three arms ran every task twice (Table 3).

Table 3 gives the result on the gold as corrected after the runs (end of this paragraph). With grep and reads alone, PLAIN found 96.6% of the callers at 89% precision. Required to use the server, the agent listed references in 93% of runs and changed task-averaged recall by +0.6 points (0.0 to 1.7) and precision by +0.7 points (0.0 to 2.0) at ×1.11 (1.07 to 1.15) the cost; permissive, it listed references in 5% of runs and matched PLAIN at ×1.03 (1.00 to 1.05) the cost. In the transcripts the agent greps the name and tells the functions apart from the call text, such as a receiver named `number_lexer`. The callers every arm missed are calls inside templates that bind to the target only in some instantiations; in both fmt tasks the enforced agent received those locations from `findReferences` and left them out, and for such dependent calls clangd's own answer is heuristic. Two caveats bound the result. Requiring that clangd resolve every textual occurrence keeps names whose uses the index resolves, which excludes many macro and dependent-template cases that are hard for a reader too. And after the runs, having seen that no arm completed some tasks, we removed generated single-header copies (Catch2, simdjson) from the gold and enforced the request's exclusion of test directories (QuantLib's `test-suite`); we also excluded one task whose conversion-operator names the matcher cannot recognise, and eight others then had fewer than two eligible callers, leaving 29 of the 38. Answer lines on generated copies are ignored when scoring precision. On the gold as mined (38 tasks) recall is 86% (PLAIN) and 88% (LSP), a difference of +1.1 points (0.0 to 2.9), at ×1.11 the cost.

6 Where the tokens and the dollars go

For a run whose context is only appended to, let P_k be the input tokens of model call k, cached or not: P_1 is the initial context (system prompt, tools, instructions, task) and $D_j = P_{j+1} - P_j$ is what call j added (the model's message and the tool results). Then $P_k = P_1 + \sum_{j<k} D_j$, and the input of a run with n calls is

$$\sum_{k=1}^{n} P_k \;=\; n\,P_1 \;+\; \sum_{j=1}^{n-1} (n-j)\,D_j, \tag{1}$$

so an item added early is read by every later call. None of our runs was compacted. Prompt caching changes the price of re-reading, not its volume: each D_j is written to the cache once and then re-read at one twenty-fifth of the write price. Generated output is billed separately and is not part of Equation (1).

We split each run's measured increments D_j among the parts added after call j in proportion to their length in characters, and price the parts at list prices assuming one cache write and then cache reads. The token parts reconcile to each run's measured total by construction; the dollar estimate is within 2.3% of the measured mean cost per arm (individual runs differ more where the cache expired during long builds). Figure 2 shows the average run. Re-reads are 86% of PLAIN's tokens (88% of LSP's) but only about 16% of PLAIN's estimated dollars, which go mostly to writing new context (56%, largely the initial context) and to output (28%). Counted once, all tool output is 4% of PLAIN's tokens. The *extra* spending of LSP splits differently: re-reads are 94% of its extra tokens but 28% of its extra dollars, and new context and output are 72%.

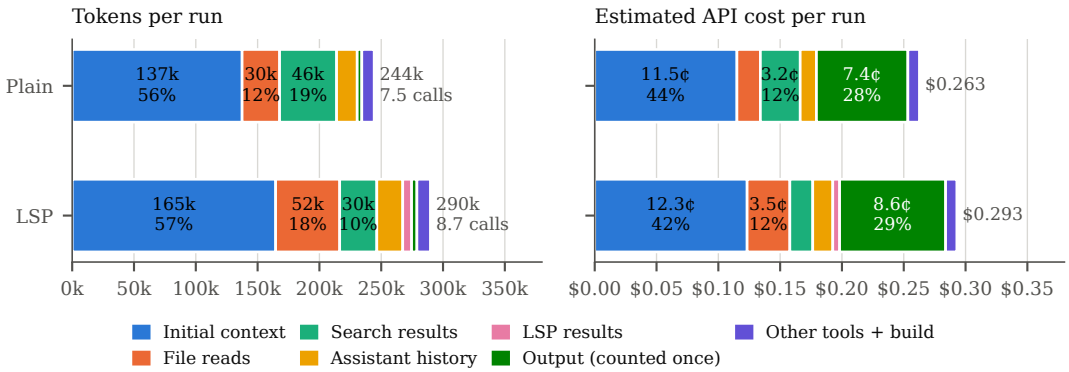


Figure 2: The average run (614 paired runs per arm). Left: tokens, each input part counted every time a call reads it and output counted once; parts are allocated by character length. Right: the same parts priced at list prices (cache write $5/M, cache read $0.20/M, output $20/M), an estimate within 2.3% of measured cost.

The language server did shorten navigation output: search and LSP results, with their re-reads, fell from 46k to 37k tokens per run, and search results cost $0.014 less, while LSP results added $0.006. But most LSP answers are locations, and LSP runs read more file content: 52k against 30k tokens, $0.016 more, 1.9 times what the shorter navigation results saved. Estimated spending on output rose by $0.012 and on the initial context, its write and its re-reads, by $0.008. Two accounting bounds on the observed runs show how little the server's own answers weigh: pricing every LSP result at zero removes 18% of the premium (cost ×1.09), and making every re-read free still leaves ×1.10. The longer prompt itself (586 tokens) explains little: holding each observed LSP run fixed, it accounts for 11% of the extra tokens and 13% of the cost gap. Across paired attempts, the difference in model calls is closely associated with the difference in tokens (Figure 3; Spearman $\rho = 0.96$); least-squares slopes are 63k tokens ($R^2 = 0.73$) and $0.03 ($R^2 = 0.64$) per extra call. These slopes describe the association; they are not the price of inserting one call, and they do not separate the workflow's parts. The number of LSP calls alone correlates more weakly ($\rho = 0.40$).

This gives a break-even rule. Holding the rest of a run fixed, shortening a result by s tokens that m later calls re-read saves s tokens at the write price and ms at the read price, about $0.024 for $s = 4{,}000$ and $m = 5$. A tool pays for itself only if these savings exceed the cost of the calls, output and reads it adds.

7 What the C++ benchmark measures

A navigation tool helps most with what the issue does not already say. To measure how much that is, we gave the same model, through the API, the navigation prompt and the issue but no repository and no tools, three times per task (2,000 output tokens), and graded the answers like the agents' (Figure 4). From the issue alone it found 63% of fmt's required sites (agents with the repository 73% and 72%) and 52% of nlohmann/json's (62% and 58%), but 30% in projects with fewer than 1,000 stars (76% and 77%). On Multi-SWE-bench C++, whose fixes predate the model, it found 55% of the sites, against 61% (Plain) and 58% (LSP), and in each try completed 69–74 of 129 tasks (agents: 81 and 83). Substantial issue-only recall also occurs on the post-cutoff set: it found 46% (agents 51% and 50%) and completed 35–36 of 66 tasks in each try (39 and 40). The probe cannot separate hints in the issue from general knowledge of a project or from inference, and some issues go further and state the fix: the screen flagged 6–8 of the 129 Multi-SWE-bench issues, 13 of our 73 post-cutoff ones and 9 of the 22 QuantLib candidates. On the 46 tasks where all three tries found no site, the agents found 47% (Plain) and 40% (LSP), a difference of −6.7 points (−21.6 to 2.9): the gap between the agents and the issue is widest there, and the server did not widen it. Post-cutoff fixes still allowed substantial issue-only localization: the probe completed 11 of the 13 repair tasks in at least two of three tries.

Most repair task–tier pairs had identical observed outcomes under the two Opus workflows: across the available runs, 30 of the 359 pairs showed any variation within or between arms, and only the 122 Multi-SWE-bench edit-only pairs had a second run of each arm. These counts describe the agreement

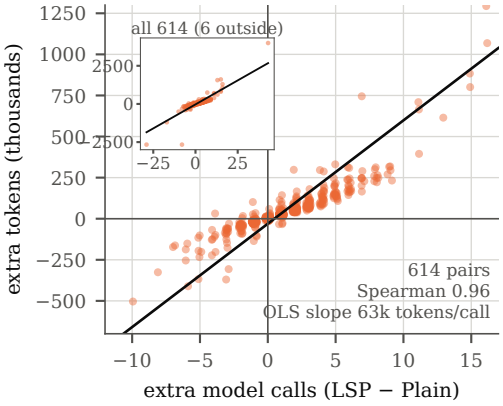


Figure 3: Paired differences (LSP minus PLAIN) for 614 task–tier pairs; the main panel enlarges the centre, the inset shows all pairs; the line is a least-squares fit to all pairs; horizontal jitter.

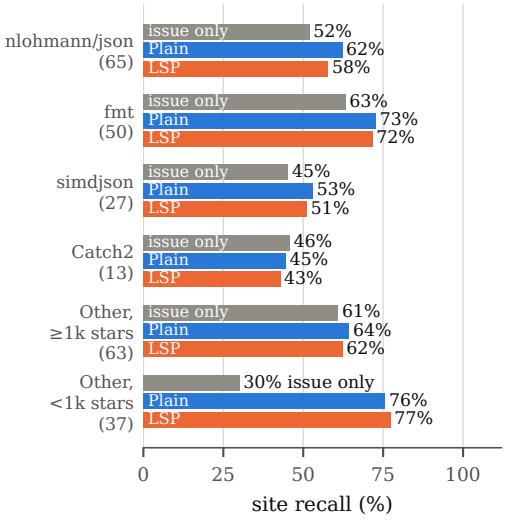


Figure 4: Site recall per repository group from the issue alone (no repository or tools; three tries pooled) and by each agent with the repository (one run), on the same tasks. Tasks per group in parentheses.

of the evaluated workflows and the available replication. Single runs are also noisy. On the seven Multi-SWE-bench edit-only tasks that only one arm solved in run 1 (six by PLAIN, one by LSP), the repeat gave 3 solved by both, 2 by LSP only and 2 by neither; every one of the seven changed outcome in at least one arm. Tasks selected because the arms disagreed drift back toward agreement, which is why the full second run is the better estimate. Assuming a discordant share of $\max(\hat{\psi}, \delta)$, an exact McNemar test at 5% has 80% power from about 6.6 points on 122 tasks and 27 on 30.

8 Grading hazards

Five problems in the C++ grading path would each have changed reported numbers (details in Appendix B): missing images that the harness builds from floating compiler tags instead of pulling the pinned official ones (one Catch2 reference fix failed under GCC 16; we pulled and pinned all 139 tags); log parsers that miss CTest's timed failure lines, so a run in which every test fails reads as "no results"; official test patches that carry production code for three Catch2 tasks (excluded); nlohmann/json tests that compile a generated single header (38 of 49 reference fixes fail unless it is regenerated; we regenerate it for both arms, which changed one verdict); and tests that need live websites (four tasks excluded). Two bugs of our own were fixed before the analysis.

9 Related work

Navigation tools for agents. Interface design changes what agents achieve [34]; AutoCodeRover searches the abstract syntax tree [38], Agentless localizes in stages [32], and RepoGraph and LocAgent use code graphs [8, 25]; added to existing agents, RepoGraph raises resolve rates and token use. MarsCode Agent combines language-server navigation with grep and diagnostics [19], InfCode-C++ reports a resolve gain from AST-structured queries on Multi-SWE-bench C++ inside its own pipeline [9], and RepoNavigator trains an agent to use one definition-jump tool [39]. Closest to us, Xu [33] compares retrieval conditions with model and harness fixed and reports task-dependent adoption and token overhead; AutoCrashFL ablates clangd definition lookup on 454 industrial C/C++ crashes and reports almost unchanged top-1 localization (134 of 454 crashes with definition lookup, 135 without), with a code read after almost every lookup [12]; and CodeCompass finds that 58% of graph-enabled runs never call the graph tool [26]. Profiling Claude Code with Serena on 60 SWE-bench tasks, TokenEconomizer

finds higher cost in four of six repositories [15]; a tree-sitter code graph served over MCP uses ten times fewer tokens at lower answer quality (83% against 92%) [29]; CodexGraph answers repository questions through a graph database [18]. A JVM industrial report claims 15–60% fewer navigation tokens [14]. Xu's adoption finding is the closest; we test it in C/C++ with a production plugin at larger scale, with an optional and an enforced policy, on natural issues and on change requests that need overload resolution, and we price whole runs.

Cost and context. Work on agent evaluation argues for reporting accuracy and cost together [10, 13]. Cache reads dominate tokens and dollars in a Sonnet 4.5 case study [6]; with caching disabled, a fixed prefix is most of the input of short trajectories [28]. Dropping old tool output from the context lowers agent cost substantially [17].

Benchmark validity. For SWE-bench [11] and its successors, including Multi-SWE-bench [36] and SWE-PolyBench [27], audits report weak tests [30, 35], solutions leaked in issues [1], and underspecified tasks or overly strict tests [23]; Liang et al. [16] probe memorization with issue-only file localization, and fresh-task pipelines address contamination [5, 37]. We extend the issue-only probe to C++ and to bugs fixed after the training cutoff.

10 Limitations of Claude Code for C/C++, and of this study

Building the LSP arm exposed four limitations of Claude Code for C/C++. The model never loaded a deferred tool. The LSP tool takes a file and position for every operation, even a workspace-wide name lookup, and 5% of calls failed because the model passed a directory. Most answers are locations, which the agent then reads. And the plugin relies on a compilation database: we supplied one and an index, so coverage followed that build configuration; without one, clangd falls back to a compile command that may miss include paths [20].

This study has limits of its own. It covers one agent and two models, mostly with one run per task. The LSP arm changes tool loading, availability, diagnostics, the image and the instruction together, and its prescribed first name lookup may shape the query mix. The permissive arm ran navigation only, a hybrid policy (the server only for ambiguous names) is untested, and the overload tasks are constructed change requests whose gold comes from the compiler's references, not natural issues; the fan-out strata are exploratory and measure lexical spread as a proxy for ambiguity. Repositories are public (private code is untested), many issues point at their fix, and the ten-line cap does not test exhaustive reference search. Only 13 edit-only and 13 end-to-end post-cutoff tasks have validated graders. The edit-only hook matches build commands by name; we found no circumvention, but it is not hardened against an adversarial agent.

11 Conclusion

Given a configured clangd plugin, the agent rarely called the server when its use was optional; required, it used the server mostly to look names up, consumed more file content, made more calls and resolved no more issues, and on the constructed overload requests it cost more without a detectable recall improvement. Beyond this plugin, a tool's savings must be priced together with the work the agent does to use its answer: the extra re-reads were most of the extra tokens but a minority of the estimated extra cost. And issue-only localization was often high, while most observed repair outcomes agreed between the workflows. Agent tools should be judged by whole-run cost and validated outcomes, on tasks that need them, with repeated runs and an audited grader.

References

[1] Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-Bench+: Enhanced coding benchmark for LLMs. arXiv preprint arXiv:2410.06992v2, 2024. URL https://arxiv.org/abs/2410.06992v2. Version 2.

[2] Anthropic. Code intelligence plugins. Claude Code documentation, 2026. URL https://code.claude.com/docs/en/plugins/code-intelligence. Living documentation; snapshot accessed 2026-11-04.

[3] Anthropic. Claude Opus 5.5: Model overview. Claude Platform documentation, 2026. URL https://platform.claude.com/docs/en/models/opus-5-5/overview. Training-data cutoff stated as June 2026; accessed 2026-11-04.

[4] Anthropic. Connect Claude Code to tools via MCP. Claude Code documentation, 2026. URL https://code.claude.com/docs/en/mcp#scale-with-mcp-tool-search. Section: Scale with MCP tool search; snapshot accessed 2026-11-04.

[5] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems (Datasets and Benchmarks Track)*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/21bec6ace947b1b58967b945c8ac0f10-Abstract-Datasets_and_Benchmarks_Track.html.

[6] Longju Bai, Zhemin Huang, Xingyao Wang, Jiao Sun, Rada Mihalcea, Erik Brynjolfsson, Alex Pentland, and Jiaxin Pei. How do AI agents spend your money? Analyzing and predicting token consumption in agentic coding tasks. arXiv preprint arXiv:2604.22750, 2026. URL https://arxiv.org/abs/2604.22750.

[7] Blackwell Systems. We measured it: LSP saves AI agents 5–34x tokens vs grep. Blackwell Systems engineering blog, 2026. URL https://blog.blackwell-systems.com/posts/agent-lsp-token-savings/.

[8] Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8697–8727, 2025. doi: 10.18653/v1/2025.acl-long.426.

[9] Qingao Dong, Mengfei Wang, Hengzhi Zhang, Zhichao Li, Yuan Yuan, Mu Li, Xiang Gao, Hailong Sun, Chunming Hu, and Weifeng Lv. InfCode-C++: Intent-guided semantic retrieval and AST-structured search for C++ issue resolution. arXiv preprint arXiv:2511.16005, 2025. URL https://arxiv.org/abs/2511.16005.

[10] Zhiyu Fan, Kirill Vasilevski, Dayi Lin, Boyuan Chen, Yihao Chen, Zhiqing Zhong, Jie M. Zhang, Pinjia He, and Ahmed E. Hassan. SWE-Effi: Re-evaluating software AI agent system effectiveness under resource constraints. arXiv preprint arXiv:2509.09853, 2025. URL https://arxiv.org/abs/2509.09853.

[11] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

[12] Sungmin Kang, Sumi Yun, Jingun Hong, Shin Yoo, and Gabin An. Finding the needle in the crash stack: Industrial-scale crash root cause localization with AutoCrashFL. In *International Conference on Software Engineering: Software Engineering in Practice*, 2026. URL https://arxiv.org/abs/2510.22530.

[13] Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter. *Transactions on Machine Learning Research*, 2025. URL https://openreview.net/forum?id=Zy4uFzMviZ.

[14] Anastasia Kazakova. How Netflix made AI coding agents more efficient with JetBrains IDE intelligence. JetBrains Blog, October 2026. URL https://blog.jetbrains.com/ai/2026/10/how-netflix-made-ai-coding-agents-more-efficient/.

[15] Aaron J. Li, Weijia Xu, Chandan Singh, Zixiao Chen, Jeevana Priya Inala, Charlotte Siska, Acar Tamersoy, Bin Yu, Ion Stoica, Michel Galley, and Jianfeng Gao. TokenEconomizer: Experience-driven black-box cost control for coding agents. arXiv preprint arXiv:2610.22732, 2026. URL https://arxiv.org/abs/2610.22732.

[16] Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. The SWE-Bench illusion: When state-of-the-art LLMs remember instead of reason. arXiv preprint arXiv:2506.12286v4, 2025. URL https://arxiv.org/abs/2506.12286v4. Version 4.

[17] Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The complexity trap: Simple observation masking is as efficient as LLM summarization for agent context management. In *Fourth Deep Learning for Code Workshop at NeurIPS*, 2025. URL https://arxiv.org/abs/2508.21433.

[18] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. CodexGraph: Bridging large language models and code repositories via code graph databases. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics (NAACL), Volume 1: Long Papers*, pages 142–160, 2025. URL https://aclanthology.org/2025.naacl-long.7/.

[19] Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. MarsCode Agent: AI-native automated bug fixing. arXiv preprint arXiv:2409.00899, 2024. URL https://arxiv.org/abs/2409.00899.

[20] LLVM clangd contributors. Compile commands. clangd design documentation, 2026. URL https://clangd.llvm.org/design/compile-commands. Living documentation; snapshot accessed 2026-11-04.

[21] Microsoft. Language Server Protocol specification—3.17. Official protocol specification, 2022. URL https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/. Accessed 2026-11-04.

[22] Robert G. Newcombe. Improved confidence intervals for the difference between binomial proportions based on paired data. *Statistics in Medicine*, 17(22):2635–2650, 1998. doi: 10.1002/(SICI)1097-0258(19981130)17:22<2635::AID-SIM954>3.0.CO;2-C.

[23] OpenAI. Separating signal from noise in coding evaluations. Official research report, 2026. URL https://openai.com/index/separating-signal-from-noise-coding-evaluations/. Published 2026-07-08.

[24] Oraios and contributors. Serena: The IDE for your coding agent. Official software documentation, 2026. URL https://github.com/oraios/serena. Living documentation; snapshot accessed 2026-11-04.

[25] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. RepoGraph: Enhancing AI software engineering with repository-level code graph. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.14684.

[26] Tarakanath Paipuru. CodeCompass: Navigating the navigation paradox in agentic code intelligence. arXiv preprint arXiv:2602.20048, 2026. URL https://arxiv.org/abs/2602.20048.

[27] Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. SWE-PolyBench: A multi-language benchmark for repository level evaluation of coding agents. arXiv preprint arXiv:2504.08703, 2025. URL https://arxiv.org/abs/2504.08703.

[28] Tommy Tran. Where the tokens go: A measured anatomy of LLM agent trajectory costs. In *Proceedings of the Second Workshop for REsearch on Agent Language Models (REALM 2026)*, pages 365–373. Association for Computational Linguistics, 2026. doi: 10.18653/2026.realm-1.24.

[29] Martin Vogel, Falk Meyer-Eschenbach, Severin Kohler, Elias Grünewald, and Felix Balzer. Codebase-Memory: Tree-sitter-based knowledge graphs for LLM code exploration via MCP. arXiv preprint arXiv:2603.27277, 2026. URL https://arxiv.org/abs/2603.27277.

[30] You Wang, Michael Pradel, and Zhongxin Liu. Are "solved issues" in SWE-bench really solved correctly? An empirical study. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*, 2026. doi: 10.1145/3744916.3764576. URL https://arxiv.org/abs/2503.15223v2. Revised text: arXiv:2503.15223v2.

[31] Roger Winter. Why you should use Language Server Protocol (LSP) with Claude Code. CircleCI blog, 2026. URL https://circleci.com/blog/claude-code-lsp/.

[32] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE), 2025. doi: 10.1145/3715754. Article FSE037.

[33] Pengcheng Xu. Does a language server save tokens for coding agents? A measurement methodology and preliminary study. arXiv preprint arXiv:2608.13568, 2026. URL https://arxiv.org/abs/2608.13568.

[34] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://arxiv.org/abs/2405.15793.

[35] Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous evaluation of coding agents on SWE-Bench. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3762–3774, 2025. doi: 10.18653/v1/2025.acl-long.189.

[36] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Ming Ding, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-2111.

[37] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. SWE-bench goes live! In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-4923.

[38] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024. doi: 10.1145/3650212.3680384.

[39] Zhaoxi Zhang, Yitong Duan, Yanzhi Zhang, Yiming Xu, Zhixiang Wang, Kun Liang, Weikang Li, Jiahui Liang, Deguo Xia, Jizhou Huang, Jiyan He, Shuxin Zheng, and Yunfang Wu. One tool is enough: Reinforcement learning of LLM agents for repository-level code navigation. In *Proceedings of the International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, 2026. URL https://proceedings.mlr.press/v306/zhang26an.html.

A Results by source and tier

Table 4: Results by source and tier (run 1). Navigation: site recall (tasks completed); repairs: tasks resolved. Disc. and p (exact McNemar) refer to completion for navigation. Δ: LSP minus PLAIN, points (navigation: paired bootstrap; repair: Newcombe method 10; pooled: issues resampled). Post-cutoff navigation accuracy uses 66 of 67 tasks.

			Accuracy		LSP vs Plain (paired)			LSP / Plain, ratio of means (95% CI)	
Source	Tier	n	Plain	LSP	Disc.	p	Δ, points (95% CI)	Tokens	Cost
Tuning set	Navigation	30	95.0% (28)	95.0% (28)	0/0	1.00	0.0 (identical)	1.54 (1.20–1.79)	1.23 (1.10–1.37)
	Edit only	30	30	29	1/0	1.00	−3.3 (−16.7 to 8.3)	1.24 (1.08–1.44)	1.12 (1.04–1.21)
	End to end	30	30	30	0/0	1.00	0.0 (−11.4 to 11.4)	1.08 (0.98–1.26)	1.04 (0.94–1.13)
Held-out set	Navigation	29	54.7% (13)	57.3% (14)	0/1	1.00	+2.7 (0.0 to 6.6)	1.28 (1.07–1.50)	1.14 (1.06–1.23)
	Edit only	29	22	26	0/4	0.12	+13.8 (−1.0 to 29.7)	1.03 (0.85–1.51)	1.04 (0.91–1.24)
Multi-SWE-bench C++	Navigation	129	61.3% (81)	57.9% (83)	3/5	0.73	−3.4 (−10.6 to 2.7)	1.22 (1.13–1.32)	1.13 (1.09–1.17)
	Edit only	122	85	80	6/1	0.12	−4.1 (−8.7 to 0.4)	1.44 (1.17–1.81)	1.19 (1.07–1.33)
	End to end	122	86	83	5/2	0.45	−2.5 (−7.1 to 2.1)	1.09 (0.93–1.26)	1.06 (0.99–1.14)
Post-cutoff bugs	Navigation	67	51.2% (39)	50.0% (40)	2/3	1.00	−1.2 (−5.8 to 3.5)	1.22 (1.13–1.32)	1.12 (1.09–1.16)
	Edit only	13	12	11	1/0	1.00	−7.7 (−33.5 to 17.2)	1.14 (0.92–1.42)	1.10 (0.99–1.20)
	End to end	13	10	11	0/1	1.00	+7.7 (−15.2 to 31.0)	1.23 (1.01–1.51)	1.18 (1.01–1.37)
Pooled, edit only + end to end		359	275	270	13/8	0.41	−1.4 (−4.2 to 1.1)		
Pooled, every tier		614						1.19 (1.11–1.27)	1.11 (1.08–1.15)

Table 5: Pooled results for each cohort. The primary cohort excludes post-cutoff and held-out issues that contain their fix, found by a retrospective screen (Section 2).

Cohort	Issues / attempts	Resolved, Plain / LSP	p	Tokens (95% CI)	Cost (95% CI)
As run	273 / 645	288 / 282 of 372	0.31	1.19 (1.11–1.27)	1.11 (1.08–1.15)
Primary: own sets screened	255 / 614	275 / 270 of 359	0.41	1.19 (1.11–1.27)	1.11 (1.08–1.15)
Multi-SWE-bench screened too	247 / 590	263 / 258 of 343	0.41	1.18 (1.10–1.27)	1.11 (1.08–1.15)
Primary without tuning set	225 / 524	215 / 211 of 299	0.53	1.19 (1.09–1.28)	1.11 (1.07–1.16)

B Grading hazards in detail

Each of the following would have changed reported numbers. *Images built locally*: at harness commit `24f493f`, a missing image is built from floating `gcc` tags rather than pulled (the README suggests pre-pulling). We first found 59 locally built images; under GCC 16 the reference fix of Catch2-2723 failed where the official GCC 14.2 image passes, and of our first 17 Catch2 verdicts one was a compiler artifact and four were patch collisions. We pulled the 139 official tags, pinned their digests and checked them before and after every grading call. *Failed-test parsing*: for fmt, nlohmann/json and simdjson the harness's

log parsers miss CTest's timed failure lines, so a run in which every test fails is reported as "no results". Our first pipeline treated that as an infrastructure error and retried it until the task dropped out of both arms; final grading records such runs as failures. *Production code in test patches*: the official split assigns files whose paths contain "test" to the test patch; for three Catch2 tasks that includes production code, so applying it can collide with a correct agent fix. We excluded the three. *Generated headers*: most nlohmann/json tests compile the generated single header; removing the header changes from the reference fixes makes 38 of 49 fail, and regenerating the header from the edited sources makes all 49 pass. We regenerate it inside the task image for both arms; this changed one verdict (edit only, 85 against 79 before and 85 against 80 after). *External dependencies*: four tasks fail with their reference fix because their tests contact live websites or their build fetches the current upstream; we excluded them. Two bugs of our own (reviewed fix sites overwritten by automatic ones; transcript output counts taken when streaming starts) were fixed before the analysis.

C Overload change requests

Mining (`analysis/bench/sem_mine.py`, inside each repository's LSP image, no network) and selection (`sem_build.py`) were fixed before any agent run. Library code is the tracked C/C++ files outside test, documentation, example, benchmark, fuzzing, tool, build and vendored directories. Function and method symbols come from clangd's document symbols of the first 400 library files in path order; candidate names must appear in two to twelve enumerated symbols before reference-based merging, and names shorter than three characters, operators, constructors and destructors are skipped. Symbols are one function when one's position is among the other's references. A name is kept when it has at least two functions and every whole-word occurrence of it in library code, comments and string literals removed, is a reference location of one of them, so that the compiler accounts for every text hit. Each non-declaration reference is assigned to the innermost enclosing function-symbol range spanning more than one line, which approximates a function body; a library reference outside such a range disqualifies the target. A target has two to eight callers, at least two decoys, no caller keyed like its own declaring sites, and no declaration or caller in vendored code; per repository at most four targets, one per name, ordered by decoys per caller. The request names the target's signature and declaring line and asks for a trailing `const char* origin` parameter passed on by every caller of this function only; the instruction header, answer format, ten-line cap, hooks and grader are those of every navigation task. The permissive arm uses the same image as LSP with the note below. After the runs we corrected the gold (`sem_gold_v2.py`, arm-blind): callers in generated single-header copies and test directories are dropped and answer lines on a generated copy are ignored; the one task whose callers are conversion operators is dropped; tasks left with fewer than two callers or two decoys are dropped (9 of 38). Both gradings are in the supplement.

D The LSP arm

User-level instruction given to the LSP arm (verbatim):

> **C/C++ code navigation: LSP only, never grep for symbols**
>
> In this C/C++ repository, find code ONLY with the LSP tool. grep, rg, ag, git grep and find are NOT allowed for finding a function, class, type, macro or variable, anywhere in the repository, including inside a single file.
>
> - Your first action for every task is LSP workspaceSymbol for each function, class, type or macro named in my request (and the identifiers in any error message or stack trace), all in one message.
> - Definitions: workspaceSymbol with qualified names (e.g. ns::Class::method), goToDefinition, goToImplementation.
> - Usages and callers: findReferences, incomingCalls, outgoingCalls. Never grep for a symbol name.
> - Types and signatures: hover. Before changing a function signature: findReferences on it.
> - Put independent LSP calls and Reads in ONE message.
> - Read code only where LSP pointed you: Read with offset/limit (or sed -n) on those lines. Never run documentSymbol on a large header.
> - Make every code change with the Edit tool, then fix the compile errors clangd reports.
>
> grep is allowed only for text that is not a code symbol (string literals, comments, CMake or config files, test names, log messages), for finding files by name (find/ls), and when LSP could not answer. Before every grep, ask: is this a symbol? If yes, use LSP.

Configuration: `ENABLE_LSP_TOOL=1` and tool search disabled, with the ten other deferred tools disallowed; clangd 21.1.8 with `-limit-results=20 -j=1` at low CPU and I/O priority; `.ixx .ipp .tpp .txx .inl .hh .tcc` mapped to C++; unused- and missing-include diagnostics off. No LSP-specific hook; both arms use the tier hooks of Section 2. The permissive arm (Section 3) has the same image and settings; its user-level note reads, verbatim: *"This repository has a clangd language server with a pre-built index, available as the LSP tool (workspaceSymbol, goToDefinition, findReferences, incomingCalls, outgoingCalls, hover, documentSymbol). After an Edit, clangd reports compile errors in that file. Use it, grep, or both, as you see fit."* The supplement holds the code, per-run data, graders and the leak-screen records; held-out issue numbers are replaced by random tokens (repositories seen only in the held-out set get pseudonyms too), and the supplement regenerates every table and figure. The repository `https://github.com/ByteAsk/lsp-claude-code-cpp` runs both arms on 20 sample tasks (Harbor task format, your own API key) and scores them as in this paper; the full task set is available on request at `contact@byteask.ai`.