Evaluating Agent Frameworks for Code-centric Software Engineering: A Comprehensive Empirical Study

Zhuowen Yin[a,b,1], Cuifeng Gao[c,1], Chunsong Fan[a], Wenzhang Yang[a,d,*] and Yinxing Xue[a,*]

[a]Institute of AI for Industries, Chinese Academy of Sciences, 168 Tianquan Road, Nanjing, 211135, Jiangsu, China
[b]Northeastern University, 96 Jinzhai Road, Shenyang, 110819, Liaoning, China
[c]Lingnan University, 8 Castle Peak Road, Tuen Mun, Hong Kong, China
[d]Jiangsu Key Laboratory of AI for Industries, Institute of AI for Industries, Chinese Academy of Sciences,

ARTICLE INFO

Keywords:
Agent framework
Software development
Vulnerability detection
Program repair

ABSTRACT

The rapid advancement of large language models (LLMs) has led to intelligent agents that reason, act, and observe in dynamic environments. Unlike static LLM-based systems, agents combine decision-making and tool use to address complex software engineering tasks. However, existing studies often focus on specific tasks or isolated aspects, leaving agents' practical capabilities insufficiently characterized.

We conduct an empirical study of seven general-purpose agent frameworks across three code-centric tasks: software development, vulnerability detection, and program repair. Each task is evaluated using widely adopted benchmarks. We analyze agent performance from three complementary perspectives: effectiveness, execution efficiency, and token-consumption overhead.

The results reveal distinct capability patterns and trade-offs. Agents achieve moderate overall effectiveness: OpenHands shows balanced results in software development, GPTSwarm attains the highest detection accuracy in vulnerability analysis, and only a subset of agents repairs about half of the issues in program repair. For efficiency, AgentOrchestra tends to produce the longest trajectories and the most correction attempts because of coordination overhead, whereas OpenHands shows stronger reflective reasoning. For overhead, program repair incurs the highest mean monetary cost per instance; OpenHands consumes the most tokens but benefits from input caching, while GPTSwarm is the most cost-efficient. We further examine how model selection, reflection mechanisms, and tool usage influence agent effectiveness and efficiency. These findings provide practical guidance for selecting and improving agent frameworks for software engineering. The replication package is available on GitHub and archived on Zenodo.

1. INTRODUCTION

With the rapid advancement of large language models (LLMs), their level of intelligence has grown remarkably, driving a new wave of innovation in intelligent agents. Agents are not simply more complex LLMs; rather, the ability to use tools—a defining feature that distinguishes humans from animals—also distinguishes agents from plain LLMs Google (2025). By leveraging external tools, agents can not only execute actions but also make decisions and manage workflows, thereby enabling them to handle more intricate tasks. The Reason–Act–Observe paradigm has emerged as a cutting-edge technique toward achieving artificial general intelligence (AGI) Gao, Geng, Hua, Hu, Juan, Liu, Liu, Qiu, Qi, Wu et al. (2025a).

Recently, increasingly elaborate and sophisticated agent frameworks have demonstrated tremendous potential in the software engineering domain Luo, Zhang, Yuan et al. (2025). They are being adopted throughout the software development life cycle Liu, Wang, Chen, Peng, Chen, Zhang and Lou (2024), particularly in code-centric software engineering tasks such as software development Qian, Liu, Liu, Chen, Dang, Li, Yang, Chen, Su, Cong, Xu, Li, Liu and Sun (2024), vulnerability detection Wei, Sun, Sun, Liu, Wu, Zhang, Zhang, Li, Liu, Li et al. (2025), and program repair Pabba, Mathai, Chakraborty and Ray (2025). Despite their widespread application, their practical capabilities in these tasks remain largely unclear to researchers and practitioners.

With the growing interest in agents, several empirical studies have begun to investigate the behavior and performance of agents from different perspectives. For instance, Gao *et al.* Gao, Li, Liu, Yu, Wang, Lin and Lai (2025b) compared the performance of multi-agent and single-agent systems across diverse domains beyond software engineering. In the context of automated program repair, Ceka *et al.* Ceka, Pujar, Ramji, Buratti, Kaiser and Ray (2025) conducted an in-depth investigation into how agent decision-making workflows affect performance, while Meng *et al.* Meng, Ma, Gao and Peng (2025) performed fine-grained experiments to assess the effectiveness of agents. However, despite these contributions, existing studies remain insufficient for a comprehensive evaluation of agent capabilities in software engineering due to two main limitations: (1) they focus on specific tasks, predominantly automated program repair; and (2) they emphasize isolated aspects, such as traceability or effectiveness, rather than providing a holistic assessment.

*Corresponding authors.
Email addresses: yinzhuowen@stumail.neu.edu.cn (Z. Yin); cuifenggao@ln.edu.hk (C. Gao); fanchunsong25@mails.ucas.ac.cn (C. Fan); wzhyang@iaii.ac.cn (W. Yang); yxxue@ustc.edu.cn (Y. Xue)
ORCID(s): 0009-0005-4063-2509 (Z. Yin); 0000-0003-0672-1485 (C. Gao); 0009-0009-4142-599X (C. Fan); 0009-0006-7836-1246 (W. Yang); 0000-0002-2979-7151 (Y. Xue)
[1]Zhuowen Yin and Cuifeng Gao contributed equally to this work.

To address these limitations, we conduct a comprehensive empirical study to systematically evaluate the practical performance of state-of-the-art agent frameworks across diverse code-centric software engineering tasks. Specifically, to ensure fair and consistent comparisons across multiple tasks, we focus on general-purpose agent frameworks that are designed to be adaptable across domains, rather than task-specific systems optimized for a single purpose. In total, seven representative frameworks are selected as the evaluation subjects. In addition, given the absence of a unified benchmark applicable to all code-centric tasks, we adopt widely used, high-quality benchmarks for each individual task: SRDD Qian et al. (2024) benchmark for software development task, LLM-SmartAudit benchmark Wei et al. (2025) for vulnerability detection task, and SWE-bench Lite SWE-bench Team (2024) benchmark for program repair task.

To systematically evaluate agent capabilities, we note that their workflows differ fundamentally from both LLM-based and traditional code-based tools. Unlike pre-trained systems, agents make autonomous and dynamic decisions during task execution. Hence, it is essential to assess not only what they achieve (functional effectiveness) but also how they achieve it (process efficiency). Moreover, the complex reasoning and decision-making inherent in agent execution often result in substantially higher token consumption compared to single-pass LLMs. Thus, we further investigate the overhead of task completion, including how token expenditure is distributed across reasoning and execution steps. Evaluating both the outcomes and the processes provides a holistic understanding of an agent framework's true level of intelligence. Consequently, our study is guided by three complementary research questions addressing the *effectiveness*, *efficiency*, and *overhead* of agent frameworks.

We investigate the following research questions and summarize the corresponding findings below.

RQ1 [Effectiveness]: How effectively do agent frameworks complete code-centric software engineering tasks?

Agents demonstrate moderate performance across all tasks. OpenHands achieves the most balanced quality in software development, while GPTSwarm leads in vulnerability detection with a 78% accuracy. In program repair, only a subset of agents successfully repair about half of the issues, leaving substantial room for improvement. We observe clear effectiveness trade-offs and underscore the ongoing challenges in fully automating complex, code-centric software engineering tasks.

RQ2 [Efficiency]: How efficiently do agent frameworks utilize reasoning cycles and external tools to complete code-centric software engineering tasks?

The execution efficiency varies across frameworks and tasks. In our evaluation, the complete SE-Agent workflow comprises three successive iterations. Its cumulative task-execution steps are obtained by summing the steps across Iter-1, Iter-2, and Iter-3, rather than considering Iter-3 alone. For program repair, the sum of its reported per-iteration average trajectory steps is 188.00, exceeding the single-run averages of the other evaluated frameworks. At the individual-run level, the longest average trajectories are recorded by OpenHands for software development, AgentOrchestra for vulnerability detection, and Trae for program repair. AgentOrchestra exhibits the longest correction sequences, whereas OpenHands ranks second, demonstrating stable and convergent behavior. AgentOrchestra exhibits high correction rates across tasks, whereas OpenHands achieves the highest software-development Quality of 0.47. These results indicate that more correction attempts do not necessarily translate into better outcomes. Overall, efficiency should be assessed by considering how a workflow uses its execution steps and feedback alongside the quality of its outcomes.

RQ3 [Overhead]: What is the token cost incurred by agent frameworks in completing code-centric software engineering tasks?

Program repair has the highest mean monetary cost per instance, with OpenHands incurring the highest overall monetary cost. Token consumption patterns vary significantly across frameworks. For instance, OpenHands uses the most tokens overall but benefits from caching, while GPTSwarm uniquely produces more output than input tokens. Breakdown analyses further reveal that planning and reflection stages dominate multi-agent workflows, in contrast to single-agent systems that concentrate their costs on execution and editing stages.

Ablation Studies: We further perform ablation studies to investigate the key factors influencing the performance of agent frameworks. Specifically, we examine three dimensions: the effect of different foundational models across various agent frameworks, the role of reflection mechanisms, and the impact of tool usage. These ablation studies provide deeper insights into the underlying reasons behind the effectiveness and efficiency of agent frameworks.

Additionally, we explore the relationship between effectiveness and efficiency from the perspectives of multi-agent and single-agent frameworks, analyzing the underlying factors driving these dynamics. Overall, this study provides actionable insights for both researchers and practitioners. For framework developers, our findings highlight opportunities to optimize agent architecture design, reasoning coordination, and token usage efficiency. For end users, the results offer practical guidance in selecting agent frameworks that best balance performance, efficiency, and cost according to their specific application needs.

Contributions. Our contributions are summarized as follows:

- We provide the first comprehensive comparison of seven general-purpose agent frameworks across three representative code-centric software engineering tasks.
- We systematically assess these agent frameworks across each task from three holistic perspectives: effectiveness, efficiency, and overhead.

- We present new empirical findings to guide practical adoption and future research. All experimental results are publicly available on GitHub and archived on Zenodo.

2. Background and Preliminaries

2.1. LLM-based Agents

The term "agent" is used with varying meanings across different communities Anthropic (2024). In some contexts, it refers to highly autonomous systems that operate over long time spans and employ multiple tools to solve complex tasks. In others, the same term may describe a more prescriptive workflow wrapped around an LLM, offering limited autonomy and decision-making capability. Such inconsistent usage often blurs the line between general LLM automation and true agent architectures Luo et al. (2025), making it difficult to evaluate and compare existing work.

Definition. To clearly establish the scope of this study, we adopt the definition of LLM-based agents from Google's recent Agent White Paper Google (2025).

> *"This combination of reasoning, logic, and access to external information that are all connected to a generative AI model invokes the concept of an agent."*

According to the white paper, an agent is viewed as a system driven by a cognitive architecture that governs its behavior, interaction, and decision-making. This architecture consists of three essential components: (1) *Model*, which serves as the central decision-making engine, typically implemented using one or multiple instruction-following language models; (2) *Tools*, which empower the agent to interact with external data and services beyond pure text generation; and (3) *Orchestration*, which manages the iterative perception–reasoning–action loop, including planning, tool invocation, action execution, and reflection until the task goal is reached. In this work, we therefore focus on agent frameworks that embody these components.

Categories. Based on the scale of collaboration, agent systems can be divided into single-agent and multi-agent paradigms Gao et al. (2025b). Single-agent systems employ a centralized architecture in which one agent independently performs perception, reasoning, and decision-making. Their simplicity and low communication overhead make them suitable for applications such as personal assistants and basic chatbots. However, a single agent can easily become a bottleneck under high concurrency, and its generalization capability is fundamentally constrained by the capacity of the underlying LLM. In contrast, multi-agent systems distribute work across multiple agents, each specializing in specific tasks to improve performance through division of labor, parallel execution, and dynamic coordination. Expanding the number of agents can further enhance system adaptability. Depending on interaction dynamics, multi-agent systems can be cooperative, competitive, or hybrid. Nevertheless, they introduce considerable coordination complexity and high resource consumption. In this paper, we investigate both single-agent and multi-agent systems to ensure a comprehensive and unbiased analysis of LLM-based agent frameworks.

2.2. Code-centric Software Engineering Tasks

The software engineering life-cycle typically comprises several sequential and iterative phases, including requirements analysis, design, implementation or development, testing and verification, deployment, and maintenance or evolution. Each phase involves distinct objectives and artifacts, ranging from specifying system requirements and designing architecture to implementing, verifying, and maintaining the software. In this study, we focus on the code-centric phases of the life-cycle, particularly those that involve direct manipulation of source code, such as *code development*, *vulnerability detection*, and *program repair*. These tasks serve as representative scenarios for evaluating the capability and adaptability of agent frameworks in software engineering contexts.

Software Development Software development is the systematic application of engineering principles, methods, and tools to the production and evolution of software Basili (1989). Its primary goal is to transform stakeholder requirements into operational, high-quality software. The process is conventionally structured by a software development life cycle that consists of interdependent phases including requirements elicitation and analysis, architectural design, implementation, verification and validation and deployment Aggarwal (2005). With the advent of generative AI, large language models have shown considerable potential to transform development workflows JACKSON, VASILESCU, RUSSO, RALPH, PRIKLADNICKI, IZADI, D'ANGELO, INMAN, ANDRADE and VAN DER HOEK (2025). Agents aim to extend these capabilities further by supporting end-to-end coverage of the development life cycle. A high-quality solution generated by an agent should satisfy the stated requirements, incur low execution or generation cost, require minimal manual refinement and scale seamlessly to larger codebases Luo et al. (2025).

Vulnerability Detection. Vulnerability detection is a critical discipline within software security, focused on the systematic identification of security-critical flaws in software systems Bessey, Block, Chelf, Chou, Fulton, Hallem, Henri-Gros, Kamsky, McPeak and Engler (2010). Its primary objective is to proactively uncover and remediate weaknesses before they can be exploited by malicious actors, thereby safeguarding the confidentiality, integrity, and availability of the system. Recent advances in artificial intelligence have significantly accelerated research in this area Chakraborty, Krishna, Ding and Ray (2021). Agent-based approaches further enhance detection by combining multi-agent co-inspection, integration of additional knowledge from tool executions, and traditional static analysis techniques Luo et al. (2025), aiming to improve accuracy, diversify the types of detected vulnerabilities, and enhance overall performance.

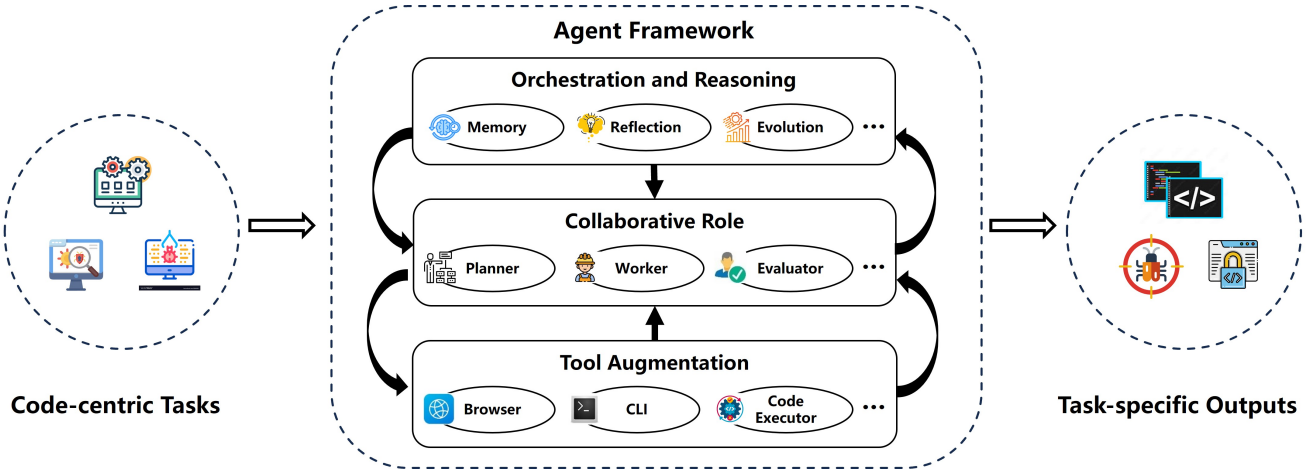


Figure 1: Agentic workflow paradigm.

Program Repair Program Repair, often referred to as Automated Program Repair (APR), is a subfield of Software Engineering that aims to automatically correct software faults Le Goues, Pradel and Roychoudhury (2019). Its primary objective is to mitigate the substantial manual effort, time, and cost associated with the debugging and maintenance phases. The typical APR pipeline involves fault localization, patch synthesis, and patch validation Liu, Koyuncu, Kim and Bissyandé (2019). Learning-based APR treats program repair either as a translation task, in which buggy code is converted to correct code, or as a generation task, where the correct code is infilled within the buggy code context Zhang, Fang, Ma, Sun and Chen (2023). LLM-based agents generally follow an iterative paradigm, refining patches based on model or tool feedback until correctness is achieved, effectively balancing scope and accuracy Luo et al. (2025).

3. Methodology

3.1. Research Questions

To derive our research questions, we first abstract a generalized agentic workflow paradigm that characterizes how agent frameworks operate in code-centric software engineering tasks, as illustrated in Figure 1. This paradigm adopts the cognitive architecture proposed in Google's Agent White Paper Google (2025), encompassing the model, tool, and orchestration components introduced in § 2.1. The agentic workflow comprises three conceptual layers that interact synergistically to enable adaptive, goal-driven task execution, as follows:

Orchestration and Reasoning. This layer governs high-level decision-making and adaptive reasoning processes. It is responsible for dynamic task planning, self-reflection, and continuous evolution of strategies based on intermediate feedback. It typically incorporates memory mechanisms to retain contextual knowledge across iterations, enabling long-term consistency and informed reasoning. Reflection modules allow agents to analyze prior failures or inefficiencies, while evolution components iteratively refine prompts, goals, or workflows to enhance overall performance.

Collaborative Role This layer defines specialized agent roles and their interactions within a cooperative framework. Common configurations include a planner that decomposes complex tasks into executable subtasks, workers that perform targeted actions or reasoning steps, and evaluators that assess the quality or correctness of intermediate outputs. This division of labor enables structured coordination and feedback loops among agents, thereby improving robustness and accountability in multi-agent collaboration.

Tool Augmentation This layer extends the agent's capabilities through external tools. These tools include browsers for retrieving up-to-date information from the web, command-line interfaces (CLI) for system-level operations, and code executors for testing or debugging generated code. Such augmentations bridge the gap between purely language-based reasoning and real-world execution, supporting more practical and context-aware problem solving.

These layers form an integrated ecosystem with rich interdependencies. The most distinctive feature of agent execution is its dynamic, adaptive intelligence, which differentiates agents from other approaches such as traditional code-based tools (e.g., static and dynamic analysis techniques) with fixed programs and static rules, or LLM-based tools that rely on manually engineered mechanisms (e.g., diversity chain-of-thought paradigms, fine-tuning, or retrieval-augmented generation), where processes and models are largely fixed.

Consequently, a comprehensive and systematic evaluation of agent frameworks requires assessing three complementary aspects: the quality of their *function*, the characteristics of their *processes* underlying task execution, and the *overhead* associated with their token consumption. In terms of *function*, we assess whether agents can effectively complete target tasks, focusing on final outcomes, as in traditional automation tools. In terms of *process*, it is equally important to examine how agents achieve these goals, with

Table 1
Comparison of architectural components across the studied agent frameworks.

Agents	Year	URL	Multi Agent	Planner	Context Builder	Worker	Evolution	Tool Selection	Test-Time Scaling
AGENTORCHESTRA	2025	https://github.com/SkyworkAI/DeepResearchAgent	✔	✔	✔	✔	✘	✔	✘
OWL	2025	https://github.com/camel-ai/owl	✔	✔	✔	✔	✘	✘	✔
SE-AGENT	2025	https://github.com/JARVIS-Xs/SE-Agent	✘	✘	✔	✔	✔	✘	✔
TRAE	2025	https://github.com/bytedance/trae-agent	✘	✘	✘	✔	✔	✔	✔
GPTSWARM	2024	https://github.com/metauto-ai/GPTSwarm	✘	✘	✔	✔	✘	✘	✘
OPENHANDS	2024	https://github.com/All-Hands-AI/OpenHands	✘	✘	✔	✔	✘	✔	✔
SWE-AGENT	2024	https://github.com/SWE-agent/SWE-agent	✘	✘	✔	✔	✘	✔	✘

particular emphasis on the efficiency and dynamics of their execution. This procedural aspect serves as a key indicator of the core intelligence level of the frameworks. In terms of *overhead*, it is not sufficient to measure only the total resources expended to complete a task; we must also analyze how resources are consumed across different stages of the reasoning process. Such a detailed examination provides deeper insights into the operational characteristics of agents and the significance of their execution strategies.

Based on this analysis, we formulate the following research questions:

- **RQ1 [Effectiveness]:** How effectively do agent frameworks complete code-centric software engineering tasks?
- **RQ2 [Efficiency]:** How efficiently do agent frameworks utilize reasoning cycles and external tools to complete code-centric software engineering tasks?
- **RQ3 [Overhead]:** What is the token cost incurred by agent frameworks in completing code-centric software engineering tasks?

3.2. Studied Agent Frameworks

To comprehensively collect state-of-the-art agent frameworks that support code-centric software engineering tasks, we began with the survey by Liu *et al.* Liu et al. (2024), which provides a comprehensive collection of research papers on LLM-based agents for software engineering. Note that this survey also includes LLM-based works that fall outside the scope of agents as defined in § 2.1. We thus carefully identified the relevant agent frameworks that meet our target criteria.

We conducted this work in September 2025, so the candidate agent frameworks were limited to those released or publicly available before September 2025. In addition, to avoid repeating prior findings and to highlight the incremental contribution of our study, we excluded agents that had already been systematically evaluated in previous empirical studies. For framework collection, we first searched Google Scholar using combinations of keywords centered on agent framework. After gathering a sufficiently large pool of candidates, we performed manual screening and deduplication based on the goals of this study, which target diverse code-centric tasks and emphasize generality and comparability. We then retained only general-purpose open-source frameworks that met the following inclusion criteria for experimental evaluation.

C1 The agent framework can be applied to all three tasks, namely software development, vulnerability detection, and program repair. This criterion excludes task-specific agents that are tailored to a single scenario, such as those exclusively designed for vulnerability repair or code generation.

C2 The agent framework is publicly available. This criterion excludes proprietary or closed-source frameworks that are not accessible for independent evaluation.

In total, as shown in Table 1, we identified 7 that satisfy the inclusion criteria and summarizes the architectural components adopted by these frameworks.

AGENTORCHESTRA Zhang, Zeng, Xiao, Li, Cui, Zhao, Hu, Liu, Zhou and An (2025) created by researchers at Skywork AI and Nanyang Technological University. It is a multi-agent framework that supports unified multimodal tool integration, flexible component composition, and efficient hierarchical collaboration. Through dynamic tool creation, intelligent retrieval, and automatic reuse.

OWL Hu, Zhou, Fan, Nie, Xia, Sun, Ye, Jin, Li, Chen, Zhang, Wang, Ye, Ghanem, Luo and Li (2025) developed by researchers from The University of Hong Kong and the CAMEL team, OWL is a hierarchical multi-agent framework. A Planner Agent decomposes tasks, a Coordinator Agent manages subtasks, and Worker Agents invoke domain-specific tools to execute work. Adding or modifying Worker Agents improves overall generalization during inference. Furthermore, OWL employs reinforcement learning with real-world feedback to optimize a domain-agnostic Planner Agent, thereby enhancing cross-domain generalization.

SE-AGENT Lin, Guo, Han, Hu, Ni, Wang, Chen, Liu, Chen, He, Jiang, Jiao, Hu and Wang (2025) jointly developed by Tsinghua University and StepFun team. It is a self-evolving, multi-step reasoning agent. SE-AGENT first constructs an initial trajectory pool by injecting five distinct reasoning strategies and controlled trajectory mutations. The agent then performs reflection and revision to identify issues, followed by cross-trajectory fusion via Crossover, Transfer,

and Restructuring. Finally, it conducts multi-dimensional trajectory selection to produce robust solutions.

TRAE Team, Gao, Tian, Meng, Wang, Hu, Xiao, Liu, Zhang, Chen, Gao, Lin, Xiong, Peng and Liu (2025) built by engineers at ByteDance for code engineering. It orchestrates a Code Agent that integrates multiple tools to iteratively generate candidate patches across prescribed steps. It then applies a two-stage pruning process—patch deduplication and regression testing—to reduce the search complexity for the optimal patch. To select candidates, a Selector Agent with repository-level understanding ranks solutions, and a voting mechanism determines the final patch.

GPTSWARM Zhuge, Wang, Kirsch, Faccio, Khizbullin and Schmidhuber (2024) proposed by researchers from KAUST and the Swiss AI Lab IDSIA. It models agents as computation graphs: nodes are functions that process multimodal data or query LLMs, edges represent interaction flows, and graphs can be recursively composed into larger hierarchies to capture inter-agent collaboration structures. GPTSWARM introduces novel graph-level optimization operators: it refines node-level LLM prompts and adjusts graph connectivity to improve agent orchestration.

OPENHANDS Wang, Li, Song, Xu, Tang, Zhuge, Pan, Song, Li, Singh, Tran, Li, Ma, Zheng, Qian, Shao, Muennighoff, Zhang, Hui, Lin, Brennan, Peng, Ji and Neubig (2025) is an open-source platform co-developed by academic and industry contributors to provide a unified, powerful, and safe environment for building and evaluating general-purpose "AI software developer" agents. 1) It centers on an event-stream interaction mechanism that precisely logs each agent action and environment observation, forming complete task trajectories. 2) It bundles a ready-to-use Python tool library that encapsulates common but complex operations—file editing, multimodal parsing (e.g., PDFs, images), and code search—into simple function calls. 3) It pioneered a multi-agent delegation mechanism that allows a generalist agent (e.g., CodeAct Agent) to outsource specialized subtasks to expert agents (e.g., a Browsing Agent), enabling capability complementarity for complex composite tasks. As a result, OPENHANDS supports diverse real-world scenarios, such as automated software development and maintenance, intelligent data analysis, efficient research assistance, and personal automation workflows.

SWE-AGENT (Yang et al., 2024) was developed by researchers at Princeton University. It is a single-agent framework tailored to software engineering. SWE-AGENT augments an agent's ability to create and modify files, navigate codebases, and run tests via an agent–computer interface (ACI). Its design mirrors human use of IDEs: (1) manual inspection of agent behaviors to identify failure modes and suggest improvements; and (2) grid search to select optimal ACI configurations. Together, these mechanisms strengthen the agent's capacity to tackle complex software engineering tasks.

3.3. Benchmark Suite

To ensure a rigorous and unbiased evaluation of the studied agent frameworks, we carefully curate a benchmark suite from publicly available software engineering benchmarks. Although agents are built on LLMs, LLM evaluation datasets and agent evaluation datasets differ fundamentally in their objectives, task granularity, and required interactions: the former measure intrinsic text understanding and generation abilities of LLMs, while the latter assess end-to-end task execution and autonomy in real workflows. Accordingly, agent evaluation typically incurs substantially higher costs than LLM evaluation. Therefore, we primarily consider the benchmarks used in prior software-engineering agent research, as catalogued in the survey by Liu *et al.* Liu et al. (2024). The selection was guided by the principles of broad community adoption, comprehensive task coverage, appropriate dataset scale, data reliability, and reproducibility to support consistent assessment and verification.

As summarized in Table 2, the final benchmark suite includes three benchmarks that span software development, vulnerability detection, and program repair tasks. In the following, we present the details of each benchmark and discuss the rationale behind its selection.

SRDD Qian et al. (2024) for Software Development. To evaluate existing LLM-based agents for end-to-end software development, as reported by Liu *et al.* Liu et al. (2024), many existing studies adopt classic code generation benchmarks such as HumanEval Chen, Tworek, Jun, Yuan, Pinto, Kaplan, Edwards, Burda, Joseph, Brockman et al. (2021) and MBPP Austin, Odena, Nye, Bosma, Michalewski, Dohan, Jiang, Cai, Terry, Le et al. (2021). However, these benchmarks mainly involve simplified programming tasks on isolated pieces of code, which do not accurately represent practical development environments. To address this limitation, more advanced benchmarks have emerged, including SRDD Qian et al. (2024) and ProjectDev Nguyen, Chau, Nguyen and Bui (2025), which support development tasks across multiple files and demonstrate increasing popularity in recent works.

Among these benchmarks, Software requirement description dataset (SRDD) has become the most frequently adopted one, since it contains the largest and most diverse collection of software development tasks with 1,200 curated prompts. Therefore, in this study, we select SRDD to evaluate agents on software development tasks. SRDD consists of 1,200 natural language prompts, organized into five major categories and 40 subcategories. The prompts were collected from ChatGPT 3.5 and thoroughly curated to ensure high task quality and relevance to realistic development scenarios.

LLM-SmartAudit Wei et al. (2025) for Vulnerability Detection. Although many benchmarks have been used to evaluate LLM-based approaches for static bug detection, as reported by Liu *et al.* Liu et al. (2024), none are specifically curated for evaluating agent frameworks. In our investigation of recent works, LLM-SmartAudit is the only one we identified that applies agents for vulnerability detection tasks.

Table 2
The benchmark suite for code-centric software engineering tasks.

Task	Benchmark Dataset	Year	URL	Instances
Software Development	SRDD Qian et al. (2024)	2023	https://github.com/OpenBMB/ChatDev/tree/main/SRDD	1,200
Vulnerability Detection	LLM-SmartAudit Wei et al. (2025)	2025	https://github.com/LLMAudit/LLMSmartAuditTool	115
Program Repair	SWEBench-Lite SWE-bench Team (2024)	2024	https://www.swebench.com/lite.html	300

Therefore, we select the dataset used by LLM-SmartAudit in this study to enable a fair and standardized comparison of different agent-based approaches under consistent evaluation conditions.

LLM-SmartAudit consists of three subset datasets: a common-vulnerability set, a real-world set, and a CVE set. Among them, only the common-vulnerability set and the CVE set provide standardized vulnerability type labels. In contrast, the real-world set is categorized based on their nature rather than their exact labels. Therefore, our evaluation focuses on the two well-labeled subsets, comprising 102 contracts from the common-vulnerability set and 13 contracts from the CVE set. In total, we analyze 115 vulnerability instances across these datasets.

SWEBench-Lite SWE-bench Team (2024) for Program Repair. For the task of program repair, many LLM-based approaches Liu et al. (2024) have been evaluated on classical benchmarks such as Defects4J Just, Jalali and Ernst (2014) and QuixBugs Lin, Koppel, Chen and Solar-Lezama (2017), which mainly target small-scale defects. In contrast, most agent-based works Pabba et al. (2025); Mu, Wang, Shi, Wang, Li and Wang (2025); Yu, Zhang, Zhao, Huang, Yao, Ding and Zhao (2025) are consistently evaluated on the derivatives of the SWE-bench benchmark Jimenez, Yang, Wettig, Yao, Pei, Press and Narasimhan (2023) (e.g., SWE-bench Lite SWE-bench Team (2024), SWE-bench Lite-S Xia, Deng, Dunn and Zhang (2024), etc.), because it is grounded in real-world development scenarios by constructing tasks directly from GitHub issues and corresponding codebases, thereby ensuring practical relevance to authentic software maintenance workflows.

SWE-bench Jimenez et al. (2023) is a comprehensive benchmark, comprising over 2,000 real-world GitHub issues drawn from 12 widely-used Python repositories. Each task includes the original issue description, the full code repository, a predefined execution environment, and associated validation tests. However, evaluating the full SWE-bench benchmark incurs substantial computational costs and includes particularly challenging or problematic tasks Xia et al. (2024), which may lead to underestimation of the practical capability of LLM-based agents.

To address these limitations, researchers have manually curated high-quality tasks that balance difficulty, provide self-contained information, include informative issue descriptions, and offer sufficient evaluation tests, which constitutes SWE-bench Lite SWE-bench Team (2024). Accordingly, we adopt SWE-bench Lite in this study to evaluate LLM-based agents on program repair tasks. This dataset comprises 300 carefully selected repair instances, covering all of the original 12 repositories while preserving the diversity and difficulty distribution of the full benchmark.

3.4. Evaluation Framework

To ensure a fair and systematic evaluation across different agent frameworks, we design a unified evaluation framework that adopts tailored assessment metrics for each research question and standardizes the experimental setup for all agents.

Evaluation Metrics. As shown in Table 3, we present the evaluation metrics corresponding to each research question. For effectiveness (RQ1), we empirically evaluate outputs produced by the agent frameworks through benchmark-specific checks, execution tests, and comparisons against reference annotations or task requirements. The evaluation procedures are as follows:

(1) For software development on SRDD, we follow the ChatDev evaluation protocol Qian et al. (2024) and compute code quality as the product of completeness, executability, and consistency. Completeness checks for placeholder markers (e.g., *TODO* comments), while executability checks whether the project's entry points run without exceptions or timeouts; both are binary metrics. Consistency measures semantic alignment between the task description and generated code using the cosine similarity of their embeddings from OpenAI's `text-embedding-ada-002` model.

(2) For vulnerability detection on LLM-SmartAudit Wei et al. (2025), each of the 115 evaluated smart contracts has one annotated vulnerability type, and each agent framework predicts one type per contract. A prediction is correct if it matches the annotated type. We compute vulnerability-type classification accuracy as $N_{\text{correct}}/115$, where N_{correct} is the number of correctly classified contracts. Under micro-averaging, each incorrect prediction contributes one false positive for the predicted type and one false negative for the annotated type.

(3) For program repair on SWE-bench Lite, we evaluate patches produced by the agent frameworks using the official Docker-based test protocol. Patches are standardized into the required JSON format, and a repair is considered successful if the patch passes the official test suite. Repair rate is the proportion of evaluated issues successfully repaired.

Efficiency (RQ2) is evaluated empirically using execution trajectories recorded during actual benchmark runs. We extract trajectory lengths and correction counts from these records to compute average trajectory steps, average correction attempts, and correction rate. Compared with absolute time, average trajectory steps can avoid statistical

Table 3
Evaluation metrics for each research question.

Research Question	Task	Metrics
Effectiveness (RQ1)	Software Development	$Quality = Completeness \times Executability \times Consistency$ Qian et al. (2024)
	Vulnerability Detection	Vulnerability-Type Classification Accuracy = $N_{\text{correct}}/115$
	Program Repair	$Repair_Rate = (Correctly_Fixed_Issues)/(Total_Issues)$ Jimenez et al. (2023)
Efficiency (RQ2)	All Tasks	Average Trajectory Steps, Correction Attempts, and Correction Rate
Overhead (RQ3)	All tasks	Monetary Cost, Token Usage

errors caused by network fluctuations and differences in host-machine performance. Correction Attempts refers to a self-correction episode in which an agent explicitly recognizes a deficiency in its current solution within the trajectory and then takes a concrete remediation action (e.g., rollback or modifying code, tests, or configuration) within the subsequent four steps. These metrics quantify how efficiently each agent framework completes the assigned tasks, capturing the dynamic decision-making behaviors intrinsic to intelligent agents. Overhead (RQ3) measures the monetary cost incurred during task completion. We also record token consumption—including both input and output tokens—and analyze how these tokens are distributed across different stages of the agent workflow. This detailed breakdown offers insights into the cost-efficiency trade-offs among the evaluated frameworks.

Monetary cost is calculated based on input, cached-input, and output token usage as follows:

$$C = \frac{(T_{\text{in}} - T_{\text{cache}})p_{\text{in}} + T_{\text{cache}}p_{\text{cache}} + T_{\text{out}}p_{\text{out}}}{10^6}.$$

Here, T_{in} denotes the total input tokens, including cached tokens T_{cache}, while T_{out} denotes output tokens. The corresponding prices p_{in}, p_{cache}, and p_{out} are expressed in USD per million tokens.

Experimental Setup. For the seven general-purpose agent frameworks, our experimental setup is primarily characterized by several key parameters, including, *Step Limit*, *Agent Quantity*, *Tool Set*, *Prompt*, *Iteration* and *Backend LLM*. The detailed configurations for each parameter are described as follows:

- **Step Limit.** For all tasks and agent frameworks, the maximum step limit was set to 150 to ensure consistent termination conditions and to keep the monetary overhead within a controllable range.
- **Agent Quantity.** We adhere to the default configurations specified by each agent framework for executing code-related tasks. Notably, in AGENTORCHESTRA's setup, all four agents are utilized for the software development task, whereas the BrowserUse agent is excluded from vulnerability detection and program repair tasks. This exclusion is due to the code-centric nature of these tasks, which rely primarily on direct source code interaction and static analysis within the repository, rendering browser capabilities unnecessary.
- **Tool Set.** In the primary RQ1–RQ3 comparison, we use the default tool set provided by each framework to evaluate its out-of-the-box behavior. In the cross-model program repair analysis, all frameworks use the SWE-Agent tool paradigm Yang et al. (2024), which standardizes repository interaction and `diff` patch generation across foundation models. For software development and vulnerability detection, we retain the original configurations of each agent without modification.
- **Prompt.** For each task, we used a common task-level base prompt across the evaluated frameworks. The base prompt follows CHATDEV Qian et al. (2024) for software development, LLM-SmartAudit Wei et al. (2025) for vulnerability detection, and SWE-AGENT Yang et al. (2024) for program repair. Within a task, we kept the task objective, instance input, and expected deliverable substantially consistent, while adapting the prompt as needed to each framework's role configuration, tool interface, and input-output format. Prompt instructions guide agent behavior but do not mechanically enforce every requested step. We recorded the actual execution trajectories for process analysis and evaluated the final outputs using the task-specific benchmark protocols.
- **Iteration.** SE-AGENT is an intentional exception to the common task-level prompt configuration. Iter-1 uses its initial, detailed task prompt with the `null` summary operator. Iter-2 uses a more concise prompt together with `alternative_strategy`, which generates alternatives from recent failures. Iter-3 uses `traj_pool_summary` to analyze failures in the trajectory pool and identify blind spots and risks. We report the three iterations separately in RQ1–RQ3. For the complete SE-AGENT workflow, task-execution steps are accumulated across all three iterations. Their comparison therefore reflects the joint effects of prompt variation and trajectory-derived reflection, rather than the effect of prompt detail alone.
- **Backend LLM.** We separate the evaluation into a primary comparison and a cross-model generalizability analysis. In the primary comparison (RQ1–RQ3), all frameworks use DeepSeek-v3.1 DeepSeek-AI (2024)

Table 4
Comparison of software development effectiveness across different task classes and metrics on the SRDD benchmark.

Agent	Game			Education			Work			Life			Creation			Average			Quality
	Comp.	Exec.	Cons.	Comp.	Exec.	Cons.	Comp.	Exec.	Cons.	Comp.	Exec.	Cons.	Comp.	Exec.	Cons.	Comp.	Exec.	Cons.	
AGENTORCHESTRA	0.82	0.50	0.78	0.90	0.58	0.77	0.83	0.56	0.78	0.91	0.58	0.78	0.81	0.49	0.77	0.86	0.55	0.78	0.36
OWL	0.69	0.78	0.79	0.61	0.72	0.78	0.60	0.71	0.79	0.73	0.76	0.78	0.65	0.38	0.78	0.66	0.70	0.78	0.36
SE-AGENT (Iter-1)	0.72	0.91	0.71	0.77	0.91	0.71	0.67	0.93	0.71	0.74	0.96	0.72	0.64	0.65	0.71	0.71	0.90	0.71	0.46
SE-AGENT (Iter-2)	0.71	0.43	0.78	0.78	0.34	0.78	0.73	0.29	0.78	0.76	0.35	0.78	0.81	0.46	0.79	0.75	0.37	0.78	0.22
SE-AGENT (Iter-3)	0.73	0.23	0.79	0.79	0.25	0.79	0.71	0.18	0.79	0.80	0.29	0.79	0.76	0.34	0.79	0.76	0.26	0.79	0.15
TRAE	0.48	0.96	0.81	0.50	0.94	0.81	0.51	0.91	0.81	0.64	0.93	0.81	0.43	0.90	0.81	0.53	0.93	0.81	0.40
GPTSWARM	0.76	0.64	0.85	0.69	0.81	0.85	0.71	0.77	0.85	0.80	0.58	0.85	0.77	0.81	0.86	0.75	0.70	0.85	0.45
OPENHANDS	0.62	1.00	0.80	0.59	1.00	0.79	0.49	1.00	0.78	0.68	1.00	0.80	0.59	1.00	0.79	0.60	1.00	0.79	0.47
SWE-AGENT	0.53	0.84	0.78	0.64	0.84	0.77	0.58	0.90	0.78	0.61	0.94	0.80	0.47	0.85	0.79	0.57	0.87	0.78	0.39
Average	0.70	0.68	0.80	0.70	0.71	0.79	0.66	0.68	0.79	0.76	0.67	0.80	0.69	0.65	0.80	0.69	0.70	0.79	0.36

as a common backend, which controls the foundation-model variable when comparing framework configurations on the full benchmark suite. The model card reports a mixture-of-experts architecture with 671 billion total parameters and 37 billion activated parameters per token DeepSeek-AI (2025). We additionally evaluate `Qwen3-Coder-Flash`, `Qwen3-Coder-Plus`, and `GLM-4.7` on representative subsets to examine whether the qualitative conclusions of RQ1–RQ3 remain supported when the foundation model changes. This supplementary analysis is reported in Section 5.3.1. For the generation parameters, we set `temperature=1`, `top_k=1`, and `max_tokens` to the default context length of each model.

4. Results and Analysis

In this section, we provide a comprehensive analysis of the empirical findings, focusing on the effectiveness (RQ1), efficiency (RQ2), and overhead (RQ3) of the seven general-purpose agent frameworks evaluated in this study.

4.1. Effectiveness of Agent Frameworks (RQ1)

To address RQ1, we leverage the benchmark suite introduced in § 3.3, which covers three representative code-centric software engineering tasks: software development (SRDD), vulnerability detection (LLM-SmartAudit), and program repair (SWE-bench Lite). Across these tasks, we evaluate seven state-of-the-art agent frameworks, as presented in § 3.2.

Software Development. The results of our study on automated software development tasks are summarized in Table 4, which presents a comparative evaluation of seven distinct agent frameworks. The evaluation was conducted using the SRDD dataset, a comprehensive collection of 1,200 instances distributed across five categories: Education (210), Work (240), Life (330), Game (270), and Creation (150).

Our methodology for this assessment was as follows: (1) We benchmarked the seven agent frameworks against the 1,200 simulated software development problems from the SRDD dataset. (2) The generated software projects—comprising Python files, metadata text files, and requirements text files—were quantitatively evaluated for Completeness, Executability, and Consistency using the official evaluation script provided by CHATDEV Qian et al. (2024). (3) A category-by-category analysis was performed to identify the specific software development tasks in which each agent excels.

As illustrated in Table 4, the quality of the software produced by the agents exhibits significant disparities across the three metrics. A key finding is that no single agent framework is omnipotent; each demonstrates a unique profile of strengths and weaknesses. This provides empirical evidence of the current limitations of agent frameworks in code generation, thereby informing future work aimed at their improvement.

A detailed analysis of the metrics reveals the following. For completeness, AGENTORCHESTRA demonstrated the highest effectiveness, achieving an impressive average score of 0.86 across the five categories, significantly outperforming the collective average of 0.69. In contrast, TRAE registered the poorest effectiveness in completeness, with an average of 0.53. In terms of executability, the OPENHANDS framework proved exceptionally proficient, attaining perfect scores (1.00) in the Game, Education, Life, and Creation categories, and a near-perfect score of 1.00 in the Work category. This outstanding result stands in stark contrast to the overall average executability of 0.79 across all agents. For consistency, GPTSWARM emerged as the leader, with a superior average score of 0.85, surpassing the overall agent average of 0.79. Across these five categories, AGENTORCHESTRA demonstrated superior performance in completeness, while OPENHANDS excelled in executability, and GPTSWARM maintained a leading position in consistency.

To assess overall effectiveness, we utilized the quality score metric proposed by CHATDEV, calculated as $Quality = Completeness \times Executability \times Consistency$. Based on this composite metric, OPENHANDS is the top-performing framework for software development tasks, achieving a quality score of 0.47.

Despite the strong effectiveness of OPENHANDS, our findings suggest that its capabilities are not yet sufficient to

Table 5
Comparison of vulnerability detection effectiveness across different vulnerability types on the LLM-SmartAudit benchmark. In the Total row, Acc = $N_{\text{correct}}/115$, where N_{correct} is the number of correctly classified contracts.

Subset Dataset	Vulnerability Type (#Contracts)	AgentOrchestra	OWL	SE-Agent (Iter-1)	SE-Agent (Iter-2)	SE-Agent (Iter-3)	Trae	GPTswarm	OpenHands	SWE-Agent
Common-Vulnerability	IO (10)	0 (0%)	1 (10%)	7 (70%)	6 (60%)	9 (90%)	2 (20%)	9 (90%)	8 (80%)	7 (70%)
	RP (10)	10 (100%)	10 (100%)	10 (100%)	10 (100%)	10 (100%)	10 (100%)	10 (100%)	10 (100%)	9 (90%)
	GL (11)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (9%)	1 (9%)	2 (18%)	1 (9%)	0 (0%)
	RE (10)	7 (70%)	10 (100%)	10 (100%)	10 (100%)	10 (100%)	10 (100%)	10 (100%)	10 (100%)	10 (100%)
	TM (10)	1 (10%)	3 (30%)	4 (40%)	5 (50%)	4 (40%)	4 (40%)	6 (60%)	1 (10%)	6 (60%)
	TOD (10)	3 (30%)	9 (90%)	9 (90%)	7 (70%)	8 (80%)	9 (90%)	8 (80%)	4 (40%)	8 (80%)
	USE (10)	0 (0%)	2 (20%)	8 (80%)	8 (80%)	8 (80%)	2 (20%)	9 (90%)	3 (30%)	9 (90%)
	TX (10)	7 (70%)	7 (70%)	7 (70%)	7 (70%)	7 (70%)	6 (60%)	7 (70%)	6 (60%)	8 (80%)
	UD (10)	9 (90%)	8 (80%)	10 (100%)	10 (100%)	10 (100%)	10 (100%)	10 (100%)	10 (100%)	9 (90%)
	USU (11)	11 (100%)	10 (91%)	11 (100%)	10 (91%)	10 (91%)	10 (91%)	10 (91%)	6 (55%)	11 (100%)
CVE	Access Control (4)	0 (0%)	1 (25%)	2 (50%)	3 (75%)	2 (50%)	2 (50%)	4 (100%)	4 (100%)	4 (100%)
	Overflow (3)	1 (33%)	3 (100%)	3 (100%)	3 (100%)	3 (100%)	3 (100%)	3 (100%)	3 (100%)	3 (100%)
	Logic Error (4)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (25%)	1 (25%)
	Delegatecall (1)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)
	Bad Randomness (1)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	0 (0%)	1 (100%)
	Total (115)	**51** (44%)	**66** (57%)	**83** (72%)	**81** (70%)	**84** (73%)	**71** (62%)	**90** (78%)	**68** (59%)	**87** (76%)

serve as a universal replacement for all other agents. Each framework possesses distinct advantages, indicating that the landscape of agent-driven software development benefits from a diversity of specialized tools rather than a single, all-encompassing solution.

Vulnerability Detection. Table 5 shows the experimental results of seven agent frameworks on vulnerability detection task. The common-vulnerability subset dataset contains 10 types of vulnerabilities, specifically Reentrancy (RE), Integer Overflow/Underflow (IO), Unchecked send (USE), Unsafe Delegatecall (UD), Transaction Order Dependence (TOD), Time Manipulation (TM), Randomness Prediction (RP), Authorization Issue using 'tx.origin' (TX), Unsafe Suicide (USU), and Gas Limitation (GL). The CVE subset dataset contains 5 types of vulnerabilities, including Access Control, Overflow, Logic Error, Delegatecall, and Bad Randomness.

We adopt the following methods to answer this question: (1) We evaluate seven agent frameworks in our assessment on the 115 vulnerable smart contracts across different vulnerability types. (2) When the vulnerability detection report generated by the agent framework is compared with the ground truth of the labeled dataset, the detection accuracy rate for each vulnerability type can be calculated. (3) Through type-by-type analysis, we can observe which vulnerability detection tasks these agents perform well in.

Each vulnerability-type row in Table 5 reports the number of correctly classified contracts and the corresponding percentage among contracts annotated with that type. The Total row uses all 115 contracts as its denominator.

For the GL vulnerability type, all seven agents demonstrated very low detection efficiency, basically failing to discover any vulnerabilities. The percentages for RP and UD approached 100% for most frameworks. We observe that the seven agents demonstrate comparable performance in the vulnerability detection task, with GPTSWARM achieving the highest score of 78%, AGENTORCHESTRA recording the lowest at 44%, and SE-AGENT in Iter-2 representing the median at 70%. However, they have not broken through 90%, which indicates that the seven agent frameworks we evaluated still have room for improvement.

Program Repair. The SWEBench-Lite benchmark comprises 12 GitHub repositories with a total of 300 instances. Our methodology for the program repair task was as follows: (1) we evaluated the seven agent frameworks on the 300 real-world issues spanning the 12 repositories; (2) the agents generated patches in diff format, which we standardized into the official JSON schema; (3) we further stratified the 300 instances by repository type across the 12 GitHub projects—for example, `django` is a web development framework repository with 114 issues; and (4) by analyzing results repository-by-repository, we identified which issues were successfully repaired by each agent.

The detailed results for program repair are summarized in Table 6, which reports the number of successfully repaired issues per agent framework for each of the 12 repositories. A repair is considered successful if the diff-format patch generated by an agent passes the official test suite. In Table 6, each row corresponds to a repository, and each cell represents the number of issues successfully repaired by a specific agent within that repository.

Table 6 reveals substantial differences in accuracy across agents. Notably, SE-AGENT (Iter-3) achieved the highest accuracy, repairing 54% of the total issues, whereas the average success rate across all agents was only 36%. Additionally, for the `django` repository, which contains the largest number of issues (114), TRAE and SE-AGENT (Iter-3) achieved the best performance, each successfully repairing 68 issues. In contrast, for `flask` repository, which contains the fewest issues (3), none of the agents succeeded in repairing any issue, an unexpected outcome suggesting that the number of issues does not necessarily correlate with task difficulty. Note that

Table 6
Comparison of program repair effectiveness across different repositories on the SWE-bench Lite benchmark.

Repository (#Issues)	AgentOrchestra	OWL	SE-Agent (Iter-1)	SE-Agent (Iter-2)	SE-Agent (Iter-3)	Trae	GPTswarm	OpenHands	SWE-Agent
astropy (6)	0 (0%)	1 (17%)	2 (33%)	3 (50%)	2 (33%)	3 (50%)	0 (0%)	3 (50%)	2 (33%)
django (114)	7 (6%)	7 (6%)	59 (52%)	67 (59%)	68 (60%)	68 (60%)	14 (12%)	61 (54%)	66 (58%)
flask (3)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
matplotlib (23)	0 (0%)	2 (9%)	9 (39%)	14 (61%)	14 (61%)	11 (48%)	0 (0%)	11 (48%)	14 (61%)
seaborn (4)	0 (0%)	1 (25%)	3 (75%)	4 (100%)	3 (75%)	3 (75%)	0 (0%)	3 (75%)	4 (100%)
pylint (6)	0 (0%)	0 (0%)	2 (33%)	3 (50%)	2 (33%)	3 (50%)	0 (0%)	2 (33%)	3 (50%)
pytest (17)	0 (0%)	1 (6%)	6 (35%)	6 (35%)	6 (35%)	6 (35%)	0 (0%)	8 (47%)	5 (29%)
requests (6)	1 (17%)	2 (33%)	1 (17%)	1 (17%)	3 (50%)	3 (50%)	0 (0%)	0 (0%)	4 (67%)
scikit (23)	0 (0%)	1 (4%)	17 (74%)	16 (70%)	15 (65%)	14 (61%)	1 (4%)	16 (70%)	16 (70%)
sphinx (16)	1 (6%)	1 (6%)	7 (44%)	9 (56%)	9 (56%)	8 (50%)	0 (0%)	3 (19%)	9 (56%)
sympy (77)	0 (0%)	15 (19%)	34 (44%)	34 (44%)	37 (48%)	33 (43%)	0 (0%)	37 (48%)	34 (44%)
xarray (5)	0 (0%)	0 (0%)	2 (40%)	2 (40%)	2 (40%)	2 (40%)	0 (0%)	2 (40%)	2 (40%)
Total (300)	**10 (3%)**	**31 (10%)**	**142 (47%)**	**159 (53%)**	**161 (54%)**	**154 (51%)**	**15 (5%)**	**146 (49%)**	**159 (53%)**

AGENTORCHESTRA, OWL, and GPTSWARM exhibit distinctly poor effectiveness. The underlying reason is that they do not leverage Patch tooling Yang et al. (2024), which leads to failures in generating correct diff-format patches. This limitation reflects a broader challenge inherent to current LLM capabilities.

In total, SE-AGENT (Iter-2) and SWE-AGENT fully repaired (100%) only 1 out of the 12 repositories, namely seaborn, which contains only 4 issues. This indicates that agent frameworks are not universally effective for program repair. Although overall effectiveness was modest, these findings provide concrete evidence of the limitations of current agent frameworks in program repair and offer actionable insights for future improvements. The extent to which these effectiveness patterns recur with other foundation models is examined in Section 5.3.1.

Answer to RQ1. The key takeaways are as follows:

- **Software Development:** A clear effectiveness trade-off is observed among completeness, executability, and consistency. OPENHANDS achieves the best overall balance, with the highest quality score of 0.47.
- **Vulnerability Detection:** The average success rate across the seven agent frameworks is approximately 66%, with GPTSWARM achieving the highest detection accuracy of 78%. Nevertheless, this performance remains suboptimal and leaves considerable room for improvement.
- **Program Repair:** Effectiveness is modest, with at most roughly half of the issues repaired by only 4 out of 7 agents, while the rest achieve low repair rates, highlighting significant potential for improvement.

4.2. Efficiency of Agent Frameworks (RQ2)

To address RQ2, we analyze the efficiency of the seven agent frameworks in the same set of code-centric software engineering tasks introduced in § 4.1. Unlike RQ1, which focuses solely on final outcomes, RQ2 delves into the agents' execution processes, providing a more nuanced understanding of their operational behavior. In contrast to traditional software engineering automation tools, which execute predefined, statically encoded procedures, agents operate in a dynamic and adaptive manner. Consequently, assessing the efficiency of these processes is essential for gaining insights into the level of intelligence and adaptability demonstrated by each framework.

Software Development. Table 7 presents the execution trajectory analysis for the software development task, revealing distinct operational characteristics across different agent frameworks. Statistical analysis shows that seven frameworks exhibit an average trajectory length of 28.31 steps, with OPENHANDS recording the longest trajectory of 81.28 steps. Note that SE-AGENT exhibits a distinctive iterative pattern: the first iteration comprises 43.9 steps, while the second and third iterations show markedly shorter trajectories of 3.70 and 3.20 steps, respectively. This progressive reduction suggests that SE-AGENT employs an adaptive refinement strategy, in which an initial comprehensive exploration is followed by focused, targeted adjustments in subsequent iterations. Such a pattern reflects efficient convergence behavior, as the agent leverages insights from earlier iterations to streamline problem-solving in later stages.

Our analysis reveals that AGENTORCHESTRA exhibits the highest correction attempts at 16.7, accounting for approximately 41.54% of its trajectory length. This substantially exceeds the average correction attempts of 4.81 and correction rate of 16.99%, representing the highest values among all evaluated agents. In contrast, SWE-AGENT achieves the lowest correction rate among the evaluated configurations at 6.93%, while SE-AGENT (Iter-3) has the fewest correction attempts at 0.70. Among the three SE-AGENT iterations, Iter-1 has the lowest correction rate at 8.20%. However, an intriguing pattern emerges across subsequent iterations: although the absolute number of correction

Table 7
Comparison of software development efficiency across different agent frameworks.

Metric	AgentOrchestra	OWL	SE-Agent (Iter-1)	SE-Agent (Iter-2)	SE-Agent (Iter-3)	Trae	GPTswarm	OpenHands	SWE-Agent	Average
ATS	40.20	19.90	43.90	3.70	3.20	28.20	7.00	**81.28**	27.40	28.31
CA	**16.70**	4.20	3.60	1.10	0.70	6.10	1.21	7.79	1.90	4.81
CR(%)	**41.54**	21.10	8.20	29.73	21.86	21.63	17.19	9.58	6.93	16.99

Table 8
Comparison of vulnerability detection efficiency across different agent frameworks.

Metric	AgentOrchestra	OWL	SE-Agent (Iter-1)	SE-Agent (Iter-2)	SE-Agent (Iter-3)	Trae	GPTswarm	OpenHands	SWE-Agent	Average
ATS	**46.50**	7.50	1.00	1.40	7.00	8.90	1.00	11.00	9.50	10.40
CA	**21.04**	0.80	0.00	0.10	1.30	2.10	0.00	3.70	0.50	3.28
CR(%)	**45.25**	10.67	0.00	7.14	18.57	23.60	0.00	33.64	5.26	31.54

attempts decreases to 1.10 in the second iteration and 0.70 in the third iteration, the correction rate paradoxically increases. This phenomenon is attributable to the substantial reduction in trajectory length across iterations, resulting in a higher proportional correction rate despite fewer absolute corrections.

Vulnerability Detection. Table 8 presents the trajectory statistics of these agents in the vulnerability detection task. The analysis reveals that seven frameworks exhibit an average trajectory length of 10.40 steps, with AGENTORCHESTRA demonstrating the longest trajectory at 46.5 steps. In contrast, SE-AGENT (Iter-1) and GPTSWARM share the shortest average trajectories at 1.00 step, while SE-AGENT Iter-2 and Iter-3 average 1.40 and 7.00 steps, respectively. Furthermore, Table 8 presents the correction attempts and corresponding correction rates for each agent framework in the vulnerability detection task. AGENTORCHESTRA exhibits 21.04 correction attempts, constituting nearly 45.25% of its total trajectory steps, which represents the highest correction-to-execution ratio among the evaluated systems. This substantially exceeds the average of 3.28 correction attempts and 31.54% correction rate. Moreover, both SE-AGENT (Iter-1) and GPTSWARM exhibit zero correction attempts, representing the most efficient execution patterns among all evaluated agents.

Program Repair. Table 9 presents the execution trajectory analysis for the program repair task. Statistical analysis demonstrates that seven frameworks exhibit an average trajectory length of 57.44 steps, TRAE records the longest average trajectory of 78.1 steps among single-agent frameworks, warranting a detailed examination of its execution pattern. Our analysis reveals that the repair process of TRAE consists of: (1) 41.1 Bash commands for environment configuration, code execution, and repair validation; (2) 30 file editing operations; (3) 6 problem analysis steps; and (4) 1 completion command, totaling 78.1 steps. This decomposition indicates that TRAE's extended trajectory primarily stems from its more comprehensive problem analysis phase and increased reliance on environment validation commands, distinguishing it from other frameworks that adopt more streamlined execution strategies. Moreover, since GPTSWARM employs the CodeReact Zhuge et al. (2024) mechanism with a predetermined constraint of exactly 3 React cycles, it exhibits the shortest trajectory length of 2.90 steps among all seven evaluated agent frameworks, representing a distinct operational paradigm that prioritizes execution efficiency through fixed iteration constraints rather than adaptive exploration.

Although TRAE exhibits the longest trajectory among all agents, its correction attempts of 19.50 do not represent the maximum value. OPENHANDS records the highest correction attempts at 25.20, accounting for 36.36% of its trajectory length, substantially exceeding averages of 11.53 attempts at 20.08% rate. In contrast, GPTSWARM demonstrates the fewest correction attempts at merely 0.5, while OWL exhibits the lowest correction rate at only 1.55%. However, these minimal correction metrics do not translate to superior performance in program repair tasks. Both GPTSWARM and OWL fail to identify errors in their self-generated content, suggesting that their low correction frequencies reflect inadequate self-monitoring capabilities rather than execution efficiency. This observation underscores a critical insight: the absence of correction attempts may indicate a deficiency in error detection mechanisms rather than optimal performance. This deficiency can be attributed to the fact that neither GPTSWARM nor OWL incorporates version control tools such as `Git`, utilized by SWE-AGENT, or employs specialized diff format validation utilities like `unidiff`. The absence of these essential tools significantly undermines their operational efficiency and error detection capabilities. The cross-model analysis in Section 5.3.1 examines whether these trajectory and correction patterns remain qualitatively consistent when the backend model changes.

Answer to RQ2. The key takeaways are as follows:

Table 9
Comparison of program repair efficiency across different agent frameworks.

Metric	AgentOrchestra	OWL	SE-Agent (Iter-1)	SE-Agent (Iter-2)	SE-Agent (Iter-3)	Trae	GPTswarm	OpenHands	SWE-Agent	Average
ATS	46.80	64.50	51.50	67.40	69.10	**78.10**	2.90	69.30	67.40	57.44
CA	21.10	1.00	6.70	9.90	9.50	19.50	0.50	**25.20**	10.40	11.53
CR(%)	**45.09**	1.55	13.01	14.69	13.75	24.97	17.24	36.36	15.43	20.08

Table 10
Comparison of monetary overhead (USD) across different agent frameworks on three tasks. The final column reports the overall mean cost per instance across the seven frameworks, treating all three SE-Agent iterations as one workflow. DeepSeek-v3.1 prices per million tokens are USD 0.56 for uncached input, USD 0.07 for cached input, and USD 1.68 for output.

Task	AgentOrchestra	OWL	SE-Agent (Iter-1)	SE-Agent (Iter-2)	SE-Agent (Iter-3)	Trae	GPTswarm	OpenHands	SWE-Agent	Mean cost per instance
Software Development	$120.69	$21.05	$43.10	$31.01	$25.37	$48.90	$13.49	**$149.49**	$29.99	$0.0575
Vulnerability Detection	$14.13	$1.89	$0.29	$0.26	$1.27	$1.91	$0.67	**$20.91**	$1.07	$0.0527
Program Repair	**$64.05**	$2.49	$27.78	$38.40	$40.73	$15.53	$2.13	$54.48	$42.16	$0.1370
Total	$198.87	$25.43			$208.21	$66.34	$16.29	**$224.88**	$73.22	—

- **Trajectory Steps:** OPENHANDS and AGENTORCHESTRA have the longest average trajectories in software development and vulnerability detection, respectively. In program repair, SE-AGENT has the highest cumulative task-execution step total across its three iterations.
- **Correction Attempts:** AGENTORCHESTRA shows the longest correction attempts in software development and vulnerability detection, while OPENHANDS records the longest in program repair.
- **Correction Rate:** Across the three tasks, AGENTORCHESTRA achieves the highest correction rate, reflecting its lower efficiency due to frequent revisions.

4.3. Overhead of Agent Frameworks (RQ3)

To address RQ3, we evaluate the costs of seven agents across three tasks. Comprehensive results are presented in Table 10. The first column indicates the three evaluation tasks: software development, vulnerability detection, and program repair. The bottom row displays the total cost for each agent across all three tasks.

The results reveal that OPENHANDS incurred the highest cost at $224.88 across the three tasks, whereas GPTSWARM incurred the lowest total cost at $16.29. Software development has the highest full-benchmark evaluation cost, totaling $483.09, where OPENHANDS consumed $149.49, the highest among all agents. The seven agents collectively spent $42.40 to complete the vulnerability detection task, with OPENHANDS leading at $20.91, while SE-AGENT (Iter-1) required only $0.29, the lowest expenditure. For the program repair task, the total cost reached $287.75. For program repair, SE-AGENT's complete three-iteration workflow incurs the highest cumulative cost at USD 106.91, while GPTSWARM has the lowest cost at USD 2.13.

Because the benchmark sizes differ substantially (1,200 software development instances, 115 vulnerability detection instances, and 300 program repair instances), full-benchmark costs do not directly indicate the cost of executing an individual task instance. We therefore additionally report mean monetary cost per instance in the final column of Table 10, calculated as the total cost across all seven frameworks for each task divided by seven times the corresponding number of instances. For SE-AGENT, we sum all three iterations to represent its complete evaluated workflow. Averaged equally across the seven frameworks, the mean costs per instance are USD 0.0575 for software development, USD 0.0527 for vulnerability detection, and USD 0.1370 for program repair. Under the benchmarks and configurations evaluated in this study, program repair has the highest mean cost per instance and vulnerability detection has the lowest. This normalization accounts for differences in instance counts; differences in benchmark content, input size, and task difficulty still limit generalization beyond the evaluated settings.

Economic costs are determined by token consumption and model pricing. Since we employed the same LLM with consistent token pricing, we focus our analysis on token consumption, where the input price is $0.56 per million tokens, and the output price is $1.69 per million tokens. Notably, since the agent often reuses the same context across multiple interactions, the model can leverage a caching mechanism to reuse previously processed inputs. Consequently, even with a large number of input tokens, the overall cost may remain low due to the lower pricing of cached tokens ($0.07 per million). Table 11 presents the token consumption results for these agents, organized into three major rows representing the three tasks. Each major row contains two sub-rows

Table 11
Comparison of token count across different agent frameworks on three tasks.

Task	Component	AgentOrchestra	OWL	SE-Agent (Iter-1)	SE-Agent (Iter-2)	SE-Agent (Iter-3)	Trae	GPTswarm	OpenHands	SWE-Agent
Software Development	Input	380.34 M	42.88 M	440.69 M	28.31 M	20.54 M	32.09 M	287.31 K	**1.26 B**	371.70 M
	Output	22.71 M	3.34 M	8.53 M	11.15 M	9.26 M	23.09 M	7.98 M	**30.54 M**	814.04 K
Vulnerability Detection	Input	36.30 M	2.64 M	216.91 K	360.03 K	7.98 M	1.57 M	130,507	**192.75 M**	4.17 M
	Output	1.60 M	632.38 K	248.32 K	202.15 K	537.87 K	841.91 K	394.12 K	**5.85 M**	227.90 K
Program Repair	Input	99.59 M	6.72 M	332.03 M	447.96 M	484.44 M	13.28 M	54.10 K	**663.55 M**	495.50 M
	Output	2.16 M	220.74 K	524.66 K	853.67 K	839.23 K	**4.82 M**	485.46 K	4.56 M	836.83 K

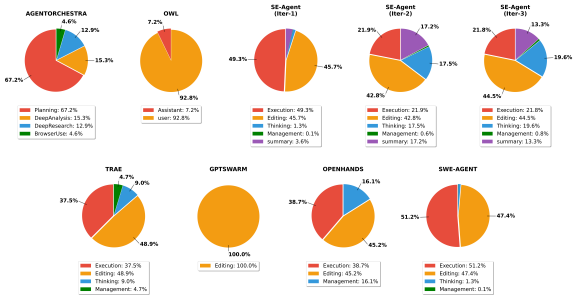


Figure 2: Breakdown of token consumption across execution stages for software development among the seven agent frameworks.

indicating input and output token counts, respectively, with each column representing the token consumption for each agent.

As shown in Table 11 OpenHands consumed 1.26 B input tokens and 30.54 M output tokens, making it the highest consumer. The main reason OpenHands consumes substantially more tokens is that it is a multi-turn coding agent rather than a single-shot generator. In each interaction round, OpenHands re-sends a long system prompt, tool specifications, prior dialogue history, execution feedback, and intermediate code-related context back to the model.GPTswarm consumed the fewest input tokens at 287.31 K, while SWE-Agent used only 814.04 K output tokens, the lowest among all agents. This discrepancy is attributable to the caching mechanism: although OpenHands processed the highest number of input and output tokens, these largely consisted of accumulated historical data that, when fed back to the agent, did not incur proportional costs. In the vulnerability detection task, OpenHands again consumed the most input tokens at 192.75 M and output tokens at 5.85 M. GPTswarm used the fewest input tokens at 130,507, while SWE-Agent consumed the least output tokens at 227.90 K. For the program repair task, OpenHands consumed 663.55 M input tokens, the highest among all agents, while Trae generated the most output tokens at 4.82 M. In contrast, GPTswarm consumed merely 54.10 K input tokens, and OWL used 220.74 K output tokens. A notable observation is that, except for GPTswarm, all agents consumed more input tokens than output tokens. For GPTswarm, this pattern is reversed, with output token consumption exceeding input token consumption.

To further investigate how token consumption is distributed across different execution stages among the seven agent frameworks, we employ the classification framework presented in Figure 2 for software development, Figure 3 for vulnerability detection, and Figure 4 for program repair. The first row comprises two multi-agent systems for which we analyze token consumption at the individual sub-agent level. Given that SE-Agent's worker agents operate analogously to SWE-Agent as single agents and utilize a summary operators for post-evaluation trajectory synthesis, we apply a unified statistical methodology to SE-Agent's worker agents along with the four single-agent systems shown in the second row, decomposing token consumption into four action categories: execution, editing, thinking, and management.

Software Development As illustrated in Figure 2, AgentOrchestra employs four distinct agents for software development tasks: Planning agent, DeepAnalysis agent, DeepResearch agent, and BrowserUse agent. The token consumption is predominantly attributed to the Planning agent, which accounts for 67.2% of the total consumption, followed by the DeepAnalysis agent at 15.3% and the DeepResearch agent at 12.9%, while the BrowserUse agent contributes the least at 4.6%. OWL comprises two agents: User agent and Assistant agent, where the User agent dominates token consumption at 92.8%, with the Assistant agent accounting for merely 7.2%.

For SE-Agent in its first iteration, token consumption is primarily distributed between editing and execution, representing 49.3% and 45.7% of the total respectively, whereas management and thinking constitute only 0.1% and 1.3%. This distribution stems from SE-Agent's well-structured prompts with examples, and the post-evaluation summary accounts for only 3.6% of total consumption. In subsequent iterations, the streamlined prompts afford SE-Agent greater flexibility, resulting in increased proportions of management and thinking activities. Nevertheless, editing and execution continue to dominate total consumption, while the reduced prompt complexity leads to elevated summary operators consumption of 17.7% and 13.3% in the second and third iterations respectively.

Figure 2 further reveals that single-agent systems allocate the majority of token consumption to editing activities in software development tasks, specifically 48.9% for Trae and 45.2% for OpenHands. GPTswarm exclusively

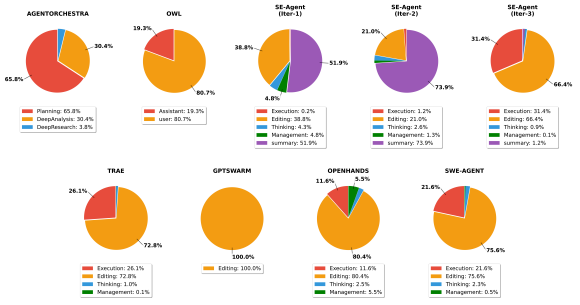


Figure 3: Breakdown of token consumption across execution stages for vulnerability detection among the seven agent frameworks.

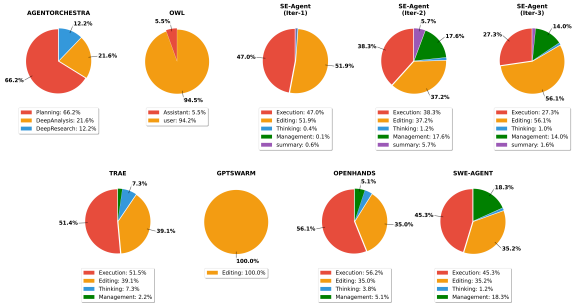


Figure 4: Breakdown of token consumption across execution stages for program repair among the seven agent frameworks.

utilizes editing due to its IO-agent architecture, resulting in 100% editing consumption. Conversely, SWE-AGENT exhibits the highest execution consumption at 51.2% of total tokens, with editing as the second-largest category at 47.4%. TRAE and OPENHANDS demonstrate execution consumption of 37.5% and 38.7% respectively. Thinking activities constitute a minimal proportion in OPENHANDS and SWE-AGENT, while accounting for 9.0% in TRAE. Similarly, management activities represent a negligible share in SWE-AGENT, whereas TRAE and OPENHANDS allocate 4.7% and 16.1% of total consumption to management respectively.

Vulnerability Detection. As depicted in Figure 3, AGENTORCHESTRA employs three agents for vulnerability detection tasks: Planning agent, DeepAnalysis agent, and DeepResearch agent. Token consumption is predominantly concentrated in the Planning agent at 65.8% of the total, followed by the deep analysis agent at 30.4% and the deep research agent at 3.8%. For OWL, the User agent accounts for 80.7% of total consumption, while the Assistant agent represents 19.3%.

In SE-AGENT's first two iterations, token consumption is overwhelmingly dominated by the summary process, constituting 51.9% and 73.9% of total consumption respectively, whereas the third iteration allocates only 1.2% to summary activities. Editing operations account for 38.8%, 21.0%, and 66.4% across the three iterations respectively. Execution activities demonstrate minimal presence in the first two iterations but increase substantially to 31.4% in the third iteration, while thinking and management activities maintain negligible proportions throughout all iterations.

Figure 3 further demonstrates that single-agent systems allocate the majority of token consumption to editing activities in vulnerability detection tasks, specifically 72.8% for TRAE, 80.4% for OPENHANDS, and 75.6% for SWE-AGENT. Execution represents the second-largest consumption category across these three agents, accounting for 26.1%, 11.6%, and 21.6% respectively. Other activities including thinking and management constitute minimal proportions across all three agents. These three agents exhibit remarkable consistency in their token consumption patterns when addressing vulnerability detection tasks.

Program Repair. As illustrated in Figure 4, AGENTORCHESTRA employs three agents for program repair tasks: Planning agent, DeepAnalysis agent, and DeepResearch agent. Token consumption is primarily attributed to the Planning agent at 66.2% of the total, followed by the DeepAnalysis agent at 21.6%, with the DeepResearch agent contributing the least at 12.2%. OWL comprises two agents: User agent and Assistant agent, where the User agent dominates token consumption at 94.2%, while the Assistant agent accounts for merely 5.5%.

For SE-AGENT in its first iteration, token consumption is predominantly distributed between editing and execution, representing 51.9% and 47.0% of the total respectively, whereas management and thinking constitute only 0.1% and 0.4%. This distribution is attributable to SE-AGENT's well-structured prompts with examples, and the post-evaluation summary accounts for only 0.6% of total consumption. In subsequent iterations, the streamlined prompts afford SE-AGENT greater flexibility, resulting in increased proportions of management and thinking activities. Nevertheless, editing and execution continue to dominate total consumption, while the reduced prompt complexity leads to elevated summary operators consumption of 5.7% and 1.6% in the second and third iterations respectively.

Figure 4 further reveals that single-agent systems allocate the majority of token consumption to execution activities in program repair tasks, specifically 51.5% for TRAE, 56.2% for OPENHANDS, and 45.3% for SWE-AGENT. Editing operations account for 39.1%, 100.0%, 35.0%, and 35.2% of total consumption for TRAE, GPTSWARM, OPENHANDS, and SWE-AGENT respectively. While thinking and management activities constitute minimal proportions in TRAE and OPENHANDS, management activities notably account for 18.3% of total consumption in SWE-AGENT. Section 5.3.1 further assesses whether the observed token and monetary-overhead differences persist across foundation models.

Answer to RQ3. The key takeaways are as follows:

- **Monetary Cost:** Under the evaluated settings, program repair has the highest mean monetary cost per instance and vulnerability detection has the lowest. Across the complete evaluated workflows, OPENHANDS has the highest total monetary cost, whereas GPTSWARM has the lowest.

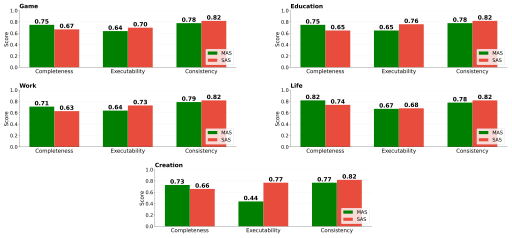


Figure 5: multi-agent systems vs single-agent systems in Software Development.

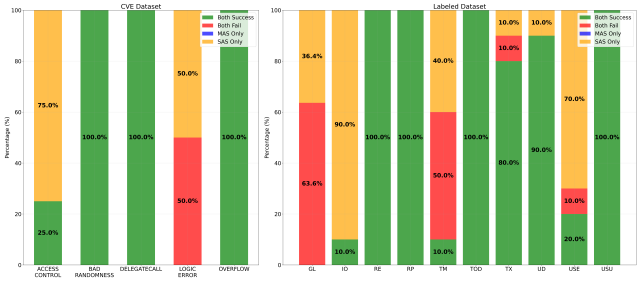


Figure 6: multi-agent systems vs single-agent systems in Vulnerability Detection.

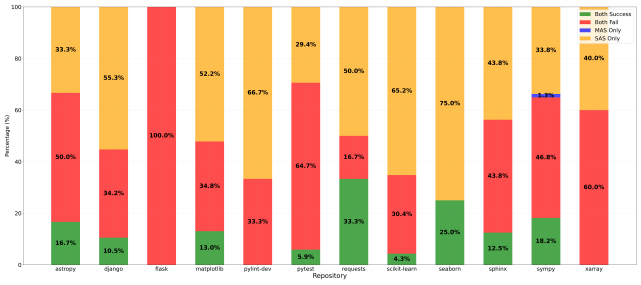


Figure 7: multi-agent systems vs single-agent systems in Program Repair.

- **Token Usage:** OPENHANDS consumes the most tokens across all three tasks, consequently incurring the highest overall cost. GPTSWARM has the lowest input-token consumption in all three tasks. It is also the only agent where input tokens are fewer than output tokens, while all other agents exhibit higher input token consumption.
- **Consumption Breakdown:** The same agent framework exhibit similar patterns across tasks. For example,in AGENTORCHESTRA, the Planning agent dominates. SE-AGENT shows varying distributions due to prompt modifications. Single-agent systems focus consumption on execution and editing actions.

5. Discussion

The RQ1–RQ3 results show that higher effectiveness does not always coincide with greater efficiency or lower overhead, motivating further analysis of how reflection, tool use, and model choice influence framework performance. We therefore discuss the relationship between steps and effectiveness, the roles of reflection and tools, robustness across models and repeated executions, and the resulting practical implications and validity threats.

5.1. More steps, better effectiveness?

The RQ2 results show that multi-agent frameworks generally execute longer trajectories than single-agent frameworks, particularly in vulnerability detection and program repair. To examine whether these additional steps improve task outcomes, we compare the aggregated effectiveness of the two groups under the primary DeepSeek-v3.1 configuration. The comparison shows that longer trajectories do not consistently lead to higher effectiveness, although the relationship varies across tasks and evaluation dimensions.

Software Development. As shown in Figure 5, multi-agent frameworks achieve higher completeness across the five software categories, suggesting that their collaborative workflows cover more task requirements. However, single-agent frameworks perform substantially better in executability and slightly better in consistency. These differences result in higher aggregated Quality for the single-agent group, with OPENHANDS obtaining the highest individual Quality of 0.47. Thus, the additional coordination steps of multi-agent frameworks improve requirement coverage but do not translate into more executable or higher-quality projects.

Vulnerability Detection. We compare the two groups across 10 vulnerability types from the common dataset and five CVE vulnerability types, as illustrated in Figure 6. Their detection results are similar for seven types, including Bad Randomness, Delegatecall, Overflow, RE, PR, TOD, and USU. Single-agent frameworks perform better for six other types, whereas no vulnerability type is detected exclusively by the multi-agent group. Because the multi-agent frameworks also use more steps in RQ2, their longer trajectories provide no aggregate effectiveness advantage for vulnerability detection.

Program Repair. The comparison across the 12 repositories in Figure 7 shows that the single-agent group achieves higher aggregated repair effectiveness, despite generally using fewer steps. The difference is associated with the ability to complete the repository inspection, editing, and patch-validation workflow. Consequently, program repair provides the clearest evidence that trajectory length alone is not a reliable indicator of effectiveness.

The trajectories indicate that the value of a step depends on its function. Multi-agent planning and delegation can broaden task coverage, but also add coordination, repeated corrections, and opportunities for error propagation. Consistent with RQ2, the evaluated multi-agent configurations generally make more corrections without achieving better aggregate outcomes.

Program repair further shows that task-aligned tools and feedback can matter more than trajectory length or agent count. AGENTORCHESTRA and OWL sometimes produce malformed `diff` patches, and GPTSWARM performs poorly

Table 12
Comprehensive Performance and Ablation Study Across SD, VD, and PR Tasks

Agent	Software Development							Vulnerability Detection				Program Repair			
	Comp	Exec	Cons	Quality	ATS	CA	CR (%)	Acc (%)	ATS	CA	CR (%)	Acc (%)	ATS	CA	CR (%)
SE-Agent(Iter1)	0.77	0.37	0.86	0.25	16.57	0.25	1.51	57.39	8.71	0.00	0.00	52.69	27.96	1.37	4.90
SE-Agent(Iter3)	0.42	0.29	0.54	0.07	12.87	0.36	2.80	53.91	5.28	0.00	0.00	48.39	43.60	2.79	6.40
GPTSwarm	0.92	0.14	0.87	0.11	10.04	0.00	0.00	41.74	1.96	0.00	0.00	41.94	29.75	1.35	4.54
GPTswarmOPT	0.68	0.38	0.85	0.22	9.96	0.04	0.40	42.61	1.94	0.00	0.00	43.01	30.65	1.49	4.86

without an interface supporting the complete repair loop, whereas SWE-AGENT benefits from specialized repository and patch operations. Additional steps can still help when they provide relevant feedback: under DeepSeek-v3.1, SE-AGENT's repair rate rises from 47.33% to 53.67%.

5.2. Impact of Reflection Mechanisms and Tool Utilization

To understand what drives the relationship between trajectory length and effectiveness observed in Section 5.1, we further analyze two mechanisms that directly shape agent execution: reflection and tool utilization.

5.2.1. Reflection Mechanisms

We compare two reflection designs using Qwen3-Coder-Flash. GPTSWARM performs batch-level optimization within one evaluation round: after every four tasks, it summarizes recent failures and updates the guidance for subsequent tasks. SE-AGENT performs three dataset-level iterations and derives reflection context from earlier trajectories. Because the SE-AGENT iterations vary both prompt content and trajectory-derived context, their differences represent the joint effect of these components rather than the isolated effect of prompt detail. The optimized GPTSWARM configuration (GPTswarmOPT) is the configuration reported as GPTSWARM in the cross-model analysis in Section 5.3.1; the shared configurations use the same experimental runs.

Finding 1: Reflection changes effectiveness in a task-dependent manner. For SE-AGENT, moving from Iter-1 to the trajectory-summary configuration in Iter-3 reduces Quality from 0.25 to 0.07 in software development and accuracy from 57.39% to 53.91% in vulnerability detection (Table 12). The generalized risk guidance can divert attention from deliverable requirements or code-level evidence, so additional reflection context does not uniformly improve outcomes.

Finding 2: Fewer steps do not necessarily indicate greater efficiency. SE-AGENT Iter-3 has shorter average trajectories than Iter-1 in software development and vulnerability detection, but also lower effectiveness (Table 12). Inspection of the trajectories shows that some runs terminate earlier or reach conclusions with less task-specific verification; the reduced step count must therefore be interpreted together with outcome quality.

Finding 3: Useful reflection must remain grounded in the current task. Trajectory summaries can help an agent avoid earlier repair attempts, but generalized warnings can also compete with current-instance evidence. In program repair, Iter-3 uses more steps and corrections than Iter-1 while attaining a lower repair rate under Qwen3-Coder-Flash (Table 12), suggesting that, for smaller models, additional historical context may increase effort without improving the quality of the final patch.

Finding 4: Reflection effectiveness is jointly determined by its design and the capability of the foundation model. SE-AGENT's repair-specific reflection Lin et al. (2025) improves its repair rate from 47.33% to 53.67% with DeepSeek-v3.1 (Table 6), but under Qwen3-Coder-Flash, Iter-3 falls from 52.69% to 48.39% and also degrades the other two tasks (Table 12). In contrast, GPTSWARM's general batch-level optimization Zhuge et al. (2024) improves effectiveness across all three tasks in Table 12. These results show that reflection effectiveness depends on both task-aligned mechanism design and the foundation model's ability to use the feedback.

5.2.2. Tool Utilization

We use GPTSWARM to examine how tool alignment changes execution. The baseline uses GPTSWARM's native `codewriting` tool, while the comparison replaces it with the SWE-Agent toolset. Both configurations use DeepSeek-v3.1 with identical generation settings, so this analysis concerns one framework–model pairing rather than cross-model robustness.

Finding: Tool effectiveness depends on alignment with the task workflow. Replacing GPTSWARM's native `codewriting` tool Zhuge et al. (2024) with the SWE-Agent toolset Yang et al. (2024) increases the program-repair rate from 3.22% to 48.39% and average steps from 2.90 to 22.18 (Table 14). The repository and patch tools generate valid `diff` files and prevent early termination caused by formatting errors, allowing the agent to complete the localization–editing–validation loop; the longer trajectory therefore represents a more complete repair process. However, the same replacement reduces software-development Quality from 0.43 to 0.17 and increases average steps from 7.00 to 9.96 because repair-oriented actions and feedback disrupt GPTSWARM's development workflow, producing more unproductive interactions without better outputs. Tool augmentation therefore improves effectiveness only when its actions, feedback, and output formats match the target workflow.

5.3. Robustness of the Main Findings

The primary RQ1–RQ3 comparison fixes DeepSeek-v3.1 as the backend so that framework differences are not

Table 13
Ablation Study of Reflection Mechanisms on Token Usage (Millions) and Total Cost (USD)

Agent	SD				VD				PR			
	In (M)	Out (M)	Cache (M)	Cost ($)	In (M)	Out (M)	Cache (M)	Cost ($)	In (M)	Out (M)	Cache (M)	Cost ($)
SE-Agent(iter1)	37.71	1.63	26.45	2.86	4.67	0.19	3.04	0.37	46.85	0.54	37.82	2.22
SE-Agent(iter3)	30.85	1.22	22.00	2.23	2.85	0.16	1.40	0.31	70.18	2.00	57.33	3.98
GPTSwarm	1.48	3.38	1.34	1.92	0.25	0.03	0.12	0.04	11.20	0.17	9.09	0.62
GPTswarmOPT	0.74	1.08	0.67	0.62	0.25	0.03	0.12	0.04	28.87	0.38	21.55	1.53

Table 14
Comprehensive Comparison of Performance, Tool Actions, and Costs: GPTSWARM vs. GPTSWARM-SWE on SD and PR Tasks

Agent	Software Development										Program Repair						
	Performance				Tool / Action			Cost			Perf	Tool / Action			Cost		
	Comp	Exec	Cons	Qual	ATS	CA	CR(%)	Input	Output	Tot($)	Acc	ATS	CA	CR(%)	Input	Output	Tot($)
GPTSwarm	**0.76**	**0.66**	**0.85**	**0.43**	7.00	1.21	17.29	28.73K	2.98M	1.35	3.22	2.90	0.50	17.24	11.42K	48.55K	0.21
GPTswarm-SWE	0.57	0.35	**0.85**	0.17	9.96	0.00	0.00	743.00K	1.77M	0.78	**48.39**	22.18	0.12	0.54	23.38M	623.61K	0.17

confounded by model choice. We complement this controlled comparison with two robustness checks. First, we replace the foundation model to test whether the substantive conclusions persist across model–framework combinations. Second, we repeat one representative setting to estimate execution-to-execution variance under a fixed configuration. These analyses address cross-model generalizability and stochastic reliability, respectively.

5.3.1. Generalizability across Foundation Models

Purpose and setup. Sections 4.1–4.3 fix DeepSeek-v3.1 for a controlled comparison; here, we test whether their conclusions recur with `Qwen3-Coder-Flash`, `Qwen3-Coder-Plus`, and `GLM-4.7` on the subsets below. For program repair, all frameworks use the SWE-Agent tool paradigm Yang et al. (2024). For scale, the public `Qwen3-Coder` checkpoints report 30.5B/3.3B and 480B/35B total/activated parameters, respectively Qwen Team (2025a,b), and `GLM-4.7` reports approximately 358B parameters Z.ai (2025). The release instructions link the 480B model to `Qwen3-Coder-Plus`, but the configurations do not verify exact API snapshots or a correspondence between `Qwen3-Coder-Flash` and the 30.5B checkpoint. Thus, this analysis evaluates the robustness of the main findings across different foundation models on the selected subsets.

The representative subsets are constructed as follows: (1) for software development, we randomly sample three SRDD instances from each of its 40 subcategories; (2) for vulnerability detection, we retain all 115 instances; and (3) for program repair, we follow the filtering method of OrcaLoca Yu et al. (2025) and use the 93-instance intersection of SWE-bench-lite and SWE-bench-verified.

RQ1: Effectiveness remains dependent on the task, framework, and foundation model. The additional-model experiments in Tables 15 and 17 support the primary conclusion that no framework is universally most effective. AGENTORCHESTRA leads software development under all three additional models, whereas the leaders change for vulnerability detection and program repair: SWE-AGENT leads vulnerability detection under both Qwen models, SE-AGENT Iter-1 leads under `GLM-4.7`, and the program-repair leaders are SE-AGENT Iter-2, SWE-AGENT, and SE-AGENT Iter-3 for the three models, respectively. The broad task dependence observed in RQ1 therefore recurs, while framework rankings and the magnitude of their differences remain model dependent.

RQ2: Longer trajectories and more corrections do not consistently imply better outcomes. Tables 16, 18, and 19 show that, across the additional models, the framework with the longest trajectory or the most correction attempts is not consistently the most effective. The identities of these frameworks also change across model–task combinations, and correction attempts disappear entirely in the vulnerability-detection runs despite substantial accuracy differences. These results support the main conclusion that efficiency depends on how a workflow uses its steps, while qualifying any ranking tied to one backend model.

RQ3: Large overhead differences persist, but their monetary effect depends on model pricing. Tables 16, 18, and 19 show substantial variation in token consumption and monetary cost across framework–model–task combinations. In software development, OpenHands has the highest estimated cost under both Qwen models, while Trae has the highest estimate under GLM-4.7. In vulnerability detection, OpenHands has the highest cost under Qwen3-Coder-Flash and GLM-4.7, and OWL under Qwen3-Coder-Plus, while the highest-cost configuration for program repair varies with the model. Because monetary cost combines execution behavior with model-specific API prices, token rankings and cost rankings need not coincide. Thus, the principal RQ3 conclusion about substantial framework overhead is supported, while exact cost rankings should be interpreted for the stated model and pricing schedule.

Overall, the additional-model experiments support the high-level conclusions of RQ1–RQ3, but they also show that framework rankings, reflection gains, correction behavior, and cost magnitudes can change with the foundation model. We therefore treat the primary DeepSeek-v3.1 comparison

Table 15
Cross-model effectiveness results for software development used to assess the generalizability of RQ1. Bold values indicate the highest Quality for each model.

Model	Metric	SWE-Agent	SE-Agent (Iter-1)	SE-Agent (Iter-2)	SE-Agent (Iter-3)	OWL	GPTswarm	AgentOrchestra	Trae	OpenHands
Qwen3-Coder-Flash	Comp	0.66	0.77	0.73	0.42	0.63	0.68	0.84	0.69	0.72
	Exec	0.39	0.37	0.37	0.29	0.34	0.38	0.73	0.32	0.31
	Cons	0.86	0.86	0.65	0.54	0.85	0.85	0.86	0.85	0.75
	Quality	0.22	0.25	0.18	0.07	0.18	0.22	**0.53**	0.19	0.17
Qwen3-Coder-Plus	Comp	0.63	0.72	0.73	0.65	0.67	0.80	0.87	0.66	0.40
	Exec	0.43	0.35	0.33	0.25	0.44	0.27	0.61	0.37	0.19
	Cons	0.80	0.86	0.86	0.86	0.86	0.86	0.85	0.86	0.35
	Quality	0.22	0.22	0.21	0.14	0.25	0.19	**0.45**	0.21	0.03
GLM-4.7	Comp	0.47	0.54	0.44	0.38	0.35	0.64	0.58	0.27	0.58
	Exec	0.26	0.34	0.35	0.35	0.22	0.25	0.56	0.28	0.10
	Cons	0.85	0.85	0.85	0.53	0.59	0.85	0.84	0.84	0.73
	Quality	0.10	0.16	0.13	0.07	0.05	0.14	**0.27**	0.06	0.04

Table 16
Cross-model execution-efficiency and resource-overhead results for software development used to assess the generalizability of RQ2 and RQ3. Costs are estimated in USD from the rounded token totals shown. Prices per million tokens (uncached input / cached input / output) are 0.30 / 0.06 / 1.50 for Qwen3-Coder-Flash, 1.00 / 0.20 / 5.00 for Qwen3-Coder-Plus, and 0.60 / 0.11 / 2.20 for GLM-4.7. Bold values indicate the lowest ATS, CA, CR, input tokens, output tokens, and cost for each model, including ties. Cache reports the cached-input volume.

Model	Metric	SWE-Agent	SE-Agent (Iter-1)	SE-Agent (Iter-2)	SE-Agent (Iter-3)	OWL	GPTswarm	AgentOrchestra	Trae	OpenHands
Qwen3-Coder-Flash	ATS	18.40	16.57	16.57	12.87	65.68	**9.96**	35.27	18.77	82.64
	CA	0.33	0.25	0.24	0.36	0.14	0.04	**0.00**	0.34	10.83
	CR (%)	1.80	1.50	1.40	2.80	0.20	0.40	**0.00**	1.80	13.10
	Input (M)	44.76	37.71	33.87	30.85	103.71	**28.87**	39.27	51.34	145.82
	Output (M)	1.86	1.63	1.52	1.22	3.33	0.38	**0.20**	2.02	4.28
	Cache (M)	31.90	26.45	22.74	22.00	77.17	21.55	11.25	38.60	123.57
	Cost	8.56	7.41	6.98	5.81	17.59	**4.06**	9.38	9.17	20.51
Qwen3-Coder-Plus	ATS	16.62	15.38	14.90	18.61	40.52	**10.00**	33.41	20.02	28.69
	CA	0.35	0.22	0.29	0.44	0.16	**0.00**	**0.00**	0.38	0.58
	CR (%)	2.10	1.40	1.90	2.40	0.40	**0.00**	**0.00**	1.90	2.00
	Input (M)	45.20	17.09	16.37	25.57	3.27	**0.74**	41.14	29.73	64.35
	Output (M)	2.15	0.91	0.93	1.16	**0.11**	1.14	0.29	1.47	3.68
	Cache (M)	38.21	13.98	13.51	22.38	2.84	0.14	22.29	25.85	35.89
	Cost	25.38	10.46	10.21	13.47	**1.55**	6.33	24.76	16.40	54.04
GLM-4.7	ATS	25.67	25.67	27.27	37.38	20.63	**10.00**	20.20	41.91	126.43
	CA	1.12	0.84	0.96	1.45	0.06	0.10	**0.00**	1.72	5.58
	CR (%)	4.40	3.30	3.50	3.90	0.30	1.00	**0.00**	4.10	4.40
	Input (M)	52.90	46.79	49.63	94.59	26.74	**0.75**	146.05	146.30	125.40
	Output (M)	2.18	2.13	2.07	2.75	**1.21**	1.86	1.75	4.26	3.67
	Cache (M)	45.61	0.00	4.12	81.84	20.91	0.68	71.23	53.49	111.24
	Cost	14.19	32.76	32.31	22.70	8.46	**4.21**	56.58	70.94	28.81

Table 17
Cross-model effectiveness results for vulnerability detection and program repair used to assess the generalizability of RQ1. Bold values indicate the highest accuracy for each model within each task.

(a) Vulnerability Detection

Agent Framework	Accuracy (%)		
	Qwen3-Coder-Flash	Qwen3-Coder-Plus	GLM-4.7
SWE-Agent	**63.48**	**73.91**	63.48
SE-Agent(Iter-1)	57.39	63.48	**64.35**
SE-Agent(Iter-2)	57.39	64.35	57.39
SE-Agent(Iter-3)	53.91	61.40	61.74
OWL	50.43	39.13	50.43
GPTswarm	42.61	52.17	53.04
AgentOrchestra	47.83	49.57	51.30
Trae	46.43	47.75	54.39
OpenHands	60.87	8.70	44.35

(b) Program Repair

Agent Framework	Accuracy (%)		
	Qwen3-Coder-Flash	Qwen3-Coder-Plus	GLM-4.7
SWE-Agent	52.69	**75.27**	74.19
SE-Agent(Iter-1)	52.69	74.19	72.04
SE-Agent(Iter-2)	**60.22**	73.12	66.67
SE-Agent(Iter-3)	48.39	63.44	**78.49**
OWL	41.94	60.22	69.89
GPTswarm	43.01	62.37	62.37
AgentOrchestra	27.96	40.86	45.16
Trae	47.31	58.06	65.59
OpenHands	46.24	15.05	18.28

Table 18
Cross-model execution-efficiency and resource-overhead results for vulnerability detection used to assess the generalizability of RQ2 and RQ3. Costs are estimated in USD from the rounded token totals shown. Prices per million tokens (uncached input / cached input / output) are 0.30 / 0.06 / 1.50 for Qwen3-Coder-Flash, 1.00 / 0.20 / 5.00 for Qwen3-Coder-Plus, and 0.60 / 0.11 / 2.20 for GLM-4.7. Bold values indicate the lowest ATS, CA, CR, input tokens, output tokens, and cost for each model, including ties. Cache reports the cached-input volume.

Model	Metric	SWE-Agent	SE-Agent (Iter-1)	SE-Agent (Iter-2)	SE-Agent (Iter-3)	OWL	GPTswarm	AgentOrchestra	Trae	OpenHands
Qwen3-Coder-Flash	ATS	9.23	8.71	5.70	5.28	22.17	**1.94**	30.23	9.94	11.70
	CA	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
	CR(%)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
	Input (M)	5.99	4.67	3.05	2.85	7.32	**0.13**	4.89	7.30	16.31
	Output (M)	0.25	0.19	0.17	0.16	0.12	**0.02**	0.12	0.35	0.32
	Cache (M)	3.86	3.04	1.58	1.40	4.56	0.00	1.99	4.84	13.31
	Cost	1.25	0.96	0.79	0.76	1.28	**0.07**	1.17	1.55	2.18
Qwen3-Coder-Plus	ATS	3.87	2.29	2.43	2.26	13.01	**1.00**	27.51	7.99	2.97
	CA	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
	CR(%)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
	Input (M)	2.11	0.89	0.96	0.87	47.74	**0.13**	4.43	6.45	17.13
	Output (M)	0.18	0.08	0.10	0.08	1.64	**0.01**	0.12	0.26	0.80
	Cache (M)	1.40	0.49	0.44	0.37	41.45	0.00	2.13	5.28	1.98
	Cost	1.89	0.90	1.11	0.97	22.78	**0.18**	3.33	3.53	19.55
GLM-4.7	ATS	4.39	3.75	3.50	3.96	25.73	**1.02**	22.12	7.71	18.34
	CA	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
	CR(%)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
	Input (M)	1.58	1.41	1.21	1.41	11.89	**0.13**	3.89	3.94	40.32
	Output (M)	0.34	0.30	0.24	0.25	0.94	**0.02**	0.12	0.37	0.69
	Cache (M)	1.00	0.90	0.77	0.88	9.31	0.00	1.44	3.38	0.00
	Cost	1.21	1.07	0.88	0.96	4.64	**0.12**	1.89	1.52	25.71

Table 19
Cross-model execution-efficiency and resource-overhead results for program repair used to assess the generalizability of RQ2 and RQ3. Costs are estimated in USD from the rounded token totals shown. Prices per million tokens (uncached input / cached input / output) are 0.30 / 0.06 / 1.50 for Qwen3-Coder-Flash, 1.00 / 0.20 / 5.00 for Qwen3-Coder-Plus, and 0.60 / 0.11 / 2.20 for GLM-4.7. Bold values indicate the lowest ATS, CA, CR, input tokens, output tokens, and cost for each model, including ties. Cache reports the cached-input volume.

Model	Metric	SWE-Agent	SE-Agent (Iter-1)	SE-Agent (Iter-2)	SE-Agent (Iter-3)	OWL	GPTswarm	AgentOrchestra	Trae	OpenHands
Qwen3-Coder-Flash	ATS	41.62	27.96	44.75	43.60	**24.41**	30.65	84.11	31.12	93.00
	CA	3.01	1.37	3.08	2.79	0.23	1.49	**0.02**	1.05	0.11
	CR (%)	7.23	4.90	6.88	6.40	0.94	4.86	**0.02**	3.37	0.12
	Input (M)	70.98	46.85	85.39	70.18	23.27	**0.74**	273.01	52.53	191.96
	Output (M)	1.02	0.54	1.58	2.00	**0.11**	1.08	1.44	0.73	3.65
	Cache (M)	57.15	37.82	70.57	57.33	6.90	0.67	103.70	41.03	164.53
	Cost	9.11	5.79	11.05	10.29	5.49	**1.68**	59.18	7.01	23.58
Qwen3-Coder-Plus	ATS	47.65	**37.00**	47.26	49.07	37.27	41.33	76.15	55.66	93.00
	CA	3.54	2.09	2.74	2.68	0.32	0.93	0.03	1.73	**0.00**
	CR (%)	7.43	5.65	5.80	5.46	0.86	2.25	0.04	3.11	**0.00**
	Input (M)	109.49	81.44	105.67	96.15	**29.58**	39.97	77.56	148.74	52.51
	Output (M)	1.08	0.72	0.98	0.86	0.70	**0.40**	1.11	1.24	2.04
	Cache (M)	97.22	72.81	93.34	84.65	25.38	36.13	35.07	134.00	32.01
	Cost	37.11	26.79	35.90	32.73	**12.78**	13.07	55.05	47.74	37.10
GLM-4.7	ATS	98.01	75.97	116.20	117.92	62.19	**47.27**	131.12	89.88	88.00
	CA	5.54	3.23	4.11	4.18	0.29	0.66	0.63	1.56	**0.00**
	CR (%)	5.65	4.25	3.54	3.54	0.47	1.40	0.48	1.74	**0.00**
	Input (M)	262.80	196.97	307.60	356.92	**26.74**	51.47	107.33	226.55	56.03
	Output (M)	2.04	2.45	2.45	2.77	**0.46**	0.87	3.26	2.03	0.55
	Cache (M)	257.66	192.89	302.59	351.05	25.33	51.47	49.34	218.89	51.43
	Cost	35.91	29.06	41.68	48.23	**4.64**	7.58	47.39	33.14	9.63

as a controlled framework evaluation and the additional-model results as evidence about the scope and boundary conditions of its conclusions.

5.3.2. Variance and Experimental Reliability

The cross-model analysis in Section 5.3.1 examines whether the main conclusions recur when the foundation model changes. This subsection addresses a different source

Table 20
Results of Five Repeated Trials on Vulnerability Detection (GPTSwarm + DeepSeek-v3.1). Acc = $N_{\text{correct}}/115$, where N_{correct} is the number of correctly classified contracts.

Trial	RQ1	RQ2			RQ3		
	Acc(%) / Count	ATS	CA	CR	Input	Output	Cache
Run 1	78.26 / 90	1.00	0.00	0.00	130,507	394,124	0
Run 2	77.39 / 89	1.00	0.00	0.00	130,507	396,351	0
Run 3	78.26 / 90	1.00	0.00	0.00	130,507	391,389	0
Run 4	77.39 / 89	1.00	0.00	0.00	130,507	397,076	0
Run 5	76.52 / 88	1.00	0.00	0.00	130,507	392,015	0

of uncertainty: variation across repeated executions while the model, framework, task, and configuration remain fixed.

Although we employ greedy decoding (*top-k* = 1) to minimize generation randomness, the inherent non-determinism of API-based large language models may still introduce slight variance. Due to constraints on time and computational resources, our primary results are based on a single complete execution. Replicating the entire suite of experiments would incur approximately a five-fold increase in financial cost and extend the timeline to around two months, a challenge further exacerbated by limited server resources and API network fluctuations.

To assess the reliability of our single-run results, we repeated the vulnerability detection task five times using GPTSwarm with DeepSeek-v3.1 on the same 115 contracts as the primary experiment. After switching from the official DeepSeek API to Alibaba Cloud Bailian, we identified a mismatch in the reasoning settings between the primary experiment and the initial repeated trials. We corrected the setting, verified that reasoning mode was enabled, and reran all five trials. We also checked how reasoning tokens, cached input, and preparatory calls were counted.

As shown in Table 20, vulnerability-type classification accuracy ranged from 76.52% to 78.26%, with a mean of 77.57% and a sample standard deviation of 0.73 percentage points. These results are close to the primary result of 90 correctly classified contracts out of 115 (78.26%). Output token counts ranged from 391,389 to 397,076, close to the 394.12K reported for the primary experiment in Table 11. Average trajectory steps, correction attempts, and correction rate were constant across the five trials.

Overall, we observe minimal variance across repeated runs, suggesting that the experimental setup exhibits reasonable stability. Although most experiments were conducted only once due to practical constraints, the observed consistency indicates that the reported results are unlikely to be substantially affected by stochastic variation.

5.4. Practical Guidance

Our results suggest four practical considerations for selecting and configuring software-engineering agents.

(1) Validate the framework–model pairing on the target task. Meng *et al.* Meng et al. (2025) show that both the underlying LLM and agent workflow affect automated bug-fixing performance. Our results in Sections 4.1 and 5.3.1 corroborate this interaction and extend it across software development, vulnerability detection, and program repair: framework rankings change with both the task and foundation model. Practitioners should therefore validate the framework and model jointly on representative target-task instances.

(2) Match reflection feedback to the task and model. SE-AGENT Lin et al. (2025) proposes trajectory-based self-evolution to improve multi-step reasoning. Our results in Section 5.2.1 qualify the generality of this benefit: its repair-specific reflection improves program repair with DeepSeek-v3.1, but its trajectory summaries reduce effectiveness across the three Qwen3-Coder-Flash tasks, whereas GPTSwarm's batch-level optimization Zhuge et al. (2024) produces smaller but broader gains. Reflection should therefore preserve current-instance evidence and be validated for the target task and model.

(3) Align tool actions and output formats with the task workflow. SWE-AGENT Yang et al. (2024) emphasizes that a specialized agent–computer interface can improve repository-level software engineering. Our tool ablation in Section 5.2.2 corroborates this finding in program repair, where repository and patch tools enable GPTSwarm to complete the repair loop. It also qualifies the transferability of specialized tools: the same repair-oriented toolset reduces software-development quality. Tool selection should therefore match the operations and output format required by the target task.

(4) Compare execution overhead and monetary cost separately. Gao *et al.* Gao et al. (2025b) emphasize the trade-off between agent effectiveness, efficiency, and deployment cost. Our results in Sections 4.3 and 5.3.1 extend this observation by showing that token consumption and monetary cost can produce different rankings because API prices vary across models and token categories. Deployment decisions should therefore report outcome quality, execution behavior, token consumption, and monetary cost separately.

These recommendations apply to the evaluated framework implementations, tasks, and model configurations. They provide selection criteria for similar tool-based software-engineering workflows rather than universal rankings of agent architectures.

5.5. Threats to Validity

Internal. Several factors may influence the internal validity of our study. First, we focus on open-source, general-purpose frameworks because their implementations can be inspected and configured across multiple tasks. Proprietary or task-specific agents may achieve stronger task-level results, but their inclusion would limit attribution and reproducibility. Second, no single benchmark covers the full software-engineering lifecycle, so we use established benchmarks for three code-centric tasks. Third, prompt quality may influence the experimental results. To mitigate this potential influence, we based our prompts on well-established task-specific templates rather than designing them from scratch. Systematic evaluation and optimization

of prompt quality remain directions for future research. Finally, most configurations are executed once. The five repeated GPTSwarm–DeepSeek-v3.1 vulnerability-detection runs in Section 5.3.2 show limited variance for that setting, but they do not quantify stochastic variation for every framework–model–task combination.

External. The primary comparison fixes DeepSeek-v3.1 to control model choice. We mitigate this limitation through supplementary experiments with three additional models, reported in Section 5.3.1, which show that the high-level RQ1–RQ3 conclusions generally recur while framework rankings and mechanism effects can change. The software development and program repair robustness analyses use subsets of the original benchmarks; their findings therefore require further validation on the full benchmarks. The findings reflect the models and API prices used at the time of the experiments; framework rankings and monetary costs may change as models and pricing evolve. In addition, released framework implementations sometimes omit components described in their publications or use simplified configurations for code tasks. Our results therefore characterize the tested releases and configurations; we report the actual roles, prompts, tools, and model identifiers to make this scope explicit.

6. Related Work

As discussed in § 3.3, the use of intelligent agents in code-centric software engineering has recently attracted increasing research attention. Numerous frameworks have been introduced to automate a range of tasks, including software development, vulnerability detection, and program repair. However, to the best of our knowledge, this work is the first to conduct a systematic empirical comparison of contemporary agent-based approaches, aiming to provide a deeper understanding of their actual capabilities and limitations across diverse software engineering scenarios.

Agents for Software Engineering. In recent years, a growing number of versatile agent frameworks have emerged and been applied to increasingly complex scientific and productivity tasks Luo et al. (2025). As software engineering represents a core pillar of productivity, enhancing its automation through agent technologies holds substantial promise for improving development efficiency and reliability. Accordingly, many studies have explored the application of intelligent agents to code-centric SE tasks, as discussed in § 3.2. Among them, some agent frameworks are designed for specific tasks, such as ExpeRepair Mu et al. (2025) and SemAgent Pabba et al. (2025), which focus exclusively on program repair. In contrast, general-purpose agent frameworks that are capable of handling diverse complex tasks exhibit stronger generalization and better alignment with real-world software engineering workflows. For instance, to support complex software engineering environments, SWE-Agent Yang et al. (2024) enhances repository-scale operations with a customized agent–computer interface that enables autonomous file manipulation, repository navigation, and test execution. Likewise, Trae Team et al. (2025) provides a research-friendly command-line interface for orchestrating real-world development workflows in a extensible manner. To align agent behavior more closely with human development practices, OpenHands Wang et al. (2025) equips agents with human-like operational abilities, such as interacting with terminals and browsing the web. To better orchestrate agent behaviors, AgentOrchestra Zhang et al. (2025) employs hierarchical task decomposition through a top-level planner directing multiple specialized workers, while OWL Hu et al. (2025) structures multi-agent collaboration following a workforce-style paradigm aimed at large-scale productivity. GPTswarm Zhuge et al. (2024) further advances scalable coordination by leveraging graph-based composition and self-organized swarm intelligence. In addition, to optimize the reasoning trajectory, SE-Agent Lin et al. (2025) proposes a self-evolving multi-step reasoning paradigm that refines solutions through strategy diversification and trajectory optimization. Overall, these developments highlight the growing convergence between agent technologies and software engineering, underscoring the need for a systematic evaluation of their actual capabilities across diverse software engineering tasks.

Benchmarks for Software Engineering Agents. Although intelligent agents have demonstrated the potential to handle multiple phases of the software engineering lifecycle in an end-to-end manner, the community still lacks a comprehensive benchmark capable of holistically evaluating their capabilities across different development stages. Existing evaluations remain fragmented and rely on task-specific datasets.

For software development tasks, widely used benchmarks include HumanEval Chen et al. (2021) and MBPP Austin et al. (2021), which focus on single-function code generation and contain only a few hundred tasks. More complex datasets requiring multi-file development have emerged, such as SRDD Qian et al. (2024), CAASD Zhang, Wang, Dong, Sun, Zhang and Pu (2024), SoftwareDev Hong, Zhuge, Chen, Zheng, Cheng, Zhang, Wang, Wang, Yau, Lin, Zhou, Ran, Xiao, Wu and Schmidhuber (2023), and ProjectDev Nguyen et al. (2025). With over a thousand tasks, SRDD achieves broader task coverage, while the other datasets include only dozens and therefore fall short in representing realistic, large-scale development workflows.

For vulnerability detection, curated datasets such as SySeVR Li, Zou, Xu, Jin, Zhu and Chen (2021) for C/C++, CWE-Bench-Java Li, Dutta and Naik (2024) for Java, and BigBench Srivastava, Rastogi, Rao, Shoeb, Abid, Fisch, Brown, Santoro, Gupta, Garriga-Alonso et al. (2023) for Python enable the evaluation of LLM-based vulnerability prediction. However, these datasets primarily measure static detection performance while overlooking the interactive reasoning and environment orchestration capabilities expected from agents. To the best of our knowledge, LLM-SmartAudit Wei et al. (2025) is the first work that introduces an agent-based approach for vulnerability detection, and its

dataset thus serves as an important reference for evaluating existing agent capabilities in this domain.

For vulnerability repair, classical benchmarks like Defects4J Just et al. (2014), BugsInPy Widyasari, Sim, Lok, Qi, Phan, Tay, Tan, Wee, Tan, Yieh et al. (2020), HumanEval-Java Chen et al. (2021), and QuixBugs Lin et al. (2017) contain small-scale defects localized in specific functions or modules. They are suitable for evaluating traditional automated repair techniques or single LLM capabilities, but insufficient for assessing multi-step repair behaviors in realistic engineering environments. To better align with real-world workflows, recent benchmarks construct repair tasks directly from GitHub issues and their associated repositories, such as SWE-bench Jimenez et al. (2023), SWE-bench Lite SWE-bench Team (2024), SWE-bench Lite-S Xia et al. (2024), and SWE-bench Verified OpenAI (2024), which are widely adopted for evaluating agent-based program repair.

Overall, no single benchmark can comprehensively evaluate the full spectrum of capabilities exhibited by software engineering agents. A unified evaluation framework that integrates diverse datasets across different task categories is needed to fully assess their robustness and generalizability throughout the entire software engineering life-cycle.

Empirical Evaluation of Software Engineering Agents. The recent surge in intelligent software engineering agents has drawn increasing attention from the research community, prompting systematic empirical studies of their capabilities. Researchers have examined agent behavior from multiple perspectives, including architectural design, reasoning processes, and task-specific effectiveness.

Regarding architectural design, studies focus on the intrinsic characteristics of agent systems, such as single-agent versus multi-agent configurations. Gao *et al.* Gao et al. (2025b) conducted an extensive comparison across 15 software engineering tasks, showing that multi-agent systems generally perform better on complex tasks due to long-horizon context tracking and role-specific error correction, whereas single-agent systems often achieve higher efficiency on simpler tasks. They further proposed a hybrid paradigm, cascading requests between multi-agent systems and single-agent systems, to balance accuracy and deployment cost.

In terms of reasoning processes, empirical analyses have investigated agent decision-making workflows. Ceka *et al.* Ceka et al. (2025) present the first systematic study of software engineering agents through the lens of execution traces. They classify decision pathways, identify core components such as bug localization, patch generation, and test generation, and analyze how reasoning about error context influences successful outcomes. These insights illuminate the internal mechanisms that drive agent performance, independent of specific task instances.

Moreover, task-specific evaluations provide complementary evidence of agent effectiveness. Meng *et al.* Meng et al. (2025) conducted fine-grained empirical studies of LLM-based agents for automated bug fixing, assessing fault localization accuracy at file- and line-levels, bug reproduction capabilities, and overall repair performance. Their findings highlight that improving agent effectiveness requires advances in both the underlying LLM models and the design of agent workflows.

Unlike previous studies, which often focus on specific tasks or architectural configurations, our work provides a systematic evaluation of general-purpose software engineering agents across a range of code-centric tasks. We assess not only their effectiveness but also their efficiency and deployment cost, offering a holistic perspective on agent performance throughout the software engineering life-cycle. This comprehensive approach allows us to identify strengths and limitations of current frameworks and informs the design of more capable and practical agent systems.

7. Conclusion

This study presents a comprehensive empirical evaluation of seven general-purpose agent frameworks across three critical code-centric software engineering tasks: software development, vulnerability detection, and program repair. By systematically examining agent performance from the perspectives of effectiveness, efficiency, and overhead, we reveal important insights into the current capabilities and limitations of intelligent agents in real-world software engineering scenarios. Our study reveals important insights into agent frameworks from three key angles. In terms of effectiveness, agents show moderate success: OpenHands balances code quality well in software development, GPTSwarm excels in vulnerability detection accuracy, and program repair remains challenging with only some agents fixing about half of the issues. Efficiency analysis shows that OpenHands, AgentOrchestra, and Trae have the longest individual-run average trajectories in software development, vulnerability detection, and program repair, respectively. In program repair, SE-Agent has the largest cumulative task-execution step total across its three iterations. AgentOrchestra has the longest correction sequences, while OpenHands ranks second, demonstrating stable and convergent behavior. Regarding overhead, program repair incurs the highest mean monetary cost per instance. The token consumption breakdown reveals multi-agent frameworks dominated by planning stages, while single-agent systems concentrate costs on execution and editing activities. These findings collectively highlight the trade-offs and potentials of current frameworks, guiding future improvements in building more capable and resource-aware software engineering agents.

Acknowledgements

This work was supported by the Jiangsu Provincial Science and Technology Major Project (Grant No. BG2025033), by the 2025 Purple Mountain Talent Program for High-Level Innovative and Entrepreneurial Talents under the project “Theoretical Research on Trustworthy Evaluation of Intelligent Software” (Grant No. E6430219G8), and by the Research Fund of Jiangsu Provincial Department of

Science and Technology and Jiangsu Key Laboratory of AI for Industries (Grant No. E6420008G8).

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During the preparation of this work, the authors used ChatGPT to assist with language editing, submission package organization, and compliance checks. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the submitted manuscript.

References

Aggarwal, K.K., 2005. Software engineering. New Age International.

Anthropic, 2024. Building effective agents. URL: https://www.anthropic.com/engineering/building-effective-agents. accessed: September 22, 2026.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al., 2021. Program synthesis with large language models. URL: https://arxiv.org/abs/2108.07732, doi:10.48550/arXiv.2108.07732, arXiv:2108.07732. arXiv preprint.

Basili, V.R., 1989. Software development: A paradigm for the future, in: [1989] Proceedings of the Thirteenth Annual International Computer Software & Applications Conference, IEEE. pp. 471–485. doi:10.1109/CMPSAC.1989.65127.

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D., 2010. A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM 53, 66–75. URL: https://doi.org/10.1145/1646353.1646374, doi:10.1145/1646353.1646374.

Ceka, I., Pujar, S., Ramji, S., Buratti, L., Kaiser, G., Ray, B., 2025. Understanding software engineering agents through the lens of traceability: An empirical study. URL: https://arxiv.org/abs/2506.08311, doi:10.48550/arXiv.2506.08311, arXiv:2506.08311. arXiv preprint.

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2021. Deep learning based vulnerability detection: Are we there yet? IEEE Transactions on Software Engineering 48, 3280–3296. doi:10.1109/TSE.2021.3087402.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.D.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al., 2021. Evaluating large language models trained on code. URL: https://arxiv.org/abs/2107.03374, doi:10.48550/arXiv.2107.03374, arXiv:2107.03374. arXiv preprint.

DeepSeek-AI, 2024. Deepseek-v3 technical report. URL: https://arxiv.org/abs/2412.19437, doi:10.48550/arXiv.2412.19437, arXiv:2412.19437. arXiv preprint.

DeepSeek-AI, 2025. DeepSeek-V3.1 model card. URL: https://huggingface.co/deepseek-ai/DeepSeek-V3.1. accessed: September 27, 2026.

Gao, H.a., Geng, J., Hua, W., Hu, M., Juan, X., Liu, H., Liu, S., Qiu, J., Qi, X., Wu, Y., et al., 2025a. A survey of self-evolving agents: On path to artificial super intelligence. URL: https://arxiv.org/abs/2507.21046, doi:10.48550/arXiv.2507.21046, arXiv:2507.21046. arXiv preprint.

Gao, M., Li, Y., Liu, B., Yu, Y., Wang, P., Lin, C.Y., Lai, F., 2025b. Single-agent or multi-agent systems? why not both? URL: https://arxiv.org/abs/2505.18286, doi:10.48550/arXiv.2505.18286, arXiv:2505.18286. arXiv preprint.

Google, 2025. Google's white paper on agents. URL: https://drive.google.com/file/d/1oEjiRCTbd54aSdB_eEe3UShxLBWK9xkt/view?pli=1. accessed: September 22, 2026.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S.K.S., Lin, Z.H., Zhou, L., Ran, C., Xiao, L., Wu, C., Schmidhuber, J., 2023. Metagpt: Meta programming for a multi-agent collaborative framework, in: International Conference on Learning Representations. URL: https://api.semanticscholar.org/CorpusID:265301950.

Hu, M., Zhou, Y., Fan, W., Nie, Y., Xia, B., Sun, T., Ye, Z., Jin, Z., Li, Y., Chen, Q., Zhang, Z., Wang, Y., Ye, Q., Ghanem, B., Luo, P., Li, G., 2025. Owl: Optimized workforce learning for general multi-agent assistance in real-world task automation. URL: https://arxiv.org/abs/2505.23885, doi:10.48550/arXiv.2505.23885, arXiv:2505.23885. arXiv preprint.

JACKSON, V., VASILESCU, B., RUSSO, D., RALPH, P., PRIKLADNICKI, R., IZADI, M., D'ANGELO, S., INMAN, S., ANDRADE, A., VAN DER HOEK, A., 2025. The impact of generative ai on creativity in software development: A research agenda. ACM Transactions on Software Engineering and Methodology doi:10.1145/3708523.

Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., Narasimhan, K., 2023. Swe-bench: Can language models resolve real-world github issues? URL: https://arxiv.org/abs/2310.06770, doi:10.48550/arXiv.2310.06770, arXiv:2310.06770. arXiv preprint.

Just, R., Jalali, D., Ernst, M.D., 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs, in: Proceedings of the 2014 international symposium on software testing and analysis, pp. 437–440. doi:10.1145/2610384.2628055.

Le Goues, C., Pradel, M., Roychoudhury, A., 2019. Automated program repair. Communications of the ACM 62, 56–65. doi:10.1145/3318162.

Li, Z., Dutta, S., Naik, M., 2024. Iris: Llm-assisted static analysis for detecting security vulnerabilities. URL: https://arxiv.org/abs/2405.17238, doi:10.48550/arXiv.2405.17238, arXiv:2405.17238. arXiv preprint.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing 19, 2244–2258. doi:10.1109/TDSC.2021.3051525.

Lin, D., Koppel, J., Chen, A., Solar-Lezama, A., 2017. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge, in: Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity, pp. 55–56. doi:10.1145/3135932.3135941.

Lin, J., Guo, Y., Han, Y., Hu, S., Ni, Z., Wang, L., Chen, M., Liu, H., Chen, R., He, Y., Jiang, D., Jiao, B., Hu, C., Wang, H., 2025. Se-agent: Self-evolution trajectory optimization in multi-step reasoning with llm-based agents. URL: https://arxiv.org/abs/2508.02085, doi:10.48550/arXiv.2508.02085, arXiv:2508.02085. arXiv preprint.

Liu, J., Wang, K., Chen, Y., Peng, X., Chen, Z., Zhang, L., Lou, Y., 2024. Large language model-based agents for software engineering: A survey. URL: https://arxiv.org/abs/2409.02977, doi:10.48550/arXiv.2409.02977, arXiv:2409.02977. arXiv preprint.

Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F., 2019. Tbar: Revisiting template-based automated program repair, in: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, pp. 43–54. doi:10.1145/3293882.3330577.

Luo, J., Zhang, W., Yuan, Y., et al., 2025. Large language model agent: A survey on methodology, applications and challenges. URL: https://arxiv.org/abs/2503.21460, doi:10.48550/arXiv.2503.21460, arXiv:2503.21460. arXiv preprint.

Meng, X., Ma, Z., Gao, P., Peng, C., 2025. An empirical study on llm-based agents for automated bug fixing. URL: https://arxiv.org/abs/2411.10213, doi:10.48550/arXiv.2411.10213, arXiv:2411.10213. arXiv preprint.

Mu, F., Wang, J., Shi, L., Wang, S., Li, S., Wang, Q., 2025. Experepair: Dual-memory enhanced llm-based repository-level program repair. URL: https://arxiv.org/abs/2506.10484, doi:10.48550/arXiv.2506.10484, arXiv:2506.10484. arXiv preprint.

Nguyen, M.H., Chau, T.P., Nguyen, P.X., Bui, N.D., 2025. Agilecoder: Dynamic collaborative agents for software development based on agile methodology, in: 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge), IEEE. pp. 156–167. doi:10.1109/Forge66646.2025.00026.

OpenAI, 2024. Introducing swe-bench verified. URL: https://openai.com/index/introducing-swe-bench-verified/. accessed: September 22, 2026.

Pabba, A., Mathai, A., Chakraborty, A., Ray, B., 2025. Semagent: A semantics aware program repair agent. URL: https://arxiv.org/ab

s/2506.16650, doi:10.48550/arXiv.2506.16650, arXiv:2506.16650. arXiv preprint.

Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z., Sun, M., 2024. Chatdev: Communicative agents for software development, in: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 15174–15186. URL: https://aclanthology.org/2024.acl-long.810/, doi:10.18653/v1/2024.acl-long.810.

Qwen Team, 2025a. Qwen3-Coder-30B-A3B-Instruct model card. URL: https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct. accessed: September 27, 2026.

Qwen Team, 2025b. Qwen3-Coder: Agentic coding in the world. URL: https://qwenlm.github.io/blog/qwen3-coder/. accessed: September 27, 2026.

Srivastava, A., Rastogi, A., Rao, A., Shoeb, A.A.M., Abid, A., Fisch, A., Brown, A.R., Santoro, A., Gupta, A., Garriga-Alonso, A., et al., 2023. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. Transactions on machine learning research URL: https://openreview.net/forum?id=uyTL5Bvosj.

SWE-bench Team, 2024. Swe-bench lite. URL: https://www.swebench.com/lite.html. accessed: September 22, 2026.

Team, T.R., Gao, P., Tian, Z., Meng, X., Wang, X., Hu, R., Xiao, Y., Liu, Y., Zhang, Z., Chen, J., Gao, C., Lin, Y., Xiong, Y., Peng, C., Liu, X., 2025. Trae agent: An llm-based agent for software engineering with test-time scaling. URL: https://arxiv.org/abs/2507.23370, doi:10.48550/arXiv.2507.23370, arXiv:2507.23370. arXiv preprint.

Wang, X., Li, B., Song, Y., Xu, F.F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., Tran, H.H., Li, F., Ma, R., Zheng, M., Qian, B., Shao, Y., Muennighoff, N., Zhang, Y., Hui, B., Lin, J., Brennan, R., Peng, H., Ji, H., Neubig, G., 2025. Openhands: An open platform for ai software developers as generalist agents. URL: https://arxiv.org/abs/2407.16741, doi:10.48550/arXiv.2407.16741, arXiv:2407.16741. arXiv preprint.

Wei, Z., Sun, J., Sun, Y., Liu, Y., Wu, D., Zhang, Z., Zhang, X., Li, M., Liu, Y., Li, C., et al., 2025. Advanced smart contract vulnerability detection via llm-powered multi-agent systems. IEEE Transactions on Software Engineering 51, 2830–2846. doi:10.1109/TSE.2025.3597319.

Widyasari, R., Sim, S.Q., Lok, C., Qi, H., Phan, J., Tay, Q., Tan, C., Wee, F., Tan, J.E., Yieh, Y., et al., 2020. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies, in: Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp. 1556–1560. doi:10.1145/3368089.3417943.

Xia, C.S., Deng, Y., Dunn, S., Zhang, L., 2024. Agentless: Demystifying llm-based software engineering agents. URL: https://arxiv.org/abs/2407.01489, doi:10.48550/arXiv.2407.01489, arXiv:2407.01489. arXiv preprint.

Yang, J., Jimenez, C.E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., Press, O., 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. URL: https://arxiv.org/abs/2405.15793, doi:10.48550/arXiv.2405.15793, arXiv:2405.15793. arXiv preprint.

Yu, Z., Zhang, H., Zhao, Y., Huang, H., Yao, M., Ding, K., Zhao, J., 2025. Orcaloca: An llm agent framework for software issue localization. URL: https://arxiv.org/abs/2502.00350, doi:10.48550/arXiv.2502.00350, arXiv:2502.00350. arXiv preprint.

Z.ai, 2025. GLM-4.7 model card. URL: https://huggingface.co/zai-org/GLM-4.7. accessed: September 27, 2026.

Zhang, Q., Fang, C., Ma, Y., Sun, W., Chen, Z., 2023. A survey of learning-based automated program repair. ACM Transactions on Software Engineering and Methodology 33, 1–69. doi:10.1145/3631974.

Zhang, S., Wang, J., Dong, G., Sun, J., Zhang, Y., Pu, G., 2024. Experimenting a new programming practice with llms. URL: https://arxiv.org/abs/2401.01062, doi:10.48550/arXiv.2401.01062, arXiv:2401.01062. arXiv preprint.

Zhang, W., Zeng, L., Xiao, Y., Li, Y., Cui, C., Zhao, Y., Hu, R., Liu, Y., Zhou, Y., An, B., 2025. Agentorchestra: Orchestrating hierarchical multi-agent intelligence with the tool-environment-agent(tea) protocol. URL: https://arxiv.org/abs/2506.12508, doi:10.48550/arXiv.2506.12508, arXiv:2506.12508. arXiv preprint.

Zhuge, M., Wang, W., Kirsch, L., Faccio, F., Khizbullin, D., Schmidhuber, J., 2024. Gptswarm: Language agents as optimizable graphs, in: Forty-first International Conference on Machine Learning, pp. 62743–62767. URL: https://proceedings.mlr.press/v235/zhuge24a.html.