Agent-Based Ensemble Reasoning for Repository-Level Issue Resolution

Zhao Tian
School of Computer Software, Tianjin University
Tianjin, China
tianzhao@tju.edu.cn

Pengfei Gao
ByteDance
Beijing, China
gaopengfei.se@bytedance.com

Junjie Chen*
School of Computer Software, Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Chao Peng*
ByteDance
Beijing, China
pengchao.x@bytedance.com

Abstract

Software issue resolution is a critical challenge in software engineering and has garnered increasing attention in recent years. With the rapid advancement of large language models (LLMs), substantial progress has been made in addressing real-world software engineering tasks. Recent studies have introduced ensemble reasoning techniques to enhance the performance of LLM-based issue resolution. However, existing prompting-based methods still face limitations in effectively exploring large ensemble spaces and lack the capacity for repository-level understanding, both of which constrain their overall effectiveness. In this paper, we propose *EnAgent*, the first agent-based ensemble reasoning approach for repository-level issue resolution. *EnAgent* formulates our goal as an optimal solution search problem and addresses two key challenges, i.e., large ensemble spaces and repository-level understanding, through modular agents for generation, pruning, and selection. We conduct extensive experiments using three leading LLMs on the widely-adopted SWE-bench benchmark, comparing *EnAgent* against four state-of-the-art ensemble reasoning techniques. Experimental results demonstrate that *EnAgent* consistently achieves superior performance, with an average improvement of 10.22% over all baselines in terms of Pass@1. *EnAgent has been integrated into Trae Agent, driving it to achieve first place on the SWE-bench Verified leaderboard as of January 2026, with a notable Pass@1 score of 78.80%.*

CCS Concepts

• **Software and its engineering** → **Automatic programming**; • **Computing methodologies** → *Natural language processing*; *Neural networks.*

Keywords

Software Issue Resolution, Large Language Model, Agent, Ensemble Reasoning

*Junjie Chen and Chao Peng are the corresponding authors.

Please use nonacm option or ACM Engage class to enable CC licenses
This work is licensed under a Creative Commons Attribution 4.0 International License.
ICSE '26, April 12–18, 2026, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787761

ACM Reference Format:
Zhao Tian, Pengfei Gao, Junjie Chen, and Chao Peng. 2026. Agent-Based Ensemble Reasoning for Repository-Level Issue Resolution. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787761

1 Introduction

Software issue resolution refers to the automated handling of newly reported bugs or feature requests during software development, aiming to ensure correct and reliable system behavior [19, 23, 27]. In recent years, this task has garnered growing interest from both academia and industry, driven by its potential to reduce developer burden and significantly enhance productivity [58, 64]. Large language models (LLMs) have shown remarkable capabilities in function-level code tasks such as code generation [49, 50] and automated program repair [5, 20]. Despite this progress, LLMs continue to face major challenges in resolving complex repository-level software issues. For example, GPT-4o achieves a resolution rate of 92.70% on the function-level HumanEval benchmark [33], but only 11.99% on the repository-level SWE-bench benchmark [58]. This performance gap underscores the difficulty of real-world issue resolution, which often requires a global understanding of large codebases, cross-file reasoning, and the detection of subtle, multi-component bugs [43, 57]. These challenges limit the practical adoption of LLM-based techniques in real-world software engineering and pose a risk to software quality if unresolved. Bridging this gap remains a critical research goal toward realizing robust, scalable, and reliable automated software issue resolution.

Recently, many techniques have been proposed to enhance the performance of LLMs in issue resolution [54, 58]. These methods primarily focus on improving patch generation, by carefully designing agent architectures and integrating external tools to assist LLMs in producing correct fixes. For example, OpenDevin[54] incorporates a planning module based on issue descriptions and coordinates multiple tools within an agent system to guide patch generation. Recent studies [8, 31, 60] have revealed a key property of LLMs: although the overall resolution rate remains stable across repeated runs, the set of issues successfully resolved varies across executions. This variability arises from the large action space and complex reasoning trajectories required for repository-level issue resolution, which often lead LLMs to pursue divergent solution paths. This

observation has led to the emergence of a new direction, i.e., **ensemble reasoning**, which builds on patch generation approaches by producing multiple independent candidate patches and selecting a consensus solution that is more robust than any individual output.

Existing ensemble reasoning methods are prompting-based, relying on LLMs not only to generate candidate patches, but also to perform selection among them. For example, Augment[8] uses the LLM-as-a-judge [53] paradigm to prompt the model to compare each candidate against the issue description and choose the best match. While prompting-based ensemble reasoning techniques have shown promise, they face fundamental limitations when applied to complex repository-level issue resolution. These limitations become especially evident in the face of two key challenges: First, prompting-based methods *struggle with the optimal solution search in large ensemble spaces*. As the number of candidate patches grows, subtle behavioral differences between them become increasingly difficult to distinguish using LLMs alone. Since patch selection is typically performed through a single prompt, surface-level syntactic comparisons often fail to capture deeper semantic nuances. Second, these methods *lack the capacity for repository-level understanding*, which is essential for accurately selecting patches in real-world software systems. Issues and fixes frequently span multiple files and modules, requiring cross-file reasoning, contextual awareness, and verification of patch correctness within the broader codebase. Prompting-based approaches operate in a stateless, single-turn manner and lack persistent memory or tool integration, making it difficult to track dependencies, execute validation steps, or build a coherent global view of the repository.

To overcome the limitations of prompting-based ensemble reasoning approaches, we propose the first agent-based ensemble reasoning approach for repository-level issue resolution, called ***EnAgent*** (**En**semble **Agent**). *EnAgent* is designed to enhance LLM-based issue resolution by formulating it as an optimal solution search problem [10, 25] and addressing the two core challenges through a modular agent-based architecture. It consists of three key components: *patch generation*, *patch pruning*, and *patch selection*. A coder agent first generates diverse candidate patches in parallel to improve the ensemble diversity in line with prior work [11, 60]. To mitigate the difficulty of optimal solution search, *EnAgent* applies a hierarchical pruning strategy that combines patch deduplication and regression testing strategies to eliminate redundant or faulty patches, thereby reducing the ensemble space while preserving promising candidates. To address the need for repository-level understanding, *EnAgent* employs a selector agent that simulates a real-world program comprehension process [15, 18], encompassing both static review and dynamic verification. Specifically, this agent iteratively gathers and analyzes relevant code snippets (such as those referenced in the issue description, modified by patches, or related through dependencies) to build static understanding, and collects execution traces from automatically generated tests to build dynamic understanding. Based on these insights, the agent applies a majority voting strategy to select the most plausible patch. Due to its generalizable and modular design, *EnAgent* potentially provides a solid foundation for advancing ensemble reasoning in other complex software engineering tasks.

We conduct extensive experiments to evaluate the performance of *EnAgent* using three LLMs (Gemini 2.5 Pro [14], Claude 3.7 Sonnet [2], and GPT-4.1 [38]) on the widely-used software issue resolution benchmark (i.e., SWE-bench [27]). Our results show that *EnAgent* consistently and significantly outperforms four state-of-the-art ensemble reasoning baselines (i.e., Augment, Augment w/ Pruning, DeiBase, and DeiBase w/ Pruning) across all evaluation settings, demonstrating its capability to enhance LLM-based software issue resolution. In particular, *EnAgent* achieves a Pass@1 improvement ranging from 5.83% to 14.60%, where Pass@1 denotes the proportion of generated patches that pass all golden tests. We further investigate the impact of ensemble size, a critical hyperparameter in ensemble reasoning. The results reveal that *EnAgent* not only outperforms all baselines at each ensemble size but also continues to improve with larger ensemble sizes, while baseline methods exhibit varying degrees of performance degradation. Additionally, we conduct comprehensive ablation studies by constructing five variants of *EnAgent*, each omitting a major component (i.e., patch pruning, patch deduplication, regression testing, selector agent, and majority voting). The results confirm the effectiveness and necessity of each part of our framework.

The main contributions of this paper are as follows:

- **Novel Approach**: We propose *EnAgent*, the first agent-based ensemble reasoning framework for repository-level issue resolution. *EnAgent* formulates our goal as an optimal solution search problem and addresses two key challenges, i.e., large ensemble spaces and repository-level understanding, through modular agents for generation, pruning, and selection.
- **Extensive Evaluation**: We conduct extensive experiments on the SWE-bench benchmark using three state-of-the-art LLMs, comparing *EnAgent* with four leading baselines. Results show that *EnAgent* consistently improves Pass@1 by 5.83%~14.60%, demonstrating its robust effectiveness across settings.
- **Practical Impact**: *EnAgent* has been integrated into the Trae Agent [22], driving it to achieve first place on the SWE-bench Verified leaderboard as of January 2026, with a notable Pass@1 of 78.80%. Our associated GitHub repository has attracted over 10,000 stars, reflecting strong community interest and adoption.
- **Data Availability**: We publicly release the source code at [26] to support reproducibility and facilitate future research.

2 Motivation

Recently, DeepMind proposes **Test-time Scaling** [47], demonstrating that leveraging ensemble learning and increasing inference resources can substantially improve performance in mathematical reasoning tasks. Subsequent studies [17, 31, 60] extend this paradigm to software engineering tasks (e.g., code generation and issue resolution). The underlying intuition is that, although an LLM's average resolution rate remains relatively stable across repeated runs, the specific issues it successfully resolves vary across executions.

Figure 1 shows the performance of two ensemble reasoning techniques (Augment and DeiBase) on the SWE-bench Verified using three LLMs (Gemini 2.5 Pro, Claude 3.7 Sonnet, and GPT-4.1). Particularly, the Mixture setting refers to a round-robin combination of these three LLMs for generating candidate patches, aiming to enhance the diversity of the ensemble space. Taking Figure 1(1) as an example, the x-axis denotes the ensemble size N (i.e., the

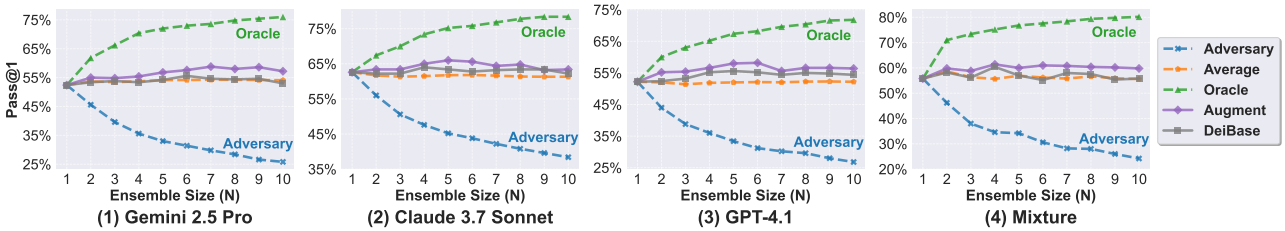


Figure 1: Influence of ensemble size on the effectiveness of existing ensemble reasoning techniques in terms of Pass@1 (↑)

number of candidate patches generated by Gemini 2.5 Pro), while the y-axis shows the Pass@1 (i.e., the proportion of patches that pass all golden tests). Following prior work [60], we report three additional metrics: (1) *Oracle*, representing the best-case performance where the correct patch is always selected if present among the N candidates; (2) *Adversary*, representing the worst-case performance where an issue is only considered solved if all N candidates are correct; and (3) *Average*, representing the expected performance of randomly selecting one patch from the N candidates.

First, we observe that as the ensemble size increases, the performance of *Average* remains relatively stable, while the *Oracle* value consistently improves. Notably, when the ensemble size reaches 10, *Oracle* achieves an average improvement of 20.80% over *Average* in terms of Pass@1. This observation strongly supports previous findings that although the overall resolution rate remains stable across repeated runs, the set of issues successfully resolved varies across executions [8, 31, 60]. This finding reflects the high diversity in the candidate patches generated by LLMs. *It motivates the significant potential of ensemble reasoning techniques as a promising direction for enhancing LLM-based software issue resolution.*

Moreover, we observe that as the ensemble size increases, the *Adversary* value consistently declines. Specifically, when the ensemble size reaches 10, *Adversary* reduces by 27.01% on average compared to the *Average* in terms of Pass@1. This indicates that although the theoretical upper bound of ensemble performance (i.e., *Oracle* = 20.80%) improves as the number of candidate patches increases, the theoretical lower bound (i.e., *Adversary* = 27.01%) deteriorates more significantly. This widening gap underscores the increasing difficulty of ensemble reasoning techniques as the ensemble size increases. Indeed, as shown in Figure 1, the performance of Augment and DeiBase initially improves with larger ensemble sizes but subsequently degrades. *It motivates the necessity of our patch pruning component, which eliminates redundant and faulty patches, thereby reducing the ensemble space while preserving promising candidates.*

Additionally, on average, both Augment and DeiBase outperform the random selection strategy (i.e., *Average*) by 3.17% and 1.06% in terms of Pass@1, respectively. These results also confirm the potential of ensemble reasoning techniques to enhance LLM performance in software issue resolution. However, in comparison, *Oracle* achieves an average improvement of 29.49% and 36.29% over Augment and DeiBase, respectively. This substantial performance gap underscores the limitations of existing ensemble reasoning methods in accurately selecting the correct patch and highlights significant potential for improvement. *It motivates the necessity of designing a more effective patch selection component to further improve LLM performance in software issue resolution.*

3 Approach

3.1 Overview

In this paper, we propose a novel ensemble reasoning approach, *EnAgent*, designed to enhance the performance of LLMs in software issue resolution. To the best of our knowledge, *EnAgent* is the first agent-based ensemble reasoning approach for repository-level issue resolution. Due to its generalizable and modular design, *EnAgent* potentially provides a solid foundation for advancing ensemble reasoning in other complex software engineering tasks. Figure 2 illustrates the overall architecture of *EnAgent*, which comprises three main components: (1) **Patch generation component** (Section 3.2) employs a coder agent to generate diverse candidate patches in parallel to improve the ensemble diversity; (2) **Patch pruning component** (Section 3.3) performs a hierarchical patch pruning that combines patch deduplication and regression testing strategies to eliminate redundant or faulty patches, thereby reducing the ensemble space while preserving promising candidates; (3) **Patch selection component** (Section 3.4) employs a selector agent that simulates a real-world program comprehension process [15, 18], constructing repository-level understanding by combining static review with dynamic verification, and ultimately selecting the correct patch through a majority voting strategy. In the following, we provide a detailed description of each component in *EnAgent*.

3.2 Patch Generation

Generate candidate patch. As illustrated in Figure 2, this component utilizes an LLM-based coder agent equipped with specialized agent tools (i.e., View, Create, Insert, StrReplace, UndoEdit, and Bash) to autonomously generate diverse candidate patches in parallel for each issue. Inspired by prior research [35, 43], the coder agent's system prompt defines a structured process comprising the following tasks: (1) analyzing the issue description to understand the target problem; (2) exploring the codebase to locate files relevant to the issue; (3) reproducing the bug to validate its manifestation; (4) diagnosing the root cause through code inspection; (5) generating a code patch to resolve the identified bug; (6) rerunning reproduction tests to verify patch correctness; and (7) summarizing the working process, emulating a realistic commit message. Finally, the coder agent effectively generates high-quality candidate patches.

Improve ensemble diversity. To further enhance the diversity of generated candidate patches, *EnAgent* employs a high-temperature sampling strategy [42, 66] across multiple independent runs of the coder agent, thereby improving the ensemble diversity. The coder agent is instantiated with three state-of-the-art LLMs (Gemini 2.5 Pro, Claude 3.7 Sonnet, and GPT-4.1), which have been extensively evaluated in our experiments. Furthermore, we introduce a Mixture setting, in which the three LLMs are used in a round-robin

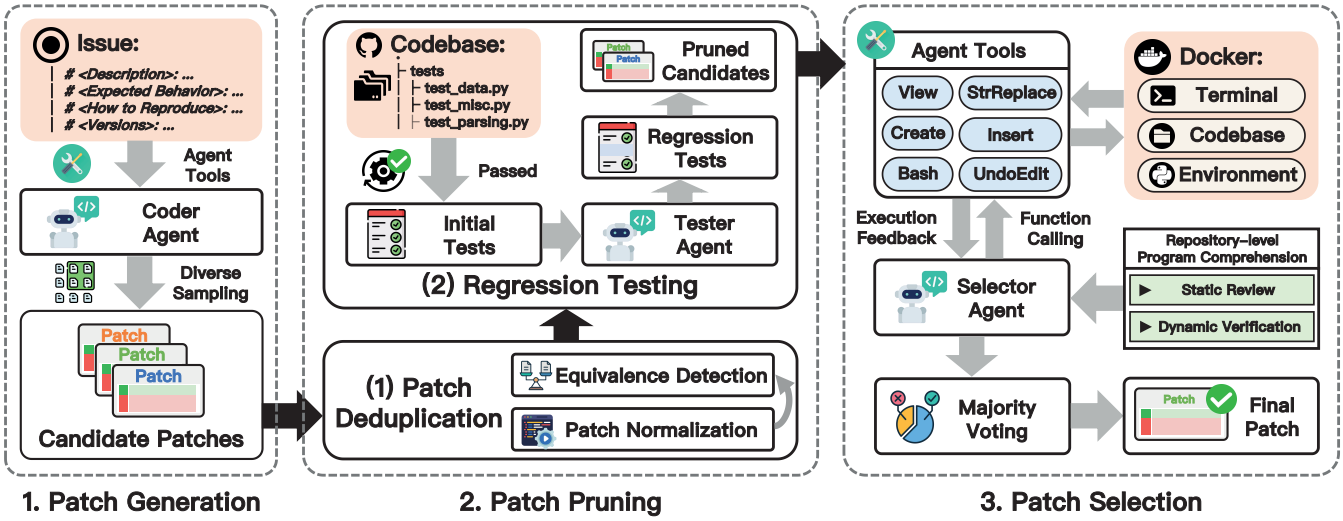


Figure 2: Overview of *EnAgent*

fashion to generate patches, further increasing the diversity of candidates. The generation process terminates once the number of generated candidate patches reaches a pre-defined ensemble size N. This ensemble size is a critical hyper-parameter in all ensemble reasoning techniques, and its impact on performance is discussed in Section 5.2. Overall, this patch generation component enables *EnAgent* to produce diverse candidate patches, whose union consistently outperforms any individual generation.

3.3 Patch Pruning

After generating candidate patches, *EnAgent* performs a hierarchical patch pruning to reduce the ensemble space, thereby enhancing the effectiveness of the subsequent patch selection. As discussed in Section 2, increasing the ensemble size raises the theoretical upper bound of performance but also lowers the lower bound more sharply, making accurate patch selection more challenging. Existing prompting-based ensemble methods struggle with the optimal solution search in large ensemble spaces. Particularly, our experiments show that, on average, 40% of the candidate patches generated by LLMs are either redundant or incorrect. To mitigate the difficulty of optimal solution search, we design a hierarchical pruning method that combines patch deduplication and regression testing strategies to eliminate redundant or faulty patches, thereby reducing the ensemble space while preserving promising candidates. Our experimental results (Section 5) confirm that both patch deduplication and regression testing strategies significantly enhance the performance of *EnAgent*, and benefit other ensemble reasoning techniques as well. Notably, we first introduce a novel patch pruning strategy for ensemble reasoning, opening a promising direction for future research. The following sections present the design and implementation of patch deduplication and regression testing strategies.

Perform patch deduplication. The primary objective of the patch deduplication strategy is to eliminate redundant candidate patches, thereby reducing the ensemble space. Inspired by prior work on equivalent mutant detection [30, 39], we eliminate redundant candidates based on patch normalization and equivalence detection. Specifically, we first implement a patch parser leveraging the Python `unidiff` [4] package to convert raw patches into structured representations, facilitating precise and reliable patch normalization. We then perform patch normalization to remove semantically irrelevant elements (such as extra spaces, line breaks, and comments) without altering program behavior, yielding a normalized form for each patch. In particular, patches that fail to parse due to syntax errors are deemed invalid and discarded. For subsequent equivalence detection, candidate patches that produce identical normalized representations are considered semantically equivalent, and redundant patches are eliminated accordingly. We acknowledge that, due to the undecidability of program equivalence [29, 52], not all redundant patches can be reliably identified and removed. However, our comprehensive empirical evaluation (Sections 5.3 and 5.4) demonstrates that our proposed strategy effectively reduces redundancy by an average of 28.90% and significantly enhances the practical performance of ensemble reasoning techniques. Furthermore, we will discuss potential enhancements to the patch deduplication process in Section 6.4 to further enhance overall effectiveness.

Perform regression testing. The primary objective of the regression testing strategy is to eliminate faulty candidate patches that fail to preserve existing functionality, thereby further reducing the ensemble space. Specifically, *EnAgent* employs a tester agent that automatically retrieves and executes all available regression tests from the original codebase, and retains those that pass as the initial regression tests. That is, regression tests that fail due to missing dependencies or project-specific setups are automatically excluded, leaving the reliable initial tests. However, not all passing tests are classified as regression tests, as issue resolution may justifiably alter certain existing functionalities, potentially causing some tests to fail. To address this, the tester agent further refines the initial tests by prompting the LLM to identify a smaller, relevant subset most likely to represent true regression tests, which are then designated as the final regression tests. This design enables *EnAgent* to effectively handle diverse, large-scale codebases while maintaining test executability, reducing the cost of executing extensive test suites, and minimizing noise from irrelevant cases.

<Codebase Path>
<Github Issue Description>
<Candidate Patches>
<Agent Tools>
 # ROLE:
Act as an expert code selector. Given a codebase, an github issue and N candidate patches proposed by your colleagues, your responsibility is to select the correct one to solve the issue.
 # WORK PROCESS:
You are given a software issue and multiple candidate patches. Your goal is to identify the patch that correctly resolves the issue. Follow these steps methodically:
1. Understand the Issue and Codebase: Carefully read the issue description to comprehend the problem. You may need to examine the codebase for context, including: (1) Code referenced in the issue description; (2) The original code modified by each patch; (3) Unchanged parts of the same file; (4) Related files, functions, or modules that interact with the affected code.
2. Analyze the Candidate Patches: For each patch, analyze its logic and intended fix. Consider whether the changes align with the issue description and coding conventions.
3. Validate Functionality (Optional but Recommended): If needed, write and run unit tests to evaluate the correctness and potential side effects of each patch.
4. Select the Best Patch: Choose the patch that best resolves the issue with minimal risk of introducing new problems.
 # FINAL REPORT:
If you have successfully selected the correct patch, submit your answer in the following format:
Status: succeed ### Result: Patch-x ### Analysis: [Explain why Patch-x is correct.]

Figure 3: Prompt template for the selector agent

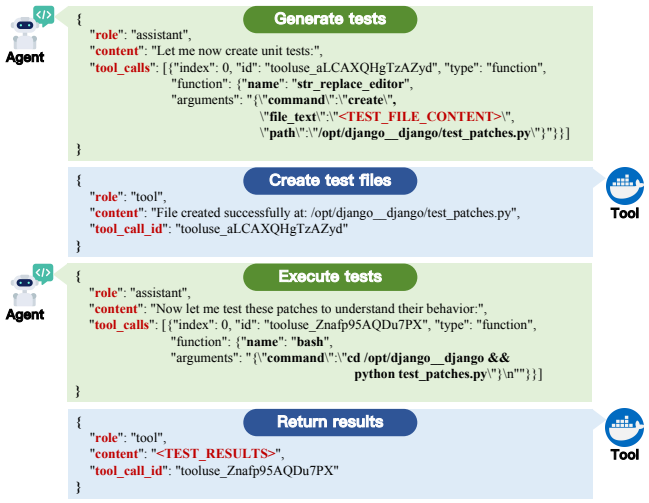


Figure 4: An example of agent to execute generated tests

Subsequently, each candidate patch is applied to the original codebase, and the tester agent executes the final regression tests individually on each patch version. Patches that fail any regression test are discarded, and only those that pass all tests proceed to the patch selection stage. To further mitigate the impact of noisy or unreliable tests, *EnAgent* adopts a conservative fallback strategy: if all candidate patches for an issue fail the selected regression tests, *EnAgent* retains the full candidate set to avoid prematurely discarding potentially correct patches. We acknowledge that the regression tests extracted from the original codebase may contain inaccuracies, potentially introducing false positives during this pruning process. Nonetheless, our experiments (Section 6.2) empirically demonstrate that the regression testing strategy effectively eliminates faulty patches, achieving a low FN rate (i.e., correct patches mistakenly discarded) of only 3.69%. Furthermore, the ablation studies (Section 5.3) confirm that this strategy substantially enhances the practical performance of ensemble reasoning techniques, yielding an average improvement of 3.42%.

3.4 Patch Selection

The primary goal of the patch selection component is to accurately identify the correct patch from the pruned candidates. Real-world issues and fixes frequently span multiple files and modules, necessitating cross-file reasoning, contextual awareness, and verification of patch correctness within the broader codebase. Therefore, effective patch selection requires an accurate repository-level understanding. However, existing prompting-based ensemble reasoning approaches are typically stateless and single-turn, lacking persistent memory and tool integration. These limitations hinder their ability to track dependencies, perform validation steps, and build a coherent, global understanding of the repository.

Obtain repository-level understanding. To address the challenge of repository-level understanding, *EnAgent* introduces a selector agent that simulates a real-world program comprehension process [15, 18]. Figure 3 shows the detailed prompt template for the selector agent. It iteratively gathers and analyzes relevant code snippets (such as those referenced in the issue description, modified by patches, or related through dependencies) to build the static understanding, and collects execution traces from automatically generated tests to build the dynamic understanding. To achieve this, we equip the selector agent with tools that provide access to the complete codebase and execution environment for each issue. Specifically, *EnAgent* integrates six agent tools: View (to list directory contents or view file contents), Create (to create new files), Insert (to edit files via text insertion), StrReplace (to edit files via string replacement), UndoEdit (to revert file modifications), and Bash (to execute shell commands). In addition to gathering and analyzing relevant code snippets for static understanding, the selector agent leverages these tools to generate and execute unit tests, thereby facilitating dynamic understanding, as illustrated in Figure 4. Specifically, the selector agent interacts with these tools through function calling [1, 34], which are automatically translated into concrete shell commands and executed within a tailored Docker-based environment. The execution results are returned in JSON format, allowing the agent to analyze the execution feedback and refine its understanding. The selector agent iteratively leverages these tools until it obtains sufficient repository-level understanding to make a final patch selection. Additionally, to control computational overhead and prevent excessive token usage, we impose an upper bound of 30 interaction rounds between the selector agent and the agent tools.

Implement majority voting. To further mitigate the risk of LLM hallucinations and enhance solution consistency, we implement a majority voting strategy in the selector agent. Specifically, given N candidate patches, the selector agent is executed in parallel for N iterations, and the patch receiving the highest number of votes is selected as the final output. To improve efficiency, if the first $\lceil \frac{N}{2} \rceil$ votes are unanimous, this consensus patch is immediately returned as the final result, and the remaining $(N - \lceil \frac{N}{2} \rceil)$ executions are skipped. In cases where the N votes are evenly distributed across multiple candidate patches, potentially indicating the presence of several correct solutions, *EnAgent* randomly selects one from among the top-voted candidates. The impact of this majority voting strategy is further analyzed in Section 5.3. Through this patch selection component, *EnAgent* can effectively select the most plausible patch to improve LLM-based issue resolution.

4 Evaluation Design

Our study aims to address the following research questions (RQs):

- **RQ1**: How does *EnAgent* perform in terms of effectiveness and efficiency compared to the state-of-the-art techniques?
- **RQ2**: How do hyper-parameters affect *EnAgent*'s effectiveness?
- **RQ3**: How does each main component in *EnAgent* contribute to the overall effectiveness?
- **RQ4**: How does the ensemble space affect the effectiveness of patch selection?
- **RQ5**: How orthogonal is *EnAgent* to existing patch generation techniques?

4.1 Benchmarks

We evaluate the effectiveness of *EnAgent* on SWE-bench [27], a widely-used benchmark for automated software issue resolution. Following existing work [55, 57, 61], our evaluation focuses on its verified version. For each issue, all evaluated techniques receive only the issue description and the original codebase as input. SWE-bench also provides some golden tests for each GitHub issue to evaluate whether the generated patch solves the corresponding issue. Note that all regression tests used by *EnAgent* are part of the original codebase, ensuring a fair and realistic evaluation setting.

4.2 Metrics

We use **Pass@1** to assess the effectiveness of the studied ensemble reasoning techniques. It measures the functional correctness of a generated patch by determining whether it successfully resolves the given software issue. For each issue, SWE-bench provides golden tests to verify patch correctness. In our evaluation, the ensemble reasoning technique selects a single patch per issue; if this patch passes all associated tests in the SWE-bench, the issue is deemed successfully resolved. It is worth noting that Pass@1 is a stringent evaluation criterion, making improvements to this metric both technically challenging and practically meaningful [43, 57]. In addition, we assess the efficiency of each technique by measuring the **average time overhead** and **average token overhead**, where the latter includes both input and output token overhead.

4.3 Compared Techniques

To comprehensively evaluate the performance of *EnAgent*, we compare it with state-of-the-art ensemble reasoning techniques:

- **Augment** [11]: implements a prompting-based ensemble technique inspired by the LLM-as-a-judge paradigm. It prompts the LLM to compare each candidate against the issue description and choose the best match.
- **DeiBase** [60]: prompts the LLM to generate detailed justifications and assign confidence scores to each candidate patch, selecting the one with the highest score as the final patch.

To further demonstrate the effectiveness and generality of the patch pruning component in our *EnAgent*, we integrate it into both ensemble reasoning techniques, resulting in two enhanced baselines: **Augment w/ Pruning** and **DeiBase w/ Pruning**. To ensure a fair comparison, all ensemble reasoning techniques are evaluated using the same set of candidate patches generated by our patch generation component.

4.4 Implementation Details

To evaluate the performance of *EnAgent*, we employ three state-of-the-art LLMs: Gemini 2.5 Pro [14] (version *gemini-2.5-pro-preview-06-05*), Claude 3.7 Sonnet [2] (version *claude-3-7-sonnet-20250219*), and GPT-4.1 [38] (version *gpt-4.1-2025-04-14*). These models are used for patch generation. Among them, Claude 3.7 Sonnet demonstrated the best performance (shown in Figure 1); therefore, for consistency and fair comparison, we adopt Claude 3.7 Sonnet as the base model for all ensemble reasoning techniques. Additionally, we set the temperature to 0.2 to ensure low variability and increase determinism in outputs, and set the maximum token limit to 4,096 to support sufficient reasoning length.

5 Results and Analysis

5.1 RQ1: Effectiveness and Efficiency

5.1.1 Process: To answer RQ1, we evaluate the performance of *EnAgent* alongside four state-of-the-art ensemble reasoning baselines (Augment, Augment w/ Pruning, DeiBase, and DeiBase w/ Pruning) across three leading LLMs (Gemini 2.5 Pro, Claude 3.7 Sonnet, and GPT-4.1). Additionally, we introduce a Mixture setting, where the three LLMs generate patches in a round-robin manner to further enhance the diversity of the candidate patches. The effectiveness of each technique is assessed on the widely-used SWE-bench Verified benchmark using two metrics (Pass@1 and the number of uniquely resolved issues). To evaluate the efficiency of each ensemble reasoning technique, we measure both the average time and token overhead. Moreover, we include three reference baselines (introduced in Section 2): *Oracle* (representing the best-case performance of an ensemble technique), *Adversary* (representing the worst-case performance of an ensemble technique), and *Average* (representing the expected performance of a random baseline). The ensemble size N is set to 3. To further mitigate the impact of randomness, each experiment is repeated three times.

5.1.2 Results: Table 1 presents a comprehensive comparison of all studied ensemble reasoning techniques in terms of both effectiveness and efficiency. First, we observe that all five ensemble reasoning techniques consistently outperform the *Average* baseline in terms of Pass@1, validating the effectiveness of ensemble reasoning techniques and reinforcing the motivation for our design of *EnAgent*. Notably, *EnAgent* achieves the best performance across all ensemble reasoning techniques. Specifically, it demonstrates an improvement of 5.01%∼12.86% compared to the baselines in terms of Pass@1. Furthermore, we collect multiple paired overall Pass@1 results for *EnAgent* and each baseline across repeated experiments and different LLMs to evaluate the statistical significance of *EnAgent*'s improvements. The *Wilcoxon Signed-Rank Test* [56] (at a significance level of 0.05) yields p-values below 8.00×10^{-6}, confirming that *EnAgent* significantly outperforms all baselines in terms of Pass@1. In addition, *EnAgent* exhibits strong stability, with an average standard deviation of only 0.19%, lower than those of other baselines (0.38%∼0.52%), thereby further reducing the impact of randomness. Moreover, as shown in Figure 5, *EnAgent* also achieves the highest number of uniquely resolved issues, further demonstrating its superior effectiveness compared to ensemble reasoning baselines.

Table 1: Effectiveness comparison in terms of Pass@1 (↑).

LLM	Adversary	Average	Oracle	Augment	Augment w/ Pruning	DeiBase	DeiBase w/ Pruning	*EnAgent*
Gemini 2.5 Pro	39.60%	53.40%	66.20%	55.40%±0.60%	59.27%±0.31%	53.53%±0.23%	57.33%±0.12%	**62.27%±0.12%**
Claude 3.7 Sonnet	50.60%	61.33%	70.00%	63.13%±0.31%	64.33%±0.42%	62.33%±0.42%	63.87%±0.31%	**66.40%±0.20%**
GPT-4.1	38.80%	51.40%	63.00%	54.87%±0.92%	56.60%±0.53%	53.13%±**0.12%**	56.00%±0.35%	**59.00%**±0.20%
Mixture	38.00%	56.33%	73.40%	58.93%±0.23%	61.07%±0.31%	55.87%±0.76%	58.07%±1.01%	**65.67%±0.23%**

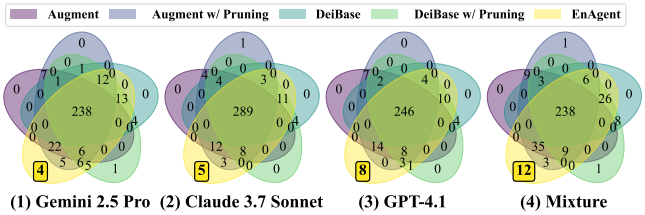


Figure 5: Number of uniquely resolved issues across five studied ensemble reasoning techniques on SWE-bench Verified

Secondly, Table 1 also shows that both Augment w/ Pruning and DeiBase w/ Pruning consistently outperform their original counterparts (Augment and DeiBase). On average, the patch pruning component improves the performance of Augment and DeiBase by 3.91% and 4.72% in terms of Pass@1, respectively. This result empirically validates the effectiveness of our patch pruning component. Furthermore, we collect multiple paired overall Pass@1 results for each baseline with and without the pruning component across repeated experiments and different LLMs to assess the statistical significance of the pruning effect. The *Wilcoxon Signed-Rank Test* [56] (at a significance level of 0.05) yields p-values below 1.30×10^{-5}, confirming that the patch pruning component significantly improves ensemble reasoning baselines in terms of Pass@1.

In terms of efficiency metrics (i.e., the average time and token overhead), DeiBase and DeiBase w/ Pruning exhibit the highest costs, primarily due to the need to separately score each of the N candidate patches, requiring repeated LLM invocations. While prompting-based Augment and Augment w/ Pruning demonstrate better efficiency than *EnAgent*, the notable performance gains of *EnAgent* justify its additional computational cost, indicating a favorable trade-off between effectiveness and efficiency. On average, *EnAgent* incurs a token overhead of approximately 25K on Gemini 2.5 Pro, 27K on Claude 3.7 Sonnet, and 22K on GPT-4.1, corresponding to costs of $0.17, $0.28, and $0.12 per issue, respectively. Furthermore, the pruning component incurs an average of 11.6s and 3.08K tokens, while the selection component requires 11.09s and 22.07K tokens. Overall, the total per-issue cost of *EnAgent* averages only $0.19, indicating that the overhead remains practical for real-world deployment. We further discuss potential efficiency improvements (such as employing trajectory summarization to reduce token overhead and refining regression test selection to lower time costs) in Section 6.4 as our future work.

5.2 RQ2: Influence of Hyper-parameter

5.2.1 Setup: The ensemble size (N) is a critical hyper-parameter in ensemble reasoning techniques. In this research question, we investigate the impact of the ensemble size on the performance of *EnAgent* and other compared ensemble reasoning techniques.

Table 2: Efficiency comparison in terms of the average time overhead (↓) and the average token overhead (↓).

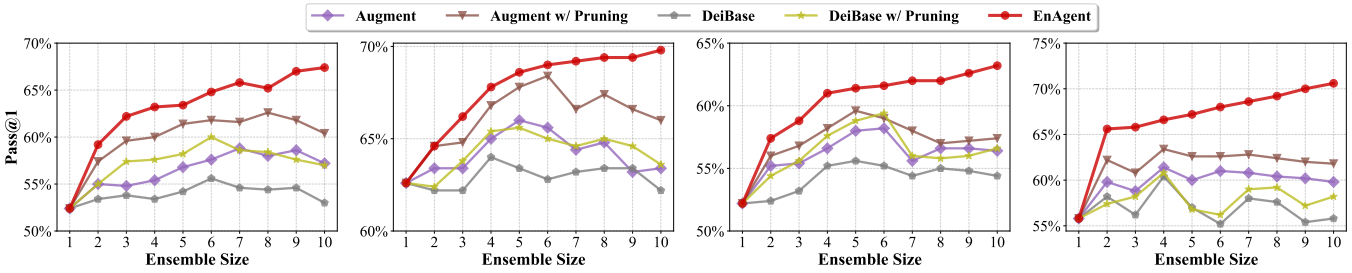
Metric	Technique	Gemini	Claude	GPT	Mixture
AVG. Time	Augment	3.04	3.58	3.12	3.48
	Augment w/ Pruning	14.61	15.15	14.69	15.05
	Deibase	25.38	30.78	26.22	25.62
	Deibase w/ Pruning	36.95	42.35	37.79	37.19
	EnAgent	22.73	22.01	22.73	23.27
AVG. Tokens	Augment	2.07	2.05	2.14	1.98
	Augment w/ Pruning	4.80	4.81	4.87	4.73
	Deibase	34.28	34.10	34.03	33.92
	Deibase w/ Pruning	35.14	35.72	35.21	35.28
	EnAgent	25.00	26.81	22.32	26.46

Specifically, for $1 \leq N \leq 10$, we evaluate and compare the effectiveness of *EnAgent* alongside four baselines in terms of Pass@1.

5.2.2 Results: Figure 6 illustrates the performance trends of different ensemble reasoning techniques as the ensemble size increases, evaluated using the Pass@1 metric. Across all ensemble size settings, *EnAgent* consistently outperforms all four baselines, achieving an average improvement of 5.83%∼14.60% in terms of Pass@1. These results demonstrate the stable effectiveness of *EnAgent* under varying ensemble size settings. Furthermore, we collect multiple paired overall Pass@1 results for *EnAgent* and each baseline across different ensemble sizes and LLMs to assess the statistical significance of *EnAgent*'s improvements. The *Wilcoxon Signed-Rank Test* [56] (at a significance level of 0.05) yields p-values below 3.74×10^{-12}, validating that *EnAgent* significantly outperforms all baselines across different ensemble sizes in terms of Pass@1.

In addition, as the ensemble size increases, the performance of the four baselines generally improves initially but subsequently declines. This trend can be attributed to two key factors: (1) while the theoretical upper bound increases with larger ensemble sizes, the lower bound degrades more rapidly (as discussed in Section 2); and (2) the increasing context length required to process more candidate patches leads to the dilution of relevant information, making accurate patch selection more difficult for LLMs. In contrast, *EnAgent* consistently demonstrates an upward trajectory in terms of Pass@1 with increasing ensemble size, demonstrating superior effectiveness and scalability. We hypothesize that further scaling may continue to enhance *EnAgent*'s effectiveness (see Section 6.4); however, this entails a trade-off between effectiveness and computational cost.

Additionally, we observe that Augment w/ Pruning and DeiBase w/ Pruning consistently outperform their original counterparts (Augment and DeiBase) across all ensemble size settings in terms of Pass@1. This result further substantiates the effectiveness of



(1) Gemini 2.5 Pro (2) Claude 3.7 Sonnet (3) GPT-4.1 (4) Mixture

Figure 6: Influence of the ensemble size in terms of Pass@1 (↑)

Table 3: Comparison between *EnAgent* and its five ablation variants in terms of Pass@1 (↑).

Technique	Gemini 2.5	Claude 3.7	GPT-4.1	Mixture
$EnAgent_{woD}$	60.00%	64.00%	57.00%	63.20%
$EnAgent_{woR}$	60.20%	64.60%	56.80%	63.40%
$EnAgent_{woP}$	58.40%	63.60%	56.20%	61.80%
$EnAgent_{A}$	59.80%	64.60%	57.20%	61.80%
$EnAgent_{woM}$	59.60%	63.60%	57.40%	62.60%
EnAgent	62.27%	66.40%	59.00%	65.67%

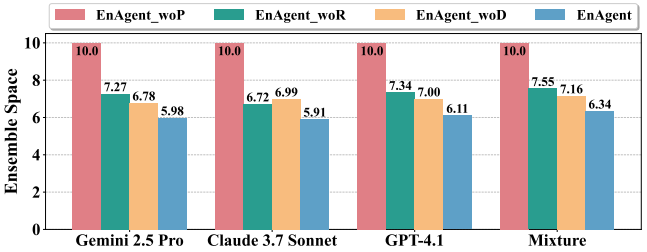


Figure 7: Influence of the patch pruning component and its patch deduplication and regression testing strategies in terms of ensemble space (↓)

the patch pruning component. On average, the patch pruning component yields improvements of 3.91% and 3.74% for Augment and DeiBase, respectively. Furthermore, we collect multiple paired overall Pass@1 results for each baseline with and without the pruning component across different ensemble sizes and LLMs to assess the statistical significance of the pruning effect. The *Wilcoxon Signed-Rank Test* [56] (at a significance level of 0.05) yields p-values below 1.95×10^{-11}, indicating that the patch pruning component significantly improves ensemble reasoning baselines across different ensemble sizes in terms of Pass@1.

5.3 RQ3: Contribution of Main Components

5.3.1 Variants: To assess the contributions of the main components in *EnAgent*, we construct and evaluate five ablation variants. For the patch pruning component, which comprises both patch deduplication and regression testing strategies, we design three variants: (1) $\boldsymbol{EnAgent_{woD}}$, which removes the patch deduplication strategy; (2) $\boldsymbol{EnAgent_{woR}}$, which removes the regression testing strategy; and (3) $\boldsymbol{EnAgent_{woP}}$, which removes the entire patch pruning component. For the patch selection component, we construct two variants: (4) $\boldsymbol{EnAgent_{A}}$, which replaces the selector agent in *EnAgent* with the advanced prompting-based Augment, allowing us to evaluate the effectiveness of the selector agent; (5) $\boldsymbol{EnAgent_{woM}}$, which removes the majority voting strategy to assess its contribution within the patch selection component.

5.3.2 Results: Table 3 presents the comparison results of *EnAgent* and its five ablation variants in terms of Pass@1. First, we observe that *EnAgent* consistently outperforms the three pruning-related variants ($EnAgent_{woP}$, $EnAgent_{woD}$, and $EnAgent_{woR}$). Specifically, *EnAgent* achieves average improvements of 5.57%, 3.73%, and 3.42% over$EnAgent_{woP}$, $EnAgent_{woD}$, and $EnAgent_{woR}$, respectively, demonstrating the contributions of the patch pruning component and its patch deduplication and regression testing strategies. These results reinforce the critical role of the patch pruning component in enhancing the overall performance of *EnAgent*. Furthermore, as shown in RQ1 and RQ2, incorporating the patch pruning component into other ensemble reasoning techniques (i.e., Augment w/ Pruning and DeiBase w/ Pruning) also leads to performance improvements, further validating its generalizability and utility.

Second, *EnAgent* achieves an average improvement of 4.08% over $EnAgent_{A}$ in terms of Pass@1, validating the effectiveness of the selector agent and further demonstrating its superiority over existing prompting-based ensemble reasoning techniques. In addition, *EnAgent* outperforms $EnAgent_{woM}$ by an average of 4.14% in terms of Pass@1, highlighting the contribution of the majority voting strategy to enhance overall performance. This result confirms the necessity of incorporating voting-based mechanisms to mitigate selection instability and reduce the impact of LLM hallucinations.

Furthermore, we collect multiple paired overall Pass@1 results for *EnAgent* and each variant across different LLMs to evaluate the statistical significance of *EnAgent*'s improvements. The *Wilcoxon Signed-Rank Test* [56] (at a significance level of 0.05) yields p-values below 7.32×10^{-3}, confirming that *EnAgent* significantly outperforms all variants in terms of Pass@1. Overall, all main components contribute substantially to the overall effectiveness of *EnAgent*.

5.4 RQ4: Influence of Ensemble Space

5.4.1 Process. To address RQ4, we first examine the impact of *EnAgent* and its three pruning-related variants (i.e., $EnAgent_{woP}$, $EnAgent_{woD}$, and $EnAgent_{woR}$ introduced in RQ3) on the ensemble space (i.e., the number of remaining candidate patches after patch pruning) and then conduct a in-depth correlation analysis between ensemble space and selection effectiveness (measured by Pass@1). Specifically, we fix the ensemble size to 10 and evaluate four techniques: $EnAgent_{woP}$ (which removes the entire patch pruning component including both patch deduplication and regression testing

Table 4: Correlation coefficients between ensemble space and selection effectiveness. A correlation coefficient with an absolute value closer to 1.0 indicates a stronger correlation.

Correlation	Gemini 2.5	Claude 3.7	GPT-4.1	Mixture
Pearson's r	0.91	0.73	0.78	0.89
Spearman's ρ	1.00	0.80	0.80	1.00
Kendall's τ	1.00	0.67	0.67	1.00

strategies), $EnAgent_{woR}$ (which removes the regression testing strategy while retaining the patch deduplication strategy), $EnAgent_{woD}$ (which removes the patch deduplication strategy while retaining the regression testing strategy), and *EnAgent* (which includes the complete patch pruning component).

To further investigate the correlation between the ensemble space and effectiveness, we perform a statistical correlation analysis [32, 46, 62] using three widely-adopted correlation coefficients: *Pearson's r* coefficient [12] (that measures linear correlation), *Spearman's* ρ coefficient [48] (that captures monotonic relationships and is closely related to Pearson's r coefficient), and *Kendall's* τ coefficient [28] (that measures rank correlation). According to established thresholds in prior work [7, 45], a coefficient with an absolute value of 1.0 denotes *perfect* correlation; values in the range [0.8,1.0) indicate *very strong* correlation, [0.6,0.8) *strong* correlation, [0.4,0.6) *moderate* correlation, [0.2,0.4) *weak* correlation, and (0,0.2) *very weak* correlation. A value of 0 indicates *no correlation.*

5.4.2 Results. As shown in Figure 7, *EnAgent*, $EnAgent_{woR}$, and $EnAgent_{woD}$ all reduce the ensemble space compared to $EnAgent_{woP}$. Specifically, they achieve average reductions of 27.80%, 30.22%, and 39.15%, respectively. These results demonstrate that the patch pruning component, along with its patch deduplication and regression testing strategies, effectively reduces the ensemble space. Combined with the findings in Table 3 (as discussed in RQ3), which show that these components contribute to improved selection effectiveness, we can qualitatively conclude that reducing the ensemble space is beneficial for enhancing patch selection performance.

To further quantitatively assess the correlation between the ensemble space and selection effectiveness, we compute the *Pearson's r*, *Spearman's* ρ, and *Kendall's* τ between the two metrics (ensemble space and Pass@1). As shown in Table 4, the absolute values of *Pearson's r* range from 0.73 to 0.91, *Spearman's* ρ range from 0.80 to 1.00, and *Kendall's* τ range from 0.67 to 1.00, respectively. These results consistently indicate a strong correlation across all three correlation metrics, suggesting that reducing the ensemble space is closely associated with improved selection effectiveness.

5.5 RQ5: Orthogonality with existing patch generation techniques

5.5.1 Process: We first evaluate the performance of two state-of-the-art patch generation techniques (SpecRover [43] and Agentless [57]). To further investigate the orthogonality between ensemble-based techniques and patch generation techniques, we combine each ensemble-based technique with the patch generation techniques and measure the resulting performance improvements. Specifically, the patch generation techniques provide additional and more diverse candidate patches for the ensemble-based techniques. The experimental settings (e.g., LLMs, ensemble size) remain consistent with those in RQ1.

Table 5: Orthogonality of ensemble-based techniques and existing patch generation techniques in terms of Pass@1. The relative improvement in this table is computed as $(combination - ensemble)/ensemble \times 100\%$**.**

Technique	Gemini 2.5	Claude 3.7	GPT-4.1
Agentless	50.40%	54.40%	40.80%
SpecRover	51.80%	55.60%	42.40%
Augment	56.00%	63.40%	55.40%
DeiBase	53.80%	62.80%	53.20%
EnAgent	62.40%	66.60%	59.20%
Augment + Agentless	56.80%(↑1.43%)	64.80%(↑2.21%)	56.80%(↑2.53%)
Augment + SpecRover	57.20%(↑2.14%)	64.60%(↑1.89%)	57.00%(↑2.89%)
Augment + Agentless + SpecRover	57.60%(↑2.86%)	65.80%(↑3.79%)	58.00%(↑4.69%)
DeiBase + Agentless	54.40%(↑1.12%)	63.60%(↑1.27%)	54.60%(↑2.63%)
DeiBase + SpecRover	54.60%(↑1.49%)	63.80%(↑1.59%)	54.40%(↑2.26%)
DeiBase + Agentless + SpecRover	55.00%(↑2.23%)	64.40%(↑2.55%)	55.60%(↑4.51%)
EnAgent + Agentless	63.20%(↑1.28%)	67.80%(↑1.80%)	61.20%(↑3.38%)
EnAgent + SpecRover	63.40%(↑1.60%)	68.20%(↑2.40%)	61.40%(↑3.72%)
EnAgent + Agentless + SpecRover	64.60%(↑3.53%)	69.00%(↑3.60%)	62.00%(↑4.73%)

5.5.2 Results: As shown in Table 5, *EnAgent* consistently outperforms Agentless and SpecRover in terms of Pass@1, achieving an average relative improvement of 28.53%. Moreover, all combined techniques achieve higher Pass@1 values than any individual patch generation or ensemble-based technique across all studied LLMs, confirming the orthogonality between the two classes of techniques. In particular, the combined technique ***EnAgent*+Agentless+SpecRover** achieves the best overall performance, with an average relative improvement of 23.76% over the individual techniques, further validating that *EnAgent* can effectively collaborate with existing patch generation techniques to achieve superior performance.

6 Discussion

6.1 Practical Impact

Building autonomous agent-based systems for software engineering tasks has emerged as a leading research direction in recent years. SWE-bench, a widely adopted benchmark, encompasses real-world software engineering tasks such as bug fixing and feature implementation. The SWE-bench leaderboard [3] continuously tracks the state-of-the-art progress in agent-based software engineering systems. **Our proposed framework, *EnAgent*, has been integrated into Trae Agent [22], driving it to achieve the first place on the SWE-bench Verified leaderboard as of January 2026, with a notable Pass@1 score of 78.80%.** This configuration of *EnAgent* slightly differs from that used in the main experiments. Specifically, the coder agent adopts the Mixture strategy, leveraging four LLMs (Claude 4 Sonnet, Claude 4 Opus, Claude 3.7 Sonnet, and Gemini 2.5 Pro) with an ensemble size of 16. Additionally, the tester and selector agents are based on Claude 4 Sonnet. The use of more advanced LLMs, a larger ensemble size, and increased candidate diversity collectively enable *EnAgent* to achieve superior performance, further demonstrating its scalability and potential for continuous improvement. Furthermore, our associated GitHub repository has attracted over 10,000 stars, indicating substantial community interest and adoption. Our findings suggest that integrating ensemble reasoning

Table 6: Quality of regression tests in terms of Accuracy (↑), Precision (↑), Recall (↑), and F1-Score (↑).

Metric	TP	TN	FP	FN	Total
# Instances	10,424	2,231	6,608	737	20,000
Accuracy = (TP+TN)/Total					63.28%
Precision = TP/(TP+FP)					61.20%
Recall = TP/(TP+FN)					93.40%
F1-Score = (2×Precision×Recall)/(Precision+Recall)					73.95%

* TP (true positive): patch passes regression tests and is correct;
* TN (true negative): patch does not pass regression tests and incorrect;
* FP (false positive): patch passes regression tests but is incorrect;
* FN (false negative): patch does not pass regression tests but is correct.

techniques into existing agent-based systems can significantly improve their ability to resolve complex real-world issues. We believe that this insight could serve as a guiding principle for the design of future agent-based software engineering frameworks.

6.2 Quality of Regression Tests

In the patch pruning component, the regression tests selected by the tester agent play a critical role in determining the overall effectiveness of *EnAgent*. To evaluate the quality of these regression tests, we assess the consistency between their pruning decisions and the ground-truth results provided by the golden tests in the SWE-bench. Specifically, we examine whether the regression tests accurately determine the correctness of the candidate patches. As shown in Table 6, the regression tests achieve an Accuracy of 63.28%, a Precision of 61.20%, a Recall of 93.40%, and an F1-Score of 73.95%, respectively. We further analyze the two types of regression test mispredictions (i.e., FP and FN), which directly affect the effectiveness of patch pruning. The overall misprediction rate ($\frac{FP+FN}{Total}$) is 36.73%. Among these, FP cases (i.e., incorrect patches mistakenly passing the regression tests) can be viewed as a conservative patch pruning strategy that preserves erroneous patches for subsequent selection. In contrast, FN cases (i.e., correct patches mistakenly discarded) are more detrimental to overall effectiveness, as they reduce the correct candidate patches. In particular, the FP rate is 33.04%, while the FN rate accounts for only 3.69%. This indicates that the risk of discarding correct patches is relatively low.

In addition, we further investigate the impact of regression tests on instance-level correctness. Specifically, each instance is associated with multiple candidate patches. If all candidate patches are correct, the instance is guaranteed to be resolved successfully; conversely, if all candidates are incorrect, the instance cannot be resolved. Therefore, for ensemble reasoning techniques, it is desirable to maximize the number of all-correct instances and minimize the number of all-incorrect instances. To this end, we evaluate the ratios of all-correct and all-incorrect instances before and after applying the regression testing strategy. As shown in Table 7, the regression testing strategy increases the ratio of all-incorrect instances by 4.45% on average, but more notably increases the ratio of all-correct instances by 12.00% on average. These findings further support the effectiveness of the regression testing strategy and provide an in-depth explanation for its positive impact on the overall performance of *EnAgent*.

Table 7: Influence of the regression testing strategy in terms of the ratios of all-correct (↑) and all-incorrect (↓) instances.

Metric	Technique	Gemini 2.5	Claude 3.7	GPT-4.1	Mixture
All-correct	w/o Regression	25.80%	38.40%	26.80%	24.20%
	w/ Regression	40.80%	48.00%	37.00%	37.40%
All-incorrect	w/o Regression	24.00%	21.60%	28.20%	19.80%
	w/ Regression	28.60%	25.80%	32.40%	24.60%

6.3 Context Window Limitations of Agents

In our evaluation, *EnAgent* does not encounter truncation issues, mainly due to the strong long-context capabilities of the underlying LLMs (200K/1M/1M tokens) and *EnAgent*'s effective program comprehension strategy, which collects contextual information in a concise and targeted manner. However, we acknowledge that context management remains a fundamental challenge for all LLM-based agents, particularly when dealing with larger-scale or more highly complex codebases. As future work, we plan to develop a trajectory summarization strategy that compresses and summarizes historical context while preserving essential information, providing a promising direction to enhance the robustness and scalability of LLM-based agents.

6.4 Future Work

EnAgent is the first agent-based ensemble reasoning approach for repository-level issue resolution. While its effectiveness has been demonstrated through comprehensive empirical studies, several aspects of *EnAgent* can be further enhanced in future work:

- **Improving efficiency**: To reduce the token overhead, we plan to explore trajectory summarization strategies [63, 65] that compress the agent's historical interactions without sacrificing essential contextual information. In addition, we plan to develop more accurate and efficient regression test selection methods and further reduce time overhead through parallel execution.
- **Scaling ensemble size**: Our current experiments (as discussed in RQ2) evaluate ensemble sizes ranging from 1 to 10, with results showing a consistent performance improvement of *EnAgent* as the ensemble size increases. In future work, we plan to investigate the potential performance gains of larger ensemble sizes, while carefully considering cost-efficiency constraints.
- **Enhancing patch deduplication**: Recent studies have explored leveraging LLMs for program equivalence detection, such as identifying equivalent mutants [36, 51]. Building on this, we plan to investigate LLM-based patch deduplication techniques to further enhance the overall performance of *EnAgent*.
- **Enhancing regression testing**: Incorporating test coverage information can help identify tests directly affected by patch modifications, ensuring that pruning focuses on issue-relevant, high-impact tests, thereby improving patch selection accuracy. Accordingly, we plan to integrate code coverage metrics to further enhance the effectiveness of regression test selection.

7 Threats and Validity

Construct Validity. This threat mainly arises from the inherent randomness of LLMs. To mitigate this, we conduct a large-scale

empirical study and release the replication package for practitioners. Moreover, we ensure consistency by repeating all experiments three times in RQ1. Notably, the standard deviations of Pass@1 for Augment, Augment w/ Pruning, DeiBase, DeiBase w/ Pruning, and *EnAgent* are only 0.0052, 0.0039, 0.0038, 0.0044, and 0.0019, respectively, indicating high robustness across runs. In addition, a *Wilcoxon Signed-Rank Test* [56] (at a significance level of 0.05) yields p-values exceeding 0.38 for all comparisons, indicating no statistically significant differences across the three repeated experimental results and further strengthening the reliability of our findings.

External Validity. This threat mainly lies in our experimental subjects. To address this, we adopt the widely-used SWE-bench benchmark and multiple evaluation metrics commonly used in the issue resolution domain [43, 54, 60, 64]. We further evaluate *EnAgent* against four state-of-the-art ensemble reasoning techniques across three leading LLMs and the widely-used SWE-bench benchmark, ensuring comprehensive comparisons. In future work, we plan to extend our evaluation to additional benchmarks and LLMs to further assess the generalizability and robustness of *EnAgent* across diverse settings.

8 Related Work

8.1 Automatic Software Issue Resolution

Automatic software issue resolution is a critical task in software engineering and has attracted increasing research interest in recent years. Numerous techniques have been proposed, among which agent-based approaches have gained particular prominence.

OpenDevin [54] (renamed OpenHands) is among the earliest agent-based frameworks, leveraging a planning mechanism based on user requirements and utilizing tools (such as file editors, terminals, and web search engines) to iteratively accomplish complex tasks. SWE-agent [58] introduces a custom agent-computer interface, enabling agents to interact with the codebase through operations such as viewing and editing files. Moatless [67] enhances issue resolution by equipping agents with code search tools and retrieval strategies to identify relevant code locations. AutoCodeRover [64] further refines the code search capability by representing software projects as abstract syntax trees, allowing agents to effectively retrieve contextual information and locate faults. Building on AutoCodeRover, SpecRover [43] improves the specification by generating function summaries, and also incorporates the generation of reproduction tests to assist in patch generation. Agentless [57] adopts a standardized operational pipeline comprising fault localization, patch generation, and patch verification, without requiring agents to dynamically decide on future actions or interact with complex external tools. MarsCode [35] Agent combines advanced code analysis techniques (i.e., code knowledge graph) with LLM capabilities to provide a systematic process for fault localization, candidate patch generation, and patch validation.

Unlike existing individual patch generation techniques, our *EnAgent* can be seamlessly integrated with them to build more effective ensemble reasoning systems (as shown in Section 5.5). Furthermore, due to its generalizable and modular design, *EnAgent* potentially provides a promising foundation for advancing ensemble reasoning in broader and more complex software engineering tasks.

8.2 Ensemble Techniques for LLM

Ensemble learning [40, 44] has been widely adopted in machine learning and deep learning models to reduce prediction bias and improve generalization by selecting a consensus solution from multiple models, thereby mitigating the limitations inherent to any single model. Traditional ensemble methods (e.g., bagging [6], boosting [21], and stacking [16]) have achieved notable success across a variety of domains, including image classification [9], natural language processing [59], and anomaly detection [24].

Recent advances have demonstrated the potential of ensemble techniques to enhance the performance of LLM-based agent systems across a range of tasks. In the mathematical reasoning domain, Snell et al. [47] conduct a comprehensive study and introduce the Best-of-N approach, which generates multiple solutions in parallel from a base LLM and employs a reward model to select the highest-scoring solution. In the domain of competition-level code generation, Li et al. [31] propose S*, which leverages both execution feedback from public and LLM-generated test cases, along with a clustering-based selection mechanism to guide the final code selection. In addition, Mahmud et al. [37] develop EnsLLM, which utilizes CodeBLEU [41] for syntactic similarity and CrossHair [13] (a property-based testing tool) for behavior similarity. Both similarities are then integrated using a voting mechanism to select the most reliable candidate solution.

In the more complex task of software issue resolution, Chen and Flaherty [8] propose Augment, an agent that applies the LLM-as-a-judge paradigm to assess the semantic alignment between the given GitHub issue and a set of candidate patches, ultimately selecting the most relevant patch. In addition, Zhang et al. [60] introduce DeiBase, which prompts the LLM to generate detailed explanations and confidence scores for each candidate patch, thereby selecting the one with the highest score. In contrast to existing ensemble techniques, we propose *EnAgent*, the first agent-based ensemble reasoning approach for repository-level issue resolution. *EnAgent* is designed to enhance LLM-based issue resolution by formulating it as an optimal search problem and enhancing LLM performance in issue resolution through a modular agent-based architecture.

9 Conclusion

In this work, we present *EnAgent*, the first agent-based ensemble reasoning approach for repository-level issue resolution, which substantially improves the effectiveness of LLMs. *EnAgent* formulates ensemble reasoning as an optimal solution search problem and tackles two fundamental challenges, i.e., large ensemble spaces and repository-level understanding, through modular agents for generation, pruning, and selection. To comprehensively evaluate its performance, we conduct extensive experiments using three leading LLMs on the widely-used SWE-bench benchmark. The experimental results show that *EnAgent* consistently outperforms four state-of-the-art ensemble reasoning baselines across all evaluation settings, demonstrating its effectiveness in enhancing LLM-based software issue resolution.

Acknowledgments

This work is supported by National Natural Science Foundation of China (Grant No. 62322208).

References

[1] Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, GP Shrivatsa Bhargav, Maxwell Crouse, Chulaka Gunasekara, et al. 2024. Granite-Function Calling Model: Introducing Function Calling Abilities via Multi-task Learning of Granular Tasks. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*. 1131–1139.

[2] Anthropic. 2025. Claude 3.7 Sonnet and Claude Code. https://www.anthropic.com/news/claude-3-7-sonnet.

[3] SWE bench Team. 2025. SWE-bench Leaderboard. https://www.swebench.com/.

[4] Matias Bordese. 2025. Unified diff python parsing/metadata extraction library. https://github.com/matiasb/python-unidiff.

[5] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2188–2200.

[6] Leo Breiman. 1996. Bagging predictors. *Machine learning* 24 (1996), 123–140.

[7] Raymond PL Buse and Westley R Weimer. 2009. Learning a metric for code readability. *IEEE Transactions on software engineering* 36, 4 (2009), 546–558.

[8] Tongfei Chen and Colin Flaherty. 2025. #1 open-source agent on SWE-Bench Verified by combining Claude 3.7 and O1. https://www.augmentcode.com/blog/1-open-source-agent-on-swe-bench-verified-by-combining-claude-3-7-and-o1.

[9] Yushi Chen, Ying Wang, Yanfeng Gu, Xin He, Pedram Ghamisi, and Xiuping Jia. 2019. Deep learning ensemble for hyperspectral image classification. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 12, 6 (2019), 1882–1897.

[10] John Clarke, Jose Javier Dolado, Mark Harman, Rob Hierons, Bryan Jones, Mary Lumkin, Brian Mitchell, Spiros Mancoridis, Kearton Rees, Marc Roper, et al. 2003. Reformulating software engineering as a search problem. *IEE Proceedings-software* 150, 3 (2003), 161–175.

[11] Augment Code. 2025. Augment SWE-bench Verified Agent. https://github.com/augmentcode/augment-swebench-agent.

[12] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. *Noise reduction in speech processing* (2009), 1–4.

[13] CrossHair. 2025. An analysis tool for Python that blurs the line between testing and type systems. https://github.com/pschanely/CrossHair.

[14] Google DeepMind. 2025. Gemini 2.5 Pro. https://deepmind.google/models/gemini/pro/.

[15] Giuseppe A Di Lucca and Massimiliano Di Penta. 2005. Integrating static and dynamic analysis to improve the comprehension of existing web applications. In *Seventh IEEE International Symposium on Web Site Evolution*. IEEE, 87–94.

[16] Saso Džeroski and Bernard Ženko. 2004. Is combining classifiers with stacking better than selecting the best one? *Machine learning* 54 (2004), 255–273.

[17] Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. 2025. CodeMonkeys: Scaling Test-Time Compute for Software Engineering. *arXiv preprint arXiv:2501.14723* (2025).

[18] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. 2001. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001.* IEEE, 602–611.

[19] Sarah Fakhoury, Saikat Chakraborty, Madanlal Musuvathi, and Shuvendu K Lahiri. 2024. Nl2fix: Generating functionally correct code edits from bug descriptions. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 410–411.

[20] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

[21] Yoav Freund and Robert E Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences* 55, 1 (1997), 119–139.

[22] Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. 2025. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370* (2025).

[23] Lianghong Guo, Wei Tao, Runhan Jiang, Yanlin Wang, Jiachi Chen, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2025. Omnigirl: A multilingual and multimodal benchmark for github issue resolution. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 24–46.

[24] Xu Han, Xiaohui Chen, and Li-Ping Liu. 2021. Gan ensemble for anomaly detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 4090–4097.

[25] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.

[26] EnAgent Homepage. 2026. https://github.com/bytedance/trae-agent/tree/main/evaluation/patch_selection.

[27] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*.

[28] Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1-2 (1938), 81–93.

[29] Jinhan Kim, Juyoung Jeon, Shin Hong, and Shin Yoo. 2022. Predictive mutation analysis via the natural language channel in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–27.

[30] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. 2017. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Transactions on Software Engineering* 44, 4 (2017), 308–333.

[31] Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E Gonzalez, and Ion Stoica. 2025. S*: Test time scaling for code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*.

[32] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.

[33] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2023), 21558–21572.

[34] Weiwen Liu, Xu Huang, Xingshan Zeng, xinlong hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong WANG, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Wang Xinzhi, Yong Liu, Yasheng Wang, Duyu Tang, Dandan Tu, Lifeng Shang, Xin Jiang, Ruiming Tang, Defu Lian, Qun Liu, and Enhong Chen. 2025. ToolACE: Winning the Points of LLM Function Calling. In *The Thirteenth International Conference on Learning Representations*.

[35] Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. 2024. Marscode agent: Ai-native automated bug fixing. *arXiv preprint arXiv:2409.00899* (2024).

[36] Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. 2023. LLMs: Understanding code syntax and semantics for code analysis. *arXiv preprint arXiv:2305.12138* (2023).

[37] Tarek Mahmud, Bin Duan, Corina Pasareanu, and Guowei Yang. 2025. Enhancing llm code generation with ensembles: A similarity-based selection approach. *arXiv preprint arXiv:2503.15838* (2025).

[38] OpenAI. 2025. Introducing GPT-4.1 in the API. https://openai.com/index/gpt-4-1/.

[39] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 936–946.

[40] Robi Polikar. 2012. Ensemble learning. *Ensemble machine learning: Methods and applications* (2012), 1–34.

[41] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[42] Matthew Renze. 2024. The effect of sampling temperature on problem solving in large language models. In *Findings of the association for computational linguistics: EMNLP 2024*. 7346–7356.

[43] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 963–974.

[44] Omer Sagi and Lior Rokach. 2018. Ensemble learning: A survey. *Wiley interdisciplinary reviews: data mining and knowledge discovery* 8, 4 (2018), e1249.

[45] B Sebastian, C Christian, and P Alexander. 2017. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada*. 16–18.

[46] Donghwan Shin, Shin Yoo, Mike Papadakis, and Doo-Hwan Bae. 2019. Empirical evaluation of mutation-based test case prioritization techniques. *Software Testing, Verification and Reliability* 29, 1-2 (2019), e1695.

[47] Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*.

[48] Charles Spearman. 1961. The proof and measurement of association between two things. (1961).

[49] Zhao Tian and Junjie Chen. 2026. Aligning Requirement for Large Language Model's Code Generation. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE)*.

[50] Zhao Tian, Junjie Chen, and Xiangyu Zhang. 2025. Fixing Large Language Models' Specification Misunderstanding for Better Code Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*.

[51] Zhao Tian, Honglin Shu, Dong Wang, Xuejie Cao, Yasutaka Kamei, and Junjie Chen. 2024. Large Language Models for Equivalent Mutant Detection: How Far Are We?. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1733–1745.

[52] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F Bissyandé, Yves Le Traon, and Koushik Sen. 2020. Selecting fault revealing mutants. *Empirical Software*

Engineering 25 (2020), 434–487.

[53] Ruiqi Wang, Jiyu Guo, Cuiyun Gao, Guodong Fan, Chun Yong Chong, and Xin Xia. 2025. Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1955–1977.

[54] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations*.

[55] You Wang, Michael Pradel, and Zhongxin Liu. 2026. Are "Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study. *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE)*.

[56] Frank Wilcoxon, SK Katti, Roberta A Wilcox, et al. 1963. *Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test.* Vol. 1. American Cyanamid Pearl River, NY.

[57] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE (2025), 24 pages.

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

[59] Hongzhi Zhang and M Omair Shafiq. 2024. Survey of transformers and towards ensemble learning using transformers for natural language processing. *Journal of big Data* 11, 1 (2024), 25.

[60] Kexun Zhang, Weiran Yao, Zuxin Liu, Yihao Feng, Zhiwei Liu, Rithesh RN, Tian Lan, Lei Li, Renze Lou, Jiacheng Xu, et al. 2024. Diversity empowers intelligence: Integrating expertise of software engineering agents. In *The Thirteenth International Conference on Learning Representations*.

[61] Kechi Zhang, Huangzhao Zhang, Ge Li, Jinliang You, Jia Li, Yunfei Zhao, and Zhi Jin. 2026. SEAlign: Alignment training for software engineering agent. *2026 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*.

[62] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 214–224.

[63] Yusen Zhang, Ansong Ni, Ziming Mao, Chen Henry Wu, Chenguang Zhu, Budhaditya Deb, Ahmed Awadallah, Dragomir Radev, and Rui Zhang. 2022. SummN: A Multi-Stage Summarization Framework for Long Input Dialogues and Documents. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1592–1604.

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

[65] Ming Zhong, Yang Liu, Yichong Xu, Chenguang Zhu, and Michael Zeng. 2022. Dialoglm: Pre-trained model for long dialogue understanding and summarization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 11765–11773.

[66] Yuqi Zhu, Jia Li, Ge Li, YunFei Zhao, Zhi Jin, and Hong Mei. 2024. Hot or cold? adaptive temperature sampling for code generation with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 437–445.

[67] Albert Örwall. 2025. Moatless Tools. https://github.com/aorwall/moatless-tools.