SWE-Exp: Experience-Driven Software Issue Resolution

Anonymous Author(s)

ABSTRACT

Recent advances in large language model (LLM) agents have shown remarkable progress in software issue resolution, leveraging advanced techniques such as multi-agent collaboration and Monte Carlo Tree Search (MCTS). However, current agents act as memoryless explorers—treating each problem separately without retaining or reusing knowledge from previous repair experiences. This leads to redundant exploration of failed trajectories and missed chances to adapt successful issue resolution methods to similar problems. To address this problem, we introduce SWE-Exp, an experience-enhanced approach that distills concise and actionable experience from prior agent trajectories, enabling continuous learning across issues. Our method introduces a multi-faceted experience bank that captures both successful and failed repair attempts. Specifically, it extracts reusable issue resolution knowledge at different levels—from high-level problem comprehension to specific code changes. Experiments show that SWE-Exp achieves state-of-the-art resolution rate (41.6% Pass@1) on SWE-bench-Verified under open-source agent frameworks. Our approach establishes a new paradigm in which automated software engineering agents systematically accumulate and leverage repair expertise, fundamentally shifting from trial-and-error exploration to strategic, experience-driven issue resolution[1].

1 INTRODUCTION

Software issue resolution, which aims to automatically localize and fix faults across interdependent source files, represents one of the most challenging tasks in automated software engineering [3, 14, 28, 30]. With the introduction of SWE-bench [14]—the standard benchmark for evaluating automated program repair (APR) on real-world GitHub issues—researchers have developed a diverse range of techniques to tackle this challenge [1, 2, 4, 19]. SWE-bench provides a comprehensive evaluation framework by pairing real-world GitHub issues with their full repository-level contexts, enabling rigorous assessment of repair methods in realistic, complex software environments.

Recently, the emergence of LLMs and multi-agent techniques has significantly advanced the task of automated issue resolution. Agent-based approaches equip LLMs with external tools for code navigation, editing, and testing, enabling them to iteratively explore and refine potential solutions [1, 2, 40]. Building on this foundation, MCTS-based systems further enhance agent capabilities by guiding exploration in a more systematic and goal-directed manner [1], improving the efficiency and completeness of the search process.

Despite remarkable progress in the issue-solving rate, current approaches suffer from a fundamental limitation: agents operate as memoryless explorers, treating each issue in isolation and failing to leverage insights from previous repair attempts [5]. This limitation creates three critical inefficiencies: 1) Redundant exploration: agents often retry ineffective trajectories across similar issues, expending computational efforts on issue resolution strategies that have proven unsuccessful in similar contexts [1, 15]; 2) Inability of knowledge transfer: agents often discard valuable insights from successful resolution trajectories, including effective issue resolution workflows, code patterns, and contextual factors influencing patch quality after each session [38, 40, 48]; and 3) Lack of strategic evolution: without systematic experience accumulation, agents are unable to develop increasingly refined issue resolution strategies or compound expertise over time. As a result, they struggle to adapt to novel or evolving issues, particularly those that are specific to individual repositories [15, 26].

To address these challenges, we propose SWE-Exp, a novel experience-enhanced approach that transforms issue resolution from isolated, stateless problem-solving into a continuous learning process. Unlike previous approaches that only utilize internal knowledge [35, 40], SWE-Exp distills structured experiences from prior resolution attempts, and leverages such accumulated knowledge to guide future repair attempts. SWE-Exp maintains an evolving experience bank that encodes knowledge across three facets: trajectory-guided problem understanding, fault localization patterns, and modification strategies. When encountering a new issue, the system retrieves relevant experiences and distills them into actionable guidance. To operationalize this knowledge more efficiently, we employ a dual-agent architecture, where an *Instructor* agent formulates high-level strategies and an *Assistant* agent executes low-level operations. By maintaining an evolving multi-facet experience bank, our approach avoids redundant exploration and leverages past knowledge more effectively.

To validate the effectiveness of SWE-Exp, we conducted extensive experiments on open-source LLM DeepSeek-V3-0324 [6] on the SWE-bench benchmark. Experimental results show that SWE-Exp achieves a resolution rate of 41.6% on the SWE-bench-verified benchmark, surpassing SOTA methods by a large margin. Furthermore, the comprehension and modification capabilities distilled by our method independently contribute to performance improvements, with their combination yielding the most significant gains.

Our main contributions can be summarized as follows:

- We present a novel framework that systematically captures and manages experiences from agent trajectories at multiple facets, enabling systematic collection of repair knowledge across different issue contexts.
- We propose an experience-driven guidance system that uses historical knowledge through dynamic retrieval to improve fault-localization accuracy and patch quality, transforming repository-level issue resolution from isolated problem-solving into continuous learning.
- Experimental validation showing that SWE-Exp achieves state-of-the-art resolution rate on open-source agent frameworks.

†Xiaodong Gu is the corresponding author.

[1]Our code and data are available at https://github.com/YerbaPage/SWE-Exp

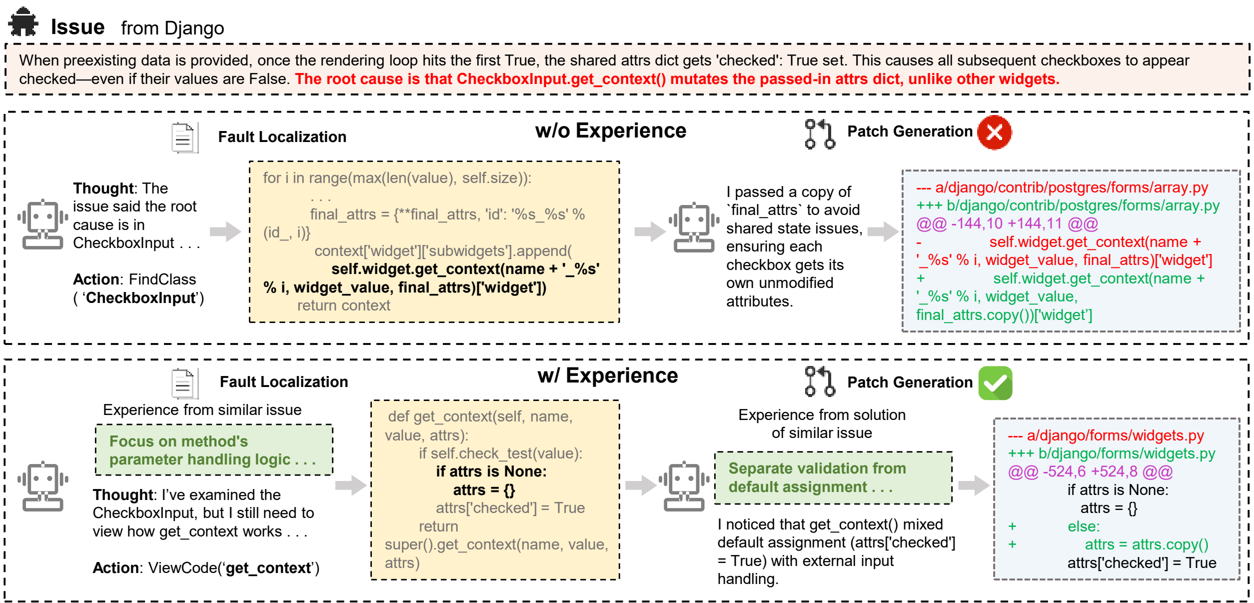

Figure 1: Motivating example of experience-guided approach on instance *django-11964*.

2 MOTIVATION

Recent advances in MCTS approaches for repository-level issue resolution have shown significant improvements in exploring solution spaces systematically [1]. These methods enable agents to backtrack and explore alternative solutions through strategic search tree expansion, addressing limitations of linear sequential processes. However, even with advanced search strategies, current agents remain fundamentally limited by their inability to learn from accumulated issue resolution experiences across different issues.

To illustrate this limitation, let us consider a concrete issue resolution scenario from the Django codebase involving a composite widget composed of multiple checkboxes. The issue manifests as all checkbox widgets appearing checked regardless of their actual values. This is caused by the `CheckboxInput` widget modifying a shared `attrs` dictionary in place. Figure 1 demonstrates how agents approach this problem with and without historical experience.

When operating without experience, the agent focuses on the surface-level symptoms mentioned in the issue description. This leads to a patch that creates a copy of `final_attrs` within the context rendering method of the composite widget. While this appears to address the immediate problem, it represents a narrow, symptom-focused solution that fails to address the root cause: the `CheckboxInput` widget's fundamental design flaw of modifying its input parameters. This approach has two critical limitations. First, it only addresses the specific context for the particular class, leaving the underlying issue unresolved for other potential widget combinations. Second, it treats the symptom rather than the disease, creating a fragile fix that may not prevent similar issues in future scenarios.

In contrast, an agent equipped with relevant experience demonstrates more strategic and insightful reasoning. Prior experience directs the agent to examine not just the surface behavior but the deeper mechanism—specifically, how method parameters are handled and potentially mutated during execution. This perspective leads the agent to scrutinize the implementation of the `CheckboxInput.get_context()` method, where the root cause resides. Drawing on its accumulated repair knowledge, the agent applies a principled fix: modifying the method to create a defensive copy of the `attrs` dictionary before performing any changes. This strategy reflects a key insight from past experience—that default assignments need to be verified and handled separately to avoid unintended side effects. Compared to symptom-level patches, this solution is significantly more robust, as it eliminates the underlying design flaw and ensures correctness across diverse usage contexts. It exemplifies how experiential knowledge enables agents to fix problems at their source, rather than applying narrow, reactive workarounds.

This example reveals a critical insight: the difference between surface-level symptom fixing and deep root-cause resolution. Without systematic experience accumulation, agents repeatedly engage in reactive issue resolution that addresses immediate symptoms without understanding underlying patterns. Experience-guided agents, however, develop pattern recognition capabilities that enable them to identify and resolve fundamental issues. This leads to more robust and generalizable solutions. This motivates our design of SWE-Exp, which transforms repository-level issue resolution from isolated symptom-focused issue resolution into a systematic, knowledge-driven process that accumulates and uses repair expertise across similar contexts.

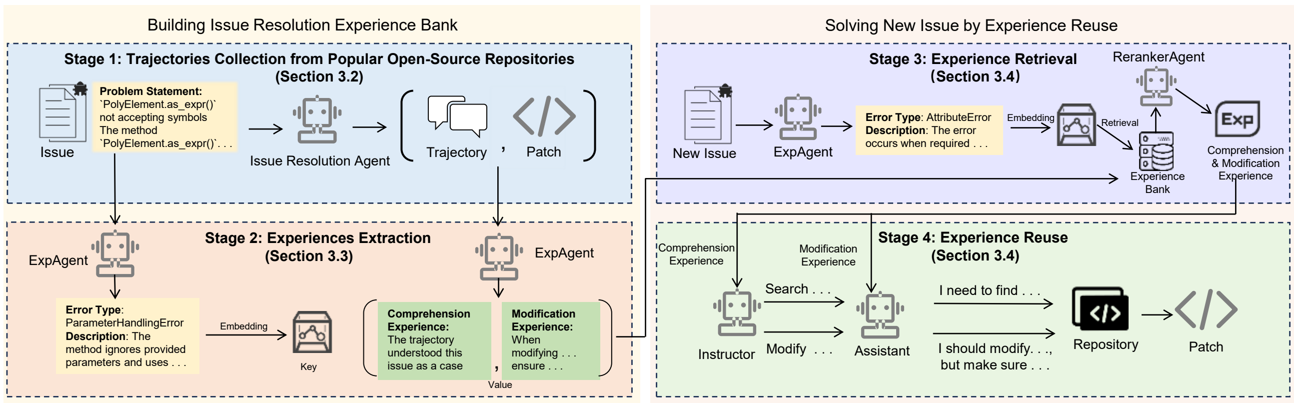

Figure 2: The framework of SWE-Exp.

3 METHODOLOGY

3.1 Conceptual Framework

To formalize our approach, consider a standard agent that resolves an issue p by generating a reasoning and action trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots)$, where s_t denotes the repository state at time step t and a_t is the corresponding action taken. Existing methods typically treat each trajectory in the agent's history $\mathcal{H} = \{\tau_1, \tau_2, \ldots\}$ as independent and do not exploit prior experiences during new problem solving. In contrast, our approach introduces an experience bank $\mathcal{B}_{\text{exp}}$ which systematically accumulates distilled knowledge from the agent's past resolution trajectories. When faced with a new issue instance p', the agent queries the memory to retrieve relevant experiences $E' = Retrieve(p', \mathcal{B}_{\text{exp}})$. The subsequent search for a new trajectory τ' is then conditioned on the retrieved experience set E'. This transforms the agent's behavior from blind exploration to experience-guided reasoning, allowing it to leverage prior insights and quickly converge toward final solutions.

The framework follows a systematic four-stage pipeline, as illustrated in Figures 2: First, the system collects repair trajectories from both successful and failed issue resolution attempts across diverse repository contexts (Section 3.2). Second, an offline experience extraction process transforms these raw trajectories into structured, multi-faceted knowledge at different abstraction levels (Section 3.3). Finally, when facing new issues, the system retrieves and adapts relevant historical experiences to provide targeted guidance, and experience-informed agents execute the issue resolution process through strategic planning and tactical implementation(Section 3.4).

3.2 Trajectories Collection

Our approach begins with the systematic collection of repair trajectories as the source for distilling experience.

Structured Trajectory Representation. Each agent repair attempt, successful or not, is represented as a sequence of tuples $\langle(d_t, a_t, s_{t+1}, f_t)\rangle_{t=0..N}$ where d_t represents high-level directives, a_t denotes specific actions taken, s_{t+1} captures the resulting repository state, and f_t denotes environment feedback. This structural representation ensures that both successful issue resolution workflows and failure patterns are preserved for later analysis.

Diverse Issue Coverage. For efficient experience extraction, trajectories are collected across multiple dimensions: different repository types (e.g., web frameworks, scientific libraries, utilities), various error categories (e.g., logic bugs, API misuse, configuration issues), and diverse issue resolution complexities (single-file modifications vs. multi-component changes). This diversity ensures that extracted experiences can generalize across different issue resolution scenarios.**[Gu: can we delete this paragraph?]**

Success Or Failure Annotation. Each trajectory is annotated with success (i.e., producing correct patches validated against ground truth) or failure. For failure trajectories, additional metadata is appended to capture the failure cause—whether due to incorrect localization, flawed modification strategy, or insufficient problem comprehension. Such a categorization enables targeted experience extraction for both positive and negative patterns.

3.3 Experiences Extraction

Raw trajectories, while comprehensive, are often too lengthy, noisy, and problem-specific to be directly reused. Our next stage aims to transform them into structured, reusable knowledge. An *Experiencer* agent is instructed to extract experiences from both success and failure trajectories.

3.3.1 Experience Representation. We define *experience* as the generalized and transferable thinking patterns extracted from an agent's past issue-solving process. It consists of two key components:

- **Perspective**: the agent's abstract understanding of the problem.
- **Modification**: the generalized strategy used to address the issue.

Formally, each experience is represented as a dictionary, where the key denotes the perspective and the value represents the corresponding modifications. For example,

```
"perspective": "The trajectory understood this issue as a
     deprecation of legacy behavior that was no longer necessary
     due to improvements in the system's handling of structured
     data. The perspective focused on transitioning users smoothly
     from an old implementation to a more direct approach.",
"modification": [
  "When deprecating functionality, it's important to first add a
      warning before removing the feature, giving users time to
      adapt their code.",
```

```
    "Removing automatic type conversions can simplify code and make
         behavior more predictable, but requires careful
         consideration of backward compatibility."
]
```

3.3.2 Offline Embedding and Storage. To facilitate efficient retrieval during issue resolution, all extracted experiences are embedded and stored in a vector database, which we refer to as the *Experience Bank*. During the offline embedding process, each experience is indexed by two metadata attributes:

- **Issue Type**: A generalized, descriptive label inferred by the agent based on the issue, such as the `AttributeError` and the `VariableReferenceError`.
- **Description**: A generalized explanation generated by the agent, describing the typical conditions and scenarios in which this type of error arises.

For instance,

```
"Issue": sphinx-doc__sphinx-8638,
"issue_type": "VariableReferenceError",
"description": "Occurs when instance variables are
     incorrectly linked to other variables of the same name
     in the project, leading to unintended documentation
     references."
```

These attributes are encoded into dense vectors using a pre-trained embedding model. The resulting embeddings are stored in the Experience Bank, enabling semantic similarity search and retrieval during online resolution tasks.

3.3.3 Multi-facet Categorization. To support efficient and context-aware reuse, extracted experiences are organized into two categories, each reflecting a distinct facet of abstraction:

Comprehension Experiences These experiences capture how past issues were interpreted and reasoned about at a conceptual level. They encode general reasoning patterns for issue understanding, such as identifying key symptoms, forming diagnostic hypotheses, and leveraging contextual or structural cues to guide early-stage exploration. For instance,

> "The issue was fundamentally about how Sphinx handles variable linking in documentation, specifically the automatic linking of similarly named variables across different contexts (instance vs global). The golden patch reveals that the solution was to modify the role assignment for variable documentation fields rather than changing the fuzzy matching logic."

> "The core misunderstanding was focusing on the cross-referencing behavior (find_obj method) rather than examining how variable documentation fields are processed and assigned roles in the Python domain."

Comprehensive experiences inform how agents interpret and navigate unfamiliar issues, helping the agent prioritize relevant information and narrow the search space effectively.

Modification Experiences These experiences encode generalized strategies for code modification based on prior patches. They include insights into how responsibility was assigned to specific code regions, which behavioral contracts were violated, and how safety, scope, and potential side effects were assessed and managed. For instance,

> "When modifying a method that accepts optional parameters, ensure the logic properly handles both the presence and absence of these parameters without accidentally overriding valid inputs.",

> "For methods that validate input parameters, structure the validation logic to clearly separate the validation step from the default value assignment to prevent unintended behavior."

Modification experiences guide not only the structure of the fix but also the underlying reasoning and design choices that informed the patch.

The multi-faceted *Experience Bank* serves as an external knowledge base that supports decision-making across the agent's debugging pipeline. During issue resolution, relevant experiences are retrieved and used to guide both high-level diagnostic reasoning and low-level code editing. This enables agents to shift from trial-and-error exploration to strategic, experience-driven behavior.

3.4 Experiences Reuse

Equipped with the experience bank, the agent is now ready to execute the core issue resolution task. This process unfolds across a standard three-stage workflow that mirrors real-world software engineering practices: Issue Understanding, Fault Localization, and Patch Generation. Each stage is enhanced by the MCTS framework [1] and informed by the experiences retrieved from the experience bank for the current problem.

The MCTS process unfolds as a search through a tree where nodes represent states of the codebase and edges represent actions (`Search` for code exploration, `View` for context examination, and `Edit` for code modification). At each step, the agent selects actions based on a modified Upper Confidence Bound for Trees (UCT) criterion that balances exploiting known high-reward paths with exploring less-visited states. Our experience-enhanced framework augments this standard MCTS exploration by retrieving relevant historical knowledge at critical decision points, providing contextual guidance that informs both action selection and value assessment. When the agent encounters decision nodes during tree expansion, semantically similar experiences from past resolution attempts are dynamically retrieved and integrated into the exploration strategy. This transforms the traditional trial-and-error nature of MCTS into a systematic, knowledge-driven process where each exploration step builds upon accumulated expertise rather than starting from scratch.

3.4.1 Experience Retrieval. The framework implements a context-aware retrieval system that seamlessly integrates with the tree search process. Before each decision, the agent retrieves N most relevant experiences from the experience bank based on the vector similarities between the new issue type and attributes with keys in the vector database.

To adapt the experiences to the current context, a rerank agent then selects K experiences that are deemed helpful for resolving the current issue. The agent analyzes the similarities and differences between past and present issues, generating contextualized guidance that preserves the essence of successful strategies while adapting to new scenarios. For comprehension experiences, the agent compares problem statements to suggest strategic approaches for problem comprehension. For modification experiences, it considers the code environment and safety patterns to inform repair decisions. To prevent data leakage, we exclude selecting experiences from the same repository.

3.4.2 Agent Role Separation. When integrating multi-facet experiences into MCTS frameworks, we observe that vanilla MCTS tends to overuse *find* actions and lacks initiative in performing actual code modifications. To address this, we introduce a hierarchical dual-agent architecture that separates high-level planning from low-level execution. The process is managed by an *Instructor* and an *Assistant*: the *Instructor* agent acts as a high-level planner, determining the strategic direction of the next action (search, view, modify, or finish), while the *Assistant* operates at a low level, executing the specific actions based on the information provided by the *Instructor*.

Such a role separation improves issue resolution by enabling more focused and interpretable repair trajectories. It offers two main advantages: (1) The *Instructor*'s decision-making is streamlined, as it no longer handles irrelevant tools or arguments. (2) Unlike vanilla MCTS, where thought and action generation are coupled, our framework decouples them, providing instruction-level control over tool usage. This allows prior experience, particularly in code modification, to be better leveraged by shaping the *Assistant*'s instructions.

4 EXPERIMENTAL SETUP

4.1 Research Questions

We evaluate SWE-Exp by addressing the following research questions:

RQ1: How effective is SWE-Exp in issue resolution compared to other approaches?

RQ2: How does each component of SWE-Exp contribute to its overall performance?

RQ3: How does the number of experiences impact the effectiveness of SWE-Exp?

4.2 Datasets

SWE-Bench-Verified Software engineering tasks provide a compelling testbed for investigating agent behavior, as they inherently involve complex reasoning, strategic decision-making, and dynamic interaction with the environment (Jimenez et al., 2024). The SWE-bench-verified benchmark exemplifies these challenges by presenting agents with authentic software bugs that require multi-step solutions: understanding natural language issue descriptions, navigating and analyzing the codebase, proposing plausible modifications, and verifying their fixes through test execution.

We adopt the SWE-Bench-Verified in particular because it focuses on issues with human-verified ground truth patches, thereby reducing label noise and ensuring higher evaluation reliability. This allows for more accurate assessment of an agent's true capability to resolve real-world software issues, without confounding effects from noisy or ambiguous labels. All baseline methods are evaluated on the same dataset.

4.3 Baselines

We compare SWE-Exp with the following baselines:

- **Agentless** [38]: A non-agentic pipeline that decomposes the repair process into distinct phases of localization, repair, and patch validation.
- **SWE-Agent** [40]: A custom agent-computer interface enabling LM agents to interact with repository environments through defined actions.
- **SWE-Search** [1]: A state-of-the-art repository-level issue resolution agent that uses Monte Carlo Tree Search (MCTS) to explore the solution space.
- **AutoCodeRover** [47]: An AST-based program improvement agent that retrieves relevant code contexts through structured API calls and performs iterative patch generation to resolve the fault.
- **Moatless Tools** [1]: A tool-augmented framework composed of lightweight modules for code retrieval, inspection, and modification, such as `FindFunction`, `SemanticSearch`, and `StringReplace`. These tools allow LMs to interact with codebase in a non-agentic yet effective manner.
- **CodeAct** [17]: A task-agnostic framework that casts repository-level coding tasks as planning problems, incrementally analyzing dependencies and orchestrating LLM-driven edits across files to reach a globally consistent state validated by external oracles.
- **OpenHands** [32]: An open-source platform for building general-purpose AI agents that solve software and web tasks through code, terminal, and browser interaction.

4.4 Implementation Details

We implement SWE-Exp by extending the SWE-Search [1] framework with our components, without using its testbed. We employ `DeepSeek-V3-0324` [6] as our agent model, configuring the agent with a temperature of 0.7 and limit the number of iterations to 20, while the remaining configurations follow SWE-Search [1]. We additionally set the maximum number of finished nodes to 2, meaning that the agent will stop early once two patches are successfully generated, even if the 20 iterations have not been reached. Due to space limitations, additional hyperparameters and prompts are provided in the supplementary material. For experience retrieval, we compute cosine similarity based on embeddings generated by the Multilingual-E5-Large model[2]. During retrieval, we first identify the top $N = 10$ issues that are most relevant in terms of error type based on cosine similarity. Subsequently, a dedicated reranking agent evaluates these candidates and selects $k = 1$ most applicable experience to guide the current resolution process. During the experience collection phase, the instructor-assistant agents are executed once to gather experiences from historical trajectories. In the experience reuse phase, when handling a specific instance, experiences originating from the same repository as that instance are excluded from the retrieval process to prevent data leakage.

Table 1: Main effectiveness results on SWE-Bench-Verified dataset.

Method	Model	Pass@1
Agentless	DeepSeek-V3-0324	36.6%
	GPT-4o (2024-05-13)	36.2%
SWE-Agent	DeepSeek-V3-0324	38.8%
	Claude-3.5 Sonnet	33.6%
	GPT-4o (2024-05-13)	23.0%
SWESynInfer	Claude-3.5 Sonnet	35.4%
	GPT-4o (2024-05-13)	31.8%
	Lingma SWE-GPT 72B	32.0%
SWE-Search	DeepSeek-V3-0324	35.4%
Moatless Tools	DeepSeek-V3-0324	34.6%
AutoCodeRover	GPT-4o (2024-05-13)	38.4%
CodeAct	GPT-4o (2024-05-13)	30.0%
OpenHands	DeepSeek-V3-0324	38.8%
SWE-Exp	DeepSeek-V3-0324	**41.6%**

5 RESULTS

5.1 RQ1: Effectiveness

Table 1 presents the comparative performance of SWE-Exp against established baselines on the SWE-Bench-Verified dataset. We measure the performance based on widely used metric Pass@1 for issue resolution. This metric captures the proportion of issues that are correctly fixed on the first attempt, in line with the evaluation standards proposed by [1, 40]. Overall, SWE-Exp achieves a Pass@1 score of 41.6%, establishing a new state-of-the-art among all methods using the DeepSeek-V3-0324 model. It surpasses the previous best result of 38.8% from SWE-Agent using the same model, indicating that experience-guided orchestration introduces significant gains even under strong agent-based setups. While larger language models generally offer stronger capabilities, our results suggest that effective orchestration plays a comparably crucial role in automated code repair. Notably, SWE-Exp achieves a Pass@1 score of 41.6% with DeepSeek-V3-0324, outperforming several competing methods that utilize more powerful foundation models. For example, AutoCodeRover [47] and CodeAct [17], both using GPT-4o (2024-05-13), obtain 38.4% and 30.0% respectively, while SWE-Agent on the same model yields only 23.0%. Similarly, although Claude 3.5 Sonnet is a high-capacity model, its performance under SWE-Agent reaches only 33.6%. These comparisons demonstrate that improvements in model architecture alone are insufficient to guarantee performance gains, and that experience-informed orchestration can compensate for, or even surpass, the advantages conferred by model scale. Focusing on methods operating under the same model, SWE-Exp further establishes a new state-of-the-art within the DeepSeek-V3-0324 setting. SWE-Exp achieves a Pass@1 accuracy of 41.6%, representing a 7.2% relative improvement over the previous best, SWE-Agent (38.8%) [40], and achieves a +17.5% relative improvement over its direct base method, SWE-Search (35.4%) [1]. In comparison to the Agentless baseline (36.6%) [38], which applies minimal orchestration over the same model, SWE-Exp still yields a +13.7% relative gain. These results affirm that our experience-guided framework enhances system effectiveness even under fixed model conditions, by transforming code repair from reactive generation into a structured, context-sensitive process.

The performance improvement stems from effective experience-driven guidance mechanisms. Trajectory-guided problem comprehension experiences enable the Instructor to develop more accurate issue understanding by leveraging patterns from analogous problems, leading to better strategic planning and fault localization hypotheses. Modification-level experiences provide the Assistant with safety patterns and repair strategies that prevent common pitfalls such as incomplete fixes or introducing regressions. This experience-informed approach transforms the repair process from exploratory trial-and-error into systematic, knowledge-guided issue resolution.

These results highlight that our proposed method provides significant and consistent performance gains over the baselines, showing the effectiveness and reliability of our approach.

Finding 1

Our approach achieves a Pass@1 score of 41.6% with DeepSeek-V3-0324, representing a 7.2% relative improvement over the previous state-of-the-art methods using the same model.

5.2 RQ2: Ablation Study

To understand the contribution of each component in SWE-Exp, we conduct ablation studies by systematically removing key components.

Table 2: Ablation study results.

Method	Pass@1	Δ
SWE-Exp	**41.6%**	-
w/o Comprehension Experience	**38.4%**	-3.2%
w/o Modification Experience	**39.0%**	-2.6%
w/o Dual-Agent	**39.4%**	-2.2%

We test the three main components of SWE-Exp: 1) **w/o Hierarchical experience bank** removes all experience components, reverting to agent specialization without experience guidance; 2) **w/o Multi-faceted Experience** no longer refers to the relevant past experiences to analyze the problem statements; 3) **w/o Modification Experience** does not use modification experiences to enhance the security and robustness of the original modification instruction.

As shown in Table 2, the removal of comprehension-related experiences leads to the most substantial performance drop among individual components, reducing Pass@1 from 41.6% to 38.4%. Excluding the modification-related experiences results in a smaller decrease to 39.0% (-2.6%), while removing the dual-agent setup leads

[2] https://huggingface.co/intfloat/multilingual-e5-large-instruct

to a Pass@1 of 39.4 (-2.4%). These results highlight the complementary roles of comprehension, modification, and coordination in the proposed approach.

The substantial impact of comprehension experiences (-3.2%) directly addresses our core motivation that existing agents operate as memoryless explorers, treating each problem in isolation. These experiences fundamentally transform how agents approach new issues by providing strategic guidance extracted from successful problem-solving patterns observed in our motivating example with CheckboxInput widgets. Without comprehension experiences, agents revert to the problematic behavior we identified—focusing on surface-level symptoms rather than understanding the underlying design patterns. Our multi-faceted experience bank design specifically captures these high-level diagnostic insights, enabling the Instructor agent to formulate more accurate hypotheses about root causes from the outset. The smaller but significant impact of modification experiences (-2.6%) demonstrates their complementary role in our dual-agent architecture, where they guide the Assistant agent in applying proven repair strategies while avoiding common pitfalls such as incomplete fixes or introducing regressions. The dual-agent framework's contribution (-2.2%) validates our architectural choice to separate strategic reasoning from tactical execution, addressing the cognitive overload problem that causes vanilla MCTS agents to over-rely on find actions while neglecting actual code modifications.

These results confirm that both comprehension and modification experiences contribute positively to system performance, with trajectory-guided problem comprehension playing a slightly more influential role. Even when only one type of experience is used, the system maintains most of its original performance and still achieves improvements over the baseline. Overall, the incorporation of hierarchical experience bank provides consistent and additive gains, validating our design for structured, stage-specific knowledge reuse.

Finding 2

Comprehension experiences contribute most significantly to performance improvements, reducing Pass@1 by 3.2% (from 41.6% to 38.4%) when removed, compared to 2.6% reduction for modification experiences (39.0%) and 2.2% for dual-agent architecture (39.4%).

5.3 RQ3: Impact of Experience Number

In this section, we analyze the impact of the number of experiences on the performance of SWE-Exp. We vary the number of experiences from 0 to 4, where 0 means no experience is used. The results are shown in Figure 3.

As shown in Figure 3, the relationship between experience number and performance demonstrates an increase-then-stable trend. Without any experiences, the system achieves 37.8% Pass@1. Performance peaks at 41.6% when using exactly 1 experience, representing a 3.8% improvement over the method without experiences. However, increasing the number of experiences beyond 1 leads to diminishing returns: using 2 experiences drops performance to 40.4%, while 3 and 4 experiences achieve 40.2% and 39.6% respectively.

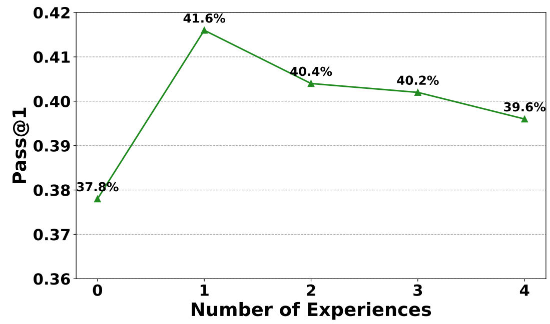

Figure 3: Impact of the number of experiences.

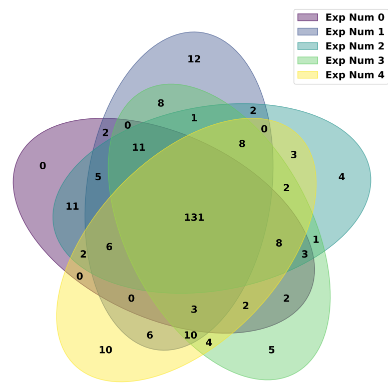

Figure 4: Unique Issues Resolved under Varying Experience Settings.

This pattern demonstrates that while relevant experiences can significantly enhance performance, excessive guidance can impair effectiveness by overwhelming or distracting the agent. The optimal configuration uses a single, carefully selected experience that provides targeted guidance without introducing cognitive burden or conflicting information. This finding underscores the importance of selective experience retrieval and highlights the need for quality over quantity in experience-driven agent systems.

When comparing the resolved instances across different numbers of retrieved experiences (from 0 to 4), we observe substantial variability in the subsets of issues successfully solved. As illustrated in Figure 4, a large number of issues (131) are consistently resolved across all settings. When utilizing past experiences, each experience demonstrates the ability to uniquely resolve specific instances. This suggests that mitigating the misleading influences within individual experiences, while allowing the accumulation of experience to contribute positively, may further enhance the agent's ability to resolve issues.

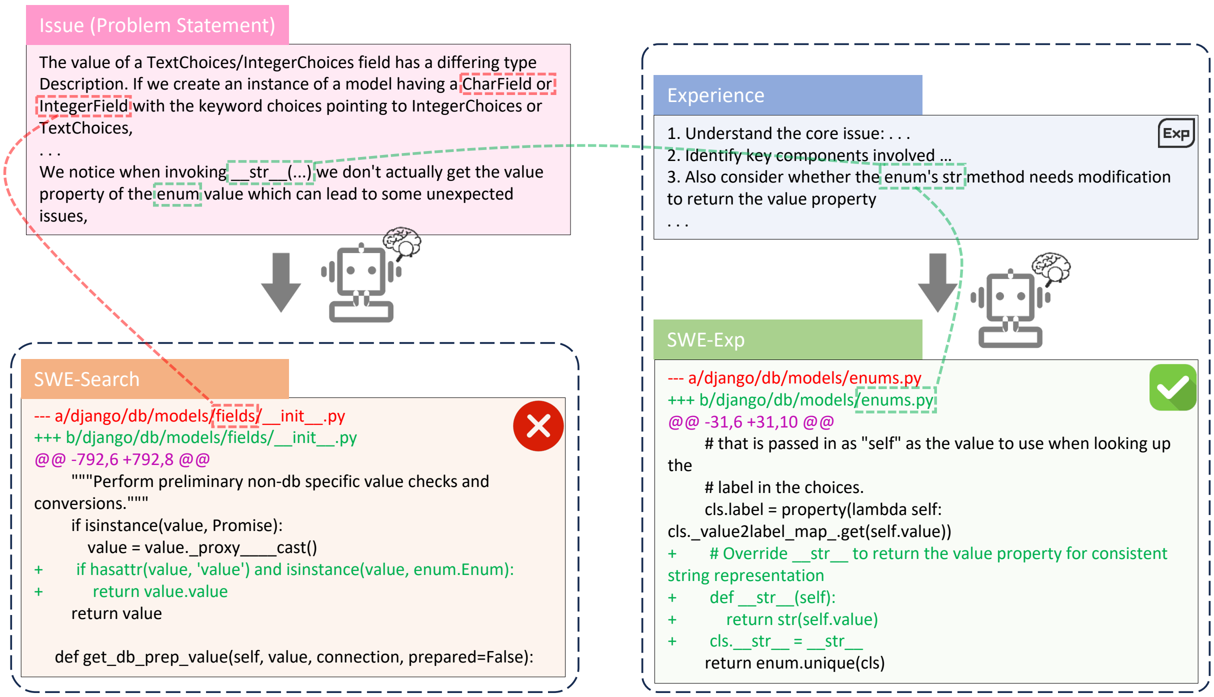

Figure 5: Case study of SWE-Exp on instance *django-11964*

> **Finding 3**
>
> Retrieving one single experience is sufficient to achieve optimal performance, reaching 41.6% Pass@1. This demonstrates that our experience retrieval mechanism can effectively identify and leverage the most relevant knowledge while avoiding information overload, enabling focused and efficient issue resolution guidance.

5.4 Case Study

To further verify the effectiveness of SWE-Exp in real-world scenarios, We compare two agent trajectories on the same SWE-bench instance—with and without experience reuse—to demonstrate how retrieved experiences influence the agent's decision-making and contribute to successful repair. The results are shown in Figure 5.

This case examines a fault related to TextChoices and IntegerChoices, which constitute the core focus of the problem statement. Although the context briefly mentions CharField and IntegerField, the key concern lies in how enum class behaves when converted to string. Specifically, invoking str function on such enum values doesn't yield expected value attribute.

An initial attempt to address this issue by agent specialization involved modifying the get_db_prep_value method within Django's model field handing code. This patch introduced a conditional check to manually extract the value attribute from enum class. Although this solution fixed the immediate problem, it did so in the wrong place. The agent produced a patch that attempts to handle enum conversion within the get_prep_value method; however, this modification fails to resolve the actual issue. The root causes of this failure is that: misleading surface-level correlations — the agent incorrectly associated the need for enum conversion with the get_prep_value method based on its docstring, without accounting for its actual invocation context and the emphasis in the problem statement.

In contrast, the subsequent fix - developed after incorporating comprehension experience — identified the enum's __str__ method as the appropriate point of intervention. The patch defined the __str__ method within the enum definition to return self.value, thereby ensuring consistent and intuitive string representations throughout the framework.

This case demonstrates the practical value of transferring knowledge across repositories. Without exposure to prior examples, especially those involving similar symptoms but differing root causes, the model might have repeated the same architectural mistake. However, by leveraging cross-repository experience, it was able to identify the correct point of intervention and propose a solution that was both technically sound and idiomatic to Django's codebase. Without referring to the prior experience, the model might have repeated the same architectural mistake. However, by leveraging cross-repository experience, it was able to identify the correct point of intervention and propose a solution that was both technically sound and idiomatic to Django's codebase.

6 DISCUSSION

6.1 Data Leakage

One critical concern about our experience-driven framework is the threat of knowledge leakage [45]. Specifically, in datasets such as SWE-bench, multiple instances from the same repository may correspond to closely related or even identical buggy code segments. If experiences are retrieved from the same repository as the target instance, especially via similarity-based matching, there is a high risk that the agent leverages repository-specific signals or implicitly accesses ground-truth-relevant information. This can lead to artificially inflated performance and fails to demonstrate the true generalizability of the extracted experiences. To avoid data leakage, we explicitly exclude all experiences from the same repository as the current instance before retrieval. This ensures that the selected experiences do not contain repository-specific artifacts or ground-truth-adjacent code snippet, allowing us to more accurately assess the cross-repository generalizability of experience reuse. Our experimental results further support the generalizability of our extracted experiences across repositories, as shown in Table 1.

6.2 Experience Quality

Although experiences have been shown to enhance the ability of agent, they could also introduce misleading thoughts—particularly at the problem comprehension stage. In early implementations, we allowed the *Instructor* to explicitly cite comprehension experiences as part of its thinking and instructions. While this made the decision process interpretable, it led to too much dependence: the Instructor continued using experience even after enough environment exploration, sometimes applying inappropriate strategies. Therefore adding experience as message context—without forcing instructor to use experiences—was the most effective, avoiding inflexible or misleading in the late issue resolution. This illustrates the misleading nature of experience during the comprehension stage: even when the environment has already been sufficiently explored, the agent may continue relying on past experience, leading to inappropriate strategies. This tendency also aligns with a broader trend observed in our quantitative evaluation in Figure 3: increasing the number of experiences beyond one led to a steady decline in performance. While a single experience boosted Pass@1 from 37.2% to 41.6%, adding more examples degraded performance, dropping to 39.8% with two experiences and further declining to 39.0% with four experiences. These results suggest that excessive experience may impair the agent's ability to focus and generalize effectively to the current issue.

Moreover, increasing the number of past trajectories used for experience generalization tended to introduce irrelevant or conflicting information, which negatively impacted the agent's effectiveness on the current issue, as the model found it harder to focus on the most relevant information and was more likely to rely on irrelevant or confusing information. In contrast, modification experience showed higher robustness: since the specific direction of the modification instruction is already decided by the Instructor, the Assistant can better assess whether a given experience makes sense or not. Overall, providing one single relevant experience alongside the interaction history yields the best performance for the Instructor.

6.3 Limitations and Future Directions

While these results are promising, SWE-Exp still faces several limitations. Its effectiveness depends on the quality of extracted experiences and their relevance to the target issue; if the retrieved knowledge fails to align with the current problem's semantics, performance may degrade. Moreover, the agent currently lacks a robust mechanism to assess the applicability of prior experiences in novel contexts, which may lead to inappropriate reuse or misleading to irrelevant patterns.

Future work may address these challenges by developing more robust experience extraction methods that better filter noise and identify transferable knowledge patterns. In addition, exploring more accurate retrieval and alignment techniques, incorporating confidence estimation or applicability scoring, and integrating formal verification can further enhance the reliability and adaptability of experience-driven agents in dynamic and unfamiliar code environment.

7 THREATS TO VALIDITY

Internal. The first internal threat comes from our reliance on a single underlying language model for all agent interactions. This choice may introduce model-specific biases and limit the generalizability of our findings across different LLM architectures. To address this concern, we follow prior work in automated program repair [38, 40] that demonstrates effectiveness on single-model evaluations, and our dual-agent architecture with experience bank can be readily adapted to other state-of-the-art models.

Another internal threat stems from potential data leakage, where SWE-bench instances may have been included in the training data of our underlying language model. While DeepSeek-V3-0324 is open-source, its training data composition is not publicly disclosed, making it impossible to verify potential overlap with our evaluation dataset. In the experiments, our approach shows consistent improvements over strong baselines that use the same underlying models, indicating that gains arise from our architectural innovations rather than training data advantages.

External. The primary external threat concerns the generalizability of our approach beyond Python repositories and the specific issue types present in SWE-bench. Our evaluation focuses exclusively on Python-based open-source projects, limiting our ability to demonstrate cross-language effectiveness. However, our approach is fundamentally language-agnostic, as it captures high-level issue resolution patterns like problem comprehension and modification experiences rather than language-specific syntax or semantics. The SWE-Exp framework should theoretically transfer to other programming languages, making cross-language evaluation a promising direction for future work.

8 RELATED WORK

8.1 Repository-Level Issue Resolution

Repository-level issue resolution automatically identifies and resolves issues across multiple files within a software project, requiring understanding of complex dependencies and maintaining code consistency [14]. Recent approaches leverage large language models to develop solution frameworks that can be categorized into agentic and non-agentic paradigms. Agentic frameworks treat language models as autonomous agents that step-by-step interact with code environments, with SWE-Agent [40] introducing a foundational agent-computer interface for repository-level interactions. Building on this foundation, several systems have enhanced specific capabilities: AutoCodeRover [47] and SpecRover [27] focus on improved localization and agent support mechanisms, while OpenHands CodeAct [17] provides comprehensive tooling frameworks.

Advanced exploration methods include SWE-Search [1], which uses Monte Carlo Tree Search for systematic solution space exploration, and CodeR [2], which employs multi-agent frameworks with pre-defined task graphs for collaborative issue resolution.

Non-agentic pipelines focus on specialized execution workflows, with Agentless [38] decomposing repair into distinct phases of localization, repair, and validation. CodeMonkeys [8] explores scaling test-time compute through iterative codebase editing with concurrent testing, while recent work [12] demonstrates that long-context language models with proper prompting can compete with complex agent systems. Training-based approaches have emerged to create SWE-bench-like instances for specialized fine-tuning [23, 24, 41], with MCTS-Refined CoT [33] using Monte Carlo Tree Search and reflection mechanisms to generate high-quality training data for substantial performance improvements.

Despite remarkable progress in performance results, existing approaches face several critical limitations that hinder their practical effectiveness. Current evaluations rely mainly on static offline datasets, raising concerns about solution memorization and configuration-specific optimizations rather than genuine algorithmic advances [45]. While graph-based methods demonstrate effective fault localization and Monte Carlo Tree Search-based exploration shows potential for higher-quality fixes, these methods often provide limited improvements in patch quality due to substantial computational costs and frequent failure to identify correct solutions after extensive search [1, 13]. Analysis of agent behavior reveals common failure patterns including overthinking and premature disengagement that further limit effectiveness [5]. Most critically, existing approaches lack systematic methods to learn from repair experiences, resulting in repeated exploration of failed strategies and missed opportunities to leverage successful patterns from previous attempts [9, 31, 48].

EvoCoder [15] introduces a promising multi-agent continuous learning framework for issue code reproduction that uses reflection mechanisms allowing LLMs to continuously learn from previously resolved problems and dynamically improve strategies for new challenges. Building on this valuable insight of leveraging historical experiences, our SWE-Exp further extends the experience-driven paradigm to the complete repository-level issue resolution workflow. SWE-Exp introduces a comprehensive experience-enhanced framework that captures and leverages structured experiences across multiple stages of the issue resolution process—from initial problem comprehension to final code modification. Through systematic distillation of multi-faceted experiences (problem comprehension and modification patterns) and their application via a dual-agent architecture, SWE-Exp transforms the entire repair workflow from isolated problem-solving into strategic, experience-guided issue resolution.

8.2 Experience Enhanced AI Agents

AI agents have fundamentally transformed how we approach complex computational tasks by providing autonomous reasoning and decision-making capabilities that can adapt to diverse problem contexts [7, 11, 29, 37, 43, 44]. In order to enhance their ability to accumulate and leverage knowledge from past experiences, experience-enhanced agent architectures are proposed [18, 20, 22, 25, 39].

Early foundational work in experience-enhanced AI agents focused on developing human-like memory systems for better long-term interactions [31, 46]. OlaGPT [39] introduced cognitive simulation by adding memory and learning from mistakes to copy human-like thought processes. Think-in-Memory (TiM) [16] introduced a two-stage framework for recalling thoughts before generation and post-thinking for memory updates. This enables LLMs to maintain evolved memory without repeated reasoning. MemoryBank [49] separated long-term and short-term memory types to create more natural human-machine interactions, while MemGPT [22] used hierarchical storage levels with context priority strategies for extended information management. OpenAI's ChatGPT also added memory functionality through external memory layers to store user-specific information across sessions [21]. These foundational approaches showed the importance of persistent memory systems but mainly focused on conversational contexts rather than task-specific problem-solving.

Modern experience-based learning frameworks have evolved to capture and use procedural knowledge from agent interactions [20, 34, 46]. ExpeL [48] introduced autonomous experience gathering through natural language insights with weighted management systems (ADD, EDIT, UPVOTE, DOWNVOTE) for non-parametric learning. Building on this, AgentRR [9] introduced comprehensive record-and-replay systems that capture both environmental interactions and internal decision processes. AutoGuide [10] automatically generates context-aware guidelines from offline experiences using contrastive learning techniques. Advanced frameworks like CAIM [36] implement advanced cognitive AI-inspired architectures with specialized Memory Controller, Memory Retrieval, and Post-Thinking modules. Recent approaches emphasize learned routine development, with ExACT [42] combining Reflective Monte Carlo Tree Search with vector database storage for dynamic search efficiency improvement. Self-improving coding systems [26] achieve autonomous code editing through LLM-driven reflection mechanisms. However, existing frameworks mainly target general-purpose tasks and lack domain-specific optimizations for software engineering scenarios [38, 47]. SWE-Exp addresses this limitation by developing specialized experience architectures designed for repository-level issue resolution, capturing both strategic repair workflows and detailed code-level patterns.

9 CONCLUSION

We presented SWE-Exp, an experience-enhanced framework that transforms repository-level issue resolution from isolated exploration into experience-driven processes. By capturing and distilling knowledge from both successful and failed repair trajectories at multiple levels including comprehension and modification experiences, our dual-agent architecture leverages historical insights to guide strategic planning and tactical execution. Experimental evaluation on SWE-bench demonstrates significant effectiveness, achieving a Pass@1 score of 41.6%, establishing a new paradigm where automated agents systematically accumulate and leverage knowledge rather than relying on trial-and-error exploration. Future work can explore more advanced experience extraction mechanisms and integration with formal verification techniques to further enhance automated software engineering capabilities.

ACKNOWLEDGMENT

This research is funded by the National Key Research and Development Program of China (Grant No. 2023YFB4503802) and the Natural Science Foundation of Shanghai (Grant No. 25ZR1401175).

REFERENCES

[1] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2024. SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement. arXiv:2410.20285

[2] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. arXiv:2406.01304 [cs]

[3] Zhi Chen, Wei Ma, and Lingxiao Jiang. 2025. Unveiling Pitfalls: Understanding Why AI-driven Code Agents Fail at GitHub Issue Resolution. arXiv:2503.12374 [cs]

[4] Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. LocAgent: Graph-Guided LLM Agents for Code Localization. arXiv:2503.09089 [cs]

[5] Alejandro Cuadron, Dacheng Li, Wenjie Ma, Xingyao Wang, Yichuan Wang, Siyuan Zhuang, Shu Liu, Luis Gaspar Schroeder, Tian Xia, Huanzhi Mao, Nicholas Thumiger, Aditya Desai, Ion Stoica, Ana Klimovic, Graham Neubig, and Joseph E. Gonzalez. 2025. The Danger of Overthinking: Examining the Reasoning-Action Dilemma in Agentic Tasks.

[6] DeepSeek-AI. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs]

[7] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2024. Improving Factuality and Reasoning in Language Models through Multiagent Debate. In *Proceedings of the 41st International Conference on Machine Learning (ICML'24, Vol. 235)*. JMLR.org, Vienna, Austria, 11733–11763.

[8] Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. 2025. CodeMonkeys: Scaling Test-Time Compute for Software Engineering.

[9] Erhu Feng, Wenbo Zhou, Zibin Liu, Le Chen, Yunpeng Dong, Cheng Zhang, Yisheng Zhao, Dong Du, Zhichao Hua, Yubin Xia, and Haibo Chen. 2025. Get Experience from Practice: LLM Agents with Record & Replay. arXiv:2505.17716 [cs]

[10] Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. 2024. AutoGuide: Automated Generation and Selection of Context-Aware Guidelines for Large Language Model Agents. arXiv:2403.08978 [cs]

[11] Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. 2023. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation. arXiv:2312.13010

[12] Mingjian Jiang, Yangjun Ruan, Luis Lastras, Pavan Kapanipathi, and Tatsunori Hashimoto. 2025. Putting It All into Context: Simplifying Agents with LCLMs. arXiv:2505.08120 [cs]

[13] Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. CoSIL: Software Issue Localization via LLM-Driven Code Repository Graph Searching. arXiv:2503.22424 [cs]

[14] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *ICLR*.

[15] Yalan Lin, Yingwei Ma, Rongyu Cao, Binhua Li, Fei Huang, Xiaodong Gu, and Yongbin Li. 2024. LLMs as Continuous Learners: Improving the Reproduction of Defective Code in Software Issues. arXiv:2411.13941 [cs]

[16] Lei Liu, Xiaoyan Yang, Yue Shen, Binbin Hu, Zhiqiang Zhang, Jinjie Gu, and Guannan Zhang. 2023. Think-in-Memory: Recalling and Post-thinking Enable LLMs with Long-Term Memory. arXiv:2311.08719 [cs]

[17] Weijie Lv, Xuan Xia, and Sheng-Jun Huang. 2024. CodeACT: Code Adaptive Compute-efficient Tuning Framework for Code LLMs. arXiv:2408.02193 [cs]

[18] Yingwei Ma, Binhua Li, Yihong Dong, Xue Jiang, Rongyu Cao, Jue Chen, Fei Huang, and Yongbin Li. 2025. Thinking Longer, Not Larger: Enhancing Software Engineering Agents via Scaling Test-Time Compute. arXiv:2503.23803 [cs]

[19] Yingwei Ma and Yue Liu. 2025. Improving Automated Issue Resolution via Comprehensive Repository Exploration. In *ICLR 2025 Third Workshop on Deep Learning for Code*.

[20] Zexiong Ma, Chao Peng, Pengfei Gao, Xiangxin Meng, Yanzhen Zou, and Bing Xie. 2025. SoRFT: Issue Resolving with Subtask-oriented Reinforced Fine-Tuning. arXiv:2502.20127 [cs]

[21] OpenAI. 2024. Memory and New Controls for ChatGPT. https://openai.com/index/memory-and-new-controls-for-chatgpt/.

[22] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560 [cs]

[23] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training Software Engineering Agents and Verifiers with SWE-Gym.

[24] Minh V. T. Pham, Huy N. Phan, Hoang N. Phan, Cuong Le Chi, Tien N. Nguyen, and Nghi D. Q. Bui. 2025. SWE-Synth: Synthesizing Verifiable Bug-Fix Data to Enable Large Language Models in Resolving Real-World Bugs. arXiv:2504.14757 [cs]

[25] Chen Qian, Yufan Dang, Jiahao Li, Wei Liu, Zihao Xie, Yifei Wang, Weize Chen, Cheng Yang, Xin Cong, Xiaoyin Che, et al. 2023. Experiential co-learning of software-developing agents. *arXiv preprint arXiv:2312.17025* (2023).

[26] Maxime Robeyns, Martin Szummer, and Laurence Aitchison. 2025. A Self-Improving Coding Agent. arXiv:2504.15228 [cs]

[27] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. SpecRover: Code Intent Extraction via LLMs. arXiv:2408.02232 [cs]

[28] Yuchen Shao, Yuheng Huang, Jiawei Shen, Lei Ma, Ting Su, and Chengcheng Wan. 2025. Are LLMs Correctly Integrated into Software Systems?. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 1178–1190.

[29] Yuling Shi, Songsong Wang, Chengcheng Wan, and Xiaodong Gu. 2024. From Code to Correctness: Closing the Last Mile of Code Generation with Hierarchical Debugging. arXiv:2410.01215 [cs]

[30] Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. 2024. Between Lines of Code: Unraveling the Distinct Patterns of Machine and Human Programmers. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 51–62.

[31] Xiangru Tang, Tianrui Qin, Tianhao Peng, Ziyang Zhou, Daniel Shao, Tingting Du, Xinming Wei, Peng Xia, Fang Wu, He Zhu, et al. 2025. Agent KB: Leveraging Cross-Domain Experience for Agentic Problem Solving. *arXiv preprint arXiv:2507.06229* (2025).

[32] 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. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741

[33] Yibo Wang, Zhihao Peng, Ying Wang, Zhao Wei, Hai Yu, and Zhiliang Zhu. 2025. MCTS-Refined CoT: High-Quality Fine-Tuning Data for LLM-Based Repository Issue Resolution. arXiv:2506.12728 [cs]

[34] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2024. Agent workflow memory. *arXiv preprint arXiv:2409.07429* (2024).

[35] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. 2025. SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution. arXiv:2502.18449 [cs]

[36] Rebecca Westhäußer, Frederik Berenz, Wolfgang Minker, and Sebastian Zepf. 2025. CAIM: Development and Evaluation of a Cognitive AI Memory Framework for Long-Term Interaction with Intelligent Agents. arXiv:2505.13044 [cs]

[37] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155 [cs]

[38] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. arXiv:2407.01489

[39] Yuanzhen Xie, Tao Xie, Mingxiong Lin, WenTao Wei, Chenglin Li, Beibei Kong, Lei Chen, Chengxiang Zhuo, Bo Hu, and Zang Li. 2023. OlaGPT: Empowering LLMs With Human-like Problem-Solving Abilities. arXiv:2305.16334 [cs]

[40] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

[41] John Yang, Kilian Leret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. SWE-smith: Scaling Data for Software Engineering Agents.

[42] Xiao Yu, Baolin Peng, Vineeth Vajipey, Hao Cheng, Michel Galley, Jianfeng Gao, and Zhou Yu. 2025. ExACT: Teaching AI Agents to Explore with Reflective-MCTS and Exploratory Learning. arXiv:2410.02052 [cs]

[43] Jusheng Zhang, Yijia Fan, Wenjun Lin, Ruiqi Chen, Haoyi Jiang, Wenhao Chai, Jian Wang, and Keze Wang. 2025. GAM-Agent: Game-Theoretic and Uncertainty-Aware Collaboration for Complex Visual Reasoning. *arXiv preprint arXiv:2505.23399* (2025).

[44] Jusheng Zhang, Zimeng Huang, Yijia Fan, Ningyuan Liu, Mingyan Li, Zhuojie Yang, Jiawei Yao, Jian Wang, and Keze Wang. 2025. KABB: Knowledge-Aware Bayesian Bandits for Dynamic Expert Coordination in Multi-Agent Systems. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=AKvy9a4jho

[45] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2025. SWE-bench Goes Live! arXiv:2505.23419 [cs]

[46] Wenqi Zhang, Ke Tang, Hai Wu, Mengna Wang, Yongliang Shen, Guiyang Hou, Zeqi Tan, Peng Li, Yueting Zhuang, and Weiming Lu. 2024. Agent-pro:

Learning to evolve via policy-level reflection and optimization. *arXiv preprint arXiv:2402.17574* (2024).

[47] 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 (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604.

[48] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. ExpeL: LLM Agents Are Experiential Learners. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 17 (March 2024), 19632–19642.

[49] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. MemoryBank: Enhancing Large Language Models with Long-Term Memory. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'24/IAAI'24/EAAI'24, Vol. 38)*. AAAI Press, 19724–19731.

A HYPERPARAMETERS OF MCTS

The Monte Carlo Tree Search (MCTS) algorithm [14] used in this study employs hyperparameters in Table 3.

Table 3: MCTS Hyperparameters

Hyperparameter	Description	Default
Main Search Parameters		
c_param	UCT exploration parameter	1.41
max_expansions	Max children per node	3
max_iterations	Max MCTS iterations	20
provide_feedback	Enable feedback	True
best_first	Use best-first strategy	True
value_function_temperature	Value function temperature	0.2
max_depth	Max tree depth	20
UCT Score Calculation Parameters		
exploration_weight	UCT exploration weight	1.0
depth_weight	Depth penalty weight	0.8
depth_bonus_factor	Depth bonus factor	200
high_value_threshold	High-value node threshold	55
low_value_threshold	Low-value node threshold	50
very_high_value_threshold	Very high-value threshold	75
high_value_leaf_bonus_constant	High-value leaf bonus	20
high_value_bad_children_bonus_constant	High-value bad children bonus	20
high_value_child_penalty_constant	High-value child penalty	5
Action Model Parameters		
action_model_temperature	Action model temperature	0.7
Discriminator Parameters		
number_of_agents	Number of Discriminator Agents	5
number_of_round	Number of debate rounds	3
discriminator_temperature	Discriminator temperature	1

B COMPARISON WITH VANILLA RAG

In Table 5, we compare two RAG-based approaches: 1. Direct Issue Patch: We use issue–patch pairs from retrieved instances with similar error types as demonstrations for in-context learning (ICL) without experiences extraction; 2. RAG w/o LLM-Reranking: We use the most similar retrieved experiences without LLM reranking for issue resolution.

Table 4: Comparison with vanilla RAG.

Method	Pass@1	Δ
SWE-Exp	**41.0%**	-
w/o Experiences Extraction	**36.0%**	**-5.0%**
w/o LLM Reranking	**38.2%**	**-2.8%**

C HEAD-TO-HEAD COMPARISON

For memory-enhanced agent, we adapted EvoCoder, a experience-enhanced agent that leverages both intra-repository and cross-repository experience to reproduce errors, for the issue resolution task, achieving 38.0% Pass@1 on SWE-bench-Verified.

For multi-agent method, we also reproduced another multi-agent approach, CodePlan, where a PlanAgent decomposes the problem statement into sequential sub-goals solved by specialized agents via moatless-tools. In contrast, our method achieves substantially better performance, while CodePlan only reached 35.2% Pass@1.

For graph-guided agent, we also evaluated LocAgent [4], a multi-agent approach that leverages code graph structures and tool-driven search. It achieved 37.4% Pass@1, still lower than our method.

To assess compatibility, we integrated Skywork-SWE-32B into our framework and evaluated it on a subset of 75 instances (25 each from Django, SymPy, and Sphinx), achieving a Pass@1 of 37/75 compared to 29/75 without our method, as summarized in Table 6. This empirical evidence indicates that our framework operates orthogonally to training-enhanced repair models, enabling seamless integration.

Table 5: Head-to-Head Comparison with representative related studies.

Method	Pass@1
SWE-Exp	42.6%
EvoCoder	38.0%
CodePlan	35.2%
LocAgent + SWE-Search	37.4%

Table 6: Experimental results of Skywork-SWE-32B.

Model	Pass@1	Δ
Skywork-SWE-32B	**49.33%**	-
w/o SWE-Exp	**38.67%**	**-10.66%**

D VARIANTS

Table 7 reports the results of our method with and without the testbed, while Table 8 compares results when experiences from the target repository are either included or excluded. Experimental results show that our method can further surpass the current method of the same model by equipping with the testbed or internal experiences.

Table 7: Experimental results w/ and w/o testbed.

Variants	Pass@1
SWE-Exp w/ testbed	42.0%
SWE-Exp w/o testbed	41.0%

Table 8: Experimental results w/ and w/o internal experiences.

Variants	Pass@1
SWE-Exp w/ internal	42.6%
SWE-Exp w/o internal	41.0%

E ADDITIONAL MODELS

As shown in Table 6, we evaluate GPT-4o with SWE-Exp on SWE-Bench-Verified. Notably, our approach continues to perform robustly on the GPT-4o model, outperforming state-of-the-art method for the same model (AutoCodeRover, 38.4%), which highlights the generalizability and effectiveness of our method.

Table 9: Experimental results with different models.

Method	Model	Pass@1
Agentless	🤠 DeepSeek-V3-0324	36.6%
	🔒 GPT-4o (2024-05-13)	36.2%
SWE-Agent	🤠 DeepSeek-V3-0324	38.8%
	🔒 Claude-3.5 Sonnet	33.6%
	🔒 GPT-4o (2024-05-13)	23.0%
SWESynInfer	🔒 Claude-3.5 Sonnet	35.4%
	🔒 GPT-4o (2024-05-13)	31.8%
	🤠 Lingma SWE-GPT 72B	32.0%
SWE-Search	🤠 DeepSeek-V3-0324	35.4%
Moatless Tools	🤠 DeepSeek-V3-0324	34.6%
AutoCodeRover	🔒 GPT-4o (2024-05-13)	38.4%
CodeAct	🔒 GPT-4o (2024-05-13)	30.0%
OpenHands	🤠 DeepSeek-V3-0324	38.8%
EvoCoder	🤠 DeepSeek-V3-0324	38.0%
CodePlan	🤠 DeepSeek-V3-0324	35.2%
LocAgent + SWE-Search	🤠 DeepSeek-V3-0324	37.4%
SWE-Exp	🤠 DeepSeek-V3-0324	**41.0%**
	🔒 GPT-4o (2024-05-13)	**40.6%**

F COST ANALYSIS AND TOOLSETS

Table 10 presents the cost comparison of DeepSeek-V3-0324 between SWE-Exp (SWE-Exp) and SWE-Search. While SWE-Exp employs a more sophisticated dual-agent architecture together with retrieval, the additional overhead is modest. Specifically, the average token usage only slightly increases (203.3K vs. 189.1K), and the average USE cost remains nearly unchanged ($0.13 vs. $0.12). Although retrieval adds 37s to the pipeline, the total wall time is only marginally longer (15min 49s vs. 12min 37s). These results highlight that the performance improvements of SWE-Exp are achieved with minimal additional computational and monetary costs.

Table 10: Efficiency Metrics.

Metrics	SWE-Exp	SWE-Search
Average Token Costs	203.3K	189.1K
Average USD Costs	$0.13	$0.12
Average Wall Time	15min 49s	12min 37s
Average Retrieval and rerank time	37.5s	0s

G IMPACT OF EXPERIENCE BANK SIZE

We conducted experiments on a subset with 75 instances (25 from Django, 25 from SymPy, and 25 from Sphinx). As shown in Figure 6, we analyzed the effect of experience-bank growth by adding experiences in increments of 100. We observed that Pass@1 steadily increases until around 300 experiences. Beyond 300 experiences, Pass@1 enters a plateau, exhibiting only minor fluctuations of 1–2 points. All experience additions follow the chronological order, simulating realistic accumulation of experience over time.

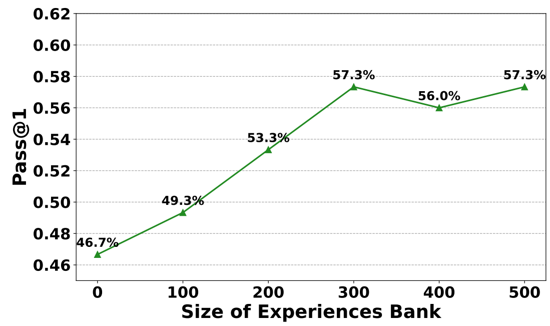

Figure 6: Impact of the number of experiences.

H PROMPT TEMPLATES

In the following section, we provide a comprehensive enumeration of all prompts employed throughout our workflow, including the system prompts used by the dual-agent architecture, the prompts designed for extracting successful and failed experiences, and those used for reusing past experiences. This detailed documentation aims to ensure reproducibility and to highlight the role of prompt engineering in the effectiveness of our method.

H.1 Instructor

Prompt 1: Instructor Prompt

```
You are an autonomous AI instructor with deep
     analytical capabilities. Operating
     independently, you cannot communicate with
     the user but must analyze the past history of
      interactions with the code repository to
     generate the next instruction that guides the
      assistant toward completing the task.

# Workflow to guide assistants in modifying code

Follow these structured steps to understand the
     task and instruct the assistant to locate
     context, and perform code modifications.

### 1. Understand the Task
    - Carefully read the <task> to determine
         exactly what is known and what still
         needs to be clarified according to the
         interaction history.
    - Focus on the cause of the <task> and
         suggested changes to the <task> that have
          been explicitly stated in the <task>.
```

```
    - Compare <task> with the code from the
        interaction history, determine what
        additional context (files, functions,
        dependencies) may be required. Request
        more information if needed.

### 2. Locate Code
    - Using your analysis, generate instructions
        to guide assistant to locate the exact
        code regions to understand or modify.
    - Once the location of the code that needs to
        be modified is determined, instruct
        assistant to modify it and provide the
        exact location.
    - Narrow down the scope of the code you need
        to look at step by step.

### 3. Modify Code
    - The generated instruction should only focus
        on the changes needed to satisfy the task
        . Do not modify unrelated code.
    - The instructions for modifying the code need
         to refer to the task and the relevant
        code retrieved, rather than being based
        on your own guesses.
    - Keep the edits minimal, correct, and
        localized.
    - If the change involves multiple locations,
        apply atomic modifications sequentially.

### 4. Iterate as Needed
    - If the task has already been resolved by the
         existing code modifications, finish the
        process without making additional changes
        .
    - If the task is not fully resolved, analyze
        what remains and focus only on the
        unresolved parts.
    - Avoid making unnecessary changes to
        previously correct code modifications.
        Subsequent edits should strictly target
        the remaining issues.
    - When modifying the input parameters or
        return values of a function or class,
        make sure to update all relevant code
        snippets that invoke them accordingly.
    - But do not take test into account, just
        focus on how to resolve the task.
    - Repeat until the task are resolved.

### 5. Complete Task
    - Once the implementation satisfies all task
        constraints and maintains system
        integrity:
      - Do not add additional test cases.
      - Stop the task.

# Additional Notes

 * **Think Step by Step**
   - Always document your reasoning and thought
       process in the Thought section.
   - Only one kind of instruction is generated
       each step.

 * **Efficient Operation**
   - Use previous observations to inform your next
       actions.
   - Avoid instructing assistant to execute
       similar actions as before.
   - Focus on minimal viable steps: Prioritize
       actions that maximize progress with
       minimal code exploration or modification.

 * **Never Guess**
   - Do not guess line numbers or code content.
   - All code environment information must come
       from the real environment feedback.

# Instructor Output Format
For each input, you must output a JSON object with
      exactly three fields:
    1. thoughts: A natural language description
        that summarizes the current code
        environment, previous steps taken, and
        relevant contextual reasoning.
    2. instructions:
        - One specific and actionable objective
            for the assistant to complete next.
            This should be phrased as a goal
            rather than an implementation detail,
             guiding what should be achieved
            based on the current context.
        - Instruction related to modifying the
            code must strictly refer to the task
            at the beginning, and you shouldn't
            guess how to modify.
        - Do not include any instructions related
            to test cases.
        - The more detailed the better.
    3. context:
        - If the next step involves retrieving
            additional context according to the
            previous observations, ensure the
            context includes the following
            specific details from the code
            environment (as applicable):
            -- Exact file path or vague file
                pattern(e.g., **/dictionary/*.py)
            -- Exact Class names from environment
                feedback
            -- Exact Function names from
                environment feedback
            -- Exact Code block identifiers from
                environment feedback (e.g.,
                method headers, class
                declarations)
            -- Exact Corresponding line ranges
                from environment feedback (
                start_line and end_line)
            -- The span ids of the code you hope
                to view
        - If the code environment is uncertain or
            specific classes and functions cannot
             be retrieved multiple times,
            -- Only output a natural language
                query describing the
                functionality of the code that
                needs to be retrieved, without
                exact file, class, function, or
                code snippets.
```

```
        - If the next step needs to modify the
            code, the context must contain
            specific file path.
        - If the task is complete, this could
            return `None`.
        - Don't guess the context, the context
            must come from the interaction with
            the code environment.
    4. type: A string indicating the kind of next
        action required. Must be one of:
        - "search": when more information is
            needed,
        - "view": when additional context not
            returned by searches, or specific
            line ranges you discovered from
            search results
        - "modify": when you have identified the
            specific code to be modified or
            generated from the code environment
            feedback.
        - "finish": when the task has been solved.

The instructor's output must follow a structured
     JSON format:
{
  "thoughts": "<analysis and summary of the
       current code environment and interaction
       history>",
  "instructions": "<next objective for the
       assistant and some insights from the
       previous actions>",
  "context": "<the description or query that
       summarizes the code environment that needs
       to be known in the next step>",
  "type": "<search | view | modify | finish>"
}
```

H.2 Assistant

Prompt 2: Assistant Prompt

```
# Guidelines for Executing Actions Based on
     Instructions:

1. Analysis First:
    - Read the problem statement in <task> to
        understand the global goal.
    - Read the instructor's instruction in <
        instruction> to understand the next
        action.

2. Analyze Environment, Interaction History and
     Code Snippet:
    - If the next action requires retrieving more
        context, carefully extract precise
        targets from the <environment>. These may
         include relevant file names, class names
        , function names, code block identifiers,
         or corresponding line ranges, depending
        on what is available in the context.
    - Actions and their arguments from the past
        interactions are recorded in <history>.
        Your next action should retrieve content
        that is not redundant with those previous
         actions.
    - If the next action involves modifying code,
        use the <environment> to get the target
        path and identify the exact code snippet
        that needs to be changed in <code>, along
         with its surrounding logic and
        dependencies. This ensures the
        modification is accurate, consistent, and
         context-aware.

2. EVERY response must follow EXACTLY this format:
   Thought: Your reasoning and analysis
   Action: ONE specific action to take

3. Your Thought section MUST include:
   - What you learned from previous Observations
   - Why you're choosing this specific action
   - What you expect to learn/achieve
   - Any risks to watch for

# Action Description
1. **Locate Code**
  * **Primary Method - Search Functions:** Use
       these to find relevant code:
      * FindClass - Search for class definitions
          by class name
      * FindFunction - Search for function
          definitions by function name
      * FindCodeSnippet - Search for specific code
           patterns or text
      * SemanticSearch - Search code by semantic
          meaning and natural language
          description
  * **Secondary Method - ViewCode:** Only use when
        you need to see:
      * Additional context not returned by
          searches but in the same file
      * Specific line ranges you discovered from
          search results
      * Code referenced in error messages or test
          failures

2. **Modify Code**
  * **Fix Task:** Make necessary code changes to
       resolve the task requirements
  * **Primary Method - StringReplace:** Use this
       to apply code modifications
    - Replace exact text strings in files with new
        content
    - The old_str argument cannot be empty.
  * **Secondary Method - CreateFile:** Only use
       when you need to need to implement new
       functionality:
    - Create new files with specified content

3. **Complete Task**
  * Use Finish when confident all applied patch
       are correct and complete.

# Important Guidelines

 * **Focus on the Specific Instruction**
   - Implement requirements exactly as specified,
       without additional changes.
   - Do not modify code unrelated to the task.

 * **Code Context and Changes**
```

```
    - Limit code changes to files in the code you
        can see.
    - If you need to examine more code, use
        ViewCode to see it.

  * **Task Completion**
    - Finish the task only when the task is fully
        resolved.
    - Do not suggest code reviews or additional
        changes beyond the scope.

# Additional Notes

  * **Think Step by Step**
    - Always document your reasoning and thought
        process in the Thought section.
    - Build upon previous steps without unnecessary
         repetition.

  * **Never Guess**
    - Do not guess line numbers or code content.
        Use ViewCode to examine code when needed.
```

H.3 Issue Agent

Prompt 3: Issue Agent Prompt

```
You are an expert error classification assistant.
     Your task is to analyze string-formatted
     issue reports and identify the type of error
     they contain.
For each input, you must output a JSON object with
      exactly two fields:

1. `issue_type`: The generalized error category in
      the format "<generalized_descriptive_name>
     Error" (e.g., "SyntaxError", "
     NullReferenceError")
2. `description`: A brief description (1-2
     sentences) of the characteristics of the
     identified error category

Your output should strictly follow JSON format
     with the following structure:
{
    "issue_type": "<generalized_descriptive_name>
         Error",
    "description": "<the brief description>",
}
```

H.4 Issue Comprehension ExpAgent

H.4.1 *Successful Experience Extraction Prompt.*

Prompt 4: Issue Comprehension ExpAgent (Success)

```
You are a bug resolution expert. You will be given
      a software issue, the corresponding golden
     patch and a trajectory that represents how an
      agent successfully resolved this issue.

## Guidelines
You need to extract two key aspects from this
     successful trajectory:
1. **perspective** - how this trajectory thought
     about this issue - that is, how the problem
     was understood in a way that **led to its
     successful resolution**. This should be
     abstract and not name specific code entities.

## Important Notes:
    - Perspective should be at the level of
        thinking, not specific implementation
        details.
    - Perspective and reasoning should be
        expressed in as generalized and abstract
        terms as possible.
    - Do not include specific object names in
        perspective.

Your output must strictly follow the JSON format
     shown below:
{
    "perspective": "<1-2 sentences to describe how
          this trajectory understood this issue>",
}
```

H.4.2 *Failed Experience Extraction Prompt.*

Prompt 5: Issue Comprehension ExpAgent (Failure)

```
You are a bug resolution expert. You will be given
      a software issue, the corresponding golden
     patch and a trajectory that represents how an
      agent attempted to resolve this issue but
     failed.

## Guidelines
You need to extract some reflections from this
     failed trajectory according to the golden
     patch:
1. **reflections** - three reflections on why this
      trajectory failed to resolve this issue, you
      need to consider the following aspects:
    - `Perspective`: Explain how should you
        correctly understand the issue according
        to the golden patch.
    - `Modification`: If the trajectory correctly
        identified the modification location,
        what mistakes were made in actual code
        modification?

## Important Notes:
    - Reflections should be at the level of
        thinking, not specific implementation
        details.
    - Reflections should be expressed in as
        generalized and abstract terms as
        possible.
    - Be comprehensive and detailed as possible.
    - Do not include specific object names in the
        output.

Your output must strictly follow the JSON format
     shown below:
{
    "perspective": [
        "<one key reflection>",
        ...
        ],
```

```
    "modification": [
        "<one key reflection>",
        ...
        ]
}
```

H.5 Modification ExpAgent

Prompt 6: Modification ExpAgent Prompt

```
You are a software patch refinement expert. You
     will be given a software issue, a successful
     trajectory that shows how the agent modified
     the code to fix the bug, and the agent-
     generated patch which successfully resolved
     this issue.

Your job is to:
1. Compare the generated patch with the issue,
     determine why this patch could resolve this
     issue and how to resolve this kind of issue.
2. Analyze the successful trajectory and decide
     which code modification is vital to resolve
     this issue.

## Guidelines
Your need to extract and summarize one key insight
      based on the agent's successful patch:
1. **experience** - abstract the reasoning behind
     this code change. What principle, pattern, or
      insight can be generalized from this fix and
      applied to future debugging cases?

## Important Notes:
    - experience explains *why* the fix worked, in
        abstract and transferable terms.
    - You could extract *at most three*
        experiences.
    - Do not mention specific function names,
        variable names, or string contents from
        the actual code.

## Output Format
Your output must strictly follow the JSON format
     shown below:
{
    "modification": {
        "experience": [
            "<1-2 sentences summarizing the
                 abstract insights learned from
                 making this fix.>",
            ...
        ]
    }
}
```

H.6 RerankAgent

Prompt 7: RerankAgent Prompt

```
You are a knowledgeable issue resolution assistant
     . Your task is to analyze a current issue and
      identify the most relevant past experience
     that can help resolve it.

You will be given:
- A `problem_statement` describing the current
     issue
- A set of past trajectories, each with:
  - `issue_id`: A unique identifier
  - `issue_description`: The description of the
       past issue
  - `experience`: Either a `perspective` (how this
        successful trajectory understood this
       issue) or `reflections` (insights gained
       from an unsuccessful trajectory)

Your job is to:
1. Compare the current `problem_statement` with
     each past trajectory's `issue_description`
     and `experience`.
2. Select up to **{k}** past experiences - choose
     only those that are clearly relevant and
     potentially helpful for resolving the current
      issue.
3. You must select **at least one** experience,
     even if fewer than {k} are strongly relevant.

You should **prioritize trajectories whose problem
     -solving approach (as described in the
     perspective) aligns closely with the current
     issue**.

You must output a JSON object with exactly two
     fields for each selection:
- `issue_id`: ID of the past issue
- `reason`: A short explanation of why this issue
     and experience was selected

Your output must strictly follow the JSON format
     below:
{{
    "issue_id": {{
        "reason": "<why you select this issue and
             corresponding experience>"
    }},
    ...
}}
```

H.7 Reuser

H.7.1 Reuse Comprehension Experience Prompt.

Prompt 8: Reuser – Reuse Comprehension Experience Prompt

```
You are a knowledgeable issue resolution assistant
     . Your task is to analyze a current issue and
      generalize the received experiences into a
     new insight that is applicable to this issue.

You will be given:
```

```
- A `problem_statement` describing the current issue
- A past trajectory with:
  - `issue_description`: The description of the past issue
  - `experience`: Either a `perspective` (how this successful trajectory understood this issue) or `reflections` (insights gained from an unsuccessful trajectory)

Your job is to:
1. Compare the current `problem_statement` with each past trajectory's `issue_description` and `experience`.
2. Adapt the old experience to the current issue and produce a new applicable experience.
3. Identify the most likely entry point in the codebase - based on the problem statement - that is critical to resolving the current issue.

You must output a JSON object with exactly one field:
- `new_experience`: A new experience statement tailored to the current issue, based on the old experience. **The more detailed the better**

Your output must strictly follow the JSON format below:
{
    "new_experience": "<the new experience>"
}
```

H.7.2 *Reuse Modification Experience Prompt.*

Prompt 9: Reuser – Reuse Modification Experience Prompt

```
You are a strategic assistant helping an agent improve its next-step instruction in a debugging task.

You are given:
- A `problem_statement`: a natural language description of the current software problem
- A `current_code_exploration_history`: The recent exploration steps taken to understand or debug the current codebase. This may include what has been examined, eliminated, or hypothesized so far.
- An `instruction`: the next step the agent is expected to take
- A list of `experiences`: each offering past insights about how to better approach the corresponding issue.

Your task is to:
1. Analyze how the current `instruction` relates to the given `issue` and `current_code_exploration_history`
2. Identify useful, transferable, generalized insights from the past experiences of **modification** type
3. Based on those insights, rewrite the instruction to make it more robust, strategically informed, and better suited to succeed in this situation

### Important Notes
    - Focus only on experience of **modification**, and ensure the improved instruction aligns with the original goal but incorporates better reasoning or coverage
    - NEVER add the content that are not related to solving the current problem

Output only the following JSON structure:
{
  "enhanced_instruction": "<A single improved and robust instruction, rewritten based on relevant experience of modification type>"
}
```