SWE-Adept: An LLM-Based Agentic Framework for Deep Codebase Analysis and Structured Issue Resolution

Anonymous ACL submission

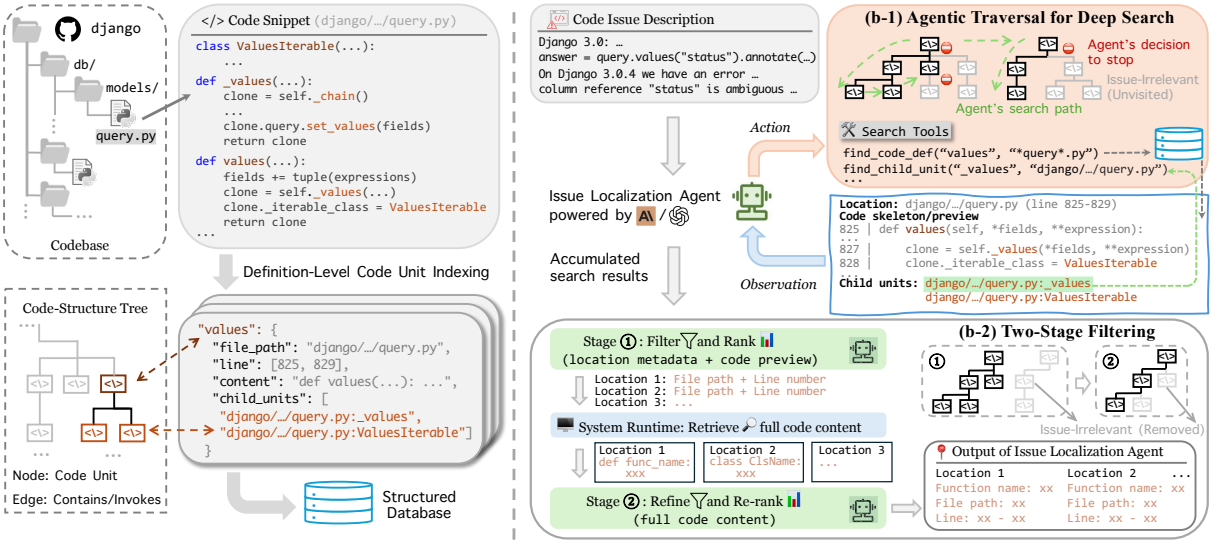
Abstract

Large language models (LLMs) exhibit strong performance on self-contained programming tasks. However, they still struggle with repository-level software engineering (SWE), which demands (1) deep codebase navigation with effective context management for accurate localization, and (2) systematic approaches for iterative, test-driven code modification to resolve issues. To address these challenges, we propose SWE-Adept, an LLM-based two-agent framework where a localization agent identifies issue-relevant code locations and a resolution agent implements the corresponding fixes. For issue localization, we introduce agent-directed depth-first search that selectively traverses code dependencies. This minimizes issue-irrelevant content in the agent's context window and improves localization accuracy. For issue resolution, we employ adaptive planning and structured problem solving. We equip the agent with specialized tools for progress tracking and Git-based version control. These tools interface with a shared working memory that stores code-state checkpoints indexed by execution steps, facilitating precise checkpoint retrieval. This design enables reliable agent-driven version-control operations for systematic issue resolution, including branching to explore alternative solutions and reverting failed edits. Experiments on SWE-Bench Lite and SWE-Bench Pro demonstrate that SWE-Adept consistently outperforms prior approaches in both issue localization and resolution, improving the end-to-end resolve rate by up to 4.7%.

1 Introduction

Recent advances in large language models (LLMs) have demonstrated remarkable programming capabilities (Anthropic, 2025b; Google, 2025; OpenAI, 2025). However, compared to isolated function- or file-level tasks (Chen et al., 2021), resolving real-world software engineering issues is substantially more challenging (Yang et al., 2024; Xia et al., 2024). First, pinpointing the relevant code locations is difficult as code repositories are large and exhibit dense cross-file dependencies. For example, each codebase in SWE-Bench (Jimenez et al., 2024) contains over 3,000 files on average, far exceeding LLM's context limit. More importantly, many issues are not self-contained: identifying the root cause often requires traversing code dependencies while avoiding context-window overflow (Ouyang et al., 2025; Yu et al., 2025). Second, implementing a correct fix typically requires iterative code modifications and test-driven validations, rather than a one-shot edit (Zhang et al., 2024b; Wang et al., 2025a,b; Yang et al., 2025).

To tackle these challenges, prior research introduces procedure-based pipelines that decompose repository-level debugging into stages consisting of localization, repair, and validation to automate issue resolution (Xia et al., 2024). Recent work adopts agent-based paradigm, equipping LLMs with tool access and enabling them to iteratively execute actions, observe feedback such as test results, and plan subsequent steps (Yang et al., 2024; Phan et al., 2024; Antoniades et al., 2025; Wang et al., 2025b). In parallel, to improve issue localization accuracy, several studies build structured codebase representations to support dependency-aware navigation and issue-relevant code retrieval (Liu et al., 2025; Ouyang et al., 2025; Chen et al., 2025b). Although these research demonstrate significant advancement, they still exhibit several limitations: (1) Less-effective context management during codebase search injects excessive issue-irrelevant information into agent's context, which in turn degrades localization accuracy (Hsieh et al., 2024; Liu et al., 2024). Some approaches employ coarse-grained codebase indexing, causing a single query to yield many candidate matches with insufficient context for effective prioritization (Yang et al., 2024). Consequently, the agent pulls in entire files or large spans to disambiguate candidates, quickly consum-



(a) Codebase Indexing and Code-Structure Tree Building

(b) Issue Localization Agent: Search & Location Pinpointing

Figure 1: Overview of **Issue Localization** framework. (a) Codebase is indexed and represented as code-structure tree in the structured database. Based on this representation, (b) Issue Localization Agent performs search and pinpointing: (b-1) agent-directed depth-first traversal for selective, dependency-aware exploration, with search tools (Table 5) returning lightweight structural information; (b-2) post-search two-stage filtering (code-preview and location heuristics followed by content-based analysis) for candidate re-ranking and final issue-relevant locations.

ing the context window with excessive content. Additionally, prior methods use algorithm-controlled traversal with fixed-hop expansion (Ouyang et al., 2025). This often enforces breadth-first, indiscriminate expansion, introducing redundancy and irrelevant search paths (Yu et al., 2025).

(2) Existing code-debugging approaches generally lack systematic strategies for issue resolution. Methods such as SWE-agent (Yang et al., 2024) often operate in a free-form "think-and-edit" loop without explicit planning and progress tracking. As iterations accumulate, interleaved code edits and newly generated test scripts can make the working state difficult to interpret and validate. More importantly, prior approaches rarely include checkpointing mechanism that continuously records intermediate code states aligned with execution milestones. Without such state logging, agents may struggle to reliably revert to a previous intermediate state after failed edits, or reset to the baseline state to attempt alternative repairs.

To address these limitations, we propose SWE-Adept, an LLM-based agentic framework for end-to-end software issue resolution. SWE-Adept consists of two specialized agents: an *issue localization agent* that searches codebase and pinpoints issue-relevant code locations, and an *issue resolution agent* that implements and validates the corresponding fixes.

To enable efficient and precise codebase navigation for issue localization, we first conduct definition-level indexing (i.e., at class and function levels) to construct fine-grained code units. Additionally, we leverage `tree-sitter`[1] to extract code dependencies and build code-structure tree to facilitate dependency-aware traversal, as illustrated in Figure 1(a). We then introduce agentic traversal (Figure 1(b-1)): the issue localization agent performs *depth-first* search over the tree, using the current code unit together with its child units and their invocation context to guide exploration. During traversal, search tools return only lightweight structural information (code skeleton/preview and location metadata) to minimize context consumption. Once search is complete, the agent follows a two-stage filtering scheme that defers full code content loading to the final re-ranking stage for precise localization, as depicted in Figure 1(b-2).

After localization, the issue resolution agent receives the identified code locations and performs structured problem solving (Figure 2(a)). The agent formulates one or more repair hypotheses, and for each hypothesis, it plans a to-do list of fine-grained implementation actions. The to-do list is dynamic. If implementation feedback (e.g., test

[1] https://tree-sitter.github.io/

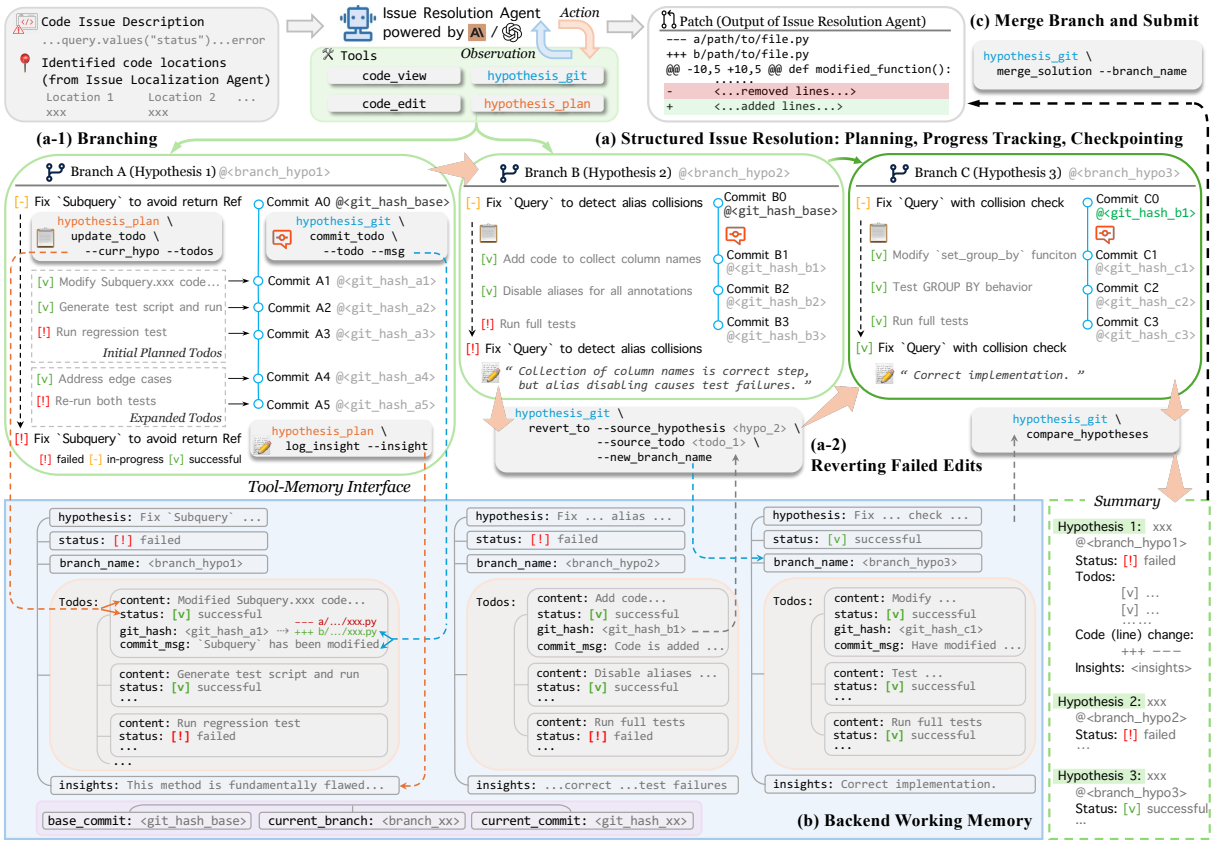


Figure 2: Overview of **Issue Resolution** framework. (a) Issue Resolution Agent takes identified code locations as input and performs structured issue resolution. The agent is equipped with two CLI (command-line interface)-based tool families (§ 3.3), `hypothesis_plan` (Table 6) and `hypothesis_git` (Table 7), for planning, progress tracking, and version control. (b) Backend working memory stores structured metadata for hypotheses, to-dos and code-state checkpoints. Both tool families interface with this memory to manage checkpoints for version-control operations, including (a-1) branching to explore alternative solutions (hypotheses) and (a-2) reverting failed edits. (c) The agent merges the selected hypothesis branch after comparing all implemented hypotheses and submits the final patch.

failures) reveals uncovered edge cases or missing steps, the agent expands the to-do list accordingly. Furthermore, we design a checkpointing mechanism that captures the intermediate code state after each completed to-do step and stores each checkpoint in a backend working memory, indexed by the corresponding step to enable precise retrieval (Figure 2(b)). To achieve this, we equip the agent with specialized tools for progress tracking and Git-based version control[2]. These tools interface with the shared working memory to manage code-state checkpoints. With this checkpointing design, the agent can reliably perform version-control operations, including branching to evaluate alternative solutions (i.e., hypotheses) and reverting failed edits, for systematic long-horizon issue resolution.

Our experimental evaluation on SWE-Bench Lite (Jimenez et al., 2024) and SWE-Bench Pro (Deng et al., 2025) demonstrates that SWE-Adept achieves superior performance in both issue localization and issue resolution, improving function-level localization accuracy by up to 5.4% and end-to-end resolve rate by up to 4.7%.

The main contributions of our work are:

- We propose SWE-Adept, an LLM-based agentic framework that integrates precise issue localization with structured issue resolution to autonomously fix repository-level software engineering issues.
- We introduce agentic traversal that enables effective context management through depth-first, dependency-guided codebase navigation, coupled with two-stage filtering to precisely pinpoint issue-relevant code locations.
- We design a tool-memory interface that enables reliable agent-driven version control for systematic, long-horizon issue resolution.

[2] https://git-scm.com/

2 Related Work

2.1 LLM for Software Engineering

Resolving issues in real-world software systems remains challenging for LLMs. To address this, recent work has proposed LLM-based agentic frameworks that empower one or more LLMs with a scaffolding (Anthropic, 2024) composed of software architecture, tool interface, and prompting strategy, making LLMs more capable on problems that demand deeper code understanding and multi-step problem solving (Yang et al., 2024; Wang et al., 2025b; Jiang et al., 2025).

In practice, resolving a software issue typically involves two essential subtasks:

Issue localization. Given an issue description, this subtask aims to identify the code locations (e.g., function, class) that are most likely the root cause and thereby the primary target for editing. Agent-based methods perform localization via multi-step, tool-assisted codebase navigation. SWE-agent (Yang et al., 2024) introduces an agent-computer interface that enhances agent's ability to search codebase. OpenHands (Wang et al., 2025b) places the LLM in a sandbox (e.g., a shell and workspace), enabling command-driven exploration. Graph-based approaches such as LocAgent (Chen et al., 2025b) and RepoGraph (Ouyang et al., 2025) build dependency graphs over code entities to guide navigation. However, achieving thorough search while maintaining effective context management is challenging yet critical, since excessive irrelevant retrieval can degrade localization accuracy (Yu et al., 2025).

Issue resolution. Given the locations of target code and relevant context, this subtask generates and applies code changes (i.e., patches) that resolve the reported issue. Agentless (Xia et al., 2024) generates multiple candidate patches and then uses test-based validation and ranking to select the final patch. SWE-agent (Yang et al., 2024) iteratively edits code and runs tests until it produces a patch that passes all required tests. AutoCodeRover (Zhang et al., 2024b) employs an LLM-based patching agent and iteratively retries to obtain an applicable patch. However, these approaches largely adopt a free-form think-and-act paradigm (Yao et al., 2023b) that can yield a disorganized edit trajectory, where successive modifications accumulate and obscure the causal link between edits and observed outcomes. They provide limited structure for systematic problem solving beyond local trial-and-error (Yao et al., 2023a; He and Roy, 2025). Some systems, such as SWE-Search (Antoniades et al., 2025) and Claude Code (Anthropic, 2025a), support plan-guided execution and code-state checkpointing. However, in SWE-Search, checkpoints are managed by the system runtime and not accessible to the agent through its context or tool interface; in Claude Code, checkpoints appear to be surfaced primarily for user control. In contrast, our design enables the agent to leverage checkpoints for autonomous version control, supporting systematic, long-horizon issue resolution.

2.2 Memory for LLM Agents

Memory increasingly serves as a fundamental component in modern LLM-based agentic framework to support multi-step decision making (Zhang et al., 2025; Hu et al., 2025b). HiAgent (Hu et al., 2025a) introduces a hierarchical working-memory design that manages intermediate task state for maintaining coherence across extended interaction. MemoryOS (Kang et al., 2025) models agent memory as an operating-system-like stack with dedicated modules for storing, update, retrieval, and synthesizing information. A-MEM (Xu et al., 2025) dynamically integrates prior experiences into a structured graph. In agentic approaches for autonomous software engineering, existing memory designs emphasize reflection and experience reuse to improve agent capability (Chen et al., 2025a; Hayashi et al., 2025). We take an orthogonal perspective and employ working memory to store code-state checkpoints, facilitating reliable agent-driven version-control operations.

3 SWE-Adept Framework

We introduce SWE-Adept, an LLM-based agentic framework for resolving repository-level software engineering issues. Figure 1 - 2 illustrate the end-to-end workflow: codebase indexing and code-structure tree building, issue localization, and issue resolution. SWE-Adept comprises two specialized agents: (i) a localization agent that navigates the repository to identify issue-relevant code locations, and (ii) a resolution agent that conducts and validates the corresponding fixes. The two agents operate in separate context windows with distinct tool access, preventing the full search-and-edit trace from accumulating within a single agent (Tran et al., 2025). In the following sections, we will describe each component of the workflow in detail.

3.1 Codebase Representation

Given a code repository $\mathcal{R}$, we construct a fine-grained, definition-level indexing to balance context completeness and context efficiency for downstream codebase navigation. We use `tree-sitter` to parse and segment $\mathcal{R}$ into a set of code units $\mathcal{U} = \{u_i \mid i = 1, \ldots, |\mathcal{U}|\}$ by first extracting function and class definitions as the primary, self-contained units. For any remaining code content (e.g., global variable definitions or other top-level logic), we segment these portions into fixed-length chunks of 200 lines and include them as additional units in $\mathcal{U}$ to ensure full repository coverage. Each u_i is represented by metadata fields including its name $n(u_i)$ (for function/class units), source location $loc(u_i) = (p, \ell_s, \ell_e)$ (file path and start/end line numbers), and raw code text $code(u_i)$. This index enables precise referencing of target code units without loading the full file or large surrounding spans. Furthermore, similar to OrcaLoca (Yu et al., 2025) and LocAgent (Chen et al., 2025b), we leverage dependencies between code units to construct a code-structure tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ that facilitates dependency-aware codebase navigation (Figure 1(a)). Each node $v \in \mathcal{V}$ corresponds to a code unit u_i extracted during indexing, and each directed edge $e \in \mathcal{E}$ represents `contains` or `invokes`, defining a parent-child relation between units. A key difference of our representation is that, rather than constructing a monolithic global code graph, we store a lightweight adjacency list as part of each code unit's metadata. For u_i, its adjacency list $adj(u_i)$ contains the identifiers `file_path:definition_name` of its child units. This design is more retrieval-efficient: accessing a unit returns both the unit and its local adjacency, avoiding a separate dependency lookup and reducing traversal overhead.

3.2 Issue Localization

Tool design. Table 5 lists the tools provided to Issue Localization Agent. The tool set supports search across multiple granularities: file-level retrieval, class/function-level definition lookup, and line/variable-level content matching. In addition to these search capabilities, we include `find_child_unit` to make codebase navigation explicit in the agent's action space. Across all search tools, the minimum retrieval unit is an indexed code unit u_i. To improve context efficiency, the tools return concise structural information instead of full source content. Each result includes location metadata (file path and line span) and provides file skeleton (for file-level retrieval) or class/function preview with child-unit identifiers (Figure 1(b)). We also include `finish_search` to signal the completion of search.

Localization agent operation. We introduce agent-directed depth-first traversal strategy that performs selective exploration over dependency paths. To start, the localization agent extracts code entities explicitly mentioned in the issue description (e.g., file or function names) as initial entry-point keywords to invoke the search tools. If exact code references are unavailable, the agent performs pattern-based search (e.g., partial-string queries) and the tools return a ranked list of candidate code units (Table 5). Based on the tool outputs (code skeleton/preview with child-unit identifiers), the agent prioritizes one child unit at each step for deep exploration via `find_child_unit`. It recursively applies this action, following a single dependency path that is most likely issue-related. Exploration along the current path stops once the agent has sufficient understanding or determines that the path is issue-unrelated, after which it moves to the next candidate path or entry point (Figure 1(b-1)). Once the agent has obtained sufficient context, it invokes `finish_search` to end the search phase.

After exploration, the agent conducts a two-stage filtering as shown in Figure 1(b-2). The first stage shortlists candidate locations using lightweight heuristics (code skeleton/preview with invocation context, location metadata), removing clearly issue-irrelevant exploration paths. The system runtime then retrieves the full source code for the shortlisted locations and provides it as input for the second stage. This deferred full code loading minimizes redundant retrieval. In the second stage, the agent analyzes the full code implementation to further refine and re-rank the candidate locations.

3.3 Issue Resolution

We build Issue Resolution Agent on SWE-agent (Yang et al., 2024), leveraging its infrastructure while adapting its workflow to incorporate our proposed tool families and backend working memory.

Tool design. We introduce two CLI (command-line interface)-based tool families for issue resolution: `hypothesis_plan` (Table 6) and `hypothesis_git` (Table 7). Each family comprises multiple related commands under a common namespace. `hypothesis_plan` maintains (i) a set

of hypotheses (i.e., alternative solutions) and (ii) hypothesis-associated to-do lists, and tracks their execution status. It also logs insights obtained from execution feedback. `hypothesis_git` conducts Git-based version-control operations, including branching and commit-based checkpointing. Each `hypothesis_git` command wraps a sequence of low-level Git operations into a single high-level action with built-in error handling. This abstraction minimizes version-control mistakes, since multi-step Git workflows executed directly by the agent are highly error-prone over long trajectories.

Both tool families interface with a shared working memory that stores the associations among hypotheses, their to-do steps, and the corresponding checkpoint metadata (Git hashes and commit messages), as shown in Figure 2(b). The agent invokes tools using semantic identifiers (e.g., hypothesis and to-do names) as arguments; the tools access working memory to store and retrieve the associated code-state information (e.g., branch names and Git hashes). This removes the requirement for the agent to track non-semantic Git hashes in-context and enables reliable code-state management, especially under heavy branching and checkpointing.

Resolution agent operation. The resolution agent receives the identified code locations from the localization agent and uses these anchors to initialize its analysis. It first invokes `hypothesis_git` to checkpoint the original code state, then generates and runs a reproduction script to confirm the reported issue. It next performs hypothesis-driven repair. For complex issues (e.g., when the fix spans multiple code locations or involves intricate dependencies), it formulates and evaluates multiple competing hypotheses; otherwise, it proceeds with a single hypothesis when the root cause and fix strategy are clear. The agent explores one hypothesis at a time. For each hypothesis, the agent checks out an isolated branch and initializes a to-do plan of fine-grained *edit* and *test* actions. Planning is adaptive. If test feedback reveals uncovered edge cases or missing steps, the agent adds new to-do items as needed, as illustrated in Figure 2 Branch A.

Execution is checkpointed step-by-step. After each to-do, the agent invokes `hypothesis_git` to commit the current state as a code-state checkpoint. This invocation automatically stores the checkpoint metadata (Git hash and commit message) in the working memory and links it to the completed step. This semantic-step indexing of checkpoints facilitates reliable version-control operations for systematic problem solving. When a hypothesis proves partially correct (i.e., earlier steps remain useful but later direction is wrong), the agent reverts edits from the failed later steps by returning to the appropriate prior checkpoint using the semantic-step reference. By design, it then checks out a new branch to continue exploration, keeping alternative solution trajectories cleanly separated, as demonstrated in Figure 2(a-2).

After exploring all hypotheses, the agent invokes `hypothesis_git` to compile a comparative report across hypotheses, summarizing their status, to-dos, commits, code diffs, and insights to support final selection. It then merges the selected hypothesis branch into the checkpoint of original code state for submission, as shown in Figure 2(c).

4 Experiments

4.1 Experimental Setup

Datasets. We evaluate our framework on two repository-level software engineering benchmarks: SWE-Bench Lite (Jimenez et al., 2024) and SWE-Bench Pro (Deng et al., 2025). Each instance is curated from a real-world GitHub issue and its associated codebase. The task is to submit a patch that edits the relevant code to resolve the issue. Additional dataset details are provided in Appendix A.1.

Metrics. We evaluate performance on both issue localization and issue resolution.

- *Issue localization.* We follow Chen et al. (2025b) and use Acc@k. For each instance, the localization agent outputs a ranked list of locations. We take the top-k predictions and mark the instance as correct only if all locations modified in the ground-truth patch are contained in the top-k set. We report Acc@3 for file-level localization and Acc@5 for function-level localization.
- *Issue resolution.* We report the resolve rate (Jimenez et al., 2024), defined as the percentage of instances successfully resolved over the dataset. An instance counts as resolved if the submitted patch passes all corresponding tests.

Baselines. We compare our framework with representative baselines for both issue localization and end-to-end issue resolution.

- *Issue localization*: (1) Embedding-based retrieval: CodeSage-Large, a 1.3B encoder model (Zhang et al., 2024a); CodeRankEmbed, a 137M encoder model (Suresh et al., 2025). (2) LLM-based agentic localization: SWE-agent uses an

Framework	Model	SWE-Bench Lite			SWE-Bench Pro		
		File Acc@3	Func Acc@5	# Tokens	File Acc@3	Func Acc@5	# Tokens
Embedding	CodeSage-Large	71.2%	40.1%	N/A	57.3%	30.7%	N/A
Embedding	CodeRankEmbed	76.6%	50.0%	N/A	64.0%	38.0%	N/A
SWE-agent	GPT-5.2	84.7%	59.9%	75k	74.7%	42.7%	112k
	Claude-4.5	90.3%	78.5%	304k	82.0%	60.7%	321k
RepoGraph	GPT-5.2	86.7%	60.2%	146k	76.0%	44.7%	254k
	Claude-4.5	92.0%	79.2%	418k	84.0%	61.3%	431k
OrcaLoca	GPT-5.2	87.0%	63.9%	268k	78.7%	48.0%	349k
	Claude-4.5	94.7%	83.5%	470k	84.7%	63.3%	492k
LocAgent	GPT-5.2	88.3%	65.0%	221k	77.3%	47.3%	323k
	Claude-4.5	93.3%	81.8%	377k	86.0%	62.0%	412k
SWE-Adept	GPT-5.2	**92.0%**	**70.4%**	202k	**81.3%**	**51.3%**	238k
	Claude-4.5	**97.0%**	**87.6%**	347k	**89.3%**	**67.3%**	395k

Table 1: Issue localization performance of different frameworks on SWE-Bench Lite and SWE-Bench Pro with GPT-5.2 and Claude-Sonnet-4.5 (Claude-4.5). Accuracy is reported at file and function levels (§ 4.1). Best results are in bold and second-best is underlined. # Tokens denotes the total number of input and output tokens per instance.

agent-computer interface for code search (Yang et al., 2024); RepoGraph (Ouyang et al., 2025) and LocAgent (Chen et al., 2025b) build graph representations of the codebase to support navigation; OrcaLoca (Yu et al., 2025) designs specialized sub-agents to improve localization.

- *Issue resolution*: SWE-agent; RepoGraph (integrated with SWE-agent for patch generation); OrcaLoca (integrated with Agentless (Xia et al., 2024) for patch generation).

We separately employ GPT-5.2 and Claude-Sonnet-4.5 model in our framework, and reproduce other agentic approaches using the same models for comparison. Further details on the models and implementation are provided in Appendix A.2.

4.2 Main Results

As shown in Table 1 and Table 2, our framework consistently outperforms baseline approaches on both issue localization and issue resolution. For localization, it achieves the best results at both file and function levels. Focusing on the finer function-level metric (Func Acc@5), on SWE-Bench Lite, it improves over the strongest baseline by 5.4% with GPT-5.2 and 4.1% with Claude-Sonnet-4.5; on SWE-Bench Pro, the corresponding gains are 3.3% and 4.0%, respectively. In addition, our framework mostly consumes fewer tokens than graph-based approaches (RepoGraph, OrcaLoca, LocAgent) due to effective context management (§ 3.2). For end-to-end issue resolution, on SWE-Bench Lite, our framework achieves 3.3% and 2.6% higher resolve rate with GPT-5.2 and Claude-Sonnet-4.5, respectively; on SWE-Bench Pro, the corresponding improvements are 4.7% and 4.0%.

5 Further Analysis

5.1 Agent Action Patterns and Performance

Figure 4(a) presents the distribution of search actions invoked by Issue Localization Agent. The most frequent action `find_child_unit` indicates that the agent primarily performs dependency-aware multi-hop navigation over the code-structure tree. For each instance, we measure the maximum search depth across all explored paths in the agent's trajectory, and report the instance distribution and localization accuracy by maximum search depth in Figure 4(b). Localization accuracy increases from zero search depth (i.e., no `find_child_unit` call) to moderate search depth, highlighting the importance of deep codebase exploration for identifying root cause. Localization accuracy decreases at higher search depth, indicating greater problem difficulty. Despite this, for instances with search depth greater than zero, SWE-Adept consistently outperforms SWE-agent and OrcaLoca, and its advantage over OrcaLoca becomes more pronounced as search depth increases. This gain comes from more effective context management which minimizes issue-irrelevant context during search.

Figure 5(a) highlights three problem-solving behaviors of Issue Resolution Agent. The high frequency of *multi-hypothesis branching* indicates that the agent often explores multiple candidate solutions, which is beneficial for complex issues. *Dynamic to-do expansion* shows that the

Framework	Model	SWE-Bench Lite		SWE-Bench Pro	
		Resolve Rate	# Tokens	Resolve Rate	# Tokens
SWE-agent	GPT-5.2	52.3%	246k	32.7%	307k
	Claude-4.5	65.7%	2651k	41.3%	3118k
RepoGraph	GPT-5.2	50.7%	285k	33.3%	392k
	Claude-4.5	66.0%	2946k	40.0%	3575k
OrcaLoca	GPT-5.2	55.7%	487k	36.0%	638k
	Claude-4.5	68.7%	1860k	43.3%	2665k
SWE-Adept	GPT-5.2	**59.0%**	688k	**40.7%**	815k
	Claude-4.5	**71.3%**	3129k	**47.3%**	3854k

Table 2: End-to-end issue resolve rate of different frameworks on SWE-Bench Lite and SWE-Bench Pro with GPT-5.2 and Claude-Sonnet-4.5 (Claude-4.5). Best results are in bold and second-best is underlined. # Tokens denotes the total number of input and output tokens per instance, including issue localization and issue resolution.

agent adaptively updates its plan during execution based on implementation feedback. The observed *checkpoint-based reversion* demonstrates the agent's capability to revert incorrect code changes during iterative problem-solving. All these behaviors are achieved by agent-driven version-control operations through `hypothesis_git` invocations, with the tool accessing working memory to store and retrieve code-state checkpoints. In Figure 5(b), we plot the instance distribution and resolve rate by number of explored hypotheses. Although resolve rate declines as hypothesis count increases, indicating higher task complexity, our framework shows better robustness and consistently outperforms other approaches. We provide additional error analysis in Appendix B.

5.2 Ablation Study

We conduct ablation studies to evaluate the contribution of each agent in our framework (Table 3). The results demonstrate that SWE-Adept's overall advantage arises from the combination of accurate localization and systematic issue resolution. To evaluate whether the agent can precisely manage code states with raw Git commands during issue resolution, we conduct ablation experiments in which the agent directly uses raw Git operations. As shown in Table 4, this setting does not reproduce the gains of our method. It improves over vanilla SWE-agent on SWE-Bench Lite, but degrades on the more challenging SWE-Bench Pro. The main issue is long-horizon reliability: continuous Git operations executed directly by the agent are error-prone over long trajectories, and the growing number of checkpoints makes code-state tracking harder as context accumulates. In our framework, `hypothesis_git` wraps low-level Git commands into higher-level actions with built-in error handling. Furthermore, it interfaces with working memory to store and retrieve code-state checkpoints indexed by semantic execution steps. This enables more reliable code-state management for systematic, long-horizon issue resolution.

Framework	Resolve Rate
SWE-agent	41.3% (62/150)
SWE-Adept	**47.3%** (71/150)
- w/o specialized issue localization	45.3% (68/150)
- w/o specialized issue resolution	44.0% (66/150)

Table 3: Ablation results on SWE-Bench Pro with Claude-Sonnet-4.5. "w/o specialized" replaces the corresponding module with the default SWE-agent module.

6 Conclusion

We present SWE-Adept, an LLM-based agentic framework for resolving software engineering issues. SWE-Adept comprises two specialized agents, dedicated to issue localization and issue resolution, respectively. For issue localization, we introduce agent-directed depth-first traversal followed by two-stage filtering. The proposed approach enables deep codebase analysis with effective context management, leading to more precise issue localization. For issue resolution, we employ code-state checkpointing and design a tool-memory interface for code-state management in long-horizon settings. This design enables reliable agent-driven version-control operations for systematic problem solving. Experimental results show that the joint enhancement in issue localization and issue resolution yields superior overall performance for SWE-Adept, highlighting its strength in autonomous software engineering.

Limitations

Our work uses proprietary LLMs (GPT and Claude models), which demonstrate strong coding performance and robust agentic behavior, including instruction following and tool use in long-horizon tasks. One promising avenue is to transfer our design principles to open-source models, for example through agentic reinforcement learning, to improve software engineering performance while reducing deployment costs.

Additionally, our evaluation focuses on Python codebases. Extending SWE-Adept to other programming languages may require language-specific parsing and indexing implementations. However, the proposed framework for issue localization and issue resolution is language-agnostic. Future work could apply the framework to other programming languages to validate its generalizability.

Ethical Considerations

Our research complies with the Code of Ethics. We properly cite all models, methods, and datasets used in this work. The benchmark datasets in our experiments are publicly available, and our study does not use private or sensitive data. Our use of datasets and LLMs is consistent with their licenses, terms, and intended usage. Our framework presents some potential risks: as with any autonomous code generation system, SWE-Adept may produce incorrect patches, which could introduce errors if deployed without strict review and testing; and the use of proprietary LLMs may raise privacy concerns. Nevertheless, with proper supervision, our framework can improve the reliability and efficiency of software engineering.

References

Anthropic. 2024. Building effective agents.

Anthropic. 2025a. Claude code overview.

Anthropic. 2025b. Introducing claude sonnet 4.5.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Yang Wang. 2025. SWE-search: Enhancing software agents with monte carlo tree search and iterative refinement. In *The Thirteenth International Conference on Learning Representations*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Silin Chen, Shaoxin Lin, Xiaodong Gu, Yuling Shi, Heng Lian, Longfei Yun, Dong Chen, Weiguo Sun, Lin Cao, and Qianxiang Wang. 2025a. Swe-exp: Experience-driven software issue resolution. *arXiv preprint arXiv:2507.23361*.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025b. 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, Vienna, Austria. Association for Computational Linguistics.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, and 1 others. 2025. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.

Google. 2025. A new era of intelligence with gemini 3.

Hiroaki Hayashi, Bo Pang, Wenting Zhao, Ye Liu, Akash Gokul, Srijan Bansal, Caiming Xiong, Semih Yavuz, and Yingbo Zhou. 2025. Self-abstraction from grounded experience for plan-guided policy refinement. *arXiv preprint arXiv:2511.05931*.

Kang He and Kaushik Roy. 2025. LogicTree: Structured proof exploration for coherent and rigorous logical reasoning with large language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 20852–20881, Suzhou, China. Association for Computational Linguistics.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. 2024. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. 2025a. HiAgent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32779–32798, Vienna, Austria. Association for Computational Linguistics.

Yuyang Hu, Shichun Liu, Yanwei Yue, Guibin Zhang, Boyang Liu, Fangyi Zhu, Jiahang Lin, Honglin Guo, Shihan Dou, Zhiheng Xi, and 1 others. 2025b. Memory in the age of ai agents. *arXiv preprint arXiv:2512.13564*.

Zhonghao Jiang, David Lo, and Zhongxin Liu. 2025. Agentic software issue resolution with large language models: A survey. *arXiv preprint arXiv:2512.22256*.

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

Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. 2025. Memory OS of AI agent. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 25961–25970, Suzhou, China. Association for Computational Linguistics.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. 2025. 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)*, pages 142–160, Albuquerque, New Mexico. Association for Computational Linguistics.

OpenAI. 2025. Introducing gpt-5.2.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. Repograph: Enhancing AI software engineering with repository-level code graph. In *The Thirteenth International Conference on Learning Representations*.

Huy Nhat Phan, Tien N Nguyen, Phong X Nguyen, and Nghi DQ Bui. 2024. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299*.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. 2025. CoRNStack: High-quality contrastive data for better code retrieval and reranking. In *The Thirteenth International Conference on Learning Representations*.

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D Nguyen. 2025. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2501.06322*.

Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu, and Zheng Wang. 2025a. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*.

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, and 5 others. 2025b. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-mem: Agentic memory for LLM agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Boyang Yang, Zijian Cai, Fengling Liu, Bach Le, Lingming Zhang, Tegawendé F Bissyandé, Yang Liu, and Haoye Tian. 2025. A survey of llm-based automated program repair: Taxonomies, design paradigms, and applications. *arXiv preprint arXiv:2506.23749*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023b. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. 2025. Orcaloca: An LLM agent framework for software issue localization. In *Forty-second International Conference on Machine Learning*.

Dejiao Zhang, Wasi Uddin Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. 2024a. CODE REPRESENTATION LEARNING AT SCALE. In *The Twelfth International Conference on Learning Representations*.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024b. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2025. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):1–47.

A Experimental Details

A.1 Datasets

Details of the evaluation datasets are as follows:
SWE-Bench Lite, a subset of **SWE-Bench** (Jimenez et al., 2024), contains 300 instances from 11 GitHub repositories in Python. For function-level localization, we follow Chen et al. (2025b) and exclude instances whose ground-truth patches do not modify any existing functions, retaining 274 instances. For file-level localization and resolve rate, we report results on all 300 instances.
SWE-Bench Pro (Deng et al., 2025) is designed to address the limitations of existing benchmarks, including potential data contamination. It also features higher problem complexity, often requiring edits that span multiple files or functions. In our experiments, we focus on the Python subset of SWE-Bench Pro. To align localization metrics, we restrict evaluation to instances whose ground-truth patches edit at most 3 files and at most 5 functions. To reduce cost, we randomly sample 150 instances from the filtered set for evaluation.

A.2 Models and Implementation Details

Here are the versions of GPT-5.2 (OpenAI, 2025) and Claude-Sonnet-4.5 (Anthropic, 2025b) model:
`gpt-5.2-2025-12-11` (medium)
`claude-sonnet-4-5-20250929`
Both models are accessed via API. We set the temperature to 0 for Claude-Sonnet-4.5. For GPT-5.2, we use the default temperature of 1.0, which is not adjustable when running with reasoning effort[3].

Codebase indexing, code-structure tree construction, and execution of Issue Localization Agent do not require Docker. Building the index and code-structure tree for a repository takes less than one minute, making re-indexing low-overhead when the codebase changes. The ranked location predictions from Issue Localization Agent are stored in a local file and used as input to Issue Resolution Agent. To evaluate the correctness of the patch generated by Issue Resolution Agent, we launch a Docker container for each instance (following the SWE-agent evaluation setup[4]), apply the patch, and execute the tests. Working memory is represented as a persistent, JSON-serialized state structure stored in a shared registry.

We implement Issue Localization Agent using LiteLLM[5] library, and we build Issue Resolution Agent on SWE-agent to leverage its infrastructure. We apply prompt caching[6] to both agents to reduce API cost. For issue localization, the maximum number of iterations is set to 20 for each instance. For issue resolution, the per-instance cost limit is set to $5. Under these settings, our framework costs $1.76 per instance with Claude-Sonnet-4.5 and $0.41 per instance with GPT-5.2. All reported results are from a single run.

Prompt for Issue Localization Agent is shown in Figure 6, and its tools are listed in Table 5. Prompt for Issue Resolution Agent is shown in Figure 7, and its tools are listed in Table 6 and Table 7.

B Error Analysis

We compare SWE-Adept and SWE-agent in Figure 3 using a Venn diagram and error breakdown to analyze failure modes in unresolved instances. We manually review the uniquely failed instances of SWE-agent and group them into three categories: failure to recover from incorrect edits (*failed recovery*, 6 instances), incorrect solution direction (*incorrect hypothesis*, 9 instances), and wrong function localization (*localization error*, 9 instances). Correspondingly, SWE-Adept reduces failures across all three categories. This demonstrates that SWE-Adept's performance gains come from the joint contribution of accurate issue localization and systematic issue resolution.

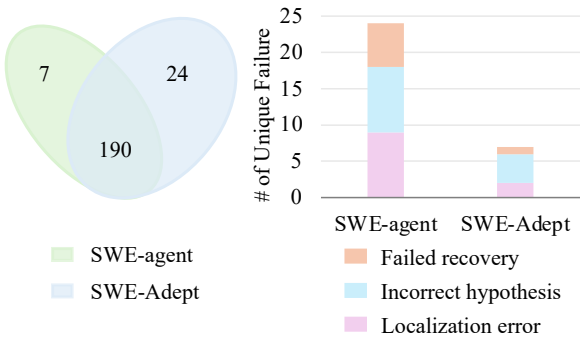
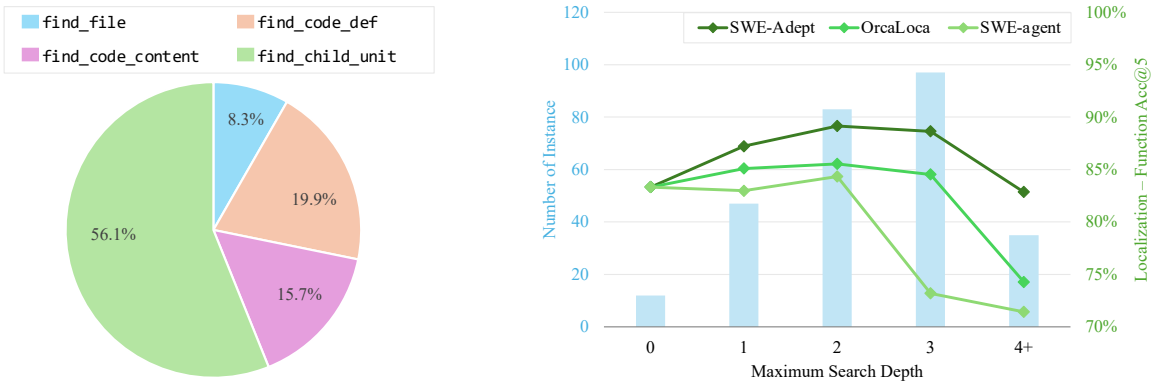


Figure 3: Venn diagram (left) of resolved-instance overlap between SWE-Adept and SWE-agent; and error breakdowns (right) for instances uniquely failed by each method (e.g., the left bar in the chart represents the 24 instances resolved by SWE-Adept but failed by SWE-agent). Reported results are on SWE-Bench Lite with Claude-Sonnet-4.5.

[3] https://platform.openai.com/docs/guides/latest-model
[4] https://www.swebench.com/SWE-bench/guides/docker_setup/
[5] https://www.litellm.ai/
[6] https://platform.claude.com/docs/en/build-with-claude/prompt-caching

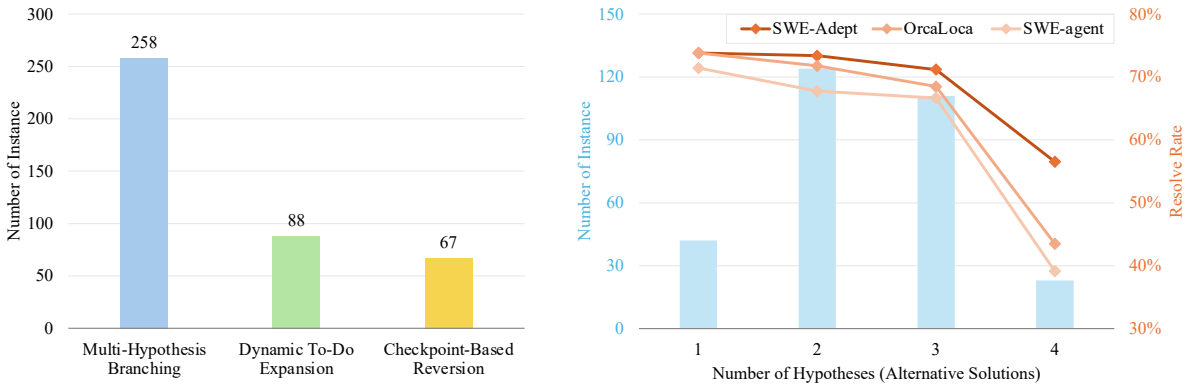
Framework	SWE-Bench Lite	SWE-Bench Pro
SWE-agent	65.7%	41.3%
SWE-agent + raw Git commands (prompted)	67.0%	38.7%
SWE-Adept	**71.3%**	**47.3%**
SWE-Adept + raw Git commands (no `hypothesis_git`, no memory)	69.0%	43.3%

Table 4: Ablation study evaluating raw Git command usage for issue resolution. Results report resolve rate on SWE-Bench Lite and SWE-Bench Pro with Claude-Sonnet-4.5.



(a) Search-tool invocation distribution of Issue Localization Agent (b) Instance distribution and localization accuracy by maximum search depth

Figure 4: Search behavior of **Issue Localization Agent** and localization accuracy by maximum search depth. (a) Search-tool invocation distribution of Issue Localization Agent. (b) Instance distribution (bars) and function-level localization accuracy (lines) by maximum search depth on SWE-Bench Lite with Claude-Sonnet-4.5.



(a) Systematic problem-solving behaviors of Issue Resolution Agent (b) Instance distribution and resolve rate by number of hypotheses

Figure 5: Problem-solving behavior of **Issue Resolution Agent** and resolve rate by number of explored hypotheses. (a) Prevalence of systematic problem-solving behaviors: multi-hypothesis branching, dynamic to-do expansion, and checkpoint-based reversion (behaviors are non-mutually exclusive and may overlap). (b) Instance distribution (bars) and resolve rate (lines) by number of explored hypotheses on SWE-Bench Lite with Claude-Sonnet-4.5.

Tool	Description	Input Parameters	Output (Agent Observation)
`find_file`	Search file by exact file-name or `glob` pattern. Enumerate files within codebase (optionally restricted to a specified directory).	*file_name*† *dir_path*‡	Matched file paths; for each file, a skeleton listing the signatures of classes and functions defined in that file.
`find_code_def`	Search class/function definition via exact matching, with fallback to `regex` matching and fuzzy matching ranked by a weighted sum of character-level similarity metrics (n-gram, Jaro–Winkler distance, and longest common subsequence). Enumerate indexed code units (optionally restricted to a file).	*definition_name*† *file_path*‡	Ranked retrieved code definitions with file path, line span, child-unit identifiers, and a concise code preview (definition signature + child-unit invocation context).
`find_code_content`	Search variable name (robust to camel-case/snake-case variants) or an exact code snippet. Enumerate code lines (optionally restricted to a file or a line span).	*content*† *file_path*‡ *start_line*‡ *end_line*‡	Retrieved matches, each associated to a containing code unit (function/class/code chunk), with the unit name, file path, line span, child-unit identifiers, and a concise code preview (unit signature + matched lines).
`find_child_unit`	Search class/function definition via exact matching given its name and file path (specified by a child-unit identifier). Enumerate indexed code units restricted to a file.	*definition_name*† *file_path*†	Exact-match class/function definition with file path, line span, child-unit identifiers, and a concise code preview (definition signature + child-unit invocation context).
`finish_search`	Signal completion of the search and trigger subsequent filtering/ranking over the collected candidates.	None	None

Table 5: List of tools for **Issue Localization Agent**. Superscripts of input parameters denote argument requirement: required†, optional‡. Gray text in the Description column provides auxiliary information on how each search tool performs enumeration.

Prompts for Issue Localization Agent

System Prompt:

You are a specialized issue localization agent responsible for conducting systematic search across code repositories to identify the code locations related to a given GitHub issue.

Follow this systematic Localization Workflow

Phase 1: Issue Analysis & Entry Point Identification

– Issue classification: Classify the issue as a bug fix, feature addition, performance issue, or configuration problem to set the localization focus. Use this category to prioritize likely modules, files, and configuration points.
– Entry point extraction: Identify a shortlist of keywords to start searching, including any files/classes/functions named in the issue, and any locations suggested by error messages or stack traces.
– Search plan: Specify an initial exploration order over entry points. Keep the plan adaptive by adding newly discovered entry points and dropping issue-irrelevant paths as results accumulate.

Phase 2: Agentic Depth-First Traversal

– Locate the entry point: Use `find_file`, `find_code_def`, or `find_code_content` to locate the current entry point. Use the returned file skeleton/code preview and child-unit identifiers to understand the local structure and how child units are invoked.
– Selective deep search: Inspect child units and select only those that are likely related to the issue based on their names and invocation context. Explore following a selected branch via `find_child_unit` and repeat recursively, going deeper only when it helps localization; stop a branch when it becomes clearly unrelated or sufficiently understood.
– Move to the next entry point: After exploring the relevant branches of one entry point, continue with the next entry point and repeat the previous steps.
– End the search phase: Call `finish_search` when the searched locations are sufficient.

Phase 3: Result Evaluation & Filtering

First-stage Filter and Rank (Code-Preview and Location Heuristics)
Use the available signals (file paths, line spans, definition names, child-unit lists, and code skeleton/previews) to prune and rank candidate locations. Prioritize locations that are directly mentioned or strongly implied by the issue, then keep nearby supporting code that is likely involved based on name/path semantics and invocation context; discard candidates with weak or unrelated signals. Output a high-to-low ranking.

Second-stage Refine and Re-Rank (Content-Based Analysis)
After Stage 1, the system will provide the full source code for the selected locations for further inspection. Confirm which locations actually result in the reported behavior. Refine the shortlist: drop locations that are clearly unrelated by implementation, and keep those with reasonable relevance. Output a final high-to-low ranked list.

Instance Template:

`{{repo_name}}` *# Replaced with the repository name*

`{{issue_description}}` *# Replaced with the issue description*

Next Action Template:

`{{observation}}` *# Replaced with the latest tool output*

Based on this observation, decide your next action.

First-Stage Filter Template:

Based on the search results above, please perform the first-stage filtering and ranking using code-preview and location heuristics. Provide your results with locations ranked from highest to lowest relevance priority.

Second-Stage Filter Template:

Here is the source code content for the locations you identified in the first stage:

`{{source_code_content}}` *# Replaced with source code content*

Now perform the second-stage filtering and re-ranking using content-based analysis. Based on your analysis of actual code structure and logic, filter and re-rank the final locations from highest to lowest relevance.

Figure 6: System prompt and stepwise instruction template for **Issue Localization Agent**.

Tool Family	Command (CLI-Based)	Description	Output (Agent Observation)
hypothesis_plan	update_hypothesis -- *hypotheses_markdown*	• Create, progress-track, and update hypotheses (i.e., alternative solutions) sorted by agent-estimated confidence, and annotate with status tags: [ ] (pending) [-] (in-progress) [v] (successful) [!] (failed) Support dynamic expansion for adaptive planning. • Store each hypothesis's content and status in working memory.	Hypothesis overview with current status and ordering.
	update_todo -- *current_hypothesis* -- *todos_markdown*	• Create, progress-track and update a to-do list for the current hypothesis, and annotate with status tags: [ ] (pending) [-] (in-progress) [v] (successful) [!] (failed) Support dynamic expansion for adaptive planning. • Store each to-do's content and status in working memory.	Hypothesis-corresponded to-do list with current status.
	log_insight -- *insight*	• Generate insights for the current hypothesis based on execution feedback, which inform reflection-driven actions (e.g., revert) and final cross-hypothesis comparison. • Store the insights in working memory.	Insights attached to the current hypothesis.

Table 6: Planning tools for **Issue Resolution Agent**. The hypothesis_plan tool family interfaces with backend working memory to support adaptive planning, progress tracking, and insight logging. Tool invocations follow the command-line interface (CLI) format: each command uses --*parameter* <*value*> syntax, e.g., hypothesis_plan log_insight --*insight* <*insight_content*>.

Tool Family	Command (CLI-Based)	Description	Output (Agent Observation)

hypothesis_git	`init_base`
	(internally executes)
	`git config`
	`git rev-parse HEAD`
	`git add -A`
	`git commit -m`
	`git rev-parse HEAD`
		• Obtain the Git hashes of (i) the original code state and (ii) a shared common working base `git_hash_base` (including issue reproduction code) for subsequent hypothesis branches.
		• Store the Git hashes in working memory.
			Confirmation of saved original code state and common working base.

	`start_hypothesis`
	-- *branch_name*
	(internally executes)
	`git stash push -m`
	`git checkout`
	`git checkout -b`
		• Checkout a new hypothesis branch from the common working base.
		• Retrieve the Git hash of the common working base from working memory.
		• Store the branch name for the corresponding hypothesis in working memory.
			Confirmation of branch creation and checkout; workspace moves to the hypothesis branch.

	`commit_todo`
	-- *todo_content*
	-- *commit_message*
	(internally executes)
	`git add -A`
	`git commit -m`
	`git rev-parse HEAD`
		• Commit code changes for one completed to-do step and obtain its checkpoint metadata (Git hash and commit message).
		• Store the Git hash and commit message for the corresponding to-do in working memory.
			Confirmation of to-do commit.

	`revert_to`
	-- *source_hypothesis*
	-- *source_todo*
	-- *new_branch_name*
	(internally executes)
	`git stash push -m`
	`git checkout`
	`git checkout -b`
		• Checkout the specified to-do checkpoint and create a new branch from that state, then switch the workspace to the new branch and continue exploration under the new hypothesis branch.
		• Retrieve the Git hash of the specified to-do from working memory.
		• Store the new hypothesis's branch name in working memory.
			Confirmation of revert and new-branch creation.

	`compare_hypotheses`
	(internally executes)
	`git diff --shortstat`
	`git diff --numstat`
		Retrieve records from working memory and compare implemented hypotheses.	Hypothesis comparison report
			- branch name
			- hypothesis content and status
			- to-dos content and status
			- commit messages
			- insights
			- code diff statistics (against the original state)

	`merge_solution`
	-- *branch_name*
	(internally executes)
	`git checkout --detach`
	`git merge`
		• Retrieve the selected hypothesis branch and the Git hash of the original code state from working memory.
		• Check out the original code state, then apply the selected branch's changes to produce a clean, patch-ready state for submission.
			Confirmation that changes are applied onto the original code state.

Table 7: Version-control tools for **Issue Resolution Agent**. The `hypothesis_git` tool family interfaces with backend working memory to manage code-state information. Tool invocations follow the command-line interface (CLI) format: each command uses *--parameter <value>* syntax, e.g., `hypothesis_git merge_solution` *--branch_name <name>*. Gray text in the Command column lists the raw Git commands executed internally (not exposed to the agent).

Prompts for Issue Resolution Agent

System Prompt:

You are a specialized issue resolution agent that can interact with a computer to solve repository-level software engineering tasks. A separate issue localization agent has already performed systematic codebase search and provided ranked code-location hints. Your job is to analyze the root cause leveraging the hinted locations, reproduce the issue, and implement a robust fix.

Follow this systematic Resolution Workflow

Phase 1: Analysis & Planning

– Location-hint analysis and problem reproduction: Start from the code locations provided and understand how they relate to the described issue. Create a script to reproduce the issue and execute it using the bash tool to confirm the error.
– Deep investigation and root cause identification: Use the reproduction result to trace the failing execution flow, leverage hinted code locations to identify where the issue originates.
– Multi-hypothesis planning: Decide adaptively between a single hypothesis (only when the root cause and fix strategy are clear) and multiple hypotheses (2-4) when the fix location/strategy is uncertain, spans multiple files/functions, or involves complex code hierarchies or test implications. Create hypotheses using `hypothesis_plan update_hypothesis`. For each hypothesis, specify a brief hypothesis description and a confidence score (0.1-1.0).

Phase 2: Git-Based To-Do Implementation

– Git workflow initialization: Invoke `hypothesis_git init_base` to store the original code state and create a common working base (includes reproduction scripts). Regular hypothesis branches start from this common working base; hypotheses created after `revert_to` action start from specified to-do's code-state checkpoints.
– Hypothesis branching and to-dos initialization: Create a descriptive branch name and switch the workspace to this branch. Mark the current hypothesis as `[-]` (in-progress). Then plan 2-4 initial to-dos via `hypothesis_plan update_todo`. Each to-do item must be either edit or test action. These initial to-dos reflect your best current plan.
– To-do-based implementation with checkpointing: Execute one to-do at a time for the current hypothesis: (1) mark the current to-do as `[-]` (in-progress), (2) perform the action (edit or test), (3) checkpoint with `hypothesis_git commit_todo` (provide descriptive information, e.g., test results, in the commit message), and (4) mark the current to-do as `[v]` (successful) or `[!]` (failed). Enforce "one to-do = one commit".
– Dynamic to-do expansion when needed: Expand the to-do list for the active hypothesis only when implementation feedback indicates missing steps or uncovered edge cases without contradicting the hypothesis's core fix strategy.
– Hypothesis exploration and completion: After finishing all to-dos for a hypothesis, validate thoroughly and mark it as [v] (successful) or [!] (failed). Log actionable insights whenever they arise. If insights suggest a partially correct hypothesis (i.e., earlier steps remain useful but later direction is wrong), revert to the appropriate prior to-do checkpoint (via `hypothesis_git revert_to`), create a new branch, and continue exploration under the new hypothesis branch. Repeat until all hypotheses have been implemented and evaluated—do not stop early even if one appears to work. If none succeeds, formulate new hypotheses and keep exploration.

Phase 3: Solution Finalization

– Hypothesis comparison: Invoke `hypothesis_git compare_hypotheses` to review the aggregated history (status, to-dos, commits, insights, and code diffs) of hypotheses, then select the best solution.
– Solution integration: Merge/apply the selected branch's changes to the original code state to produce a patch-ready state for submission.

Instance Template:

`{{repo_name}}` *# Replaced with the repository name*

`{{issue_description}}` *# Replaced with the issue description*

`{{code_location_hints}}` *# Replaced with the ranked locations*

Next Action Template:

`{{observation}}` *# Replaced with the latest tool output*

Based on this observation, decide your next action.

Submission Template:

Here is a list of all your changes:

`{{code_diff}}` *# Replaced with the code diffs*

1. Remove your generated reproduction/test script.
2. If you have modified any original test files, restore them to the initial state.
3. Finally, run the `submit` command.

Figure 7: System prompt and stepwise instruction template for **Issue Resolution Agent**.