SWELSP: SYMBOL-INDEXED NAVIGATION FOR EFFICIENT CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents spend substantial token budgets exploring repositories through text searches and code reads. These operations return line-based fragments, often prompting additional queries to identify complete definitions and follow relationships between symbols. Existing approaches reduce this overhead through context pruning or structured navigation, but may introduce additional dependencies, such as auxiliary models or dedicated analysis pipelines. Moreover, providing navigation tools alone does not ensure their effective use within existing agent workflows. We present SWELSP, a symbol-indexed navigation interface that connects Language Server Protocol to agents' established workflows. It augments bash outputs with symbol identifiers that agents can directly use to retrieve definitions and references. Our evaluation shows that SWELSP can reduce API cost by 10.1–21.3% on SWE-bench Multilingual, and 19.8–29.6% on SWE-QA while maintaining task performance on both benchmarks by reducing intermediate searches and windowed code reads of text-based exploration. It also achieves the lowest cost per resolved task against popular open-source tools such as Serena and AiderMap in openhands.

1 INTRODUCTION

Coding agents are increasingly used in software development, making their execution efficiency an important concern (Robbes et al., 2026; Murphy-Hill et al., 2026). These agents inspect repositories, modify code, and execute tests to iteratively resolve engineering tasks (Yang et al., 2024). Such interactions incur substantial token consumption, with input tokens accounting for much of the cost (Bai et al., 2026). When the interaction history retains retrieved code, it is included again in subsequent model inputs, extending its token cost beyond the initial retrieval. Reducing this overhead while preserving task success is therefore a practical objective for coding agents (Kapoor et al., 2024).

Problem. Traditional agent's repository exploration introduces unnecessary context because line-based access does not directly expose program structure. Figure 1 illustrates this problem as the agent tries to fix an issue involving accepting multiple IP addresses in the `default_bind` option. The agent first searches for the option, then inspects its handler and follows the address-parsing logic across files. At each step, the agent locates candidate lines but leaves the agent to determine which surrounding code to read. The resulting windows can include unrelated code snippets or truncate the target definition, requiring further searches and reads. As a result, the agent repeatedly processes code retrieved merely to establish the boundaries and context of its actual targets.

Existing approaches tried to reduce this overhead by making code access more selective. Model-based pruning filters retrieved content using relevance estimates (Wang et al., 2026; Shi et al., 2025), while structured access exposes definitions and relationships through repository graphs (Chen et al., 2025; Ouyang et al., 2025), language-server-backed symbol operations (Oraios, 2026), or compact repository maps (Aider-AI, 2026). However, selective retrieval can require an auxiliary model or language-specific analysis. Even when precise navigation tools are available, agents may continue using familiar shell commands, adding tool calls without replacing existing searches and reads.

Challenges. Reducing code exploration cost in practice requires addressing two challenges. *First, obtaining focused context introduces additional model or analysis dependencies.* SWE-Pruner requires an auxiliary trained model, while the LocAgent and RepoGraph pipelines primarily target Python and require adaptation for other languages. The challenge is to decrease token cost without training an auxiliary model or developing a separate analysis pipeline for each language. *Second, more*

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

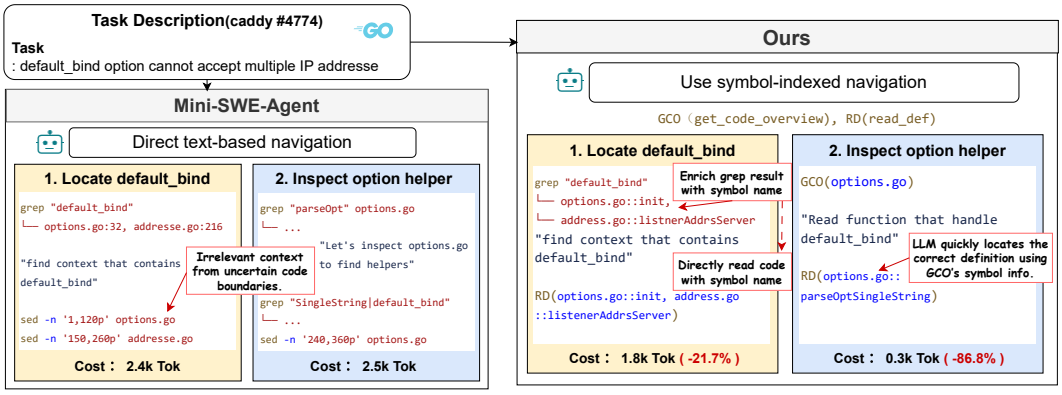


Figure 1: SWELSP reduces exploration tokens by replacing repeated text searches with direct symbol-indexed retrieval.

precise retrieval does not necessarily yield more efficient agent behavior. Agents autonomously select tools, so a new navigation interface may remain unused or introduce redundant queries alongside familiar shell commands (Yang et al., 2024; Xu, 2026). The challenge is to make symbol-based retrieval directly usable within existing exploration steps so that retrieval precision translates into fewer unnecessary interactions.

Approach. We introduce SWELSP, a symbol-indexed navigation interface for AI agents that reuses existing language analysis and connects precise code retrieval to agents' established exploration workflows. To address the first challenge, SWELSP obtains symbol outlines, definitions, and references through the Language Server Protocol (LSP), avoiding a separate analysis pipeline for each language (Microsoft, 2022). To address the second, SWELSP integrates two mechanisms: *Symbol-based output enrichment* and *symbol-indexed navigation*. Symbol-based output enrichment associates code locations in grep results and traceback outputs with symbol identifiers, providing retrieval guidance. Symbol-indexed navigation lets agents use these identifiers to retrieve definitions and references, while file outlines expose additional symbols for selective inspection. Together, these mechanisms connect discovery steps to targeted code retrieval, reducing intermediate searches and windowed reads. As shown in Figure 1, this workflow can reduce exploration tokens.

We evaluate SWELSP on SWE-bench Multilingual (Jimenez et al., 2024) and SWE-QA (Peng et al., 2026) with three backbone models, DeepSeek-V4-Flash, DeepSeek-V4-Pro, and GPT-5.5. We compare against SWE-Pruner on both benchmarks and additionally against LocAgent on SWE-QA. We also compare against Serena and AiderMap using OPENHANDS on SWE-bench Multilingual. In the MINI-SWE-AGENT experiments, SWELSP reduces API cost by 10.1–21.3% while preserving or improving Pass@1 across all three models. On SWE-QA, it reduces API cost by 19.8–29.6% across the models, with answer-quality scores above the vanilla agent. Further analysis shows that Search/read output falls by 17–41% across these two settings, and SWELSP also achieves the lowest cost per resolved instance compared to renowned open-source symbolic tools in OPENHANDS such as Serena and AiderMap across all three models. Our contributions are as follows:

- We present SWELSP, which integrates symbol-indexed navigation into the agent's established workflow. Our source code is available at https://anonymous.4open.science/r/swelsp.
- We demonstrate consistent cost savings across two agent frameworks, three backbone models, and both issue-resolution and code-QA tasks. Further comparison shows that SWELSP still achieves a lower cost per resolved task than prominent symbol-based open-source tools with OPENHANDS.

2 BACKGROUND

Repository exploration. Coding agents solve software engineering tasks through iterative tool use and environment feedback (Yao et al., 2023; Yang et al., 2024; Wang et al., 2025). Repository exploration supplies task-relevant context that may be distributed across files (Zhang et al., 2023). To accommodate diverse tasks, agent frameworks such as OpenHands provide general-purpose

navigation interfaces, including line-based read and shell access (Wang et al., 2025). For repository exploration, these interfaces enable agents to use utilities such as `find` to locate files and `grep` to search their contents. Operating on file paths and text patterns without requiring language-specific parsing, these utilities can be applied across diverse programming languages and to non-code text such as configuration files and documentation. Commercial coding agents, including Claude Code and Cursor, also provide shell execution alongside dedicated file-search and reading tools (Anthropic, 2026; Anysphere, 2026).

Symbol-based retrieval. Symbol-based retrieval organizes code access around named program entities, such as functions and classes (Zhang et al., 2024). Existing approaches use program structure and dependency relations to guide code retrieval and navigation (Cheng et al., 2024; Ouyang et al., 2025; Chen et al., 2025). The Language Server Protocol (LSP) provides standardized interfaces for querying document symbols, definitions, and references (Microsoft, 2022). These interfaces expose symbol locations and source ranges that tools can use to retrieve code according to definition boundaries. We next analyze how window-based and symbol-based retrieval differ in cumulative input token cost when reading the complete source of the same target function.

3 Token Cost of Context Retrieval

Setup. We compare text-based and symbol-based retrieval for retrieving the complete source of a target function whose starting position is already known. Let $L > 0$ denote the target's length in code tokens and $p > 0$ the initial context length. We model text-based retrieval as consecutive, non-overlapping reads of a fixed window of $w > 0$ code tokens, beginning at the target's start. Each read returns a full window, and retrieval stops once the target is fully covered. Each retrieval round consists of one model invocation followed by one tool response. To isolate the effect of windowing, both methods share the same p and add $a > 0$ retained non-code tokens per round, including tool-call messages and response metadata. We assume that all messages are retained without truncation or compression. We measure cumulative input tokens through the model invocation that consumes the final retrieval result. Thus, k retrieval rounds require $k + 1$ model invocations.

Text-based retrieval. Retrieving the complete function requires $k = \lceil L/w \rceil$ rounds. Each round adds $a + w$ tokens to the history. Thus, the input length after t rounds is

$$P_0 = p, \qquad P_t = P_{t-1} + a + w = p + t(a + w). \tag{1}$$

Summing over the $k + 1$ model invocations gives

$$I_{\text{text}} = \sum_{t=0}^{k} P_t = \sum_{t=0}^{k} \big[p + t(a + w)\big] = (k + 1)p + \frac{k(k+1)}{2}(a + w). \tag{2}$$

The first term accounts for repeated input of the initial context. The second accounts for the growing history: messages from earlier retrieval rounds are included in subsequent model invocations.

Symbol-indexed retrieval. Symbol-based retrieval uses program structure to identify the target's source range. We consider an operation that, given correct boundary information, extracts exactly the complete target source in one round. This assumes that the target fits within the operation's output limit. Its two model invocations receive p and $p + a + L$ input tokens, respectively, yielding

$$I_{\text{sym}} = p + (p + a + L). \tag{3}$$

Cost comparison. Subtracting Eq. equation 3 from Eq. equation 2, the excess cumulative input cost of text-based retrieval is

$$\Delta I = I_{\text{text}} - I_{\text{sym}} = \underbrace{kw - L}_{\text{excess code}} + \underbrace{(k-1)(p+a)}_{\text{extra rounds}} + \underbrace{\frac{k(k-1)}{2}(a+w)}_{\text{repeated history}}. \tag{4}$$

The first term, $kw - L$, measures code returned outside the target. Without knowing the target's endpoint, the text-based agent reads a fixed window; when $L < w$, this introduces $w - L$ unnecessary code tokens into the model's context. When $L > w$, retrieving the complete function requires $k > 1$ rounds. The second term accounts for the additional repetitions of the initial context and the non-code

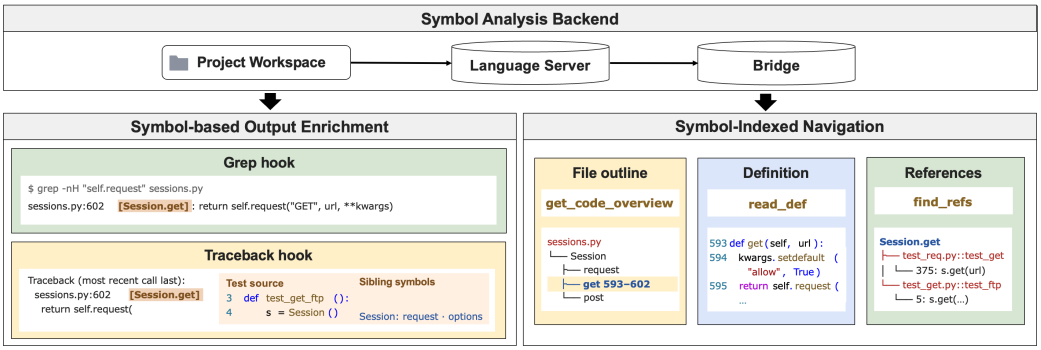


Figure 2: Overview of SWELSP.

overhead introduced by these extra rounds. The third term captures repeated processing of previously returned code and non-code messages as the history grows. Thus, under the stated assumptions, text-based retrieval incurs strictly greater input cost whenever $L \neq w$, with equality only when the target exactly fills one window ($L = w$). By returning the complete target in one operation, symbol-based retrieval avoids excess code and the repeated processing caused by additional rounds.

4 METHOD

The main goal of SWELSP is to integrate symbol-indexed code navigation into coding agents' existing workflows. As shown in Figure 2, symbol-based output enrichment exposes symbols associated with search results and execution failures, providing inputs for subsequent navigation. Symbol-indexed navigation uses these symbols to retrieve definitions and references, while file outlines provide an additional entry point for discovering candidates. A language-server bridge resolves the symbols and source locations used by both components.

4.1 SYMBOL ANALYSIS BACKEND

The symbol-analysis backend provides the symbol information used by output enrichment and navigation. It obtains symbol hierarchies and source locations from language servers, then uses a bridge to organize them into symbol names, source ranges, and enclosing relationships. These representations allow the output hooks and navigation tools to refer to the same program entities. During workspace initialization, SWELSP starts the language server for the workspace project and initializes its repository analysis. The same language-server session is reused throughout agent execution. When source files change, the bridge synchronizes the affected files with the language server and refreshes their symbol information before subsequent navigation queries. This keeps symbol ranges aligned with the updated source without restarting the language server.

Language server responses. SWELSP obtains symbol hierarchies, definition targets, and reference locations from three categories of LSP queries. Given a file f, `textDocument/documentSymbol` can return a hierarchy of `DocumentSymbol` records or a flat list of `SymbolInformation` records. We describe symbol resolution using the hierarchical form below. We represent each symbol as $s = (f, n, \kappa, \mathrm{loc}_{\mathrm{start}}, \mathrm{loc}_{\mathrm{end}}, \mathcal{S})$ where n is the scoped name path, κ its kind, $\mathcal{S}$ its child symbols, and $(\mathrm{loc}_{\mathrm{start}}, \mathrm{loc}_{\mathrm{end}})$ its complete source extent. The name path includes enclosing symbols, such as `ClassName/method1`, so that methods with the same name in different classes can be distinguished. Together, the file path f and name path n form the symbol address used by the navigation tools. A `DocumentSymbol` record also provides a selection range, typically identifying the symbol's name. We retain this range to obtain an identifier position for subsequent LSP queries.

The other two queries take a source position as input. Given a file–position pair $(f_{\mathrm{in}}, \mathrm{loc}_{\mathrm{in}})$, `textDocument/definition` returns the corresponding definition locations. These locations identify navigation targets but need not delimit their complete source bodies, requiring post-processing. Given the same form of input, `textDocument/references` returns the locations that the lan-

guage server resolves as references to the same program entity. The bridge uses these locations to recover the surrounding symbol context. When a query finds no target, it returns no locations.

Bridge. The bridge interprets these raw responses through two operations: **(1)** hierarchy traversal and **(2)** location-to-symbol mapping. Together, these operations connect the location-centered responses produced for IDE navigation to the symbol-level context consumed by the agent.

Hierarchy traversal. Given the root symbols returned for a file f, the bridge recursively traverses each child $s' \in \mathcal{S}(s)$ and retains the resulting parent–child relationships. Traversal continues until $\mathcal{S}(s) = \varnothing$. The resulting tree preserves symbol names, kinds, source extents, and nesting relationships, and can therefore be presented as a file outline without including implementation bodies.

Location-to-symbol mapping. Given a source location ℓ in file f, the bridge obtains the array of symbols under f through `textDocument/documentSymbol`. It selects a top-level symbol s whose source extent $[\mathrm{loc}_{\mathrm{start}}(s), \mathrm{loc}_{\mathrm{end}}(s))$ contains ℓ. The bridge then recursively descends into a child $s' \in \mathcal{S}(s)$ whose source ranges contain ℓ It returns the enclosing symbol reached by this.

4.2 Symbol-based Output Enrichment

Symbol-based output enrichment turns locations reported in shell output into entry points for symbol-indexed navigation. Implemented as post-execution hooks in the agent harness, it augments text-search results and tracebacks with symbol information before they are returned to the agent. This exposes symbol-based navigation targets where the agent would otherwise perform additional searches.

Hook design. After each Bash action, the harness evaluates the activation conditions of the available enrichment hooks. The search-result hook is activated when the command invokes a supported text-search utility, such as `grep` or `rg`. The traceback hook is activated when the command output matches a supported traceback format. Both hooks proceed only when the output is nonempty and contains at least one parseable file-and-line location (f, loc) pointing to a supported source file.

Search result hook. The search-result hook turns text matches into entry points for symbol-indexed navigation. Before a search result is returned to the agent, the hook extracts the file and line number from each matched line, resolves its smallest enclosing symbol s, and appends the name $n(s)$ to the original location. For example, it transforms `src/foo.py:42` into `src/foo.py:42::method1`. The enriched result preserves the original result while exposing the symbol address $(f, n(s))$, which can be passed directly to the symbol-indexed navigation tools.

Traceback hook. The traceback hook shortens the transition from an execution failure to its relevant source context. It parses file-and-line locations from the traceback and applies the same location-to-symbol mapping used by the search-result hook. Each resolved frame is annotated with the name of its smallest enclosing symbol. In addition to annotation, the hook retrieves the complete definitions of symbols containing a reported failure location, including the failing test and the method containing the reported failure location. The resulting output combines frame locations with symbol addresses and available source context, allowing the agent to inspect the failure and continue navigation from those addresses. These additions let the agent inspect the immediate failure context and reuse the exposed symbol names for subsequent navigation without translating traceback lines into separate search and source-reading actions.

4.3 Symbol-indexed Navigation

Symbol-indexed navigation exposes the backend's interpreted structures through three agent-invoked tools. These tools map source files and symbol names to file outlines, complete definitions, and reference locations. Because the tools and output hooks use the same file paths f and symbol names n, a symbol exposed by one component can be used directly as input to another.

File outline. `get_code_overview` maps a source file f to its symbol tree. Using the hierarchy traversal, it returns symbol names, source extents, and nesting relationships without returning implementation bodies. The outline lets the agent discover candidate symbols and select one by name before retrieving its definition or references, avoiding an initial read of the entire file.

Definition. `read_def` retrieves source definitions for a symbol name path n. Given n and an optional file constraint f, the bridge resolves matching symbols and their source ranges. For each matching

symbol s, the runtime reads the source range $[\mathrm{loc}_{\mathrm{start}}(s), \mathrm{loc}_{\mathrm{end}}(s))$ from its local source file and returns the definition with its location. The name path may come from a file outline, an enriched shell result, or another navigation result. Combining symbol lookup and source reading in one tool call avoids repeated adjustment of line-based reading windows. For a single resolved target whose body fits the output budget, this operation corresponds to the one-call retrieval of L code tokens assumed in Eq. equation 3.

Reference. `find_refs` maps a symbol name n to its semantic reference locations. Given n and an optional file constraint f, the tool resolves the target symbol and uses a position on its identifier token to query `textDocument/references`. Unlike text search, these results represent occurrences resolved by the language server as references to the same program entity. The bridge applies this location-to-symbol mapping to each reference location and groups the results by file and enclosing symbol. Each result retains the original usage location together with the name of the code entity containing that usage. These names provide targets for subsequent definition retrieval, allowing the agent to inspect callers without independently searching for the code surrounding every reference.

The navigation tools do not impose a fixed exploration sequence. The agent determines which symbol to investigate and whether to inspect its definition, references, or surrounding file structure. Appendix C presents representative agent trajectories showing how file outlines, enriched shell outputs, definition retrieval, and reference lookup compose during repository exploration.

5 EXPERIMENTS

Our experiments show that SWELSP reduces task cost and the context returned during code exploration. We examine these benefits through task performance, estimated API cost, and search/read tool output. We further compare SWELSP with widely used open-source symbol-based tools to assess whether its efficiency gains extend beyond research baselines. We study three questions: **(RQ1)** Does SWELSP reduce task cost while preserving task performance? **(RQ2)** Does SWELSP reduce the context returned by search/read tools during code exploration? **(RQ3)** Does SWELSP offer efficiency gains over widely used open-source symbol-based tools such as Serena and AiderMap? Additional analyses of output enrichment and runtime overhead are provided in Appendices B and D.

Benchmarks. We evaluate SWELSP on two repository-level tasks: SWE-bench Multilingual (Jimenez et al., 2024) for issue fixing and SWE-QA (Peng et al., 2026) for code question answering. For SWE-bench Multilingual, we use all 300 instances. Following the benchmark selection of SWE-Pruner (Wang et al., 2026), we use 144 questions from three Python repositories: Streamlink, Reflex, and Conan. For evaluation of SWE-QA, we strictly follow the benchmark's official instructions.

Agents and research baselines. We compare SWELSP with the vanilla agents and SWE-Pruner, which conduct code pruning with a lightweight model, using Mini-SWE-Agent (Yang et al., 2024) on SWE-bench Multilingual and OpenHands (Wang et al., 2025) on SWE-QA. On SWE-QA, we also compare with LocAgent (Chen et al., 2025), which uses code graph for localization.

Comparison with open-source tools. To assess whether SWELSP offers efficiency benefits beyond existing structured code access, we compare it with Serena (Oraios, 2026) and Aider's repository-map component (AiderMap) (Aider-AI, 2026). This comparison uses OpenHands on the same 300 SWE-bench Multilingual instances, with vanilla OpenHands included as a baseline. We follow the official installation instructions for Serena and AiderMap and retain their default tool configurations, without task- or model-specific tuning.

Models and prompting. We evaluate three backbone models, DeepSeek-V4-Flash (version 0731), DeepSeek-V4-Pro (version 0813), and GPT-5.5, referred to as V4-FLASH, V4-PRO, and GPT-5.5 in tables and figures. For issue-resolution tasks, we add an instruction prohibiting agents from retrieving existing upstream solutions through GitHub or external APIs, addressing one route to benchmark leakage (Skalse et al., 2022; Adams et al., 2026). Appendix F presents the instance template and this restriction. We retain the default settings for temperature, reasoning effort, and all other parameters.

Evaluation metrics. For SWE-bench, we report Pass@1, the percentage of instances resolved in one attempt. For SWE-QA, we retain the benchmark's default LLM judge, DeepSeek-V3 (0324), which evaluates answers against references on correctness, completeness, clarity, relevance, and reasoning (1–10 each). We report the mean across criteria and judged answers. All methods use the

same evaluation configuration. Efficiency metrics include interaction rounds, total and cache-miss tokens, estimated API cost ($), and cost per resolved instance ($). Table annotations report relative percentage changes for efficiency metrics, percentage-point differences for Pass@1, and absolute differences for judge scores, using vanilla as the reference unless otherwise specified. Arrows indicate change direction; green and red denote improvement and degradation. Arrows(↓↑) indicate change direction compared to vanilla agent.

5.1 Comparison of Cost Saving (RQ1)

Across two tasks, two agent harnesses, and three backbone models, SWELSP consistently lowers API cost by 10.1–29.6% without degrading task performance.

Table 1: Efficiency comparison on SWE-bench Multilingual.

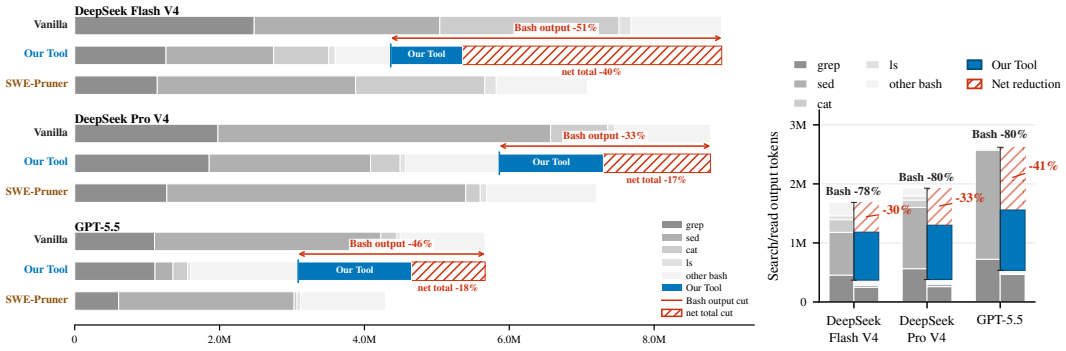
Agent	Rounds	Pass@1	Tokens	Cache-miss	Cost ($)	Cost/Resolved ($)
MINI-SWE-AGENT(V4-FLASH)	78.7	70.7	10.609 M	0.158 M	0.093	0.131
+ SWE-PRUNER	75.4	**74.0** ↑3.3	9.088 M ↓14.3%	0.153 M ↓2.8%	0.080 ↓13.3%	0.109 ↓17.2%
+ SWELSP	**59.3**	71.0 ↑0.3	**8.329 M** ↓21.5%	**0.128 M** ↓18.7%	**0.073** ↓21.3%	**0.103** ↓21.6%
MINI-SWE-AGENT(V4-PRO)	45.9	73.3	3.272 M	0.101 M	0.152	0.207
+ SWE-PRUNER	**43.0**	70.0 ↓3.3	**2.531 M** ↓22.7%	0.087 M ↓14.2%	0.125 ↓17.4%	0.179 ↓13.5%
+ SWELSP	45.1	**75.0** ↑1.7	2.676 M ↓18.2%	**0.083 M** ↓17.9%	**0.123** ↓18.7%	**0.164** ↓20.6%
MINI-SWE-AGENT(GPT-5.5)	**18.9**	71.7	0.417 M	0.029 M	0.525	0.733
+ SWE-PRUNER	20.9	71.7 - 0.0	0.394 M ↓5.4%	**0.026 M** ↓11.0%	0.521 ↓0.9%	0.727 ↓0.9%
+ SWELSP	19.8	**72.7** ↑1.0	**0.393 M** ↓5.7%	0.027 M ↓8.3%	**0.472** ↓10.1%	**0.650** ↓11.4%

As shown in Table 1, SWELSP reduces API cost by 10.1–21.3% while preserving Pass@1 on all three models, and achieves the lowest cost per resolved instance in every setting. By resolving search hits directly to targeted definitions and references, SWELSP cuts the code agents retrieve and carry forward, reducing tokens by 5.7–21.5% (and rounds from 78.7 to 59.3 with V4-FLASH). Its savings are also more stable than SWE-Pruner's, which drop to 0.9% on GPT-5.5. Although SWE-Pruner attains higher Pass@1 with V4-FLASH, SWELSP remains cheaper per resolved instance.

Table 2: Efficiency comparison on SWE-QA.

Agent	Rounds	Judge	Tokens	Cache-miss	Cost ($)
OPENHANDS (V4-FLASH)	25.1	8.93	0.349 M	0.017 M	0.0035
+ SWE-PRUNER	23.5	8.72 ↓0.2	0.406 M ↑16.4%	0.022 M ↑28.5%	0.0041 ↑18.2%
+ LocAgent	22.7	8.99 ↑0.1	0.813 M ↑133.2%	0.044 M ↑157.4%	0.0081 ↑132.3%
+ SWELSP	**17.5**	**9.17** ↑0.2	**0.235 M** ↓32.5%	**0.014 M** ↓16.0%	**0.0024** ↓29.6%
OPENHANDS (V4-PRO)	15.9	8.76	0.251 M	0.018 M	0.017
+ SWE-PRUNER	18.4	9.27 ↑0.5	0.334 M ↑32.9%	0.024 M ↑29.1%	0.022 ↑29.2%
+ LocAgent	14.8	**9.38** ↑0.6	0.420 M ↑67.1%	0.033 M ↑82.4%	0.029 ↑68.7%
+ SWELSP	**12.4**	9.37 ↑0.6	**0.167 M** ↓33.4%	**0.015 M** ↓17.5%	**0.014** ↓20.2%
OPENHANDS (GPT-5.5)	**8.3**	9.04	0.145 M	0.024 M	0.220
+ SWE-PRUNER	11.9	**9.30** ↑0.3	0.192 M ↑31.8%	0.023 M ↓4.4%	0.251 ↑13.7%
+ LocAgent	10.4	9.26 ↑0.2	0.229 M ↑57.5%	0.034 M ↑43.6%	0.299 ↑35.6%
+ SWELSP	10.2	9.23 ↑0.2	**0.144 M** ↓0.8%	**0.017 M** ↓28.9%	**0.177** ↓19.8%

On SWE-QA (Table 2), SWELSP is the only configuration that lowers cost, reducing it by 19.8–29.6% while improving judge scores over vanilla. In contrast, SWE-Pruner and LocAgent increase cost by up to 29.2% and 132.3%, respectively. The case of GPT-5.5 illustrates that token savings need not require fewer interaction rounds. SWELSP takes an average of 1.9 rounds more, yet reduces total tokens by 0.8% and cache-miss tokens by 28.9%, resulting in 19.8% API cost savings. This pattern is consistent with more selective code retrieval reducing the context carried through the interaction history, offsetting the overhead of additional rounds. The savings exceed those in issue resolution for every model, likely because QA centers on retrieving code rather than editing and testing, making search and read a larger share of the workflow. For complete five crietrion scores, refer Table 5.



(a) SWE-Multilingual (b) SWE-QA

Figure 3: Output token breakdown of search/read tools. Gray segments represent outputs from shell-based search/read commands, while blue segments represent outputs from SWELSP. It includes output added by the enrichment hooks, including source code injected by the traceback hook.

5.2 COMPARISON OF CODE EXPLORATION COST (RQ2)

Beyond end-to-end cost, we examine where the savings originate. SWELSP reduces the context returned during code exploration, and this reduction grows with repository size.

Search/read tool output. Figure 3 breaks down search/read tool output in agent trajectories. SWELSP takes over much of shell-based retrieval, cutting shell output by 33–51% on SWE-bench and 78–80% on SWE-QA; after counting SWELSP's own output, the net reduction remains 17–40% and 30–41%, respectively. This matches Eq. equation 4: returning only the target symbol avoids code outside it, and because less code enters the history, every later turn also reprocesses fewer tokens. Figure 13 includes LocAgent in the SWE-QA comparison.

Savings grow with repository size. Figure 4 shows that search/read savings tend to increase with repository size. In larger repositories, a text-based code search matches more near-miss candidates, so conventional agents need more retrieval rounds, whose cost grows as described at equation 2. Meanwhile, symbol-based navigation resolves a definition or its references in a single call regardless of repository size, so its advantage widens as the search space grows. This trend is descriptive, since repository size also varies with language and task difficulty.

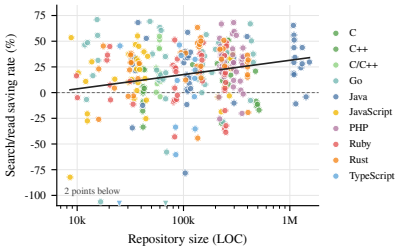


Figure 4: Savings vs. repository size.

(a) Task performance and cost

Agent	Pass@1	Tokens	Cost	Cost/Resolved
OpenHands (V4-FLASH)	45.3	3.212 M	0.032	0.071
+ SWELSP	46.7 ↑1.3	**2.874 M** ↓10.5%	**0.029** ↓10.0%	**0.062** ↓12.6%
+ Serena	**48.7** ↑3.3	3.699 M ↑15.1%	0.037 ↑13.9%	0.075 ↑6.1%
+ AiderMap	**48.7** ↑3.3	4.890 M ↑52.2%	0.047 ↑44.5%	0.096 ↑34.6%
OpenHands (V4-PRO)	63.3	3.977 M	0.190	0.300
+ SWELSP	**68.0** ↑4.7	**2.875 M** ↓27.7%	**0.152** ↓20.0%	**0.224** ↓25.5%
+ Serena	66.3 ↑3.0	4.233 M ↑6.4%	0.199 ↑4.7%	0.300 ↓0.1%
+ AiderMap	60.3 ↓3.0	4.950 M ↑24.4%	0.204 ↑7.1%	0.338 ↑12.4%
OpenHands (GPT-5.5)	**74.7**	0.715 M	0.668	0.895
+ SWELSP	73.3 ↓1.3	**0.563 M** ↓21.3%	**0.548** ↓18.0%	**0.748** ↓16.5%
+ Serena	74.3 ↓0.3	1.051 M ↑47.1%	0.896 ↑34.1%	1.206 ↑34.7%
+ AiderMap	73.3 ↓1.3	0.979 M ↑37.1%	0.851 ↑27.3%	1.160 ↑29.6%

(b) Search/read output

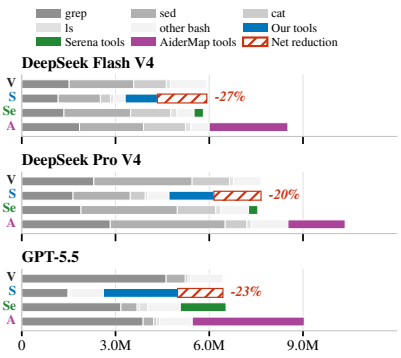


Figure 5: Comparison with widely used tools, showing (a) overall result and (b) search/read output. V, S, Se, and A denote vanilla, SWELSP, Serena, and AiderMap.

5.3 Comparison with Widely Used Symbol-based Tools (RQ3)

We next ask whether SWELSP retains its advantage over widely used open-source symbol-based tools for agents. Serena (Oraios, 2026) exposes LSP-based code operations, and Aider (Aider-AI, 2026) builds repository maps from code structure, with about 29.6k and 49.1k GitHub stars, respectively. We integrate Serena and Aider's repository-map component (AiderMap) into OpenHands and evaluate them on SWE-bench Multilingual. We use OpenHands instead of MINI-SWE-AGENT because integrating these tools into MINI-SWE-AGENT would require modifying its source code.

As shown in Figure 5(a), SWELSP achieves the lowest token usage, API cost, and cost per resolved instance for every model, reducing cost per resolved instance by 12.6–25.5% relative to vanilla while keeping Pass@1 changes range from –1.3 to +4.7 percentage points relative to vanilla. Serena and AiderMap instead increase tokens and API cost in all three settings, although they attain higher Pass@1 with V4-FLASH. Figure 5(b) explains this gap. Serena and AiderMap add symbol-level context on top of shell search and reads rather than replacing them, so their total search/read output matches or exceeds vanilla. SWELSP instead replaces much of the shell output while keeping its own output small, cutting combined search/read output by 20–27%. Its search-result annotations expose symbol addresses that let agents fetch targeted definitions and references directly, skipping later text searches and windowed reads. The advantage thus comes not from providing program structure alone, but from letting symbol retrieval substitute for text exploration instead of adding to it.

6 Related Work

Prior work lowers the cost of repository exploration by trimming retrieved text, locating code through program structure, or exposing structure through additional agent tools.

Reducing exploration context. SWE agents gather context through iterative search and reading (Yang et al., 2024; Wang et al., 2025; Xia et al., 2025), and retrieved text is re-sent with every later model call (Bai et al., 2026; Salim et al., 2026). Prior work reduces this cost after retrieval by pruning or compressing tool output (Wang et al., 2026; Shi et al., 2025) or by masking, summarizing, or learning to manage the interaction history (Lindenbauer et al., 2025; Liu et al., 2026; Gao et al., 2026). These methods select from text already fetched, so the extra rounds needed to reach it remain. SWELSP instead changes what is fetched, returning definitions and references at symbol boundaries.

Structured retrieval and localization. Another line of work uses program structure to find code precisely, including cross-file retrieval for completion (Zhang et al., 2023; Ding et al., 2024; Wu et al., 2024; Liu et al., 2024) and localization over code graphs, or dense indexes for issue resolution (Zhang et al., 2024; Ouyang et al., 2025; Chen et al., 2025; Liu et al., 2025; Jiang et al., 2025; Yu et al., 2025). These approaches build and maintain their own representation behind a separate query interface, often for a single language. Language servers have also been used as a static-analysis backend, but mainly to guide specific purposes of task completion such as unit test generation (Agrawal et al., 2023; Go et al., 2026; 2025). Serena and Aider's repository map expose program structure through additional tools (Oraios, 2026; Aider-AI, 2026), yet agents may keep relying on lexical search when such tools are available (Xu, 2026), and both tools increase cost in our evaluation (Section 5.3). SWELSP reuses these language server, but with symbol-based output enrichment making symbol-indexed retrieval the default exploration step rather than an optional tool.

7 Conclusion

We presented SWELSP, a symbol-indexed navigation interface that reuses existing language servers. It connects code locations in search results and tracebacks to symbols that agents can use to retrieve definitions and references, reducing the need for intermediate searches and window adjustments. Under a fixed-window retrieval model, our analysis identifies input-token overhead from excess code, additional retrieval rounds, and repeated processing of retained history. On SWE-bench Multilingual, SWELSP reduces estimated API cost by 10.1–21.3% while preserving or improving Pass@1 across all three models. On SWE-QA, it reduces cost by 19.8–29.6% with answer-quality scores above vanilla. Compared with the widely adopted Serena and AiderMap, which increase cost, SWELSP achieves the lowest cost per resolved instance. These results suggest that efficient agents need symbol-based retrieval to replace text exploration rather than add to it.

AI USE STATEMENT

In this work, we used generative AI tools to polish the manuscript and assist with developing and checking the theoretical derivations, and drafting sections of the paper. We did not use generative AI tools to generate synthetic datasets, for research ideation, or to find related work. The authors reviewed all AI-assisted content, including the theoretical derivations and manuscript text, and take full responsibility for the final paper, including any text, claims, and artifacts produced with the assistance of generative AI.

REPRODUCIBILITY STATEMENT

We provide the complete source code for the implementations described in this paper, instructions and procedures for reproducing the experiments, and the raw data underlying all reported experimental results. Further, we explicitly explain every detail of the experiment, such as the environment, specific templates in Appendix F. Together, these materials support independent reproduction and verification of our results. We also report the API cost calculation protocol for the LLM version that we have used. It can contribute to more reproducible and reliable results in our paper.

REFERENCES

Connor Adams, Aalhad Patankar, Alex Martin, and Poolside Evals Team. Through the looking glass of benchmark hacking. Poolside Blog, May 2026. URL https://poolside.ai/blog/through-the-looking-glass.

Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. Monitor-guided decoding of code lms with static analysis of repository context. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

Aider-AI. Aider: Ai pair programming in your terminal. https://github.com/Aider-AI/aider, 2026. GitHub repository. Accessed: 2026-09-20.

Anthropic. Claude code. https://claude.com/product/claude-code, 2026. Product documentation. Accessed: 2026-07-08.

Anysphere. Cursor ai code editor, 2026. URL https://cursor.com. An AI-first code editor fork of VS Code.

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, 2026. URL https://arxiv.org/abs/2604.22750.

Zhaoling Chen, Robert 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 Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8697–8727, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.426. URL https://aclanthology.org/2025.acl-long.426/.

Wei Cheng, Yuhan Wu, and Wei Hu. Dataflow-guided retrieval augmentation for repository-level code completion. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7957–7977, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.431. URL https://aclanthology.org/2024.acl-long.431/.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Cocomic: Code completion by jointly modeling in-file and cross-file context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pp. 3433–3445, 2024.

Shuzheng Gao, Wenhao Zeng, Zhaojian Yu, Jianqiao Wangni, Chaozheng Wang, Kai Cai, Shilin He, and Michael R Lyu. Swe-mem: Learning adaptive memory management for long-horizon coding agents. *arXiv preprint arXiv:2606.28434*, 2026.

Gwihwan Go, Chijin Zhou, Quan Zhang, Yu Jiang, and Zhao Wei. Lspai: An ide plugin for llm-powered multi-language unit test generation with language server protocol. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, pp. 1440–149, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712760. doi: 10.1145/3696630.3728540. URL https://doi.org/10.1145/3696630.3728540.

Gwihwan Go, Quan Zhang, Chijin Zhou, Zhao Wei, and Yu Jiang. Lsprag: Lsp-guided rag for language-agnostic real-time unit test generation. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, ICSE '26, pp. 4026–4038, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400720253. doi: 10.1145/3744916.3773189. URL https://doi.org/10.1145/3744916.3773189.

Zixiao Huang, Wen Zeng, Tianyu Fu, Tengxuan Liu, Yizhou Sun, Ke Hong, Xinhao Yang, Chengchun Liu, Yan Li, Quanlu Zhang, Guohao Dai, Zhenhua Zhu, and Yu Wang. Reducing latency of llm search agent via speculation-based algorithm-system co-design, 2025. URL https://arxiv.org/abs/2511.20048.

Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. Issue localization via llm-driven iterative code graph searching. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 3034–3045. IEEE, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. Ai agents that matter, 2024. URL https://arxiv.org/abs/2407.01502.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An llm compiler for parallel function calling. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

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 *NeurIPS 2025 DL4C Workshop*, 2025. URL https://arxiv.org/abs/2508.21433.

Shukai Liu, Bo Jiang, Jian Yang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 20604–20617, 2026.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *arXiv preprint arXiv:2406.07003*, 2024.

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: Human Language Technologies (Volume 1: Long Papers)*, pp. 142–160, 2025.

Microsoft. Language server protocol specification, version 3.17. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/, 2022. Accessed: 2026-07-08.

Emerson Murphy-Hill, Jenna Butler, and Alexandra Savelieva. Adoption and impact of command-line ai coding agents: A study of microsoft's early 2026 rollout of claude code and github copilot cli, 2026. URL https://arxiv.org/abs/2607.01418.

Oraios. Serena: The ide for your coding agent. https://github.com/oraios/serena, 2026. GitHub repository. Accessed: 2026-09-20.

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 Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 30098–30121, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/4a4a3c197deac042461c677219efd36c-Paper-Conference.pdf.

Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. SWE-QA: Can language models answer repository-level code questions? In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 8230–8245, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.402. URL https://aclanthology.org/2026.findings-acl.402/.

Romain Robbes, Théo Matricon, Thomas Degueule, Andre Hora, and Stefano Zacchiroli. Agentic much? adoption of coding agents on github, 2026. URL https://arxiv.org/abs/2601.18341.

Mohamad Salim, Jasmine Latendresse, SayedHassan Khatoonabadi, and Emad Shihab. Tokenomics: Quantifying where tokens are used in agentic software engineering. *arXiv preprint arXiv:2601.14470*, 2026.

Yuling Shi, Yichun Qian, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. Longcodezip: Compress long context for code language models. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 141–153. IEEE Press, 2025. doi: 10.1109/ASE63991.2025.00020. URL https://doi.org/10.1109/ASE63991.2025.00020.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward hacking. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2407.16741.

Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. Swe-pruner: Self-adaptive context pruning for coding agents, 2026. URL https://arxiv.org/abs/2601.16746.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. *arXiv preprint arXiv:2403.10059*, 2024.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

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

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. Orcaloca: an llm agent framework for software issue localization. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25. JMLR.org, 2025.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151/.

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*, ISSTA 2024, pp. 1592–1604, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706127. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

APPENDIX

Appendix Contents

A. System Architecture 14
B. Ablation Study 15
C. Case Studies 15
D. Runtime Overhead 19
E. Discussion 20
F. Experiment Details 20

A SYSTEM ARCHITECTURE

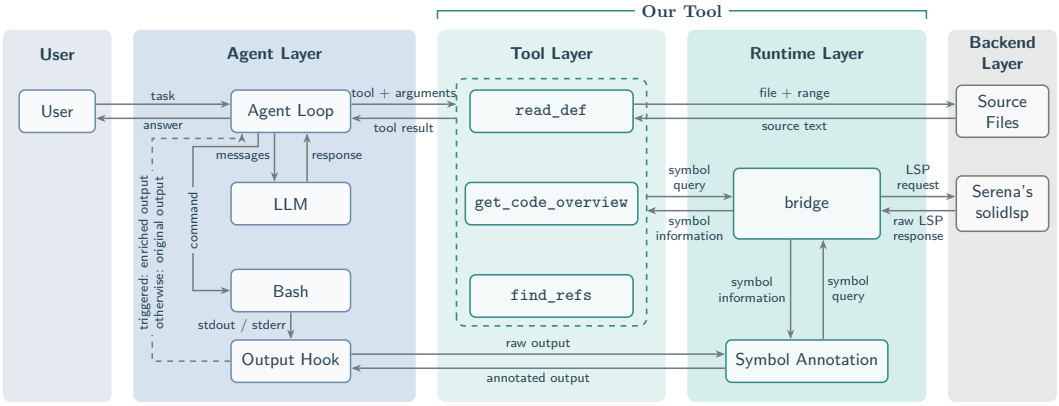


Figure 6: System architecture of SWELSP, showing the navigation path through the Tool Layer and the annotation path through the Output Hook. Other tool calls and direct code edits are omitted for simplicity.

SWELSP connects agent exploration to program symbols through two paths: explicit navigation calls and automatic annotation of shell output. Figure 6 organizes them into four layers. The Agent Layer runs the interaction loop, the Tool Layer exposes navigation operations, the Runtime Layer processes symbol queries and annotations, and the Backend Layer provides analysis results through LSP and source files.

In the navigation path, the agent invokes a tool with its arguments. `get_code_overview` returns a file outline, `read_def` retrieves symbol definitions, and `find_refs` returns reference locations with their enclosing symbol context. The LSP bridge converts document-symbol records and source locations into symbol names and ranges that the tools can use. For `read_def`, the runtime first obtains the matching symbol's file and range, then reads that range directly from the local source file.

In the annotation path, the Output Hook checks the host integration's annotation conditions after each Bash command. If they are not met, it returns the original output unchanged. Otherwise, Symbol Annotation maps the reported source locations to symbol context, and the hook returns the enriched output, exposing addresses for later navigation without a separate request.

We reuse only Serena's `solidlsp` layer (Oraios, 2026), which launches language servers and relays LSP requests and responses. All other components are our own, including the bridge in Section 4.1, the navigation tools and their descriptions, the symbol-address format, result formatting, and the output hooks.

B Ablation Study

We examine how output enrichment affects symbol-based navigation on SWE-bench Multilingual with Mini-SWE-Agent. We select V4-Flash and GPT-5.5 because their baseline token usage differs by about 25× (10.61 M vs. 0.417 M tokens per instance). This contrast lets us examine enrichment under both high and low token usage. We compare three configurations:

- **Vanilla:** the Mini-SWE-Agent agent with shell-based code navigation.
- **Nav-only:** SWELSP with `get_code_overview`, `read_def`, and `find_refs`, but without output-enrichment hooks.
- **SWELSP:** navigation tools with hooks that attach symbol context to supported shell outputs, providing addresses for subsequent navigation calls.

Table 3: Ablation of output enrichment on SWE-bench Multilingual with Mini-SWE-Agent.

Agent	Pass@1	Tokens	Cost ($)	Cost/Resolved ($)
Vanilla (V4-Flash)	70.7	10.609 M	0.093	0.131
+ Nav-only	65.7	12.464 M	0.109	0.166
Δ *(Vanilla→Nav-only)*	↓5.0	↑17.5%	↑17.5%	↑26.5%
+ SWELSP	**71.0**	**8.329 M**	**0.073**	**0.103**
Δ *(Nav-only→SWELSP)*	↑5.3	↓33.2%	↓33.0%	↓38.0%
Vanilla (GPT-5.5)	71.7	0.417 M	0.525	0.733
+ Nav-only	67.0	**0.393 M**	0.475	0.710
Δ *(Vanilla→Nav-only)*	↓4.7	↓5.5%	↓9.5%	↓3.2%
+ SWELSP	**72.7**	**0.393 M**	**0.472**	**0.650**
Δ *(Nav-only→SWELSP)*	↑5.7	↓0.2%	↓0.6%	↓8.4%

Results. Output enrichment improves Pass@1 over Nav-only for both models, but its effect on token usage is model-dependent (Table 3). For V4-Flash, Nav-only increases tokens by 17.5% and lowers Pass@1 by 5.0 percentage points relative to vanilla. Adding enrichment reduces tokens by 33.2% and raises Pass@1 by 5.3 percentage points relative to Nav-only.

For GPT-5.5, Nav-only already reduces tokens by 5.5%, but lowers Pass@1 by 4.7 percentage points relative to vanilla. Full SWELSP raises Pass@1 from 67.0% to 72.7% while using essentially the same number of tokens as Nav-only (a 0.2% reduction). Our trajectory analysis suggests that the more complete context supplied by enrichment supports additional code exploration, whose token consumption partly offsets the savings from more targeted retrieval. Thus, the main benefit in this setting is improved task performance at comparable token usage, rather than a substantial further reduction in tokens. As a result, SWELSP saved the cost per resolved instance by 8.4% compared to Nav-only. These results indicate that enrichment can improve efficiency of coding agent either by reducing exploration overhead or by supporting effective exploration at comparable token usage.

C Case Studies

We use recorded agent trajectories to illustrate how SWELSP reduces tool output during code navigation. Each figure states the task and the information available at the start of the excerpt, then compares conventional navigation on the left with SWELSP on the right. We align the excerpts by information goal and report output tokens for the illustrated navigation steps, rather than for the complete task.

Case 1. Figure 7 shows an issue from Caddy, a web server written in Go. The goal is to fix a bug in which quoted expressions fail unless the user writes the keyword `expression` before them. Earlier steps have found `MatchExpression` in a test file, and the agent now needs two pieces of code: the parser for `MatchExpression` and the function that builds the parser's input. Both agents reach the same code in three steps, so we compare the tool output of each step.

Step 1, locating the parser. The vanilla agent greps for the parser name and for `MatchExpression`, which returns every line that mentions these names. The result gives a file and a line number

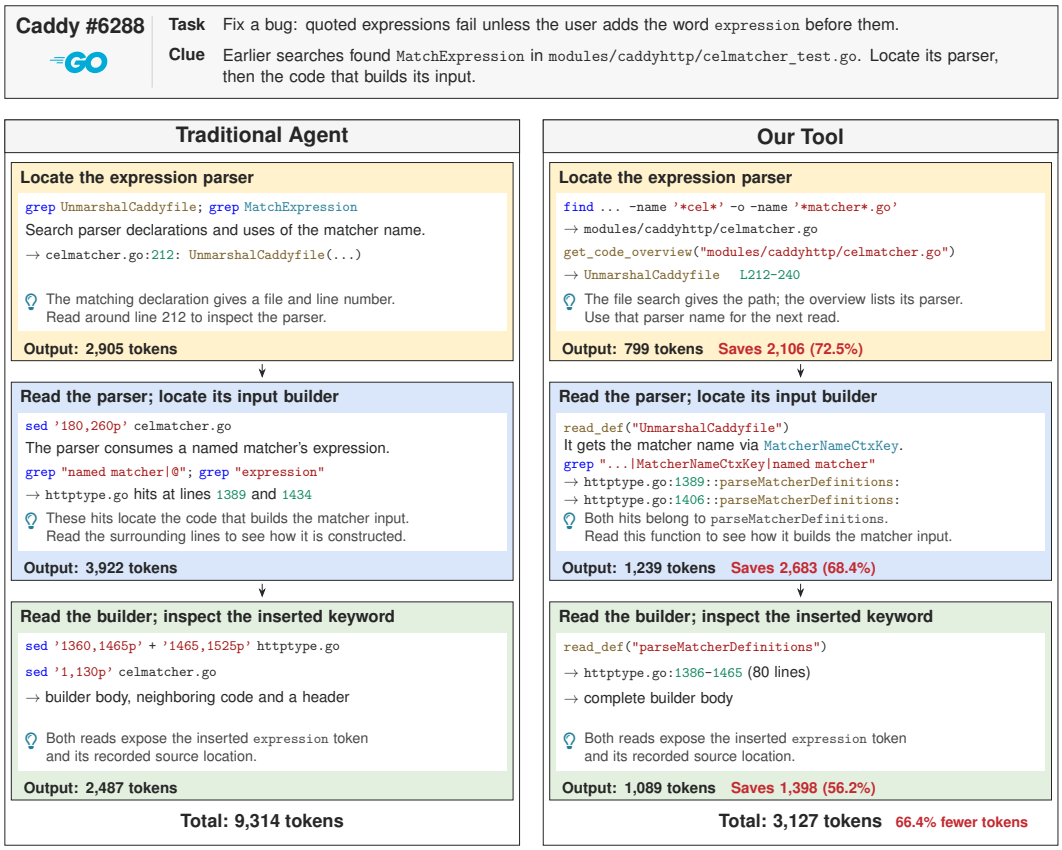


Figure 7: Navigation in Caddy #6288. The agent locates the parser for `MatchExpression` and then the function that builds its input. The vanilla agent (left) and SWELSP (right) reach the same code in three matched steps. Numbers report tool-output tokens for each step.

(`celmatcher.go:212`), but not where the parser ends. SWELSP finds the file by name and requests its outline with `get_code_overview`. The outline lists each symbol with its line range, including `UnmarshalCaddyfile` at lines 212–240, without returning any function bodies. This step costs 799 tokens instead of 2,905.

Step 2, reading the parser and finding its input builder. Knowing only line 212, the vanilla agent guesses a window and prints lines 180–260. This 81-line window contains the 29-line parser together with unrelated code around it. The agent then greps for strings seen in the parser and gets two line numbers in `httptype.go`, which do not say which function contains them. SWELSP reads exactly the parser with `read_def`. The parser body shows that the matcher name comes from `MatcherNameCtxKey`, so the agent greps for that key. Output enrichment appends the enclosing function to each hit, as in `httptype.go:1389::parseMatcherDefinitions`. The agent thus learns that both hits belong to one function and obtains the name it should read next. This step costs 1,239 tokens instead of 3,922.

Step 3, reading the input builder. The vanilla agent again has only line numbers. It prints lines 1360–1465 and then 1465–1525, since a line number does not reveal where the function ends, and it also reads the first 130 lines of `celmatcher.go`. Together, these reads return 296 lines to inspect an 80-line function. SWELSP calls `read_def` on `parseMatcherDefinitions` and receives exactly its 80 lines, which include the code that inserts the `expression` token. This step costs 1,089 tokens instead of 2,487.

Summary. Across the three steps, tool output falls from 9,314 to 3,127 tokens, a 66.4% reduction. The savings follow the decomposition in equation 4. Outlines and `read_def` return code bounded by symbol boundaries, whereas line-number windows also return surrounding code, which is the

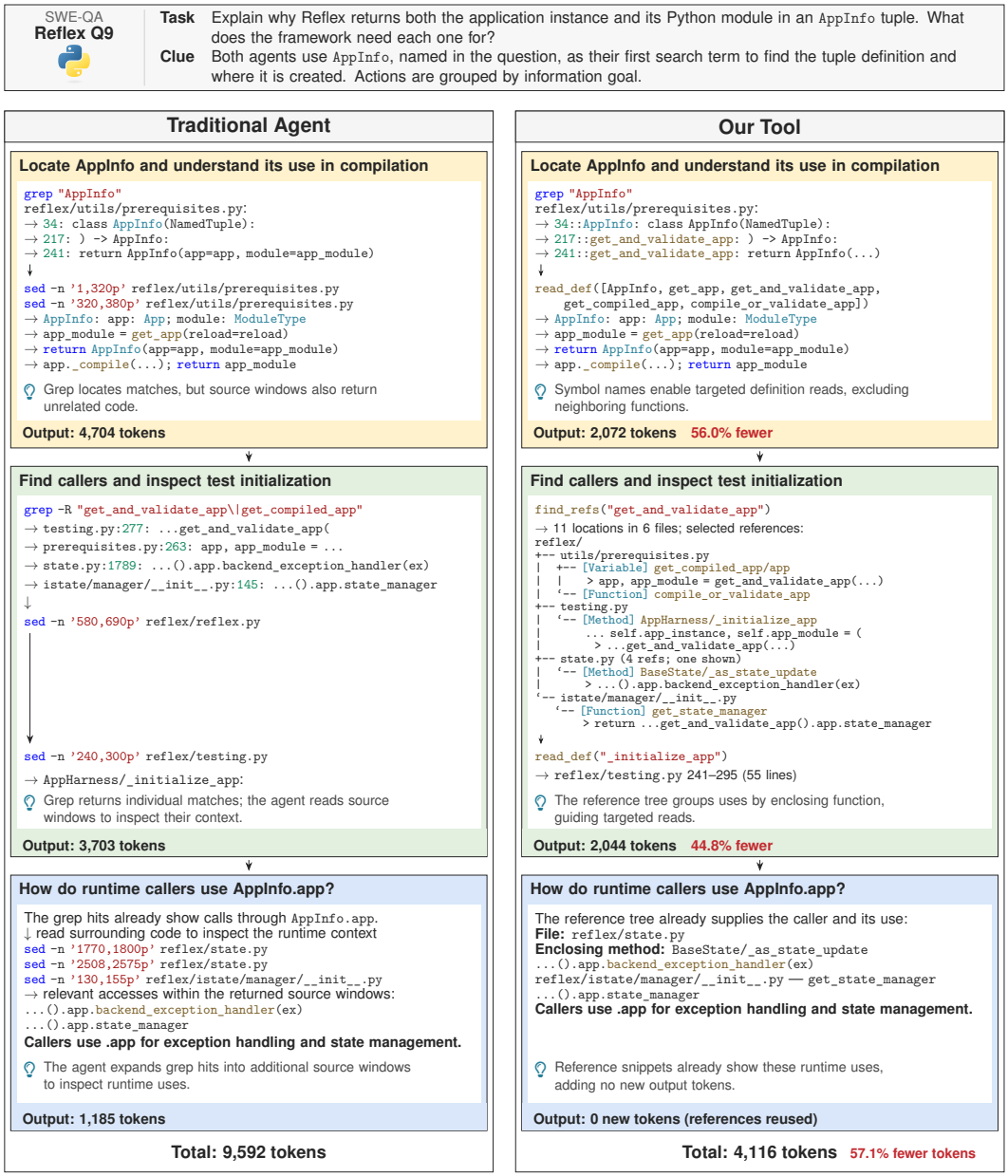


Figure 8: Navigation for SWE-QA Reflex Q9, which asks why `AppInfo` holds both the application instance and its module.

excess-code term. Enrichment turns a search hit into a symbol name, so the agent moves from a hit to the right definition without guessing or reading a second one, which avoids the extra-round term.

Case 2. Figure 8 shows a question from SWE-QA on Reflex, a Python web framework. The question asks why Reflex returns both the application instance and its Python module in an `AppInfo` tuple, and what each one is used for. Answering it requires three pieces of information, namely where `AppInfo` is created, which callers use it, and how they use its `app` field at runtime. Both agents start by searching for `AppInfo`, which the question names.

Step 1, locating `AppInfo` and its role in compilation. Both agents grep for `AppInfo` and find matches in `prerequisites.py`, three of which the figure shows. The vanilla agent then prints lines 1–380 of that file to see the code around these matches. This returns the tuple and the loading functions, but also every unrelated function in between. For SWELSP, output enrichment la-

bels for each match with its enclosing symbol, for example `217::get_and_validate_app`. The complete grep output also labels matches inside `get_app`, `get_compiled_app`, and `compile_or_validate_app`. With these names, the agent reads all five symbols in a single `read_def` call, which returns only their definitions. Both agents obtain the same facts, and this step costs 2,072 tokens instead of 4,704.

Step 2, finding callers and inspecting test initialization. The vanilla agent greps for the two loader functions and gets a flat list of matching lines. Each hit gives a file and a line number but not the function it belongs to, so the agent reads source windows in `reflex.py` and `testing.py` to find out. SWELSP calls `find_refs` on `get_and_validate_app`. The result is a reference tree that groups the 11 call sites by file and enclosing symbol, and shows the calling line under each symbol. From this tree, the agent sees that `AppHarness/_initialize_app` stores both the instance and module, and reads that 55-line code with `read_def`. This costs 2,044 tokens instead of 3,703.

Step 3, examining how runtime callers use `AppInfo.app`. The earlier grep hits already show calls such as `().app.backend_exception_handler(ex)`, but not the functions they appear in. The vanilla agent therefore reads three more source windows in `state.py` and `istate/manager/__init__.py` to see the surrounding code. SWELSP needs no new tool call. The reference tree from Step 2 already lists these calls under their enclosing symbols, `BaseState/_as_state_update` and `get_state_manager`. The agent concludes that callers use `app` for exception handling and state management, adding 1,185 tokens for the vanilla agent and none for SWELSP.

Summary. Across the three steps, tool output falls from 9,592 to 4,116 tokens, a 57.1% reduction. Enrichment and `read_def` again bound reads to symbol definitions, as in Case 1. This case adds a second effect. A grep hit reports only a line, so the vanilla agent must read a window around it to learn which function contains it. The reference tree from `find_refs` returns each call site together with its enclosing symbol, so one call answers questions that would otherwise need several reads, including later ones.

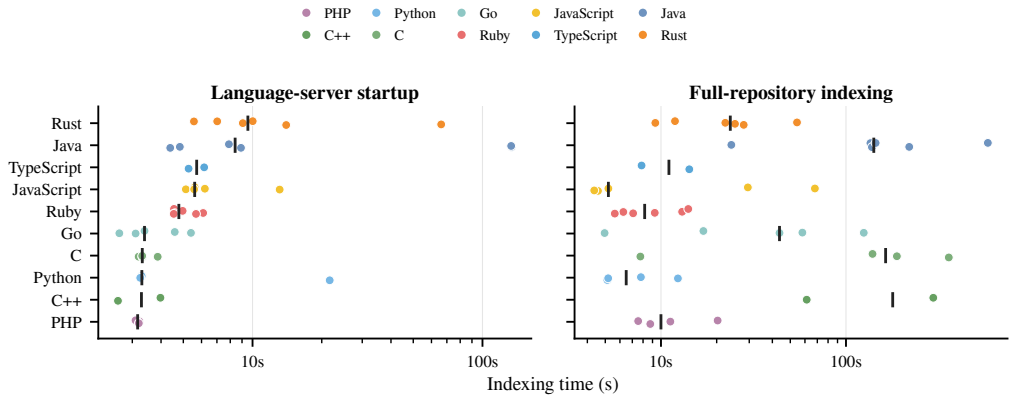


Figure 9: Language-server startup and full-indexing time across 44 measured repositories from SWE-bench Multilingual and SWE-QA, grouped by indexing language. Vertical bars of each figure mark per-language medians.

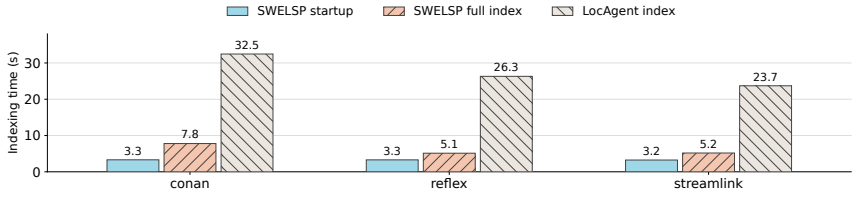


Figure 10: Preprocessing time comparison with LocAgent on SWE-QA repositories.

D RUNTIME OVERHEAD

SWELSP adds little time relative to agent execution. We measure two sources of overhead, repository preparation before navigation and output enrichment during execution, and report them separately because they measure different operations.

D.1 INDEXING OVERHEAD

We measure two preparation costs. Cold startup is the wall-clock time from launching the bridge until SWELSP can answer queries; this may include initializing multiple language servers in mixed-language repositories. Full repository indexing measures the time to collect document symbols for the configured language across the repository. The measurements cover 41 SWE-bench Multilingual repositories and three SWE-QA repositories, totaling 44 repositories. Figure 9 groups repositories by their dominant source-file language. The Python group includes Ruff and the three SWE-QA repositories.

SWELSP supports navigation without building a full repository index. We therefore report startup as the preparation time before queries can be served, and full indexing as an eager-preprocessing reference for comparison with LocAgent. Across the 44 measured repositories, median cold startup is 4.9 s and median full indexing is 15.6 s (Figure 9). For the 41 SWE-bench Multilingual repositories, the median repository-level ratios of startup and full-indexing time to the recorded GPT-5.5 agent runtime are 0.9% and 3.3%, respectively. Full-indexing time varies widely across repositories and reaches 584.6 s in the worst case.

Comparison with LocAgent. We compare preprocessing time on the three SWE-QA repositories. LocAgent preprocessing includes graph construction, graph serialization, and BM25 index construction. SWELSP full indexing takes 5.1–7.8 s, compared with 23.7–32.5 s for LocAgent. LocAgent preprocessing takes 4.2, 5.1, and 4.6 times as long on Conan, Reflex, and Streamlink, respectively (Figure 10).

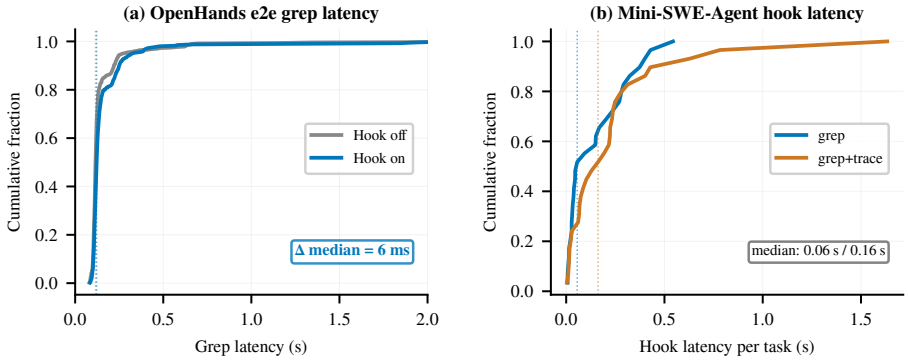


Figure 11: Hook latency with V4-FLASH on 30 SWE-bench Multilingual tasks. (a) End-to-end grep latency with hooks enabled or disabled. (b) Cumulative hook-processing time per task.

D.2 HOOK OVERHEAD

The output hooks run after each supported shell command and add symbol annotations to its output. We measure their cost in two ways. The first measures how long the agent waits for a grep command, and the second times only the hook code itself. Both use V4-FLASH.

Grep latency seen by the agent. In OpenHands, we run the same 30 SWE-bench Multilingual tasks twice, once with hooks enabled and once with hooks disabled. The median grep latency is 0.122 s with hooks and 0.116 s without, so hooks add about 6 ms to a typical command (Figure 11(a)). The slowest 1% of commands take at least 1.389 s with hooks and 0.659 s without.

Time spent inside the hooks. To isolate the hooks' own cost, we time the hook code in Mini-SWE-Agent. We sum this time over all hook calls in a task. We test two hook settings. `grep` activates only the search result hook. It annotates symbols in search results such as grep, and `grep + trace`

means full activation of our hook settings. Per task, hooks spend a median of 0.06 s with grep annotation and 0.16 s with trace analysis added, and never more than 1.64 s (Figure 11(b)). This is negligible compared with the median task runtime of over 750 s.

E Discussion

Symbol-guided navigation vs. symbol-based editing. Symbol information can support editing as well as navigation. Serena, for example, exposes symbol-body replacement and renaming (Oraios, 2026). In our preliminary runs, agents did not use these interfaces consistently, which suggests a mismatch with their established text-editing habits. For example, changing one condition through whole-symbol replacement requires rewriting the entire function, whereas a targeted patch changes only the relevant lines. We therefore apply symbols to navigation and keep the familiar editing interfaces. This observation concerns specific interfaces and current agent behavior rather than symbol-based editing in general, and making such editing work well for agents remains future work.

Other LSP capabilities. Language servers also offer quick fixes and refactorings through the code-action interface. A future extension could let the agent apply such an action directly when an existing transformation suffices, skipping an LLM generation step and the latency and cost it adds (Kim et al., 2024; Huang et al., 2025). Hover information could likewise expose documentation and types, helping agents understand an API without reading its implementation. Beyond standard LSP responses, combining symbol resolution with control-flow analysis could yield semantically annotated control-flow graphs (Go et al., 2026), although this requires additional program analysis.

Limitations. Our approach depends on the language server's support and resolution accuracy. Dynamic features, such as runtime attribute creation and reflective calls, can make definitions and references difficult to resolve statically. Missing dependencies and incomplete project configuration can interfere with the language server's analysis, thereby limiting SWELSP's ability. In this case, agents may still need textual search and execution-based evidence.

F Experiment Details

F.1 Setup

Execution environment. We run agents and repository tools on an Ubuntu server with 256 CPU cores and 1.0 TiB of RAM. Each SWE-bench instance runs in a separate container based on the benchmark's `x86_64` image, with a six-hour timeout. We evaluate generated patches using the SWE-bench evaluation harness v4.1.0, with a 1,800-second timeout per instance.

Prompt templates. Prompt 1 shows the instance template adapted from Mini-SWE-Agent, with `{{task}}` replaced by the benchmark task description. We append an instruction prohibiting retrieval of upstream fixes through repository fetching, GitHub queries, or external APIs. This restriction is intended to prevent agents from using existing solutions instead of solving the task from the provided checkout. Every SWE-Multilingual instance used this prompt templates as task instructions.

Prompt 1: Mini-SWE-Agent instance template with an added restriction on reward hacking.

```
<pr_description>
Consider the following PR description:
{{task}}
</pr_description>

<instructions>
# Task Instructions

## Overview

You're a software engineer interacting continuously with a computer by submitting commands.
You'll be helping implement necessary changes to meet requirements in the PR description.
Your task is specifically to make changes to non-test files in the current directory in
↪  order to fix the issue described in the PR description in a way that is general and
↪  consistent with the codebase.
<IMPORTANT>This is an interactive process where you will think and issue AT LEAST ONE
↪  command, see the result, then think and issue your next command(s).</important>
```

```
For each response:

1. Include a THOUGHT section explaining your reasoning and what you're trying to accomplish
2. Provide one or more bash tool calls to execute

## Important Boundaries

- MODIFY: Regular source code files in /testbed (this is the working directory for all your
↪  subsequent commands)
- DO NOT MODIFY: Tests, configuration files (pyproject.toml, setup.cfg, etc.)

## Recommended Workflow

1. Analyze the codebase by finding and reading relevant files
2. Create a script to reproduce the issue
3. Edit the source code to resolve the issue
4. Verify your fix works by running your script again
5. Test edge cases to ensure your fix is robust

## Command Execution Rules

You are operating in an environment where

1. You issue at least one command
2. The system executes the command(s) in a subshell
3. You see the result(s)
4. You write your next command(s)

Each response should include:

1. **Reasoning text** where you explain your analysis and plan
2. At least one tool call with your command

**CRITICAL REQUIREMENTS:**

- Your response SHOULD include reasoning text explaining what you're doing
- Your response MUST include AT LEAST ONE bash tool call. You can make MULTIPLE tool calls
↪  in a single response when the commands are independent (e.g., searching multiple files,
↪  reading different parts of the codebase).
- Directory or environment variable changes are not persistent. Every action is executed in
↪  a new subshell.
- However, you can prefix any action with `MY_ENV_VAR=MY_VALUE cd /path/to/working/dir &&
↪  ...` or write/load environment variables from files

Example of a CORRECT response:
<example_response>
I need to understand the Builder-related code. Let me find relevant files and check the
↪  project structure.

[Makes multiple bash tool calls: {"command": "ls -la"}, {"command": "find src -name '*.java'
↪  | grep -i builder"}, {"command": "cat README.md | head -50"}]
</example_response>

## Environment Details

- You have a full Linux shell environment
- Always use non-interactive flags (-y, -f) for commands
- Avoid interactive tools like vi, nano, or any that require user input
- You can use bash commands or invoke any tool that is available in the environment
- You can also create new tools or scripts to help you with the task
- If a tool isn't available, you can also install it

## Submission

When you've completed your work, you MUST submit your changes as a git patch.
Follow these steps IN ORDER, with SEPARATE commands:

Step 1: Create the patch file
Run `git diff -- path/to/file1 path/to/file2 > patch.txt` listing only the source files you
↪  modified.
Do NOT commit your changes.

<IMPORTANT>
The patch must only contain changes to the specific source files you modified to fix the
↪  issue.
Do not submit file creations or changes to any of the following files:

- test and reproduction files
- helper scripts, tests, or tools that you created
- installation, build, packaging, configuration, or setup scripts unless they are directly
↪  part of the issue you were fixing (you can assume that the environment is already set
↪  up for your client)
- binary or compiled files
```

```
</IMPORTANT>

Step 2: Verify your patch
Inspect patch.txt to confirm it only contains your intended changes and headers show `---
↪  a/` and `+++ b/` paths.

Step 3: Submit (EXACT command required)
You MUST use this EXACT command to submit:

```bash
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && cat patch.txt
```

If the command fails (nonzero exit status), it will not submit.

<CRITICAL>
- Creating/viewing the patch and submitting it MUST be separate commands (not combined with
↪ &&).
- If you modify patch.txt after verifying, you SHOULD verify again before submitting.
- You CANNOT continue working (reading, editing, testing) in any way on this task after
↪ submitting.
</CRITICAL>
</instructions>

INTEGRITY RULES (strict):
- Solve the issue ONLY from the code in this checkout and the failing behavior you observe.
- Do NOT look up the official fix: no cloning/fetching the upstream repository, no querying
↪ GitHub or any external API for commits, pull requests, or diffs, and no applying
↪ patches obtained from outside this checkout.
```

Token and cost accounting. We record the token usage from providers, and we classify these tokens into cache-miss, cache-hit, and output tokens. For SWE-Pruner, token accounting includes the pruned observations delivered to the agent, but excludes discarded text and auxiliary pruning-model inference. Then, we calculate API cost as $C = 10^{-6} \sum_t (p_m M_t + p_h H_t + p_o O_t)$, where M_t, H_t, and O_t denote the numbers of cache-miss input tokens, cache-hit input tokens, and output tokens, respectively, in the t-th model invocation. The corresponding fixed per-million-token rates, p_m, p_h, and p_o, are summarized in Table 4.

Table 4: Model-token prices for standardized cost estimation (USD per million tokens).

Model	Cache-miss	Cached	Output
DeepSeek-V4-Flash	0.040	0.008	0.080
DeepSeek-V4-Pro	0.578	0.018	1.734
GPT-5.5	5.000	0.500	30.000

F.2 Additional Results

Table 5: Per-criterion judge scores on SWE-QA. Each criterion is scored from 1 to 10, and Judge is their mean.

Agent	Corre	Compl	Relev	Clari	Reaso	Total	Avg
SWE-QA(V4-Flash)	8.95	8.81	9.34	8.56	9.01	44.7	8.93
+ SWE-Pruner	8.80 ↓0.15	8.63 ↓0.18	9.13 ↓0.21	8.31 ↓0.25	8.75 ↓0.26	43.6 ↓1.1	8.72 ↓0.21
+ LocAgent	9.06 ↑0.11	8.92 ↑0.11	9.36 ↑0.02	8.55 ↓0.01	9.06 ↑0.05	45.0 ↑0.3	8.99 ↑0.06
+ SWELsp	**9.30** ↑0.35	**9.04** ↑0.23	**9.62** ↑0.28	**8.68** ↑0.12	**9.23** ↑0.22	**45.9** ↑1.2	**9.17** ↑0.24
SWE-QA(V4-Pro)	8.83	8.52	9.24	8.39	8.85	43.8	8.76
+ SWE-Pruner	9.39 ↑0.56	9.09 ↑0.57	9.77 ↑0.53	8.79 ↑0.40	9.32 ↑0.47	46.4 ↑2.6	9.27 ↑0.51
+ LocAgent	**9.49** ↑0.66	**9.16** ↑0.64	**9.90** ↑0.66	**8.93** ↑0.54	9.42 ↑0.57	**46.9** ↑3.1	**9.38** ↑0.62
+ SWELsp	**9.49** ↑0.66	9.15 ↑0.63	9.86 ↑0.62	8.87 ↑0.48	**9.49** ↑0.64	**46.9** ↑3.1	9.37 ↑0.61
SWE-QA(GPT-5.5)	9.14	8.86	9.63	8.55	9.05	45.2	9.04
+ SWE-Pruner	**9.42** ↑0.28	**9.15** ↑0.29	9.84 ↑0.21	8.75 ↑0.20	**9.34** ↑0.29	**46.5** ↑1.3	**9.30** ↑0.26
+ LocAgent	9.38 ↑0.24	8.86 - 0.00	**9.92** ↑0.29	**8.90** ↑0.35	9.24 ↑0.19	46.3 ↑1.1	9.26 ↑0.22
+ SWELsp	9.32 ↑0.18	8.85 ↓0.01	**9.92** ↑0.29	8.85 ↑0.30	9.20 ↑0.15	46.1 ↑0.9	9.23 ↑0.19

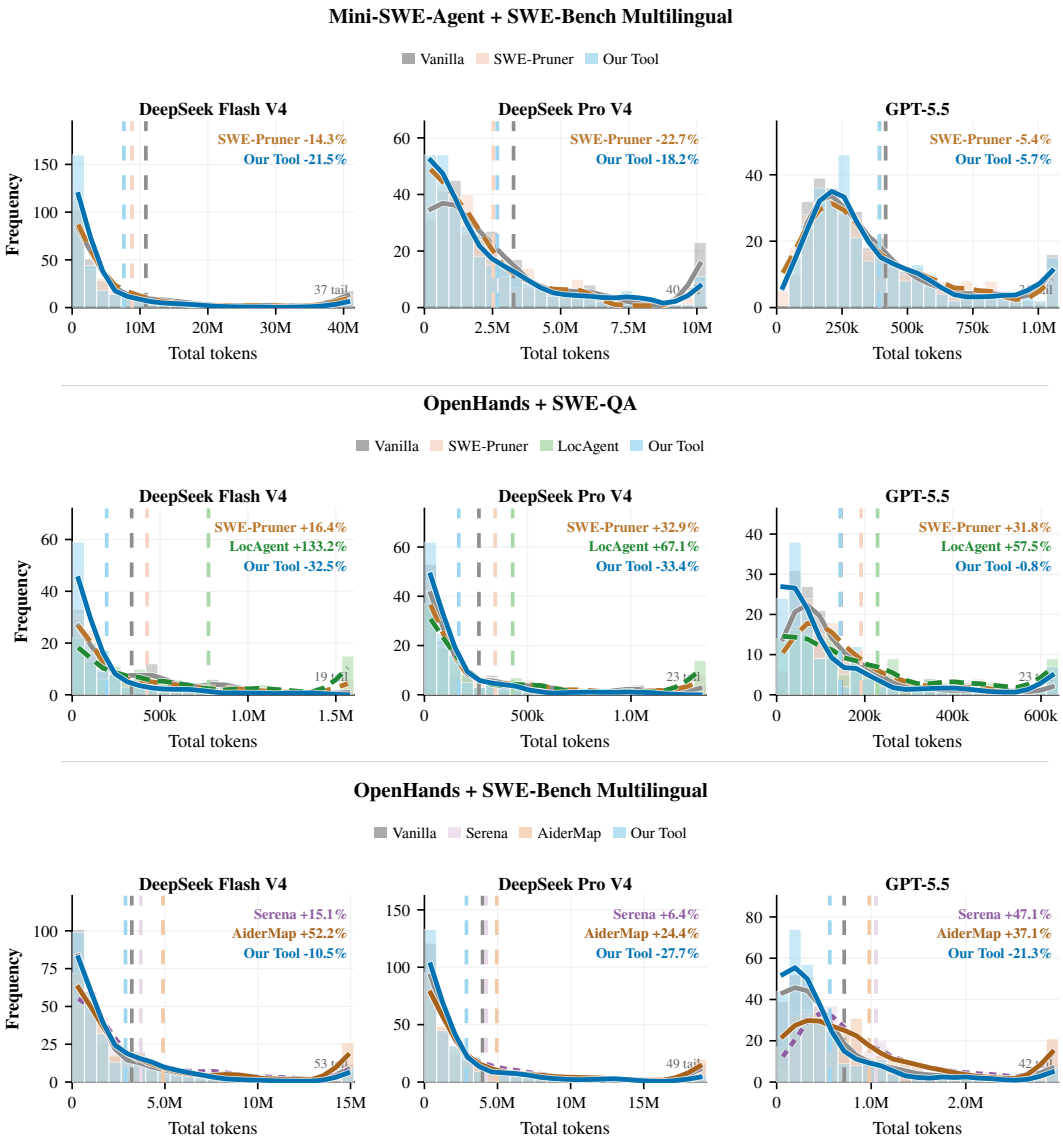


Figure 12: Per-instance total-token distributions of SWE-Multilingual Bench and SWE-QA experiments with for all configurations in each model panel. Dashed lines mark mean token counts.

Search/read output on SWE-QA. Figure 13 extends Figure 3b with SWE-Pruner and LocAgent. Each bar separates ordinary shell output from the output of LocAgent or SWELSP, and the hatched region shows the net reduction relative to vanilla after counting that output.

Token distributions. Figure 12 shows the distribution of total tokens per instance in each evaluation setting. Each row is an agent–benchmark pair, and each column is a backbone model.

Per-criterion judge scores on SWE-QA. The SWE-QA judge compares each answer with a reference answer and scores it on five criteria. Correctness checks whether the answer is factually right, completeness whether it covers the key points of the reference, clarity whether it is well organized, relevance whether it stays on the question, and reasoning whether its explanation is coherent. Table 5 reports the mean of each criterion over judged answers. SWELSP scores at or above vanilla on every criterion except completeness with GPT-5.5, where it is lower by 0.01.

Per-language analysis. We check whether token savings and cost efficiency improve together within each language. Cost efficiency is the number of resolved instances divided by the total estimated cost. Each point compares a configuration with vanilla for the same model and language, and a point in the

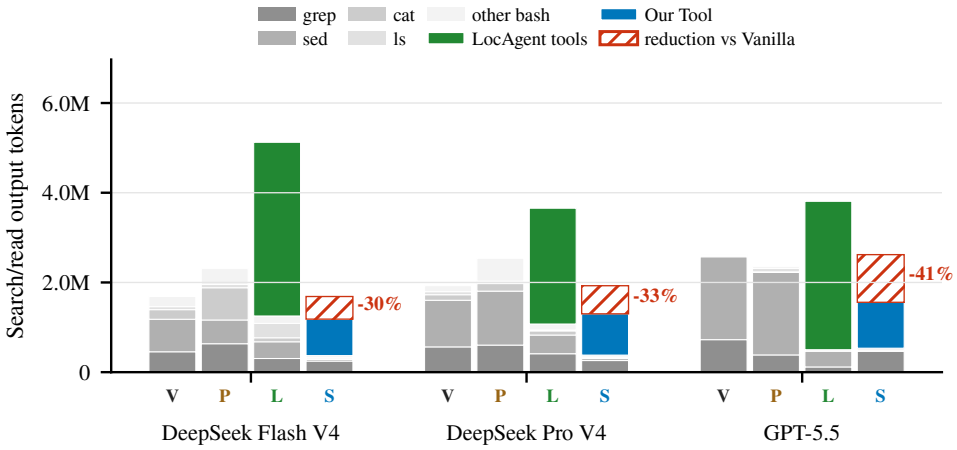


Figure 13: Search/read tool-output tokens on SWE-QA. V: vanilla; P: SWE-Pruner; L: LocAgent; S: SWELSP. Hatched regions mark the reduction relative to vanilla.

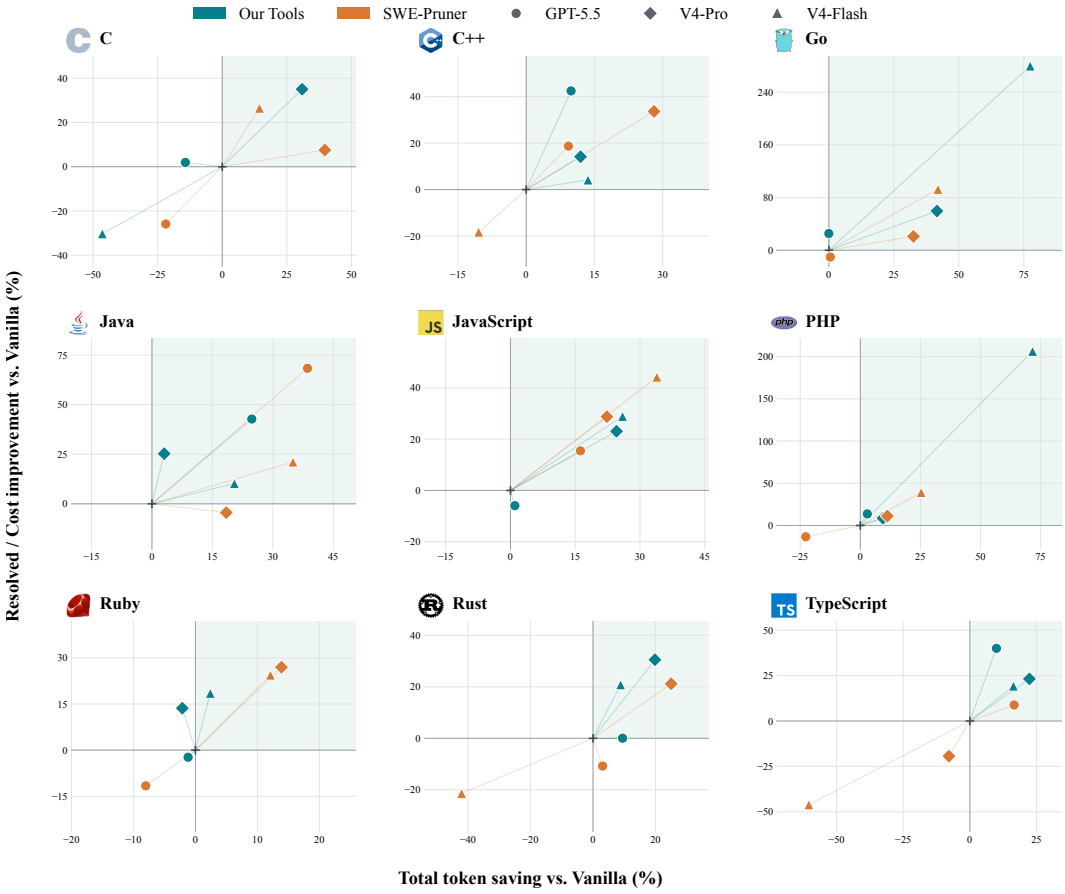


Figure 14: Token savings and improvement in resolved instances per dollar for Mini-SWE-Agent on SWE-bench Multilingual. Each point is normalized to vanilla for the same model and language, and the shaded upper-right quadrant indicates improvement in both metrics.

upper-right quadrant improves both metrics. With Mini-SWE-Agent, SWELSP improves both metrics in 21 of the 27 model–language combinations (Figure 14), and the improvements are more consistent for the two DeepSeek models than for GPT-5.5. With OpenHands, all configurations cover the same 300 instances, and SWELSP improves both metrics in 20 of the 27 combinations (Figure 15).



Figure 15: Token savings and improvement in resolved instances per dollar for OpenHands on SWE-bench Multilingual.