Reusing Past Repairs Through Hierarchical Trajectory Abstraction for Coding Agents

Yisen Xu[1], Jiayuan Zhou[2], Ruiqi Pan[3], Tse-Hsun (Peter) Chen[1]
[1]SPEAR Lab, Concordia University, Montreal, Canada
[2]Waterloo Research Center, Huawei, Canada
[3]Huawei Technologies Co., Ltd.
yisen.xu@mail.concordia.ca, jiayuan.zhou1@huawei.com, panruiqi@huawei.com, peterc@encs.concordia.ca

***Abstract*—Although LLM-driven repair agents can tackle complex, repository-level issues, they treat every issue independently and discard the procedural knowledge accumulated from previous repairs. We introduce *STAIR*, a framework that converts historical repair trajectories into hierarchical, reusable plans that can be adapted to steer future repairs. Each past trajectory is transformed into a multi-level tree that ranges from fine-grained diagnostic actions to high-level repair strategies, encoding experience at several granularities. When a new issue arrives, *STAIR* selects relevant plan nodes from multiple abstraction levels, tailors them into executable, issue-specific plans, and supplies them to the agent through its prompt. On SWE-bench Verified, *STAIR* integrated with Lingxi reaches 81.2% Pass@1 using MiniMax M2.5 and 79.2% using GPT-5. The generated plans also generalize across agents: without any code change, they lift the Pass@1 of a structurally different agent, mini-SWE-agent v2, from 75.8% to 81.0%. Ablation experiments further show that mixing multiple abstraction levels surpasses any single level and that raw, unabstracted trajectories transfer substantially worse.**

***Index Terms*—automated program repair, LLM agents, hierarchical abstraction, experience reuse**

I. Introduction

Software bugs are inevitable in modern software development, and resolving them consumes a substantial portion of developer effort [1], [2]. In practice, experienced developers rarely approach each bug in isolation. Instead, they draw on accumulated debugging experience, reusing fault localization strategies, editing patterns, and verification procedures that have proven effective for similar problems in the past [3]. More importantly, this reuse is done at multiple levels simultaneously. A developer may recall both the specific API call that fixed a similar null pointer exception in the same repository, and the general strategy of tracing data flow backward from the failure site. Hence, effective problem-solving depends on integrating both fine-grained specifics and coarse-grained transferable patterns [4], [5].

Recent advances in large language models (LLMs) have led to a new generation of autonomous repair agents that can resolve real-world repository-level issues [6]–[9]. These agents interact with codebases through tool calls, execute tests, and iteratively refine patches. However, most existing repair agents treat each issue as an independent task. After each run, they discard the procedural knowledge embedded in past repair trajectories, such as effective localization strategies and common failure modes. While some studies [10]–[12] have begun to explore knowledge reuse for issue resolution, they face three key limitations.

First, existing approaches use an LLM to summarize historical repairs represented in two forms: issue–patch pairs [11] and agent trajectories [10], [12]. Summaries generated from issue–patch pairs cannot capture the intermediate reasoning that led to the fix, while summaries generated from trajectories condense the repair process without preserving how its reasoning, actions, and verification steps build on one another. Second, they represent each repair at a fixed level of detail, forcing a trade-off between concrete but non-transferable guidance and general but less actionable strategies [4], [13]. Third, they insert the retrieved historical repair summaries into the prompt as-is, leaving the agent to bridge the gap between a past fix and the current repository on its own. Thus, these summaries are generic guidance rather than an issue-specific repair plan.

To address these limitations, we propose *STAIR*, a framework that transforms successful historical repair trajectories into reusable procedural knowledge and adapts this knowledge to new issues. *STAIR* collects these trajectories by running an existing repair agent on previously resolved issues, then processes them offline before any target repair begins. Rather than reducing each repair to a short textual summary, *STAIR* segments each trajectory by repair stage and abstracts it into a multi-level hierarchy that captures how concrete agent actions connect to broader repair strategies. Lower levels of this hierarchy retain repository-specific details such as file paths and error messages, while higher levels capture transferable strategies that generalize across issues. Given a new issue, *STAIR* retrieves relevant procedural knowledge across these levels and uses an LLM to adapt it to the target issue and repository, producing stage-specific plans.

We evaluate *STAIR* on SWE-bench Verified [14], a benchmark of 500 real-world GitHub issues with developer-written test suites. *STAIR* achieves a Pass@1 of 81.2% with MiniMax M2.5 and 79.2% with GPT-5. We also compare *STAIR* with agents that reuse historical knowledge, including SWE-Exp [10], Lingxi [11], and ExpeRepair [12], and show that *STAIR* consistently outperforms them.

To assess generalizability, we transfer *STAIR*'s plans to

another agent scaffold, mini-SWE-agent v2 [6] without any modification. The plans improve its Pass@1 from 75.8% to 81.0% while slightly reducing token usage, demonstrating that the learned plans capture reusable repair strategies rather than agent-specific heuristics. Our ablation study further shows that collapsing the hierarchy to any single abstraction level degrades performance substantially, with the raw-trajectory level alone yielding the largest drop. The findings further demonstrate the benefits of our multi-level abstraction.

In summary, our paper makes the following contributions:

- We propose *STAIR*, an approach that abstracts historical repair trajectories into a multi-level knowledge hierarchy. This hierarchy aggregates recurring repair patterns across similar issues at multiple granularities, enabling both repository-specific guidance and cross-issue transfer.
- *STAIR* achieves 81.2% Pass@1 on SWE-bench Verified. Ablation studies confirm that both the multi-level hierarchy and plan adaptation contribute to the observed gains.
- The generated plans transfer to a structurally different repair agent without any modification, improving its Pass@1 by 5.2% while reducing token usage, confirming that the learned knowledge is agent-agnostic.

Paper Organization. Section II discusses related work. Section III details the design of *STAIR*. Section IV presents the evaluation results. Section V discusses potential threats to validity. Section VI concludes the work.

II. Related Work

In this section, we review prior work most relevant to *STAIR* in two areas: LLM-based autonomous repair agents and trajectory-based learning and experience reuse in LLM agents.

A. LLM-Based Autonomous Repair Agents

Agent Architectures for Issue Resolution. A growing body of work has explored how to equip LLMs with tool-use capabilities to resolve real-world repository-level issues. Early approaches demonstrate that even lightweight designs can be effective. Agentless [15] adopts a fixed and lightweight three-phase pipeline of fault localization, patch generation, and validation without any agent-driven planning. SWE-agent [6] proposes an agent-computer interface that grants the model direct access to bash commands for navigating repositories, editing files, and running tests. AutoCodeRover [7] augments agents with AST-based code search to enable structured codebase exploration.

As benchmarks become more challenging, recent agents adopt increasingly sophisticated strategies. OpenHands [16] provides an open platform where agents write and execute code to resolve issues. Prometheus [17] constructs a unified knowledge graph over the repository to support memory-enhanced navigation within a multi-agent system. TRAE [18] improves resolution rates through test-time scaling via modular solution generation, and live-SWE-agent [8] enables the agent to autonomously modify its own scaffold during runtime. Despite their strong results, these agents treat each issue as an independent task and discard all procedural knowledge after every run.

Early Attempts at Knowledge Reuse. A few recent agents have begun to incorporate historical experience into the repair process. Lingxi [11], which serves as the base agent in our work, takes historical issue reports and their patches, uses an LLM to infer the reasoning behind each fix, and summarizes it into knowledge descriptions that are retrieved to guide issue resolution. SWE-Exp [10] uses an LLM to summarize each past repair trajectory independently into a natural-language description and retrieves relevant ones to augment the agent's prompt. ExpeRepair [12] stores successful repair cases from past trajectories and retrieves similar ones as reference examples when fixing a new issue. It appends general repair insights summarized by an LLM to the agent's prompt.

However, these approaches all inject retrieved knowledge directly into the agent's prompt without adapting it to the target issue, producing generic hints rather than issue-specific repair plans. They also represent knowledge at a single level of abstraction, unable to simultaneously provide concrete localization guidance and transferable high-level strategies. Moreover, none evaluates whether the resulting knowledge transfers across repair agents with different scaffolds. We address these limitations by constructing a multi-level knowledge hierarchy that aggregates recurring repair patterns across similar issues at multiple granularities and adapts them into stage-specific plans covering fault localization, planning, and code editing. We further show that the resulting plans transfer to a different repair agent without modification.

B. Trajectory-Based Learning and Experience Reuse in LLM Agents

Experience Extraction from Agent Trajectories. Recent work has explored enabling LLM-based agents to improve from their own trajectories without parameter updates. Existing methods extract knowledge from trajectories in various forms. ExpeL [19] distills trial-and-error experiences into natural-language insights. Reflexion [20] generates verbal self-reflections from failed attempts to guide subsequent retries. AutoGuide [21] derives conditional guidelines from offline trajectories to serve as context-aware prompts. These methods demonstrate that agent trajectories provide valuable guidance for future execution.

However, the extracted knowledge is organized as a flat list, with all entries at the same level of abstraction and no association with repair stages. This leads to three limitations. First, the agent cannot distinguish between high-level strategies and low-level actions, so all guidance is treated as equally specific. Second, it cannot distinguish between guidance for fault localization and patch generation, making retrieval stage-agnostic and imprecise. Third, knowledge is derived from individual trajectories in isolation, without capturing patterns that generalize across issues. *STAIR* addresses these limitations by organizing trajectory-derived knowledge into a hierarchical structure indexed by repair stage and aggregated across issues.

This allows the agent to retrieve knowledge that matches the required abstraction level, repair stage, and recurring patterns observed across issues.

Workflow and Plan Induction from Trajectories. More recent work moves toward inducing structured workflows or plans from agent trajectories. Agent Workflow Memory (AWM) [22] identifies reusable workflows from successful web navigation trajectories and provides them as high-level guidance during future tasks. SAGE [23] applies a similar idea to software engineering. The agent abstracts a concise plan from its past trajectories and uses it to guide a refined second attempt on the same issue. SE-Agent [24] revisits past trajectories and improves them through revision, recombination, and refinement. SWE-PRM [25] trains a process reward model that monitors trajectories during execution and intervenes when agents exhibit inefficient behaviors.

These approaches show that structured trajectory reuse can improve both effectiveness and efficiency. However, prior studies focus on improving a single task. Namely, the agent learns from its own trajectory of the same issue and cannot be reused across issues. *STAIR* builds on this idea and studies how procedural knowledge learned from prior issues transfers to new issues.

Positioning of Our Work. Our work draws on the principles of case-based reasoning [5], particularly the idea that effective knowledge reuse requires hierarchical abstraction that spans multiple levels of granularity [4], [26]. Unlike prior trajectory-based learning methods that produce flat knowledge representations or single-level abstractions, *STAIR* organizes historical repair trajectories into a multi-level abstraction tree ranging from concrete diagnostic steps to general repair strategies. Unlike SAGE and AWM, which improve performance on the same task through self-reflection, *STAIR* learns reusable plans from previously resolved issues and transfers them to unseen issues. Furthermore, *STAIR* adapts retrieved guidance into stage-specific plans tailored for the target issue. Our findings show that these plans transfer across structurally different repair agents and help achieve state-of-the-art results.

III. Methodology

Figure 1 illustrates the overall workflow of *STAIR*. During offline knowledge construction, *STAIR* divides each successful historical repair trajectory into three stages: localization, planning, and execution-and-verification. It then abstracts the information within each stage at multiple levels, from concrete agent actions to broader procedural strategies. The resulting abstractions form a hierarchical repository of reusable procedural knowledge.

Given a new issue, *STAIR* retrieves relevant procedural knowledge for each repair stage, potentially drawing different stages from different historical issues. The retrieved knowledge provides stage-specific *guidance* that combines broadly transferable strategies with more actionable procedures. An LLM then adapts this guidance to the target issue and repository to generate issue-specific *plans* for localization, planning, and execution-and-verification. These plans translate the retrieved procedural knowledge into concrete actions that the repair agent can follow.

A. Procedural Knowledge Construction

STAIR collects historical issue-fixing trajectories by running Lingxi's repair scaffold without its knowledge-reuse mechanism on previously resolved issues. For each historical issue u, the system produces a trajectory $\tau(u)$ that captures the agent's reasoning, tool interactions, edit decisions, and verification feedback across the entire repair process. To enable structured reuse, *STAIR* organizes this trajectory in a stage-aware manner. Specifically, each trajectory $\tau(u)$ is partitioned into stage-specific segments $\tau_s(u)$ corresponding to localization, planning, and execution-and-verification [27], [28]. Each segment retains the same types of information (i.e. observation, thought, and action), but is restricted to the corresponding stage. This stage-wise decomposition bridges the rich, heterogeneous information in the full trajectory with the stage-specific abstraction and retrieval used later.

In our implementation, to facilitate a controlled comparison with Lingxi [11], we follow its protocol for selecting historical issues. For each target issue, candidate historical issues are drawn from the same repository. The historical issues are restricted to those created before the target issue to avoid data leakage. They are then ranked by embedding similarity and filtered by an LLM-based relevance verifier. The repair trajectories of the selected historical issues are generated by running Lingxi without its knowledge-reuse mechanism.

B. Hierarchical Trajectory Abstraction

Raw repair trajectories are often lengthy and noisy, containing fine-grained tool outputs, failed attempts, and redundant actions that are difficult to reuse directly. To improve reusability, *STAIR* transforms each trajectory into a multi-level abstraction tree. This design reflects how developers work in practice: they move between high-level strategies and low-level actions, reusing past solutions at the level of detail that fits the current problem [4], [5], [26]. Accordingly, lower levels retain concrete, step-level actions, while higher levels capture more general repair strategies. We refer to the elements at each level of this tree as *procedural knowledge nodes* (or simply *nodes*), each capturing procedural knowledge at a given level of abstraction.

Within each stage of the repair workflow, *STAIR* constructs a hierarchy of these procedural knowledge nodes. The leaf nodes correspond to individual trajectory steps, each containing the agent's thought, action, and observation. Each higher-level node summarizes a group of related nodes from the level below, representing broader procedures and repair strategies.

Formally, let $\mathcal{L}_0$ denote the leaf-node set constructed from the original trajectory steps, where each step contains the agent's thought, action, and observation. At abstraction level $\ell \geq 0$, an LLM-based grouping operator $G_\ell(\cdot)$ partitions $\mathcal{L}_\ell$ into m_ℓ non-overlapping groups, where each group $C_{\ell,j} \subseteq \mathcal{L}_\ell$:

$$G_\ell(\mathcal{L}_\ell) = \{C_{\ell,1}, C_{\ell,2}, \ldots, C_{\ell,m_\ell}\}. \quad (1)$$

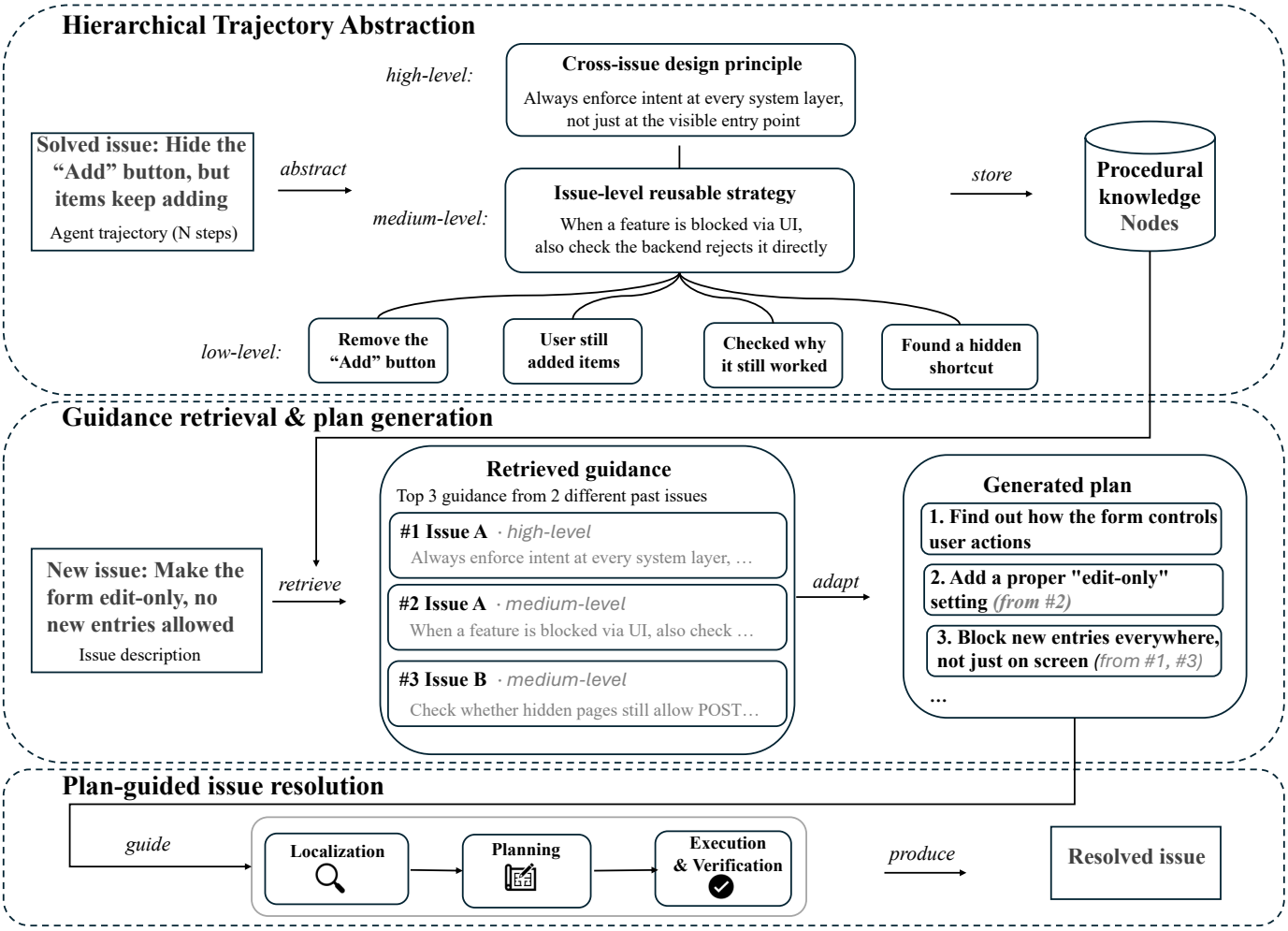

Fig. 1. An overview of *STAIR*.

An LLM-based abstraction operator $A_\ell(\cdot)$ then abstracts each group into a parent node, producing the next level:

$$\mathcal{L}_{\ell+1} = \{A_\ell(C_{\ell,j}) \mid 1 \le j \le m_\ell\}. \tag{2}$$

For instance, as shown in Figure 1, the agent's trajectory for a solved issue includes several low-level steps: removing the "Add" button, discovering that users could still add items, investigating why, and finding a hidden shortcut. $G_\ell(\cdot)$ groups these steps because they share a common sub-goal of understanding why blocking the UI entry point did not prevent the undesired behavior. $A_\ell(\cdot)$ then abstracts the group into a medium-level strategy: *when a feature is blocked via UI, also check that the backend rejects it directly*. At higher levels, this is further generalized into a cross-issue design principle: *always enforce intent at every system layer, not just at the visible entry point*.

This process repeats until $|\mathcal{L}_{\ell+1}| \le K$, where K is the number of root nodes (set to 2 in our experiments). The resulting layers collectively form the hierarchical abstraction of the original trajectory. Since a repair trajectory naturally consists of distinct phases (localization, planning, and execution with verification), *STAIR* first segments the full trajectory $\tau(u)$ into stage-specific sub-trajectories $\tau_s(u)$, and applies the above hierarchical abstraction procedure independently to each one. This preserves stage-level separation, enabling more targeted retrieval during downstream guidance generation.

In our implementation, both the grouping operator $G_\ell(\cdot)$ and the abstraction operator $A_\ell(\cdot)$ are implemented via LLM-based generation using GPT-5 [29]. Starting from the raw trajectory steps $\mathcal{L}_0$, the two operators are applied in alternation: $G_\ell(\cdot)$ partitions the current-level nodes into consecutive groups that share a coherent procedural intent (Figure 2), and $A_\ell(\cdot)$ abstracts each group into a structured node containing the repair intent, key actions, applicable conditions, and common pitfalls (Figure 3). The output nodes $\mathcal{L}_{\ell+1}$ then become the input to the next iteration. Before each iteration, an LLM-based decision prompt evaluates the current nodes and selects the appropriate abstraction level ℓ from three views: low-level for concrete sub-goals grounded in project-specific artifacts, medium-level for project-agnostic investigation strategies, and

Task: Partition L_ℓ into consecutive groups sharing a coherent procedural intent at the granularity of *[Abstraction level ℓ]*.

Inputs: *[Issue report u] [Current-level nodes L_ℓ] [Abstraction level $\ell \in$ {low, medium, high}]*

Grouping guidance:

- When ℓ = *low*: group by concrete sub-goals
- When ℓ = *medium*: group by investigation strategies
- When ℓ = *high*: group by general problem-solving principles

Rules

- Groups must be consecutive ranges (no reordering, no gaps)
- Every node belongs to exactly one group

Output: Reasoning and a list of groups, each with *[start, end, purpose]*.

Fig. 2. Simplified prompt template for the grouping operator $G_\ell(\cdot)$. The same template is applied iteratively at each abstraction level.

Task: Summarize a group $C_{\ell,j}$ into a single parent node at the granularity of *[Abstraction level ℓ]*.

Inputs: *[Issue report u] [Group steps $C_{\ell,j} \subset L_\ell$] [Abstraction level $\ell \in$ {low, medium, high}]*

Abstraction guidance:

- When ℓ = *low*: produce a concrete sub-goal grounded in project artifacts
- When ℓ = *medium*: extract a project-agnostic investigation strategy
- When ℓ = *high*: distill a general problem-solving principle

Rules

- Interpret nodes holistically using reasoning traces, tool usage, and existing abstractions
- Concreteness decreases with ℓ: low-level may name specific APIs; high-level must be fully generic

Output: A structured node with *[level, node_type, summary, intent, rationale, when_to_apply, key_actions, pitfalls]*.

Fig. 3. Simplified prompt template for the abstraction operator $A_\ell(\cdot)$. The same template is applied iteratively at each abstraction level.

high-level for general problem-solving principles that transfer across projects. While all levels share the same node schema, the increasing abstraction naturally filters out implementation details and surfaces reusable procedural knowledge, as illustrated in Figure 1.

C. Guidance Retrieval and Plan Generation

As described in Section III-A, each repair trajectory is segmented into three stages: localization, planning, and execution with verification. To resolve a new issue, *STAIR* retrieves relevant procedural knowledge from similar historical issues for each stage and organizes them into guidance, which is then adapted into issue-specific plans. We define these two concepts as follows.

Guidance. For a target issue u and repair stage s, the *guidance* $\mathcal{G}_s(u)$ is an ordered sequence of *abstract* procedural knowledge nodes retrieved from similar historical issues, arranged according to their original execution order in the source trajectories. Guidance captures historically effective repair procedures relevant to the target issue at stage s.

Plan. For a target issue u and repair stage s, the *plan* $P_s(u)$ is a structured, issue-specific action sequence generated by adapting the guidance $\mathcal{G}_s(u)$ to the concrete symptoms and repository context of u.

Retrieval. Given a target issue u, *STAIR* retrieves candidate procedural knowledge nodes from similar historical issues separately for each repair stage. Each node n is associated with a specific repair stage and can originate at any level of the abstraction hierarchy, as described in Section III-B, for similar historical issues. This allows the retrieved guidance to combine low-level actionable details with high-level repair strategies.

Retrieval begins with a similarity-based scoring step that computes an initial relevance score $r(u, n)$ based on textual similarity matching between the issue report and node content. To refine this ranking, an LLM-based verifier evaluates the top candidates to assess their procedural relevance to the target issue, filtering out nodes that are lexically similar but not applicable in the current repair context. The retained nodes for stage s are then ordered according to their execution sequence in the source trajectories, forming the guidance $\mathcal{G}_s(u)$, which is a coherent sequence that preserves the procedural structure of historical repair processes.

Plan Generation. While the retrieved guidance captures useful repair procedures, it remains abstract and does not yet specify concrete actions for the current issue. *STAIR* therefore adapts this guidance into executable plans. For each stage s, *STAIR* provides the LLM with the issue report u and the corresponding guidance $\mathcal{G}_s(u)$ and asks it to generate a structured plan $P_s(u)$. The issue report conveys the concrete symptoms and constraints of the current issue, while the guidance supplies historically effective repair strategies. During plan generation, the model selectively instantiates applicable procedures and discards irrelevant ones, producing a plan that preserves useful procedures while grounding actions in the current repository.

D. Plan-Guided Issue Resolution

The stage-specific plans generated in the previous step are executed to resolve the target issue. For each repair stage s, the plan $P_s(u)$ provides structured procedural guidance that translates the retrieved repair knowledge into concrete actions while remaining adaptable to the current repository context.

During localization, the plan $P_{loc}(u)$ directs the agent toward the most relevant failure evidence, focuses analysis on suspicious regions of the codebase, and organizes diagnosis around a smaller set of plausible fault hypotheses, narrowing the search space from the full repository to a targeted set of candidate locations. During planning, the plan $P_{plan}(u)$ specifies the intended modification strategy, expected repair scope, and constraints that should guide subsequent edits.

Based on this plan, the agent proposes candidate modifications and constructs a candidate patch. During execution-and-verification, the patch is validated through test-based checking, and the agent revises the patch when verification feedback indicates that the current repair attempt is insufficient.

Although the plans guide the repair process, they do not fully determine it. The repair run still depends on evidence uncovered during localization and feedback obtained during verification. Newly observed evidence may refine the repair strategy, while failed verification may require revisiting earlier decisions. In this way, historical procedural knowledge informs the repair process without forcing it to follow a rigid script.

E. Using Existing Repair Agent as the Scaffold

STAIR is not tied to a particular repair agent and can augment agents that accept external repair guidance through their prompts. We use Lingxi [11] as the underlying agent scaffold because it also reuses procedural knowledge from historical issues. This allows a controlled comparison between Lingxi's original knowledge-reuse mechanism and *STAIR*'s hierarchical procedural knowledge, while keeping the underlying repair workflow fixed. Lingxi also achieves competitive performance on SWE-bench Verified [14], making it a strong underlying agent for our evaluation.

To integrate *STAIR*, we retain Lingxi's internal decision logic, tools, and stage structure, disable its original knowledge-reuse mechanism, and insert the corresponding *STAIR*-generated plans into the prompts for each stage at runtime. To evaluate whether the generated plans transfer beyond Lingxi, we also integrate them with *mini-SWE-agent v2* [6] in RQ2.

IV. Evaluation

A. Benchmark and Metrics

We conduct all experiments on **SWE-bench Verified** [14], a human-curated subset of 500 instances from the original SWE-bench. Each instance corresponds to a real-world GitHub issue paired with a developer-written test suite that distinguishes correct patches. We use **Pass@1** as our evaluation metric, defined as the percentage of instances for which the LLM agent produces a correct patch on the first attempt.

B. Models and Baselines

We evaluate *STAIR* using two state-of-the-art backbone models: **GPT-5** [29] (High Reasoning, commercial) and **MiniMax M2.5** [30] (High Reasoning, open-weight). For cross-scaffold transferability (RQ2), we apply the procedural plan generated by *STAIR* to *mini-SWE-agent v2* [6]. We use the same settings and prompts across models and agent scaffolds to ensure a fair comparison.

C. Implementation Details

We implement *STAIR* using LangGraph [31], which orchestrates the entire hierarchical repair pipeline, including agent workflow coordination, trajectory storage, and stage-wise retrieval described in Section III. The historical trajectories are collected from previously resolved issues by running Lingxi without its knowledge-reuse mechanism [11], and abstracted using the pipeline described in Section III-A. To prevent temporal leakage, the historical issue candidate pool for each target instance is restricted to issues created before the target issue's creation time.

D. Environment

As described in Section III-E, *STAIR* uses Lingxi [11] as the underlying repair agent. We follow Lingxi's agent configuration, including its tool definitions, prompting templates, and stage structure. All experiments are executed via API calls to GPT-5 and MiniMax M2.5. Resolving a single SWE-bench Verified instance takes 10–15 minutes on average, end-to-end. The per-instance cost of issue resolution, including retrieval, target-specific plan adaptation, analysis, patching, and verification, is $0.84 with MiniMax M2.5 and $1.60 with GPT-5. Hierarchical trajectory abstraction is constructed once using GPT-5 as an offline preprocessing step, and the resulting abstraction nodes are reused across all backbone LLMs. This preprocessing costs approximately $0.44 per source trajectory. In our experiments, *STAIR* retrieves and abstracts 479 similar historical trajectories in total, resulting in a total preprocessing cost of approximately $211. This cost is incurred once and amortized over all subsequent evaluations.

RQ1: How does STAIR perform compared to other state-of-the-art autonomous repair agents?

Motivation and Approach. The goal of this RQ is to evaluate whether *STAIR*'s hierarchical trajectory abstraction and plan adaptation improve repository-level issue-resolution rates compared to state-of-the-art repair agents. We compare *STAIR* with nine existing agents on SWE-bench Verified, organized into three groups.

The first group uses the same backbone models as *STAIR* (GPT-5 and MiniMax M2.5):

- *OpenHands* [16]: an open-source platform for AI agents that interact with the world by writing code, executing commands, and browsing the web.
- *Prometheus-v1.2.1* [17]: a multi-agent platform that leverages unified knowledge graphs to automatically bug fixing.
- *mini-SWE-agent v2* [6]: a minimal bash-only agent scaffold developed by the SWE-bench team.

The second group includes other top-performing repair agents on the SWE-bench leaderboard that use different backbone models:

- *TRAE* [18]: an ensemble reasoning agent that applies test-time scaling through modular solution generation and selection.
- *Sonar Foundation Agent* [9]: a single-agent tool-calling framework that combines bash execution with AST-based code search.
- *live-SWE-agent* [8]: a self-evolving agent that autonomously improves its own scaffold on-the-fly during runtime.

TABLE I
COMPARISON WITH STATE-OF-THE-ART REPAIR AGENTS ON SWE-BENCH VERIFIED (PASS@1, %). HR DENOTES HIGH REASONING.

Scaffold	Backbone Model	Pass@1 (%)
Existing Agents (same backbone)		
OpenHands [16]	GPT-5	71.8
Prometheus-v1.2.1 [17]	GPT-5	74.4
mini-SWE-agent v2 [6]	MiniMax M2.5 (HR)	75.8
Existing Agents (other top repair agents)		
TRAE [18]	Doubao-Seed-Code	78.8
Sonar Foundation Agent [9]	Claude 4.5 Opus	79.2
live-SWE-agent [8]	Claude 4.5 Opus (Medium)	79.2
Experience Reuse Agents		
SWE-Exp [10]	Claude 4 Sonnet	73.0
ExpeRepair [12]	Claude 4 Sonnet + o4-mini	74.6
Lingxi [11]	Claude 4 Sonnet	74.6
Lingxi	MiniMax M2.5 (HR)	74.6
Lingxi	GPT-5 (HR)	75.6
Our Agents		
STAIR$_{GPT5}$	**GPT-5 (HR)**	**79.2**
STAIR$_{MiniMax}$	**MiniMax M2.5 (HR)**	**81.2**

The third group includes experience reuse agents that retrieve knowledge from historical issues to guide repair:

- *SWE-Exp* [10]: uses an LLM to summarize past repair trajectories into natural-language descriptions and retrieves relevant ones to augment the agent's prompt.
- *Lingxi* [11]: takes historical issue reports and their patches, uses an LLM to infer the reasoning behind each fix, and retrieves relevant knowledge descriptions to guide issue resolution.
- *ExpeRepair* [12]: retrieves past repair examples from its own trajectories as few-shot demonstrations and appends LLM-summarized repair insights to the agent's prompt.

Results. Table I presents the performance of *STAIR* alongside state-of-the-art agents on SWE-bench Verified. STAIR$_{MiniMax}$ achieves a Pass@1 of **81.2%**, outperforming all other repair agents. In comparison, STAIR$_{GPT5}$ achieves a Pass@1 of 79.2%, ranking second only behind STAIR$_{MiniMax}$. This finding shows that *STAIR* consistently delivers strong performance across different backbone models.

When compared with repair agents that use the same backbone models, *STAIR* achieves higher resolved rate. On GPT-5, *STAIR* (79.2%) outperforms *OpenHands* (71.8%) by 7.4% and *Prometheus-v1.2.1* (74.4%) by 4.8%. On MiniMax M2.5, *STAIR* (81.2%) exceeds *mini-SWE-agent v2* (75.8%) by 5.4%.

Among other top repair agents, *live-SWE-agent* and *Sonar Foundation Agent* achieve the highest Pass@1 of 79.2%, both built on Claude 4.5 Opus, one of the strongest backbones used across all prior repair agents [6], [8]. *TRAE* (Doubao-Seed-Code) reaches 78.8%. *STAIR*, by contrast, surpasses all baselines while relying on comparatively weaker backbones: MiniMax and GPT-5. We chose these backbones to assess *STAIR*'s contribution independently of model capacity. Nonetheless, *STAIR* outperforms the strongest Claude-based baselines, indicating that the observed gains are from the proposed framework rather than from the backbone choice.

Compared with experience-reuse agents that also leverage historical issues, *STAIR* outperforms all three across every backbone configuration. On Claude 4 Sonnet, SWE-Exp achieves 73.0%, and both ExpeRepair and Lingxi achieve 74.6%. To enable a controlled comparison on the same backbone LLM, we also run Lingxi (with its own procedural knowledge) with MiniMax M2.5 and GPT-5, obtaining 74.6% and 75.6%, respectively. *STAIR* outperforms these same-backbone Lingxi baselines by 6.6% (81.2% vs. 74.6% on MiniMax M2.5) and 3.6% (79.2% vs. 75.6% on GPT-5), confirming that *STAIR*'s procedural knowledge provides more effective guidance than Lingxi's.

> *STAIR* achieves a Pass@1 of 81.2% with MiniMax M2.5 and 79.2% with GPT-5 on SWE-bench Verified, achieving top performance among compared agents under reported settings.

TABLE II
TRANSFERABILITY OF *STAIR*'S PROCEDURAL PLANS TO *mini-SWE-agent v2* ON SWE-BENCH VERIFIED (500 INSTANCES). *Avg. Steps* AND *Avg. token* DENOTES THE AVERAGE NUMBER OF STEPS THE AGENT TOOK AND THE TOTAL TOKEN USAGE PER INSTANCE.

Configuration	Resolved	Avg. steps	Avg. token
mini-SWE-agent v2	379/500 (75.8%)	60.45	1,244,923
+ *STAIR* Plan	405/500 (81.0%)	58.18	1,179,885

RQ2: Can STAIR's plans transfer to other repair agents?

Motivation and Approach. *STAIR* learns and constructs procedural knowledge within its own agent scaffold, raising a key question: does it capture generalizable repair strategies, or does it encode scaffold-specific heuristics? To answer this, we transfer the stage-specific plans distilled from *STAIR*'s historical trajectories into *mini-SWE-agent v2* [6], a repair agent with a different architecture, tool interface, and prompting strategy. *mini-SWE-agent v2* is a widely adopted and lightweight repair agent developed by the SWE-bench team. It achieves over 74% on SWE-bench Verified with a minimal architecture that relies solely on bash, making it an ideal candidate for evaluating whether procedural plans generalize across structurally different repair agents.

We prepend an `<external_reference_plans>` block to the agent's input at each repair stage, while leaving *mini-SWE-agent v2*'s source code, tool definitions, and output verification logic unchanged. We use MiniMax M2.5 as the backbone model, as it achieves the best performance in RQ1 (Table I).

Results. Table II reports the transfer results. Injecting *STAIR*'s procedural plans improves the Pass@1 of *mini-SWE-agent v2* from 75.8% (379/500) to 81.0% (405/500), an absolute gain of 5.2%. At the same time, the average token usage *decreases* from 1,245k to 1,180k per instance. This reduction is driven by fewer agent steps (average steps decreased from 60.45 to

TABLE III
GAIN/REGRESSION DISTRIBUTION ON SWE-BENCH VERIFIED AFTER ADDING *STAIR*'S GENERATED PLANS. *Shared* = SOLVED BY BOTH; *Gained* = SOLVED ONLY WITH *STAIR*'S PLAN; *Regressed* = SOLVED ONLY BY BASELINE. BASELINES ARE *Lingxi* FOR *STAIR* AND VANILLA *mini-SWE-agent v2* FOR *mini-SWE-agent v2* + PLANS.

Comparison	Shared	Gained	Regressed	Net
STAIR$_{MiniMax}$ vs. LINGXI$_{MiniMax}$	363	43	10	+33
mini-SWE-agent v2 + Plans vs. vanilla	372	33	7	+26

58.18), as the injected plans guide the agent toward more targeted exploration and reduce unnecessary search.

These results demonstrate that procedural plans learned from *STAIR* transfer effectively across repair agents without any modification (the plans are only prepended to the prompt). Despite being derived from a different agent, the plans improve both effectiveness (higher Pass@1) and efficiency (lower token usage), suggesting that they capture reusable repair strategies rather than agent-specific heuristics.

> *STAIR*'s procedural plans are agent-agnostic, improving *mini-SWE-agent v2*'s Pass@1 by 5.2% (75.8% to 81.0%) while slightly reducing average token usage per instance due to fewer agent steps.

RQ3: How does plan guidance affect agent behavior?

Motivation and Approach. Procedural plans can improve repair outcomes, but their role in agent behavior remains unclear. In particular, we seek to understand how plans contribute to successful repairs and, in regressed cases (i.e., previously passed issue becomes failed), whether the failures are attributable to the plan itself or to downstream execution errors. To study these questions, we manually analyze gained and regressed instances for two comparisons: *STAIR* vs. *Lingxi* (which uses its own knowledge-reuse mechanism), and *mini-SWE-agent v2* with *STAIR*'s plans vs. vanilla *mini-SWE-agent v2*. For each instance, two authors independently examined the generated plan, the produced patch, and the full agent trajectory, then discussed disagreements to reach a consensus label. The resulting inter-rater agreement achieved a Cohen's kappa of 0.85, indicating a strong agreement [32]. Finally, we compare each plan-augmented run with its corresponding baseline to assess how the plan affected the agent's behavior.

Results. Table III summarizes the gain/regression breakdown. In both repair agents, gains after adding *STAIR*'s plans substantially outnumber regressions (43 vs. 10 and 33 vs. 7), confirming that the plans provide a strong net benefit. We next analyze the two groups separately to understand the mechanisms behind these outcomes.

Why Gained. We group the instances into three categories according to the plan's main contribution. Each category reflects a different problem that the plan corrects in the baseline behavior.

- **Original Fix Targets the Wrong Fault Location (Wrong Location)**: the baseline targets the wrong component. Without *STAIR*'s plan, the agent cannot recover from this early mistake and all subsequent steps are wasted. The plan guides the agent to the correct location from the start (e.g., fixing the upstream serializer rather than the downstream loader).
- **Original Fix Covers Only Part of the Affected Code (Partial Fix)**: the baseline identifies the correct root cause but modifies only a subset of the relevant code locations. Adding the plan ensures all affected components are covered. For example, in `django-11138`, a timezone conversion bug affected MySQL, Oracle, and SQLite backends, but the baseline only fixed the first two. The plan framed it as a cross-backend audit, pushing the agent to cover all three.
- **Original Fix Modifies More Code Than Necessary (Over-modification)**: the baseline applies a fix that changes more code than required, potentially introducing regressions. Adding the plan restricts the repair to the minimal necessary change (e.g., extending a matching condition instead of rewriting the entire discovery pipeline).

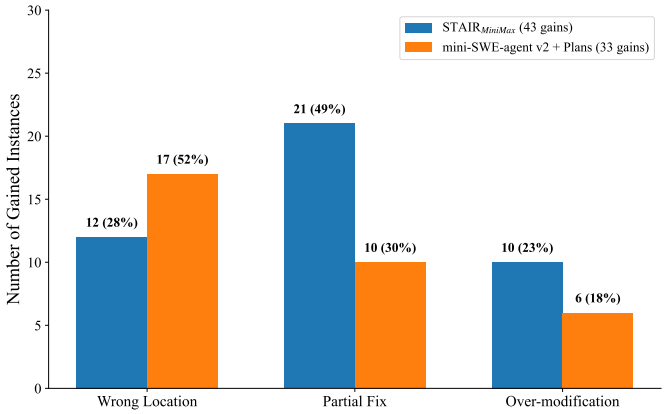

Fig. 4. Distribution of gain categories for STAIR$_{MiniMax}$ and *mini-SWE-agent v2* + Plans.

Figure 4 shows the distribution of gain categories. For STAIR$_{MiniMax}$ (43 instances), *Partial Fix* dominates (49%), followed by *Wrong Location* (28%) and *Over-modification* (23%). For *mini-SWE-agent v2* + Plans (33 instances), *Wrong Location* dominates (52%), followed by *Partial Fix* (30%) and *Over-modification* (18%).

This difference is also associated with the different workflows of the two agents. For *STAIR*, the largest share of gains comes from *Partial Fix*, suggesting that the agents in *STAIR* often reach the correct repair direction but benefit from plans that help it cover all affected locations. For *mini-SWE-agent v2*, the largest share of gains comes from *Wrong Location*, suggesting that plans are most useful in helping the agent focus on the correct problem early.

Why Regressed. We manually analyze all 17 regressed instances (10 from STAIR$_{MiniMax}$ and 7 from *mini-SWE-agent v2* + Plans). We find that regressions fall into two categories: *plan-related* failures, where the injected plan provides incom-

plete coverage of the required changes, and *non-plan-related* failures, where the plan is reasonable but the agent fails during execution.

In the first case (8/17), the procedural plan captures the main repair direction but does not fully specify the complete scope of changes needed for a correct fix. This typically happens when the issue requires coordinated updates across multiple files, entry points, while the plan identifies only the primary fix location. Across both agents, 8 out of 17 regressions fall into this category (5 from STAIR$_{MiniMax}$ and 3 from *mini-SWE-agent v2* + Plans).

In the second case (9/17), the procedural plan matches the issue, but the agent fails to follow it correctly during execution. These failures include deviating from the intended plan, introducing unnecessarily complex edits, or making local implementation mistakes such as using the wrong API or passing incorrect arguments. The remaining 9 regressions fall into this category (5 from STAIR$_{MiniMax}$ and 4 from *mini-SWE-agent v2* + Plans).

These findings suggest that the procedural plans are largely effective, with only a few that miss part of the required fix scope.

> Plans improve repair mainly by correcting fault localization, expanding repair coverage, and reducing unnecessary edits. Regressions, in contrast, arise either from incomplete plan coverage or from execution mistakes that are not caused by the plan itself.

RQ4: What is the individual contribution of each component in STAIR?

Motivation and Approach. *STAIR*'s procedural knowledge is constructed through hierarchical trajectory abstraction, where raw repair trajectories are grouped into procedural segments and abstracted into nodes at multiple levels. This hierarchy captures both concrete step-level actions and high-level repair strategies.

To evaluate the contribution of this abstraction, we conduct an ablation study along two dimensions: (1) raw trajectories vs. abstracted trajectories, and (2) different abstraction levels (low-level, medium-level, high-level, and multi-level). We implement the following variants:

- **No Abstraction (Raw Trajectories):** directly uses unprocessed trajectory logs without hierarchical abstraction.
- **Low-Level Abstraction Only:** uses only fine-grained nodes that preserve concrete, repository-specific details (e.g., file paths, function signatures, error messages).
- **Medium-Level Abstraction Only:** uses only coarse-grained nodes that capture general repair strategies independent of specific repositories.
- **High-Level Abstraction Only:** uses only the most abstract nodes that capture general problem-solving principles transferable across projects.
- **Full Hierarchical Abstraction (Full System):** combines nodes from multiple abstraction levels, reflecting the full *STAIR*.

TABLE IV
ABLATION STUDY ON A DIFFICULTY-STRATIFIED SUBSET OF 125 SWE-BENCH VERIFIED INSTANCES. Δ DENOTES THE ABSOLUTE DIFFERENCE FROM THE FULL HIERARCHICAL ABSTRACTION.

Variant	Resolved (%)	Δ
Full Hierarchical Abstraction	**100/125 (80.0)**	–
Baseline (No Abstraction)		
Raw Trajectories	72/125 (57.6)	−22.4
Ablations (Abstraction Level)		
Low-Level only	80/125 (64.0)	−16.0
Medium-Level only	76/125 (60.8)	−19.2
High-Level only	73/125 (58.4)	−21.6

Due to the cost of running full-benchmark evaluations for each ablation variant, we conduct the study on a difficulty-stratified subset of 125 instances rather than all 500. We allocate approximately three-quarters of the subset (92 instances, 73.6%) to the medium-and-hard tiers (≥15-minute estimated fix time as annotated by SWE-bench Verified) and one-quarter (33 instances, 26.4%) to the easy tier, following a roughly 3:1 ratio between non-trivial and trivial instances. This sampling strategy concentrates analytical power on instances where design choices are most likely to produce observable behavioral differences. All experiments use MiniMax M2.5 (High Reasoning) as the backbone.

Results. Table IV summarizes the ablation results along two dimensions. Replacing hierarchical abstraction with raw trajectories reduces the resolution rate from 80.0% to 57.6% (-22.4%). The findings indicate that raw trajectories (which may contain redundant actions, failed exploration paths, and issue-specific artifacts) do not transfer across issues. *STAIR*'s abstraction filters out such noise and restructures trajectories into reusable procedural knowledge, substantially improving generalization.

Different abstraction levels capture complementary aspects of procedural knowledge. Using a single abstraction level consistently underperforms the full hierarchical design. Low-level-only achieves 64.0% (Δ=−16.0%), medium-level-only 60.8% (Δ=−19.2%), and high-level-only 58.4% (Δ=−21.6%).

This trend reveals a clear progression: low-level abstractions are the most effective among single-level variants because they retain concrete, executable details needed for repository-specific edits. High-level abstractions perform the worst, as they capture only coarse strategies and lack grounding for direct action. Medium-level abstractions fall in between. Our findings show that effective repair requires procedural knowledge at multiple levels of abstraction, and collapsing to any single level leads to incomplete guidance and degraded performance.

Using raw trajectories without abstraction yields the largest performance drop (−22.4% in Pass@1), showing that issue-specific actions in the raw trajectories do not generalize well to new issues. The full hierarchical abstraction outperforms all single-level variants, demonstrating that integrating multiple levels of procedural knowledge is necessary for effective repair.

V. Threats to Validity

A. Internal Validity

To reduce the potential biases in manual analysis, two authors independently labeled all gained and regressed instances and resolved disagreements by consensus, achieving a Cohen's kappa of 0.85, indicating strong agreement. The hierarchical abstraction process relies on LLM-based generation, including both the grouping and abstraction operators. As a result, the quality of the resulting multi-level nodes may vary with the complexity and length of the input trajectory. The plan compatibility filter also uses an LLM-based evaluator and may occasionally discard useful plans or retain inapplicable ones. The transfer results in RQ2 show that the extracted plans improve a different repair agent, indicating that the approach does not overfit to a specific agent or workflow.

B. External Validity

Our evaluation is conducted on SWE-bench Verified (500 Python instances from 12 repositories), so empirical results may not directly generalize to other languages, benchmarks, or proprietary codebases. However, the framework is not inherently tied to SWE-bench Verified: it applies to any issue-resolution setting that provides executable tests for patch validation. The procedural knowledge is constructed from trajectories collected by Lingxi, and a different base agent may produce trajectories of different quality, potentially affecting the resulting plans. That said, the successful transfer of plans to mini-SWE-agent v2 in RQ2, without any modification, suggests that the plans capture agent-agnostic repair strategies rather than scaffold-specific artifacts.

C. Construct Validity

Our study evaluates the effectiveness of *STAIR* using Pass@1 and token consumption as primary metrics. While Pass@1 reflects whether a generated patch passes the test suite, it does not guarantee full semantic correctness beyond the provided tests. This limitation may allow behaviorally incorrect patches to be accepted if the tests do not fully capture the intended behavior [33]–[35]. The underlying LLMs may affect the results. However, we conducted our experiments on one commercial and one open weight model (GPT-5 and MiniMax M2.5) and we found that the results are consistent.

VI. Conclusion

In this paper, we introduced *STAIR*, a framework that abstracts historical repair trajectories into multi-level hierarchical representations and adapts them into reusable plans to guide future repairs. *STAIR* abstracts raw repair trajectories into multi-level representations ranging from concrete diagnostic steps to general repair strategies, retrieves relevant guidance for each stage of the repair workflow, and adapts it into issue-specific executable plans.

We evaluated *STAIR* on SWE-bench Verified, where it achieves a Pass@1 of 81.2% with MiniMax M2.5 and 79.2% with GPT-5. An ablation study shows that combining multiple abstraction levels outperforms any single level, and that raw trajectories without abstraction generalize poorly, with the largest performance drop of 22.4%. The generated plans also transfer to mini-SWE-agent v2 without any modification, improving its Pass@1 from 75.8% to 81.0% while reducing token usage, demonstrating that the learned knowledge is agent-agnostic. More broadly, our results show that the effectiveness of experience reuse depends less on the source of historical knowledge than on how it is structured and applied. Raw trajectories and flat summaries both contain useful information, yet they transfer poorly when injected as-is. Organizing knowledge into multiple abstraction levels and adapting it into stage-specific plans bridges this gap, turning historical experience into actionable guidance that generalizes across issues, models, and agent scaffolds.

References

[1] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of applications software maintenance," *Commun. ACM*, vol. 21, no. 6, pp. 466–471, 1978. [Online]. Available: https://doi.org/10.1145/359511.359522

[2] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: a survey," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, p. 1219. [Online]. Available: https://doi.org/10.1145/3180155.3182526

[3] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 913–923. [Online]. Available: https://doi.org/10.1109/ICSE.2015.101

[4] K. Branting, "Stratified case-based reasoning in non-refinable abstraction hierarchies," in *Case-Based Reasoning Research and Development, Second International Conference, ICCBR-97, Providence, Rhode Island, USA, July 25-27, 1997, Proceedings*, ser. Lecture Notes in Computer Science, D. B. Leake and E. Plaza, Eds., vol. 1266. Springer, 1997, pp. 519–530. [Online]. Available: https://doi.org/10.1007/3-540-63233-6_521

[5] A. Aamodt and E. Plaza, "Case-based reasoning: Foundational issues, methodological variations, and system approaches," *AI Commun.*, vol. 7, no. 1, pp. 39–59, 1994. [Online]. Available: https://doi.org/10.3233/AIC-1994-7104

[6] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html

[7] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 1592–1604. [Online]. Available: https://doi.org/10.1145/3650212.3680384

[8] C. S. Xia, Z. Wang, Y. Yang, Y. Wei, and L. Zhang, "Live-swe-agent: Can software engineering agents self-evolve on the fly?" *arXiv preprint*, 2025.

[9] H. Ruan, "Introducing sonar foundation agent," 2025, accessed: Nov. 14, 2025. [Online]. Available: https://www.sonarsource.com/blog/introducing-sonar-foundation-agent/

[10] S. Chen, S. Lin, X. Gu, Y. Shi, H. Lian, L. Yun, D. Chen, W. Sun, L. Cao, and Q. Wang, "Swe-exp: Experience-driven software issue resolution," *CoRR*, vol. abs/2507.23361, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2507.23361

[11] X. Yang, J. Zhou, M. Pacheco, W. Zhu, P. He, S. Wang, K. Liu, and R. Pan, "Lingxi: Repository-level issue resolution framework enhanced by procedural knowledge guided scaling," *CoRR*, vol. abs/2510.11838, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2510.11838

[12] F. Mu, J. Wang, L. Shi, S. Wang, S. Li, and Q. Wang, "EXPEREPAIR: dual-memory enhanced llm-based repository-level program repair," *CoRR*, vol. abs/2506.10484, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2506.10484

[13] E. D. Sacerdoti, "Planning in a hierarchy of abstraction spaces," *Artif. Intell.*, vol. 5, no. 2, pp. 115–135, 1974. [Online]. Available: https://doi.org/10.1016/0004-3702(74)90026-5

[14] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66

[15] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Demystifying llm-based software engineering agents," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, pp. 801–824, 2025. [Online]. Available: https://doi.org/10.1145/3715754

[16] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, and et al., "Openhands: An open platform for AI software developers as generalist agents," in *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. [Online]. Available: https://openreview.net/forum?id=OJd3ayDDoF

[17] Y. Pan, Z. Chen, S. Lu, Z. Chu, X. Li, H. Li, Y. Feng, C. L. Goues, F. Sarro, M. Monperrus, and H. Ye, "Prometheus: Towards long-horizon codebase navigation for repository-level problem solving," 2025. [Online]. Available: https://arxiv.org/abs/2507.19942

[18] T. R. Team, P. Gao, Z. Tian, X. Meng, X. Wang, R. Hu, Y. Xiao, Y. Liu, Z. Zhang, J. Chen, C. Gao, Y. Lin, Y. Xiong, C. Peng, and X. Liu, "Trae agent: An llm-based agent for software engineering with test-time scaling," 2025. [Online]. Available: https://arxiv.org/abs/2507.23370

[19] A. Zhao, D. Huang, Q. Xu, M. Lin, Y. Liu, and G. Huang, "Expel: LLM agents are experiential learners," in *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, M. J. Wooldridge, J. G. Dy, and S. Natarajan, Eds. AAAI Press, 2024, pp. 19 632–19 642. [Online]. Available: https://doi.org/10.1609/aaai.v38i17.29936

[20] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html

[21] Y. Fu, D. Kim, J. Kim, S. Sohn, L. Logeswaran, K. Bae, and H. Lee, "Autoguide: Automated generation and selection of context-aware guidelines for large language model agents," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/d8efbb5dd415974eb095c3f06bff1f48-Abstract-Conference.html

[22] Z. Z. Wang, J. Mao, D. Fried, and G. Neubig, "Agent workflow memory," in *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, ser. Proceedings of Machine Learning Research, A. Singh, M. Fazel, D. Hsu, S. Lacoste-Julien, F. Berkenkamp, T. Maharaj, K. Wagstaff, and J. Zhu, Eds. PMLR / OpenReview.net, 2025. [Online]. Available: https://proceedings.mlr.press/v267/wang25bx.html

[23] H. Hayashi, B. Pang, W. Zhao, Y. Liu, A. Gokul, S. Bansal, C. Xiong, S. Yavuz, and Y. Zhou, "Self-abstraction from grounded experience for plan-guided policy refinement," *CoRR*, vol. abs/2511.05931, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2511.05931

[24] J. Lin, Y. Guo, Y. Han, S. Hu, Z. Ni, L. Wang, M. Chen, H. Liu, R. Chen, Y. He, D. Jiang, B. Jiao, C. Hu, and H. Wang, "Se-agent: Self-evolution trajectory optimization in multi-step reasoning with llm-based agents," *CoRR*, vol. abs/2508.02085, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2508.02085

[25] S. Gandhi, J. Tsay, J. Ganhotra, K. Kate, and Y. Rizk, "When agents go astray: Course-correcting SWE agents with prms," *CoRR*, vol. abs/2509.02360, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2509.02360

[26] R. Bergmann and W. Wilke, "On the role of abstraction in case-based reasoning," in *Advances in Case-Based Reasoning, Third European Workshop, EWCBR-96, Lausanne, Switzerland, November 14-16, 1996, Proceedings*, ser. Lecture Notes in Computer Science, I. F. C. Smith and B. Faltings, Eds., vol. 1168. Springer, 1996, pp. 28–43. [Online]. Available: https://doi.org/10.1007/BFb0020600

[27] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *J. Syst. Softw.*, vol. 171, p. 110817, 2021. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110817

[28] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019. [Online]. Available: https://doi.org/10.1145/3318162

[29] OpenAI, "Openai GPT-5 system card," *CoRR*, vol. abs/2601.03267, 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2601.03267

[30] Minimax, "Minimax m2.5," 2026, accessed: Feb. 12, 2026. [Online]. Available: https://www.minimax.io/news/minimax-m25/

[31] L. Inc., "Langgraph," 2024, accessed: 2024-12-02. [Online]. Available: https://langchain-ai.github.io/langgraph/

[32] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[33] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 254–265. [Online]. Available: https://doi.org/10.1145/2568225.2568254

[34] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 532–543. [Online]. Available: https://doi.org/10.1145/2786805.2786825

[35] X. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, p. 163. [Online]. Available: https://doi.org/10.1145/3180155.3182536