HISTORY-AWARE AND SCOPE-CONSTRAINED SKILL EVOLUTION FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM)-based coding agents have made substantial progress in automated software engineering, and recent studies further enable them to accumulate reusable knowledge from completed task trajectories. However, as code repositories continuously evolve, accumulated knowledge may become outdated or even mislead agents on new tasks, making it necessary for knowledge to evolve alongside the repository. Existing knowledge self-evolution methods typically update knowledge based on its current state and newly acquired experience, overlooking both the history of successive updates and the scope within which knowledge remains applicable. This leads to *knowledge degeneration* as repeated updates make knowledge increasingly redundant or overly task-specific, and *cross-repository contamination* as repository-specific experience is inappropriately used to modify knowledge beyond its applicable scope. To address these limitations, we propose EvoGraph, a history-aware and scope-constrained framework for knowledge self-evolution in coding agents that represents reusable knowledge as skills. To mitigate knowledge degeneration, EvoGraph maintains an Evolution History Graph (EHG) that preserves historical skill versions and their revision, derivation, and consolidation relations, enabling proposed updates to be assessed against earlier versions and evolution-related skills. To reduce cross-repository contamination, the EHG further records the repository scope of each skill, based on which EvoGraph constrains both skill retrieval and skill updates, restricting inappropriate knowledge reuse and modification across repositories. Experiments on SWE-bench Verified and Pro with Qwen3.6-27B, GPT-5 mini, and DeepSeek-V4-Flash show that EvoGraph achieves average relative improvements of 6.90%–14.04% in resolution rate over all baselines across the three models. All code and data are available at `https://anonymous.4open.science/r/EvoGraph-B1C4`.

1 INTRODUCTION

Recent advances in large language models (LLMs) have substantially accelerated progress in automated software engineering (Jiang et al., 2025). A wide range of coding agents have emerged, such as OpenHands (Wang et al., 2025a) and Claude Code (Anthropic, 2026). Given a natural-language task description, these agents autonomously explore the repository context, edit code files, and execute tests to accomplish complex software-engineering tasks, such as bug fixing (Chowdhury et al., 2024; Deng et al., 2025b) and feature addition (Deng et al., 2025a; Zhou et al., 2026b). Beyond solving each task in isolation, recent studies further enable coding agents to accumulate knowledge across tasks, distilling reusable experience from completed task trajectories to guide subsequent ones (Zhao et al., 2024; Wang et al., 2025b; Qiu et al., 2026; Chen et al., 2025). However, these methods implicitly assume that accumulated knowledge remains valid over time, whereas real-world code repositories continuously evolve through frequent code changes. As interfaces change, new features are added, and obsolete code is removed, previously accumulated knowledge may become outdated and even mislead the agent on new tasks. Therefore, accumulated knowledge should *evolve* alongside the repository to remain effective.

Existing studies on knowledge self-evolution (Fang et al., 2025) typically represent accumulated knowledge as memory (Zhou et al., 2026a) or skills (Zhou et al., 2026c). Both paradigms distill knowledge from completed task trajectories, typically through LLM-based reflection (Li et al., 2026), and accordingly add new entries to or revise existing entries in the knowledge base (Zhao et al., 2024).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

However, these methods share a common assumption: the knowledge base is updated and maintained solely on the basis of its current state and newly acquired experience, independent of the *history* of changes through which that state was reached and of the *scope* within which each entry was acquired and remains applicable.

Such an assumption leads to two limitations in knowledge self-evolution. First, *history-blind updates can lead to knowledge degeneration*. Since updates focus on the current knowledge and new task experience, the cumulative effects of successive additions and edits can be overlooked. Similar experiences may be repeatedly added (Zhang et al., 2026c) and existing knowledge increasingly tailored to individual tasks (Yu et al., 2026), resulting in a more redundant knowledge base and overly task-specific knowledge. This redundancy makes it harder to retrieve the most relevant knowledge (Qiu et al., 2026), while increasingly task-specific instructions can misguide the agent when applied to new tasks, reducing task resolution rates (Lin et al., 2026). Second, *scope-unaware updates can cause cross-repository contamination*. When updates overlook the provenance and applicability of knowledge entries, repository-specific experience may be used to revise shared knowledge or entries intended for other repositories, thereby misleading coding agents in those contexts and lowering their resolution rates (Kim et al., 2026).

To address these limitations, we propose EvoGraph, a history-aware and scope-constrained framework for knowledge self-evolution in coding agents that represents reusable knowledge as skills (Yu et al., 2026; Yang et al., 2026b; Jiang et al., 2026; Li et al., 2026). At the core of EvoGraph is an *Evolution History Graph (EHG)* that encodes both the evolution history and the repository scope of accumulated skills. To mitigate knowledge degeneration, the EHG preserves historical skill versions and records their revision, derivation, and consolidation relations. This enables EvoGraph to compare proposed updates with earlier versions and evolution-related skills, helping identify whether a revision narrows a skill's previous applicability or an addition duplicates existing knowledge. By retaining these references across successive updates, EvoGraph helps detect cumulative degeneration that may be difficult to recognize from the current skill alone, such as a general fault-localization skill that is narrowed step by step to a single API, with each revision appearing reasonable relative to its immediate predecessor. To reduce cross-repository contamination, the EHG records the repository scope of each skill, either general or specific to one repository, and EvoGraph accordingly imposes repository-scoped constraints that govern both which skills a task may retrieve and reuse and which knowledge entries may be modified based on its experience. These constraints prevent tasks from reusing knowledge specific to unrelated repositories and keep repository-specific experience from being inappropriately used to modify shared knowledge or entries associated with other repositories.

We evaluate EvoGraph on two popular coding benchmarks, SWE-bench Verified (Chowdhury et al., 2024) and SWE-bench Pro (Deng et al., 2025b), with three advanced LLMs, and perform comprehensive comparisons with state-of-the-art coding agents and self-evolution methods. Across Qwen3.6-27B, GPT-5 mini, and DeepSeek-V4-Flash, EvoGraph achieves average improvements of 8.02% and 14.77% in resolution rate over all baselines on SWE-bench Verified and Pro, respectively.

In summary, our main contributions are as follows:

- We build an Evolution History Graph that encodes both the evolution history and the repository scope of skills, preserving historical skill versions with their evolution relations and recording whether each skill is general or specific to one repository, to mitigate knowledge degeneration and cross-repository contamination, respectively.
- We propose EvoGraph, a knowledge self-evolution framework for coding agents based on the EHG. Extensive experiments on SWE-bench Verified and Pro show average relative improvements of 6.90%–14.04% in resolution rate over all baselines across three backbone LLMs.

2 Related Work

2.1 Automated Software Issue Resolution

Rapid advances in LLMs have enabled coding agents to resolve repository-level software issues by generating code patches from natural-language issue descriptions (Jiang et al., 2025; Jimenez et al., 2024; Liu et al., 2024). Existing approaches can be broadly categorized into **agent-based** and **workflow-based** methods. **Agent-based** methods (Yang et al., 2024; Wang et al., 2025a; Zhang

et al., 2024) equip LLM agents with tools for code navigation, editing, and execution, allowing them to explore repositories and iteratively refine patches based on environmental feedback. For example, SWE-agent (Yang et al., 2024) equips agents with tools for repository exploration, code editing, and execution, while terminal-native agents such as Claude Code (Anthropic, 2026) provide these capabilities through command-line-based agent–computer interfaces. In contrast, **workflow-based** methods, such as Agentless (Xia et al., 2025a) and Moatless (Örwall, 2024), decompose issue resolution into predefined stages such as localization, patch generation, and validation, improving stability and reproducibility through structured pipelines. In addition, post-training methods further improve software-engineering capabilities through supervised fine-tuning (Pan et al., 2025; Xie et al., 2025) and reinforcement learning (Wei et al., 2025). Unlike existing issue-resolution approaches, which primarily optimize the resolution of individual issues, EvoGraph represents cross-instance knowledge as skills in the EHG, which preserves their historical versions and typed relations and records their repository scopes, enabling continual organization, evolution, and reuse.

2.2 Self-Evolving Coding Agents

Self-evolving coding agents improve their behavior based on previous interactions and carry these improvements over to subsequent tasks. According to what is evolved, existing approaches can be broadly categorized into **knowledge evolution** and **agent framework evolution** methods. **Knowledge evolution** methods keep the agent unchanged and accumulate reusable knowledge from completed tasks, which is retrieved and provided to the agent on subsequent tasks. It can be further divided into **memory-based** and **skill-based** methods. **Memory-based** methods (Ouyang et al., 2026; Chen et al., 2025; Mu et al., 2026) store declarative experience, such as reflections, past solutions, or insights distilled from successful and failed trajectories, and retrieve relevant entries to inject into the agent's context. For example, ExpeRepair (Mu et al., 2026) maintains an episodic memory of concrete repair demonstrations and a semantic memory of abstract reflective insights to guide subsequent repairs. **Skill-based** methods (Ni et al., 2026; Yang et al., 2026a) encapsulate procedural knowledge as self-contained modules with applicability conditions and actionable instructions, which the agent loads on demand (Zhang et al., 2025a; Ling et al., 2026). These methods acquire skills from execution feedback or accumulated trajectories and increasingly maintain them continually, adding or updating skills as new experience arrives (Zhang et al., 2026a; Li et al., 2026). For example, CodeSkill (Li et al., 2026) extracts multi-granularity procedural skills from coding-agent trajectories and learns to evolve and maintain a compact skill library. In contrast, **agent framework evolution** methods (Zhang et al., 2026b; Wang et al., 2026b) evolve the agent itself by iteratively modifying its code, such as its tools and workflows, and retaining the variants validated on coding benchmarks. For example, Huxley-Gödel Machine (Wang et al., 2026b) further guides this self-modification by estimating each variant's potential from the performance of its descendants.

Different from existing self-evolving methods, which evolve knowledge solely based on its current state and newly acquired experience, EvoGraph models how each piece of knowledge arrived at its current form through the EHG. The EHG preserves historical skill versions and records their revision, derivation, and consolidation relations, allowing proposed updates to be checked against earlier versions to detect cumulative degeneration such as redundancy and over-specialization. Moreover, the EHG attaches a repository scope to each skill and constrains both retrieval and updates accordingly, separating general from repository-specific knowledge to prevent cross-repository contamination.

3 EvoGraph

3.1 Overview

We introduce EvoGraph, a history-aware and scope-constrained framework for knowledge self-evolution in coding agents, to mitigate knowledge degeneration and cross-repository contamination. As shown in Figure 1, EvoGraph stores all reusable skills in an *Evolution History Graph (EHG)* that models both the evolution history and the repository scope of skills (Section 3.2). To address history-blind updates, the EHG retains every skill version as a node instead of overwriting it, and connects versions through *revision*, *derivation*, and *consolidation* edges, so that a proposed update can be compared with the path through which a skill reached its current form. To address scope-unaware updates, each node records the repository scope in which it applies, either general or specific to one repository, so that both access to and modification of knowledge can be bounded by repository. For

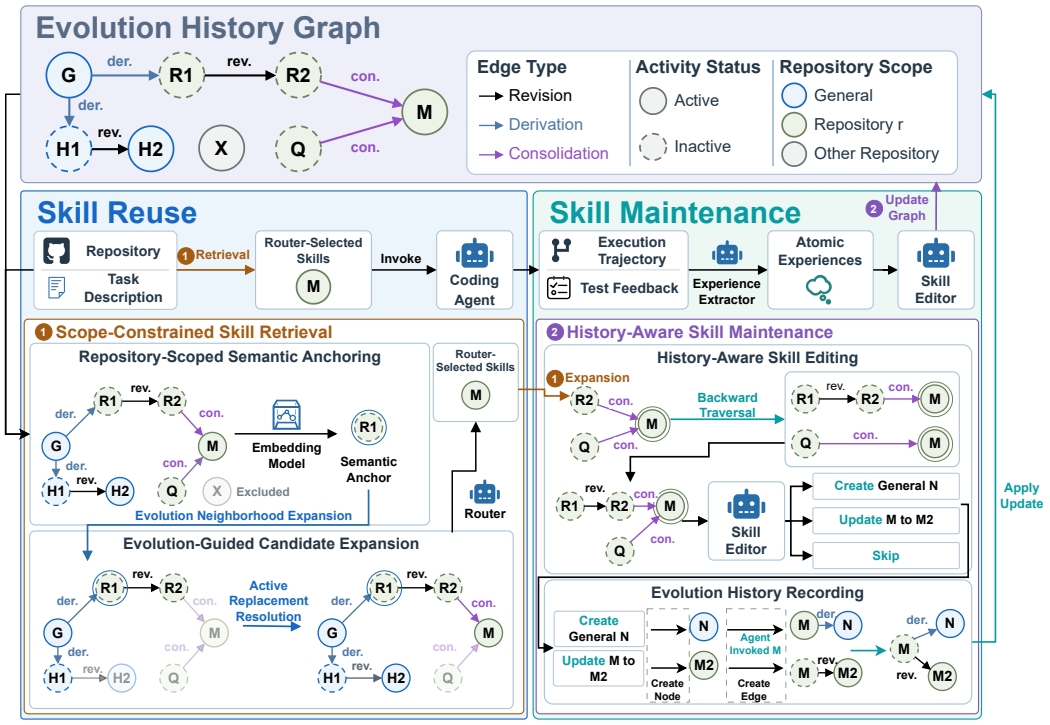

Figure 1: Overview of EvoGraph. Skill reuse follows the EHG to select skills that guide rollout. Skill maintenance extracts experience, edits skills informed by historical versions, and records evolution.

skill reuse, EvoGraph uses scope-constrained skill retrieval to acquire active skills from the current repository scope (Section 3.3). After each task, EvoGraph uses history-aware skill maintenance to revise the relevant skills in light of their evolution history and records the changes in the EHG (Section 3.4). The detailed workflow is in Appendix A.1.

3.2 Evolution History Graph

Existing self-evolution methods keep knowledge in a flat collection and overwrite entries in place, so they lose both how each entry has changed and where it applies. To detect knowledge degeneration, we need to check whether a new addition repeats guidance that an earlier version once contained and whether a revision narrows conditions that earlier versions covered, which requires keeping earlier versions and recording how each version was produced. To prevent cross-repository contamination, we further need to record where each version applies, so that both skill reuse and skill modification can be limited to general and current-repository knowledge. These requirements are naturally expressed as a graph, in which nodes store the contents and repository scope of each version and edges record how versions evolve from one another. Thus, we propose the EHG, a directed graph of skill versions connected by typed evolution relations.

Formally, after the t-th round of skill maintenance, the EHG can be represented as $\mathcal{G}_t = (V_t, E_t)$. Each node $v \in V_t$ is a skill version associated with a four-tuple $v = \langle c(v),\, d(v),\, \sigma(v),\, a(v) \rangle$, where $c(v)$ is the skill content, consisting of its `SKILL.md` instructions and any accompanying scripts and resources; $d(v)$ is the routing description, which specifies the skill's name, description, and when-to-use guidance; $\sigma(v)$ is the repository scope, which is either GENERAL or a specific repository; and $a(v) \in \{\text{ACTIVE}, \text{INACTIVE}\}$ is the activity status. Only active versions can be executed or edited, while inactive versions are retained to detect whether later updates repeat previously covered guidance. Each edge $\langle u, \ell, v \rangle \in E_t$ indicates that version v evolved from version u through a relation of type $\ell \in \{\text{REVISION}, \text{DERIVATION}, \text{CONSOLIDATION}\}$. According to what they capture, these relations fall into two groups. *Revision* and *consolidation* edges describe how versions are replaced over time. When a skill is revised, a *revision* edge links its previous version to the new one, and when several skills are merged, a *consolidation* edge links the active version of each of them to the

merged result. In contrast, *derivation* edges trace where the experience behind a version came from, linking each version invoked during a rollout to every version produced from that rollout's experience. Accordingly, a *skill* is a sequence of versions linked by *revision* edges. Through these designs, the EHG provides evolution history that supports comparison across successive updates and repository scope that bounds knowledge access and modification throughout the self-evolution lifecycle.

3.3 SKILL REUSE

To prevent cross-repository contamination and to reach usable skills even when an earlier version best matches the task, we propose the *scope-constrained skill retrieval* algorithm, which takes a task description q and repository r as input and retrieves from the EHG a set of active skills S_t that are appropriate to expose to the coding agent. As detailed in Algorithm 1 (Appendix A.3), the algorithm first uses *repository-scoped semantic anchoring* to limit retrieval to the current repository scope, then applies *evolution-guided candidate expansion* to follow evolution relations from the matched versions to versions that are currently active, and finally lets a router select S_t from the resulting candidates.

Repository-Scoped Semantic Anchoring. We first restrict the EHG to the *scoped EHG* $\mathcal{G}_t(r) = (V_t(r), E_t(r))$, which keeps only versions whose scope is either general or the current repository r, together with the edges among them,

$$\begin{aligned} V_t(r) &= \{v \in V_t \mid \sigma(v) = \textsc{general} \ \vee \ \sigma(v) = r\}, \\ E_t(r) &= \{\langle u, \ell, v\rangle \in E_t \mid u, v \in V_t(r)\}. \end{aligned} \tag{1}$$

Thus, versions scoped to other repositories are excluded to prevent cross-repository contamination. To further narrow down the skills that may be used, we first select semantic anchors related to the current task, from which the subsequent expansion delimits the subgraph containing candidate versions. Specifically, since the routing description $d(v)$ of a skill version v states when it should be used, we rank versions in $\mathcal{G}_t(r)$ by the embedding similarity between q and $d(v)$ and take the Top-N as *semantic anchors* A_t, following prior work (Xia et al., 2026).

Evolution-Guided Candidate Expansion. The retrieved semantic anchors may be incomplete or unusable: versions of related skills can fall below the similarity cutoff, and some anchors may be inactive versions. We therefore further expand the subgraph around the semantic anchors through two graph-based expansion operations. To recover omitted related skills, we perform *evolution neighborhood expansion*, which adds the parents, children, and siblings of each anchor in $\mathcal{G}_t(r)$, so as to include the versions an anchor was built from, the versions later built on it, and alternative versions built from the same source. It expands only one hop from the original anchors, so that candidates stay close to what matched the task. To make inactive candidates usable, we then perform *active replacement resolution*, which follows *revision* and *consolidation* edges from each inactive candidate until an active version is reached, so as to map the inactive candidate to the version that replaced it. This step ignores *derivation* edges, because a derived skill does not replace the skill that guided it. All versions reached during evolution neighborhood expansion and active replacement resolution, together with the evolution edges among them, form the candidate subgraph $\mathcal{C}_t$. Appendix A.4 walks through them on a concrete example.

The expansion may yield multiple active versions that are relevant to the task, some of which may overlap in content. We therefore use an LLM *router* to make the final selection, following prior work (Chen et al., 2025). Based on q and the routing descriptions and relations in $\mathcal{C}_t$, the router selects the final skills S_t from the active versions in $\mathcal{C}_t$. Then, the coding agent loads these skills from $\mathcal{G}_t$ and performs the rollout, producing the execution trajectory τ_t and test feedback f_t for skill maintenance. Appendix A.2 specifies the retrieval and execution interfaces.

3.4 SKILL MAINTENANCE

To mitigate knowledge degeneration and to keep repository-specific experience within its applicable scope, we propose *history-aware skill maintenance*, which takes repository r, router-selected skills S_t, execution trajectory τ_t, and test feedback f_t as input and updates the EHG from $\mathcal{G}_t$ to $\mathcal{G}_{t+1}$. Skill maintenance first uses *history-aware skill editing* to revise skills against their earlier versions within the current repository scope, and then applies *evolution history recording* to materialize the edited skills as new versions and evolution relations in the EHG.

History-Aware Skill Editing. We first use an *experience extractor* to convert τ_t and f_t into *atomic experiences*, each expressing reusable knowledge with references to supporting trajectory steps or repository code. We then construct a *maintenance subgraph* within $\mathcal{G}_t(r)$ by applying the evolution-guided candidate expansion of skill reuse (see Section 3.3) to the router-selected skills S_t. Starting from the active versions in the expanded subgraph, we perform *backward history traversal*, which follows incoming *revision* and *consolidation* edges until no new predecessor remains, and add the collected predecessors to the maintenance subgraph.

An agent *skill editor* then compares each atomic experience with the active skills and their predecessors in the maintenance subgraph. To prevent redundancy, the editor chooses the *skip* action for covered experience, the *update* action for a supported refinement of an existing skill, and the *create* action only for an independently reusable capability. If the history shows that apparently new guidance was removed by earlier edits, the editor assesses whether to restore it within the existing skill rather than duplicate it. To prevent over-specialization, the editor compares each proposed update with the skill's predecessors to identify conditions accumulated from individual tasks, keeping a new condition only when it is supported by the skill's objective and by the trajectory steps or repository code referenced in the atomic experience, and preserving earlier guidance that remains applicable.

Since editing targets come only from $\mathcal{G}_t(r)$, the editor cannot modify skills local to other repositories. For each update or creation, the editor also determines its scope by checking whether the change depends on repository-local APIs, contracts, paths, or conventions. A generally applicable change may refine a general skill, whereas a repository-dependent change updates an r-scoped skill or creates one if no eligible target exists. Appendix A.5 details the extractor and editor interfaces.

Evolution History Recording. Instead of modifying versions in place, the skill editor edits a copy of an active version in the maintenance subgraph for an update and drafts the first version of a new skill for a create. Each drafted version, and each edited copy that differs from the version it was copied from, is added as a node v with $a(v) = \text{ACTIVE}$, whose $c(v)$, $d(v)$, and $\sigma(v)$ are taken from the editor's output; for an update, the copied version is marked inactive and linked to v by a *revision* edge. We further record $\langle u, \text{DERIVATION}, v \rangle$ for every $u \in I_t$ and $v \in O_t$, where O_t denotes the nodes added in this round and I_t denotes the skill versions actually invoked during the rollout rather than those merely selected by the router. Appendix A.6 specifies the version bookkeeping.

Since each maintenance round only compares experience with skills related to the current task, overlapping active skills may accumulate across rounds. We therefore periodically *consolidate* same-scope skills with essentially the same instructions into a new active node, deactivating each source node and linking it to the replacement by a *consolidation* edge (Appendix A.7).

4 Experiments

4.1 Experimental Setup

Benchmarks & Metrics. Following prior work (Xia et al., 2025b; Xiao et al., 2026), we mainly evaluate EvoGraph on SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024) and SWE-bench Pro (Deng et al., 2025b), two widely used benchmarks for real-world issue resolution (detailed in Appendix B). We use *resolution rate* (Resolved%) as the main metric to measure the percentage of tasks for which the agent's final patch passes all the benchmark's associated tests. We also report *average token* (#Token) consumption to measure the inference cost of agents.

Baselines. To evaluate the effectiveness of EvoGraph, we compare it with three categories of baselines: (1) *general-purpose coding agents*, including mini-SWE-agent (Yang et al., 2024) and Claude Code (Anthropic, 2026); (2) *online self-evolution method*, including Live-SWE-agent (Xia et al., 2025b); and (3) *knowledge accumulation-based self-evolution methods*, including ReasoningBank (Ouyang et al., 2026) and MemGovern (Wang et al., 2026a). These methods (detailed in Appendix C) represent state-of-the-art approaches in their respective categories and have been widely adopted in prior work (Chen et al., 2026; Wong et al., 2025).

Implementation Details. Following prior work (Merrill et al., 2026; Tang et al., 2026; Xu et al., 2026), we implement EvoGraph on top of Claude Code and run all experiments with Harbor (Harbor Framework Team, 2026), an open-source framework for building and running agent evaluations at scale. We use three LLMs as backbone models, including Qwen3.6-27B, GPT-5 mini, and DeepSeek-

Table 1: Overall performance of EvoGraph on SWE-bench Verified and SWE-bench Pro.

Method	SWE-bench Verified			SWE-bench Pro		
	Resolved (%)	Δ	#Token (M)	Resolved (%)	Δ	#Token (M)
Qwen3.6-27B						
mini-SWE-agent	65.60	↑8.54%	0.982	45.69	↑14.38%	2.103
Claude Code	66.00	↑7.88%	1.132	46.65	↑12.03%	2.937
Live-SWE-agent	62.60	↑13.74%	0.918	43.77	↑19.40%	0.912
MemGovern	63.60	↑11.95%	1.103	47.19	↑10.74%	3.140
ReasoningBank	64.40	↑10.56%	0.682	39.83	↑31.21%	0.964
EvoGraph	**71.20**	–	1.277	**52.26**	–	3.457
GPT-5 mini						
mini-SWE-agent	53.80	↑10.04%	0.305	34.33	↑14.36%	0.466
Claude Code	53.40	↑10.86%	0.919	35.02	↑12.11%	2.707
Live-SWE-agent	54.80	↑8.03%	0.261	31.60	↑24.24%	0.423
MemGovern	52.80	↑12.12%	0.757	33.92	↑15.74%	2.067
ReasoningBank	54.40	↑8.82%	0.362	33.79	↑16.19%	0.445
EvoGraph	**59.20**	–	1.024	**39.26**	–	3.210
DeepSeek-V4-Flash						
mini-SWE-agent	73.20	↑3.55%	1.825	47.88	↑10.28%	3.645
Claude Code	73.40	↑3.27%	2.727	47.19	↑11.89%	4.911
Live-SWE-agent	73.60	↑2.99%	2.901	48.70	↑8.42%	5.301
MemGovern	72.00	↑5.28%	2.263	48.15	↑9.66%	4.269
ReasoningBank	73.80	↑2.71%	2.456	47.60	↑10.92%	4.583
EvoGraph	**75.80**	–	3.099	**52.80**	–	5.218

V4-Flash, since they cover both open-source and closed-source models across different parameter scales. Each open-source model is deployed with vLLM (Kwon et al., 2023). To prevent data leakage during skill evolution, we order the instances in both SWE-bench Verified and Pro chronologically by the timestamps of their base commits, so that repository information from future versions cannot enter the EHG prematurely. Following knowledge accumulation-based methods (Ouyang et al., 2026; Wang et al., 2026a), to initialize the EHG, we first apply EvoGraph to 1,000 randomly sampled SWE-smith (Yang et al., 2025) instances before the evaluation (see Appendix D.1). These instances come from repositories that do not appear in either benchmark, and we keep only general skills that are not tied to any specific repository, so this initialization does not leak benchmark information. Each run of EvoGraph starts from its own copy of this initialized EHG and continues to evolve it online during evaluation, so that no knowledge is shared across runs. Appendix D.2 further provides model deployment and baseline configuration details.

4.2 Overall Effectiveness

Table 1 compares the end-to-end performance of all methods on both benchmarks and backbone models. Experimental results show that EvoGraph achieves the highest resolution rate in all six settings, with improvements of 2.71%–13.74% and 8.42%–31.21% over the baselines on SWE-bench Verified and SWE-bench Pro, respectively, demonstrating robust improvements across different LLM backbones. Compared with *knowledge accumulation-based self-evolution methods*, i.e., ReasoningBank and MemGovern, EvoGraph achieves average improvements of 8.57% and 15.74% on SWE-bench Verified and SWE-bench Pro, respectively, whereas these methods even underperform the best general-purpose coding agent in 8 of the 12 settings. This suggests that accumulated knowledge can mislead the agent when its evolution history and repository scope are overlooked, while the history-aware EHG and scope-constrained skill retrieval and maintenance enable EvoGraph to turn it into consistent gains. A case study in Appendix E.2 further illustrates how the two complement each other during skill maintenance: repository scope keeps Django-specific repair instructions out of general skills, while evolution history leads the editor to extend an existing skill instead of creating an overlapping one. The smallest gain appears with DeepSeek-V4-Flash on SWE-bench Verified. This is expected, as SWE-bench Verified has been reported to approach saturation for strong models (OpenAI, 2026). While EvoGraph incurs additional token consumption, the consistent resolution gains indicate an acceptable tradeoff.

Table 2: Ablation results on SWE-bench Verified and SWE-bench Pro using Qwen3.6-27B.

	SWE-bench Verified		SWE-bench Pro	
Variant	Resolved (%)	Rel. Δ	Resolved (%)	Rel. Δ
EvoGraph	**71.20**	–	**52.26**	–
w/o EHG	66.00	↓7.30%	46.65	↓10.73%
w/o Scope-Constrain	70.00	↓1.69%	49.66	↓4.98%
w/o History-Aware Maintenance	66.80	↓6.18%	47.74	↓8.65%
w/o EHG Initialization	69.60	↓2.25%	51.71	↓1.05%

4.3 Ablation Study

EvoGraph builds on the EHG and consists of two modules operating on it, i.e., scope-constrained skill retrieval (Section 3.3) and history-aware skill maintenance (Section 3.4). To analyze the contributions of the EHG and the two modules to EvoGraph's effectiveness, we construct one variant for each of these three components. We first remove the EHG (*w/o EHG*), so that neither module can operate and EvoGraph reduces to the vanilla Claude Code. For scope-constrained skill retrieval, we discard the repository-scoped constraints (*w/o Scope-Constrain*), allowing retrieval to expose skills specific to other repositories. Because skill maintenance draws its editing targets from the same scoped EHG, such skills also become modifiable. For history-aware skill maintenance, we freeze the initialized EHG during evaluation (*w/o History-Aware Maintenance*), so that experience from earlier evaluation tasks is never incorporated into skills. Besides, *w/o EHG Initialization* starts from an empty EHG instead of the SWE-smith-initialized one to examine the auxiliary initialization stage. We evaluate all variants on SWE-bench Verified and SWE-bench Pro using Qwen3.6-27B, as EvoGraph achieves the largest absolute gains over the strongest baseline with this model.

Table 2 reports the results, showing that all three components contribute to the effectiveness of EvoGraph. Removing the EHG causes the largest degradation, with relative drops of 7.30% and 10.73% on SWE-bench Verified and SWE-bench Pro, respectively, indicating that organizing skills by their evolution history and repository scope provides effective guidance beyond the capability of the underlying coding agent. Removing *History-Aware Maintenance* leads to nearly comparable drops of 6.18% and 8.65%, as the frozen EHG outperforms *w/o EHG* by merely 0.80 and 1.09 points. This shows that the gains of EvoGraph mainly come from continuously refining skills against their evolution history, which keeps accumulated knowledge aligned with the evolving repositories rather than relying on a fixed set of transferred skills. Removing *Scope-Constrain* reduces the resolution rate by 1.69% and 4.98%, suggesting that restricting both skill retrieval and maintenance to general and current-repository skills helps mitigate cross-repository contamination. The larger drop on SWE-bench Pro is consistent with its more diverse repositories, where repository-specific skills are more likely to be exposed to and modified by unrelated tasks. In comparison, *w/o EHG Initialization* causes smaller drops of 2.25% and 1.05%, indicating that the transferred general skills provide a useful starting point, while most of the effective knowledge is acquired through history-aware skill maintenance during evaluation.

5 Analysis

Evolution Trend of the Knowledge Base. To investigate how the knowledge base grows during continual self-evolution, we track the knowledge items accumulated by EvoGraph, ReasoningBank, and Live-SWE-agent on SWE-bench Pro with the three backbone models used in our experiments. We count *knowledge items* (i.e., active skills in EvoGraph, memory items in ReasoningBank, and saved tool scripts in Live-SWE-agent). As shown in Figure 2, EvoGraph keeps the growth of knowledge items within a controllable range, ending with 216, 241, and 207 items with GPT-5 mini, DeepSeek-V4-Flash, and Qwen3.6-27B, respectively. In most cases, it reduces the number of knowledge items by a factor of 9.10 to 11.86 relative to the baselines. ReasoningBank grows linearly to 2,193 items with every model because it requires the LLM to insert three knowledge items after each task. Such evolution tends to accumulate duplicate knowledge from similar tasks, which introduces redundancy and can make retrieval and reuse more difficult. Live-SWE-agent also accumulates 2,858 and 2,238 knowledge items with DeepSeek-V4-Flash and Qwen3.6-27B, respectively. We manually inspect 50

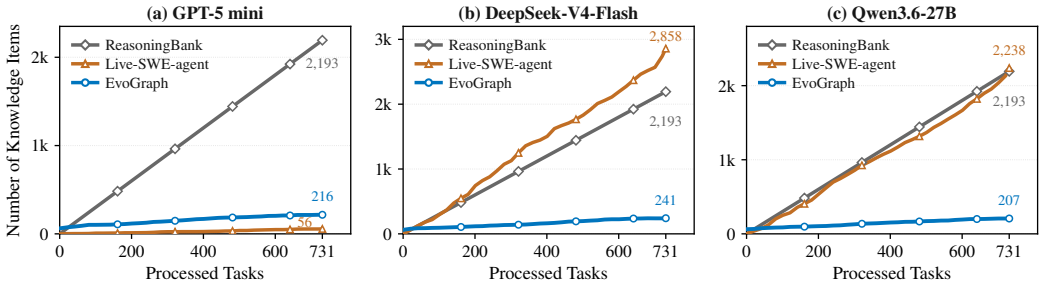

Figure 2: Growth of knowledge items on SWE-bench Pro across three models.

randomly sampled scripts written by Live-SWE-agent with each of the three models and find that 77.33% of them focus on specific types of edit, view, and search operations and 95.33% hard-code information of the task they were written for (see Table 3 in Appendix E.1), suggesting that evolution without history tends to make knowledge overly task-specific. However, Live-SWE-agent saves only 56 knowledge items with GPT-5 mini, and we examine the cause of this behavior. We manually inspect 50 randomly sampled one-off scripts written by GPT-5 mini and find that 94.00% of them are run as inline Python instead of being saved for accumulation. Since the model itself rarely regards these scripts as worth keeping, scripts may not serve well as carriers of reusable knowledge. In contrast, history-aware skill maintenance lets EvoGraph incorporate new experience into existing skills or skip it when already covered, rather than storing each experience as a new item. Across the three models, EvoGraph revises existing skills 433 to 489 times, roughly twice the number of its active skills, and skips 134 to 174 experiences already covered by existing skills (see Table 4 in Appendix E.1), indicating that the smaller knowledge base of EvoGraph does not come from learning less, but from revising existing skills with new experience.

Cross-Repository Knowledge Contamination. To investigate whether repository scope prevents knowledge learned in one repository from misleading tasks in another, we analyze the skill calls made by EvoGraph and its *w/o Scope-Constrain* variant from the ablation study on SWE-bench Pro with Qwen3.6-27B. We count *contaminated skill calls*, i.e., calls to skills whose content is specific to a repository other than the current task's repository, such as its contracts or APIs. EvoGraph reduces contaminated calls from 341 of 407 to 17 of 441, lowering the contamination rate from 83.78% to 3.85%, a relative reduction of 95.4%, which is consistent with the 4.98% drop of *w/o Scope-Constrain* on SWE-bench Pro in Table 2. The remaining 17 contaminated calls all come from three general skills, which pass the scope constraint but illustrate their otherwise general guidance with contracts and APIs from a specific repository. Meanwhile, total skill calls slightly increase from 407 to 441, indicating that restricting the scope does not prevent the agent from recognizing and invoking the retrieved skills.

6 Conclusion

This paper presents EvoGraph, a history-aware and scope-constrained knowledge self-evolution framework that keeps the skills accumulated by coding agents evolving alongside continuously changing repositories. It maintains an Evolution History Graph that preserves historical skill versions and their evolution relations, and applies repository-scoped constraints to both skill retrieval and maintenance, mitigating knowledge degeneration and cross-repository contamination. EvoGraph benefits both task resolution and the accumulated knowledge itself. Across three backbone LLMs, it achieves the highest resolution rate in all six settings on SWE-bench Verified and SWE-bench Pro, with relative improvements up to 31.21% over all baselines. On SWE-bench Pro, it maintains 9.10–11.86$\times$ fewer knowledge items than the baselines, and reduces the cross-repository contamination rate of skill calls by 95.4% relative to its variant without scope constraints. These results show that accounting for both the evolution history and the repository scope of accumulated knowledge enables coding agents to benefit more reliably from past experience.

AI USE STATEMENT

In this work, we used generative AI tools to assist with translation and to polish the writing of the paper. We have not used generative AI tools to develop conceptual frameworks, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, or interpret results, and generating synthetic data sets, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, cleaning and reformatting datasets, and supporting qualitative and thematic data analysis are not applicable to this work. Note that the LLMs used in our experiments (Qwen3.6-27B, GPT-5 mini, and DeepSeek-V4-Flash) serve as backbone models of the evaluated coding agents; they are objects of study rather than tools used to conduct the research. We have reviewed all AI-assisted work. The authors manually checked every AI-translated and AI-polished passage against the original text to ensure that it did not alter technical content, experimental results, or citations. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Anthropic. Claude code, 2026. URL https://code.claude.com/. Accessed September 3, 2026.

Silin Chen, Shaoxin Lin, Yuling Shi, Heng Lian, Xiaodong Gu, Longfei Yun, Dong Chen, Lin Cao, Jiyang Liu, Nu Xia, and Qianxiang Wang. SWE-exp: Experience-driven software issue resolution. *arXiv preprint arXiv:2507.23361*, 2025. URL https://arxiv.org/abs/2507.23361.

Yang Chen, Aliya Ahmad, Yiheng Zhou, and Reyhaneh Jabbarvand. Unlocking model potentials through adaptive multi-agent scaffolding for efficient issue resolution. *arXiv preprint arXiv:2606.25514*, 2026. URL https://arxiv.org/abs/2606.25514.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Le Deng, Zhonghao Jiang, Jialun Cao, Michael Pradel, and Zhongxin Liu. Nocode-bench: A benchmark for evaluating natural language-driven feature addition. *arXiv preprint arXiv:2507.18130*, 2025a.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-bench pro: Can AI agents solve long-horizon software engineering tasks?, 2025b. URL https://arxiv.org/abs/2509.16941.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, et al. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*, 2025.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. URL https://doi.org/10.5281/zenodo.20953922.

Guanyu Jiang, Zhaochen Su, Xiaoye Qu, and Yi R. (May) Fung. XSkill: Continual learning from experience and skills in multimodal agents. In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2603.12056.

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

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Kangsan Kim, Minki Kang, Taeil Kim, Yanlai Yang, Mengye Ren, and Sung Ju Hwang. Memory transfer learning: How memories are transferred across domains in coding agents. *arXiv preprint arXiv:2604.14004*, 2026. URL https://arxiv.org/abs/2604.14004.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Yanzhou Li, Yiran Zhang, Xiaoyu Zhang, Xiaoxia Liu, and Yang Liu. CODESKILL: Learning self-evolving skills for coding agents. *arXiv preprint arXiv:2605.25430*, 2026. URL https://arxiv.org/abs/2605.25430.

Huawei Lin, Peng Li, Jie Song, Fuxin Jiang, and Tieying Zhang. MUSE-Autoskill: Self-evolving agents via skill creation, memory, management, and evaluation. *arXiv preprint arXiv:2605.27366*, 2026. URL https://arxiv.org/abs/2605.27366.

George Ling, Shanshan Zhong, and Richard Huang. Agent skills: A data-driven analysis of Claude skills for extending large language model functionality. *arXiv preprint arXiv:2602.08004*, 2026. URL https://arxiv.org/abs/2602.08004.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024. URL https://arxiv.org/abs/2409.02977.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Fangwen Mu, Junjie Wang, Lin Shi, Song Wang, Shoubin Li, and Qing Wang. Experepair: Dual-memory enhanced llm-based repository-level program repair. *Proceedings of the ACM on Software Engineering*, 3(FSE):3954–3976, 2026.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2Skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026. URL https://arxiv.org/abs/2603.25158.

OpenAI. Why SWE-bench verified no longer measures frontier coding capabilities, 2026. URL https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/.

Albert Örwall. Moatless tools. GitHub repository, 2024. URL https://github.com/aorwall/moatless-tools.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2509.25140.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-gym. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025. URL https://arxiv.org/abs/2412.21139.

Libin Qiu, Zhirong Gao, Junfu Chen, Yuhang Ye, Weizhi Huang, Xiaobo Xue, Wenkai Qiu, and Shuo Tang. AutoRefine: From trajectories to reusable expertise for continual LLM agent refinement. *arXiv preprint arXiv:2601.22758*, 2026. URL https://arxiv.org/abs/2601.22758.

Yuheng Tang, Kaijie Zhu, Bonan Ruan, Chuqi Zhang, Michael Yang, Hongwei Li, Suyue Guo, Tianneng Shi, Zekun Li, Christopher Kruegel, Giovanni Vigna, Dawn Song, William Yang Wang, Lun Wang, Yangruibo Ding, Zhenkai Liang, and Wenbo Guo. DevOps-Gym: Benchmarking AI agents in software DevOps cycle. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=bP48r4dt7Z.

Qihao Wang, Ziming Cheng, Shuo Zhang, Fan Liu, Rui Xu, Heng Lian, Kunyi Wang, Xiaoming Yu, Jianghao Yin, Sen Hu, et al. MemGovern: Enhancing code agents through learning from governed human experiences. *arXiv preprint arXiv:2601.06789*, 2026a. URL https://arxiv.org/abs/2601.06789.

Wenyi Wang, Piotr Piekos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-Gödel Machine: Human-level coding agent development by an approximation of the optimal self-improving machine. In *International Conference on Learning Representations*, 2026b. URL https://arxiv.org/abs/2510.21614.

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. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=OJd3ayDDoF.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911, 2025b. URL https://proceedings.mlr.press/v267/wang25bx.html.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025. URL https://arxiv.org/abs/2502.18449.

Sherman Wong, Zhenting Qi, Zhaodong Wang, Nathan Hu, Samuel Lin, Jun Ge, Erwin Gao, Wenlin Chen, Yilun Du, Minlan Yu, and Ying Zhang. Confucius Code Agent: Scalable agent scaffolding for real-world codebases. *arXiv preprint arXiv:2512.10398*, 2025. URL https://arxiv.org/abs/2512.10398.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE), 2025a. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-SWE-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025b. URL https://arxiv.org/abs/2511.13646.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. SkillRL: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026. URL https://arxiv.org/abs/2602.08234.

Chuan Xiao, Zhengbo Jiao, Shaobo Wang, Wei Wang, Bing Zhao, Hu Wei, Linfeng Zhang, and Lin Qu. Socratic-SWE: Self-evolving coding agents via trace-derived agent skills. *arXiv preprint arXiv:2606.07412*, 2026. URL https://arxiv.org/abs/2606.07412.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. SWE-fixer: Training open-source LLMs for effective and efficient GitHub issue resolution. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 1123–1139, 2025. doi: 10.18653/v1/2025.findings-acl.62. URL https://aclanthology.org/2025.findings-acl.62/.

Jingxuan Xu, Ken Deng, Weihao Li, Songwei Yu, Huaixi Tang, Haoyang Huang, Zhiyi Lai, Zizheng Zhan, Yanan Wu, Chenchen Zhang, et al. SWE-Compass: Towards unified evaluation of agentic coding abilities for large language models. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/64552.

Chen Yang, Jiashuo Tian, Ziqi Wang, Xinyin Liu, Meiru Ye, and Junjie Chen. Learning globally reusable skills for coding agents. *arXiv preprint arXiv:2608.06153*, 2026a. URL https://arxiv.org/abs/2608.06153.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2504.21798.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, Bo Zhang, and Liang He. AutoSkill: Experience-driven lifelong learning via skill self-evolution. *arXiv preprint arXiv:2603.01145*, 2026b. URL https://arxiv.org/abs/2603.01145.

Simon Yu, Gang Li, Weiyan Shi, and Peng Qi. PolySkill: Learning generalizable skills through polymorphic abstraction for continual learning. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/e350897ed832f9cad6f2e223e7acad6d-Abstract-Conference.html.

Barry Zhang, Keith Lazuka, and Mahesh Murag. Equipping agents for the real world with agent skills. Anthropic Engineering, October 2025a. URL https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills.

Haozhen Zhang, Quanyu Long, Jianzhu Bao, Tao Feng, Weizhi Zhang, Haodong Yue, and Wenya Wang. MemSkill: Learning and evolving memory skills for self-evolving agents. *arXiv preprint arXiv:2602.02474*, 2026a. URL https://arxiv.org/abs/2602.02474.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. Darwin Gödel Machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, 2026b. URL https://arxiv.org/abs/2505.22954.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, 2026c. URL https://arxiv.org/abs/2510.04618.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. Qwen3 Embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025b. URL https://arxiv.org/abs/2506.05176.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, et al. Externalization in llm agents: A unified review of memory, skills, protocols and harness engineering. *arXiv preprint arXiv:2604.08224*, 2026a.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. FeatureBench: Benchmarking agentic coding for complex feature development. In *International Conference on Learning Representations*, 2026b. URL https://arxiv.org/abs/2602.10975.

Yingli Zhou, Wang Shu, Yaodong Su, Wenchuan Du, Yixiang Fang, and Xuemin Lin. A comprehensive survey on agent skills: Taxonomy, techniques, and applications. *arXiv preprint arXiv:2605.07358*, 2026c.

A METHOD DETAILS

This section gives the complete workflow for Section 3, followed by implementation details for skill reuse and skill maintenance. The retrieval and maintenance mechanisms are described in the main text.

A.1 END-TO-END WORKFLOW

Retrieval and execution. Each skill version is stored in a `SKILL.md` file, optionally accompanied by scripts and supporting resources. For task description q in repository r, the framework constructs the scoped EHG $\mathcal{G}_t(r)$ using Equation 1. It selects the Top-N ($N = 3$) semantic anchors by the cosine similarity between embeddings of q and the routing descriptions $d(v)$ for $v \in V_t(r)$. The evolution neighborhood expansion and active replacement resolution in Algorithm 1 construct the candidate subgraph $\mathcal{C}_t$. The router receives the task description, repository identifier, and candidate subgraph, including the versions' routing descriptions, activity statuses, and typed relations, and returns the router-selected skills S_t. The coding agent receives their contents from the corresponding active nodes of $\mathcal{G}_t$ through its native skill interface and invokes them as needed while producing a patch. The framework records actions and observations as the execution trajectory τ_t, including skill invocation records identifying the versions actually invoked. Available test feedback f_t contains test outcomes and output logs.

Maintenance. Given the execution trajectory τ_t and test feedback f_t from the rollout, an experience extractor produces atomic experiences. The framework then reuses the evolution neighborhood expansion and active replacement resolution in Algorithm 1 with the router-selected skills S_t as starting points within the scoped EHG $\mathcal{G}_t(r)$, and augments the result with the predecessor versions collected by backward history traversal, producing the maintenance subgraph. The skill editor assesses each experience against the active editing targets and historical references in the maintenance subgraph, choosing *update*, *create*, or *skip* for each under repository scope constraints. After editing, the framework compares each edited copy with the active version it was copied from to identify the actual changes, adds a new active node for each changed copy and drafted version, marks each updated predecessor inactive and links it to its new version by a *revision* edge, and records *derivation* edges from the invoked skill versions I_t to the produced versions O_t. The output is the updated EHG $\mathcal{G}_{t+1}$, which serves as the sole persistent state for subsequent tasks. A separate consolidation pass periodically merges same-scope skills with essentially the same instructions (Appendix A.7).

A.2 SKILL RETRIEVAL AND EXECUTION INTERFACES

This subsection specifies how the skills and candidate subgraph described in Section 3.3 are supplied to the router and the coding agent. Each EHG node stores a skill folder containing its `SKILL.md` file and optional scripts and supporting resources. Semantic ranking uses the routing descriptions of versions in the scoped EHG, while the coding agent loads the folders of the selected active nodes through its native skill interface. The router receives the task description, repository identifier, and candidate subgraph with node routing descriptions, activity statuses, and typed relations. Historical versions provide context for selection but are not executable choices.

Duplicate handling. The anchor set, the expanded neighborhood, and the retained set in Algorithm 1 are sets, so a version reached from several anchors or along several paths, including an active version to which several inactive candidates resolve, appears only once in $\mathcal{C}_t$ and is exposed to the router as a single candidate.

Empty-input behavior. If the scoped EHG $\mathcal{G}_t(r)$ contains no versions, no semantic anchors are selected and $\mathcal{C}_t$ is empty. If $\mathcal{C}_t$ contains no active versions, or the router selects none of them, S_t is empty and the coding agent performs the rollout without additional skills.

A.3 SKILL RETRIEVAL ALGORITHM

Algorithm 1 gives the complete procedure of scope-constrained skill retrieval described in Section 3.3.

Algorithm 1 Scope-Constrained Skill Retrieval: Reuse

Require: EHG $\mathcal{G}_t$, task description q, repository r
Ensure: Expanded candidate subgraph $\mathcal{C}_t$ and router-selected skills S_t

Repository-Scoped Semantic Anchoring
$\mathcal{G}_r = (V_r, E_r) \leftarrow \mathcal{G}_t(r)$ ▷ Equation 1
Rank $v \in V_r$ by cosine similarity between embeddings of q and $d(v)$
$A_t \leftarrow$ Top-N ranked versions ▷ Active and historical versions are eligible
Evolution-Guided Candidate Expansion
$U \leftarrow A_t$ ▷ Evolution neighborhood expansion
for each anchor $u \in A_t$ **do**
  $P_u \leftarrow \{p \in V_r \mid \exists \ell : \langle p, \ell, u \rangle \in E_r\}$ ▷ One-hop parents
  $C_u \leftarrow \{c \in V_r \mid \exists \ell : \langle u, \ell, c \rangle \in E_r\}$ ▷ One-hop children
  $B_u \leftarrow \{s \in V_r \setminus \{u\} \mid \exists p \in P_u, \ell : \langle p, \ell, s \rangle \in E_r\}$ ▷ Siblings
  $U \leftarrow U \cup P_u \cup C_u \cup B_u$
end for
$L_{\text{rep}} \leftarrow \{\text{REVISION}, \text{CONSOLIDATION}\}$ ▷ Active replacement resolution
$W \leftarrow U, F \leftarrow U$ ▷ Retained versions and pending versions
while $F \neq \emptyset$ **do**
  Remove a version u from F
  if $a(u) =$ INACTIVE **then**
    $T \leftarrow \{v \in V_r \mid \exists \ell \in L_{\text{rep}} : \langle u, \ell, v \rangle \in E_r\}$
    $F \leftarrow F \cup (T \setminus W)$
    $W \leftarrow W \cup T$
  end if
end while
$\mathcal{C}_t \leftarrow \mathcal{G}_r[W]$ ▷ Retain visited versions and all induced edges
$S_t \leftarrow \text{Router}(q, r, \mathcal{C}_t)$ ▷ Select only active versions in $\mathcal{C}_t$
return $(\mathcal{C}_t, S_t)$

A.4 Skill Retrieval Example

Consider the skill versions in Figure 1, where G, $H1$, and $H2$ have general scope, $R1$, $R2$, M, and Q are scoped to repository r, and X belongs to another repository. Here, $R1$, $R2$, $H1$, and Q are inactive historical versions, while G, $H2$, and M are active. First, the scoped EHG $\mathcal{G}_t(r)$ excludes X before semantic ranking. Suppose the routing description of the inactive version $R1$ matches the task, giving the semantic anchor set $A_t = \{R1\}$. Second, evolution neighborhood expansion adds its parent G through the incoming *derivation* edge and its child $R2$ through the outgoing *revision* edge. It also adds the sibling $H1$ through $R1 \leftarrow G \rightarrow H1$, since $R1$ and $H1$ share the same source of invoked guidance. The resulting neighborhood is $U = \{G, R1, R2, H1\}$. Third, active replacement resolution follows $R1 \rightarrow R2 \rightarrow M$ through *revision* and *consolidation*, and $H1 \rightarrow H2$ through *revision*. These paths stop at the active versions M and $H2$, while the already active G remains a candidate. The other consolidation parent Q is not added, because reaching M does not trigger a backward search for its parents. Thus, the retained set is $W = \{G, R1, R2, H1, H2, M\}$, and $\mathcal{C}_t$ contains these versions and the edges among them.

A.5 Experience Extraction and Editing Interfaces

This subsection specifies the evidence and skill contents supplied to the extractor and editor in Section 3.4. The experience extractor operates in the repository that produced the rollout, with access to the execution trajectory, available test outcomes and output logs, and relevant repository code. It is instructed to extract reusable action sequences, recurring error patterns, API usage, and behavioral contracts. Each atomic experience must include references to supporting trajectory steps or repository code, allowing the editor to check its applicability against the original evidence.

The editor receives the atomic experiences and the maintenance subgraph, including the contents of active editing targets and read-only historical versions. The backward history traversal described in Section 3.4 can be implemented with a pending set and a visited set, recording each predecessor only once. The subgraph retains all edges between collected versions, including their *derivation* relations. The editor checks coverage against the active versions and its own edited copies as editing proceeds, so later decisions can account for changes already made during the same maintenance round. When

there are no selected skills and the maintenance subgraph is empty, supported experience can still produce a new skill.

For each atomic experience, the editor outputs one action: an *update* returns the edited copy of the target active version, a *create* returns the first version of a new skill, and a *skip* returns no version. Each returned version specifies its skill content $c(v)$, routing description $d(v)$, and repository scope $\sigma(v)$. For each update or creation, the editor determines this scope by checking whether the change depends on repository-local APIs, contracts, paths, or conventions. A generally applicable change may refine a GENERAL skill, whereas a repository-dependent change updates an r-scoped skill or, if no eligible target exists, creates one scoped to r.

A.6 Version Storage and Change Recording

This subsection specifies the bookkeeping used for history-preserving skill evolution in Section 3.4. The framework compares each edited copy with the active version it was copied from and records the actual changes rather than the editor's proposed actions, so an unchanged skill produces no new version. If a skill is edited several times within one maintenance round, its final contents form the new version for that round. For an updated skill, the *revision* edge connects its pre-edit version to its final new version. Each new version stores the skill content, routing description, repository scope, and activity status defined in Section 3.2, together with a version number used for bookkeeping. Replaced versions remain in the EHG as historical snapshots.

Skill invocation records in the trajectory identify the versions in I_t used for *derivation* edges. The router-selected set S_t is recorded separately because selection does not imply invocation. If no skill was invoked, I_t is empty, and no *derivation* edge is added, so a newly created skill starts as a root node.

A.7 Consolidation Candidate Construction

Every $\Delta = 100$ maintenance rounds, an independent consolidation pass constructs candidate groups from the EHG. For each node, we collect its current active replacement and those of its direct *derivation* children, following outgoing *revision* and *consolidation* edges until active nodes are reached. After deduplication, nodes with identical scopes form local comparison sets. For each active skill, we take the union of the other skills appearing with it in any local comparison set and nominate the one with the highest routing-description embedding cosine similarity. Skills with no comparison candidates are skipped. We treat nominations as undirected links and use their connected components as non-overlapping candidate groups.

For each candidate group, the consolidation agent operates within a history range consisting of the group's active skills together with all their predecessor versions collected by the backward history traversal of Section 3.4, which follows incoming *revision* and *consolidation* edges. It receives the complete contents of the versions in this range, where the predecessors serve only as read-only history, and may select multiple disjoint subsets of the group's active skills to merge, each containing at least two skills. For each subset, it checks whether the skills provide essentially the same instructions for the same symptoms and whether one replacement can retain the sources' specialized steps, constraints, and stopping conditions while keeping conditions that require different actions distinguishable. For an accepted merge, the replacement is added to the EHG as a new active node, and the selected source versions become inactive and are linked to it by the *consolidation* relations defined in the main text. Skills that are not selected or fail these checks remain unchanged.

B Benchmark Details

- **SWE-bench Verified** (Chowdhury et al., 2024) is a 500-instance subset of SWE-bench (Jimenez et al., 2024) for real-world GitHub issue resolution, from which human annotators removed instances with underspecified descriptions or overly specific test cases.
- **SWE-bench Pro** (Deng et al., 2025b) contains 731 instances from diverse repositories in multiple programming languages, reflecting realistic industrial scenarios with large-scale codebases and increased complexity. Beyond the largely localized bug fixes of SWE-bench Verified, its tasks also

cover feature implementation and refactoring, and typically require coordinated edits across several files.

C Baseline Details

- **mini-SWE-agent** (Yang et al., 2024) is a lightweight coding agent that uses only shell commands to inspect files, modify code, and execute tests, appending each action and observation to a linear interaction history. It is the officially recommended evaluation scaffold for both SWE-bench Verified and SWE-bench Pro.
- **Claude Code** (Anthropic, 2026) is a state-of-the-art commercial coding agent that combines repository exploration, file editing, and command execution to carry out software-engineering tasks.
- **Live-SWE-agent** (Xia et al., 2025b) enables online self-evolution: starting from a minimal bash-based agent, it adapts its own scaffold while solving an issue by creating and revising executable tools in response to environmental feedback, without a separate offline scaffold-optimization stage. In our continual evaluation, the tool scripts it saves are retained for subsequent tasks and are counted as its knowledge items in Section 5.
- **ReasoningBank** (Ouyang et al., 2026) distills reusable reasoning strategies from both successful and failed agent trajectories into structured memory items. It retrieves relevant items through embedding-based similarity to guide new tasks and incorporates lessons from completed trajectories into the memory bank for future reuse. In our experiments, it is built on mini-SWE-agent.
- **MemGovern** (Wang et al., 2026a) converts human debugging experience from GitHub issues, pull requests, and patches into structured experience cards that separate failure symptoms from repair knowledge. It enables agents to search for relevant cards and browse their root-cause analyses and repair strategies as the diagnosis develops.

D Implementation Details

D.1 EHG Initialization

SWE-smith (Yang et al., 2025) is a large-scale dataset of executable software-engineering task instances constructed from GitHub repositories for training coding agents. Before benchmark evaluation, for each backbone, we apply EvoGraph with that backbone to 1,000 instances randomly sampled from SWE-smith. The resulting execution trajectories and test feedback provide the experience from which skill maintenance extracts reusable skills. Because the source repositories are disjoint from those in SWE-bench Verified and SWE-bench Pro, this initialization stage does not access benchmark repositories. We then retain only the skills with GENERAL scope, whose applicability is repository-independent, and discard repository-scoped versions together with their incident relations. The retained skills keep their full evolution history, including their inactive historical versions and the *revision*, *derivation*, and *consolidation* relations among the retained versions, so that maintenance during evaluation can compare new experience with how these skills evolved during initialization. Each benchmark–backbone evaluation run starts from and updates only its own copy of the EHG initialized with the same backbone, so no skills or graph changes are shared across runs or backbones.

D.2 Model and Baseline Configuration

GPT-5 mini is accessed through its official API, while the open-source models, Qwen3.6-27B and DeepSeek-V4-Flash, are deployed with vLLM (Kwon et al., 2023) on eight NVIDIA H100 GPUs. In each run, all LLM-based components of EvoGraph, including the router, experience extractor, skill editor, and consolidation agent, use the same backbone as the coding agent. EvoGraph and ReasoningBank use Qwen3-Embedding-8B (Zhang et al., 2025b) for retrieval to ensure a fair comparison. All baselines use their default configurations, including their released initialization and memory protocols and hyperparameters. The SWE-smith initialization described above applies only to EvoGraph.

E Analysis Details

E.1 Supplementary Statistics

Table 3 reports the manual classification of the Live-SWE-agent scripts sampled in Section 5, and Table 4 reports the graph and maintenance statistics of EvoGraph on SWE-bench Pro.

Table 3: Classification of sampled Live-SWE-agent scripts. Operation type and task specificity are separate classifications of the full sample.

Script category	GPT-5 mini	DeepSeek-V4-Flash	Qwen3.6-27B	Total
Sampled scripts	50	50	50	150
Operation type				
Edit/view/search scripts	48	35	33	116
Task specificity				
With hard-coded task information	50	43	50	143
General, parameterized scripts	0	7	0	7

Table 4: EvoGraph graph and maintenance counts on SWE-bench Pro across three backbone models.

Statistic	GPT-5 mini	DeepSeek-V4-Flash	Qwen3.6-27B
Total nodes	721	747	644
Active skills	216	241	207
Revision edges	489	476	433
Skip actions	134	166	174

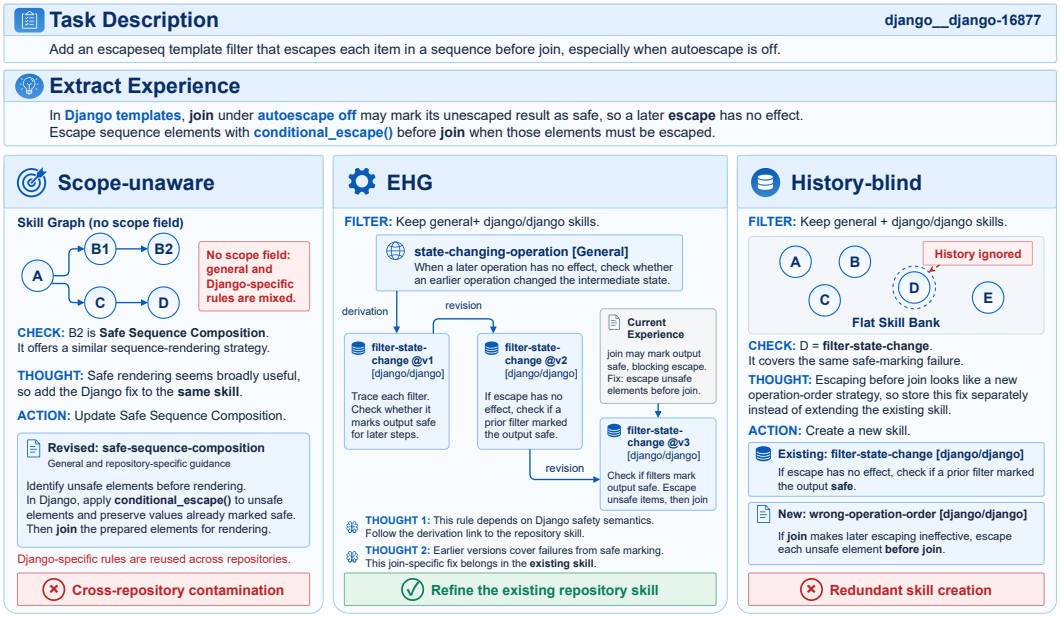

Figure 3: Maintenance decisions for the Django escaping case. Scope keeps local instructions within their applicable repository, while evolution history connects the new repair to an existing skill. Skill contents are condensed summaries.

E.2 Case Study

The task `django__django-16877` illustrates two decisions made when incorporating new experience: which repository the instructions apply to, and which existing skill they should extend. Figure 3

contrasts maintenance without repository scope, without evolution history, and with both. All three paths use Qwen3.6-27B on SWE-bench Verified. The central path is the maintenance performed by EvoGraph in the main experiment, and the left path comes from the *w/o Scope-Constrain* variant in the ablation study. The right path is a controlled replay of the central path: it reuses the same execution trajectory and the same EHG state, but withholds historical versions and evolution relations from the skill editor, isolating the effect of evolution history on the maintenance decision.

Task and extracted experience. The task requests an `escapeseq` filter for Django templates. A template filter transforms a value, and a chain of filters applies these transformations from left to right. The `join` filter combines a sequence into a string, while `escape` converts special HTML characters into escaped text. Django also tracks whether a value is marked as safe, meaning that later conditional escaping should leave it unchanged. This mark matters because it can change what a subsequent filter does even when the visible string remains the same.

Consider `values|join:", "|escape` with `autoescape off`. First, `join` combines the elements without escaping them and marks the combined string as safe. Then `escape` receives that marked string and leaves it unchanged, so the original elements are never escaped. The corrected chain, `values|escapeseq|join:", "`, processes each element before combining them. Here, `escapeseq` uses `conditional_escape()` to escape elements requiring escaping and preserve values already marked as safe. By the time `join` marks the combined result as safe, the required escaping has already occurred.

The reusable diagnostic lesson is to inspect earlier state changes when a later operation has no effect. The Django-specific lesson identifies the relevant state as safe marking and supplies the repair of escaping elements before `join`.

Existing skill lineage. The general skill `state-changing-operation` instructs the agent to trace earlier operations and check whether a state change explains an ineffective later operation. Its GENERAL scope (labeled `GLOBAL` in Figure 3) makes this diagnostic guidance available across repositories. A *derivation* relation from this general skill to version 1 of `filter-state-change`, scoped to `django/django`, records that version 1 was produced from the experience of an earlier rollout in which `state-changing-operation` was invoked. Version 1 directs the agent to inspect whether a filter marks its output as safe. A revision produces version 2, which directs this check toward filter chains in which later escaping has no effect.

Without repository scope, local instructions enter shared guidance. In the *w/o Scope-Constrain* variant, repository scope no longer restricts which skills are retrieved or edited, so general instructions are not kept apart from repository-specific ones. The editor selects *Safe Sequence Composition*, whose guidance concerns preparing elements before rendering and combining them. Because the new repair also concerns preparing and combining elements, the editor adds the use of `conditional_escape()` and the preservation of safe values to this skill. The updated skill now mixes a general rendering objective with instructions that depend on Django's API and safe-marking behavior, and because it remains available in other repositories, these local instructions can enter tasks whose libraries do not provide the same API or behavior. Assigning the repair to `django/django` instead keeps these instructions available for Django tasks while leaving general skills such as *Safe Sequence Composition* free of Django-specific instructions.

Without evolution history, the repair is stored separately from its diagnosis. In the replayed right path, the editor retains repository scope and sees the current `filter-state-change` skill, but not its earlier versions or evolution relations. The editor interprets the new instruction to escape before joining as a separate operation-order strategy. It therefore creates `wrong-operation-order` within the Django scope. The existing skill diagnoses how a preceding filter can mark its output safe and make later escaping ineffective, while the new skill repairs a `join`-induced occurrence of that same failure, so the diagnosis and its corrective action are now stored in separate skills with overlapping applicability.

With scope and history, the repair completes the existing skill. In the central path, repository scope identifies Django as the appropriate place for the repair. The *derivation* relation shows that the Django skill originated from a rollout guided by the general state-change diagnostic skill. Comparing the experience with the active version 2 and its predecessor version 1, the editor finds that the new repair continues the diagnosis these versions developed, so it treats the repair as a supported refinement of `filter-state-change` rather than an independently reusable capability and chooses *update*

instead of *create*. The resulting version 3 keeps the instruction to inspect safe marking when escaping has no effect and adds the action of escaping unsafe elements before `join`, so one skill now tells the agent what symptom to investigate, why it occurs, and how to repair it. Version 3 replaces version 2 as the active version, while the earlier versions remain as history.