

Trajectory-Guided Fault Localization for Agent Skill Evolution

Yu Ge¹, Linna Xie¹, Zhong Li^{1*}, Yu Pei², Tian Zhang^{1†}

¹ Nanjing University

² The Hong Kong Polytechnic University

Abstract

Agent skills provide reusable guidance for code agents, but incomplete or unsuitable guidance can impair task execution. To reduce the manual effort of skill refinement, recent approaches use LLMs to generate revisions from execution feedback. However, grounding these revisions in explicit behavioral evidence remains challenging. To address this gap, we propose SKILLMORPH, a skill-evolution approach based on trajectory-guided fault localization in agent skills. Its core idea is to link execution evidence to specific skill contents before generating revisions. Specifically, SKILLMORPH compares failure and success evidence in abstracted trajectories across repeated runs and tasks, incorporating changes between evolution loops to identify suspicious actions. It then uses these suspicious actions to localize edit sites in the skills and generate corresponding revisions. Experiments on SWE-Skills-Bench and CannBot show that the skills evolved by SKILLMORPH consistently achieve higher trial-level accuracy and execution consistency than the original skills and those from four existing skill-evolution methods. We have also applied SKILLMORPH to automated kernel generation with an AI operator-development team, which has accepted 6 skill-revision pull requests.

Keywords: Agent Skills, Skill Evolution, Fault Localization, Trajectory Analysis

1 Introduction

Agent Skills have become a common paradigm for supporting large language model (LLM)-based agents in domain-specific tasks such as software engineering [12], data analysis [15], and document processing [47]. Specifically, a skill packages reusable instructions, scripts, and reference materials for a specialized workflow. When carrying out a task, the agent loads the relevant skills and takes actions under their guidance, leveraging the experience encoded in those skills to solve the task more effectively. In practice, however, incomplete guidance or unsuitable workflows can lead the agent to take unsuitable actions, which limits skill effectiveness or even impairs task execution [10, 12]. Addressing these deficiencies generally relies on domain experts to inspect the agent’s actions, trace the problematic ones back to the skill content that guided them, and iteratively revise and evaluate the guidance [1, 47]. This manual process makes skill refinement costly and time-consuming, and difficult to scale as task and skill collections grow.

To reduce manual effort, several approaches have recently been proposed for *skill evolution*, which aims to improve agent performance by automatically refining skills based on execution feedback [44, 46, 1, 33, 23, 31, 48]. Most of these techniques follow an iterative process with two steps: (1) executing a set of evolution tasks using the current skill set to collect execution feedback, and (2) prompting an LLM with this feedback to produce a revision of the skill content. Despite the effectiveness of existing skill-evolution approaches, three challenges remain.

First, it is challenging to ground skill revision in explicit behavioral evidence. Existing approaches generally prompt an LLM to generate revisions directly from execution feedback, such as task outcomes, case-level scores, or whole trajectories [44, 17, 1, 10, 49]. Consequently, which behaviors are problematic is implicitly determined by the LLM as it reads the feedback and writes the revision, leaving no record for later inspection or comparison. This results in two consequences. On the one hand, without an

*Corresponding author.

†Emails for all authors: yuge@smail.nju.edu.cn, xieln@smail.nju.edu.cn, lizhong@nju.edu.cn, yupei@polyu.edu.hk, ztluck@nju.edu.cn.

explicit link between problematic behaviors and the skill content that needs correction, the revisions are likely to alter content that merely appears relevant to the observed problem while leaving the content requiring correction unchanged. Recently, ContractSkill [24] achieved explicit localization by representing a skill as an ordered sequence of steps with postconditions and using a deterministic verifier to identify the failing step. However, it assumes that the skill can be expressed in such a structured form and that a suitable verifier is available, which limits its general applicability. On the other hand, without explicit behavioral evidence, earlier revisions are difficult to reassess when a problem recurs. Several recent approaches retain historical records across evolution loops, but these records capture only scores or task-level status rather than linking targeted behaviors to edited regions [44, 17]. Such coarse records are insufficient to distinguish an incorrect edit location from an insufficient correction at the right location. As a result, later loops may repeatedly modify the same content without resolving the deficiency, or fail to notice a regression introduced by an earlier revision.

Second, it is challenging to distinguish deficiencies of the skills from incidental behaviors of the agent. A deficiency is a property of the skills themselves and is shared by every execution under them, whereas agent behavior is stochastic and tasks differ in their conditions. However, existing approaches typically attribute problems from limited evidence. For example, SkillAdaptor attributes a failed trajectory to its earliest actionable step [46], and other approaches prompt an LLM to reflect on a minibatch of rollouts [44, 17]. In both cases, the LLM has to judge which observed problems are real, without a measure of how consistently a behavior is associated with failures across independent runs. Consequently, revisions can fit individual cases without addressing deficiencies shared by other tasks of the same class, which limits the improvement on unseen tasks.

Third, existing approaches struggle to address deficiencies that span multiple skills. They generally revise only one skill or one document at a time. For example, SkillAdaptor revises only the highest-weighted skill identified by its attribution mechanism [46], and SkillOpt and SkillAdam edit a single skill document [44, 17]. In practice, however, a task often involves several skills whose contents jointly guide the agent’s behavior. Consequently, updating one artifact in isolation can leave erroneous or conflicting content elsewhere, allowing the original problem to persist or recur.

Our Work. In this work, we propose to evolve skills via *trajectory-guided fault localization*, which identifies the skill contents that require revision. The key idea is that before generating any revision, we explicitly determine the edit-sites, i.e., the skill contents that should be revised, and ground this decision in behavioral evidence from execution trajectories. This idea draws on program repair, where evidence from passing and failing tests guides the localization of suspicious code before candidate patches are generated [38, 36]. Analogously, we treat each agent execution on a task as a test execution and the actions in its trajectory as the entities covered by that execution. We localize through actions rather than treating skill contents directly as covered entities, because natural-language guidance lacks the explicit execution semantics of code. Specifically, an agent loads a skill document as a whole, and whether and how it follows each instruction is left to its own interpretation, thus there is no observable notion of which contents an execution exercises. In contrast, actions are directly observable in trajectories, differ across executions, and are shaped by the skill content. Therefore, a deficiency in a skill manifests as suspicious actions, and the trajectory records the guidance the agent consulted when taking them, allowing these actions to be traced back to the responsible skill content. By recording these actions, the edit sites derived from them, and the evidence behind both, the localization decisions become inspectable and can be reassessed when later executions reveal a persistent problem. An action also provides a common anchor for locating regions across multiple skills that jointly guide it, thus enabling coordinated revisions to those contents.

We realize this idea in SKILLMORPH. SKILLMORPH first abstracts raw trajectories into sequences of state transitions, making actions and their observed effects explicit units for inspection and comparison across executions. Using these abstractions, SKILLMORPH localizes edit sites in two stages before generating each revision. The first stage identifies suspicious actions, i.e., those more likely to reflect deficiencies in the skills than incidental agent behaviors. Our key insight is that reliable skill evolution requires both stability, by building on effective revisions across evolution loops to avoid recurring failures, and generality, by targeting deficiencies that persist across executions and tasks. Accordingly, SKILLMORPH progressively aggregates behavioral evidence across executions, evolution loops, and tasks. Specifically, it first compares repeated executions of the same task and contrasts how each action occurs in failing and passing executions, where an action that introduces an error that is later recovered from in a successful execution also counts as failure evidence. It then examines how the

evidence changes across evolution loops to reveal whether earlier revisions took effect, and finally aggregates the evidence across tasks to prioritize problems shared by multiple tasks. The second stage maps each suspicious action to edit sites in the skills. Our observation here is that although a trajectory does not directly reveal which instructions the agent followed, it records the skills and files consulted, narrowing the search for guidance that may have shaped an action. Based on this, an LLM-based agent uses the action’s execution context to progressively narrow the search from skills to files to specific skill content, and outputs the edit sites along with a rationale linking the skill content to the action. Finally, the edit sites, their localization rationale, and the supporting behavioral and historical evidence guide the generation of revision proposals, which are evaluated and consolidated into a patch that evolves the skill set.

Results. We evaluate SKILLMORPH on a repository-level software engineering skill benchmark SWE-Skills-Bench and a complex kernel-generation skill system CannBot. Compared with the original skills, SKILLMORPH achieves relative improvements of 15.84% in pass@3 and 30.03% in pass³ on SWE-Skills-Bench, and 16.67% and 100% on CannBot, respectively. SKILLMORPH also achieves better overall performance than the four compared skill-evolution methods. Moreover, in a deployment with our collaboration partner, SKILLMORPH produced six skill-revision pull requests that have been accepted for future AI operator development, further showing its practical value.

Summary. The main contributions of this paper are as follows:

- **Approach.** We propose SKILLMORPH, a skill-evolution approach based on trajectory-guided edit-site localization that links execution evidence to skill contents before generating revisions.
- **Evaluation.** We conduct an empirical study on two software engineering skill benchmarks, demonstrating the effectiveness of SKILLMORPH. In particular, SKILLMORPH has been deployed in an AI operator development team, further confirming its practicality.

2 Preliminaries

2.1 Code Agent with Skill

In this work, we focus on code agents [34, 43] that use skills for software engineering tasks and refer to the collection of skills available to an agent as its *skill set*. A skill set provides procedural knowledge and project-specific context for a class of related tasks without modifying the underlying model parameters [19, 42]. Following the Agent Skills specification [2], each skill is a file-based package with a `SKILL.md` document as its entry point, as illustrated in Figure 1. The `SKILL.md`’s front matter specifies the skill’s name and description, while its body provides natural-language instructions. The package may also contain supporting resources, such as reference documents, templates, and scripts, referenced by path. Throughout this paper, *skill* denotes the entire package, and *artifact* denotes an individual file within it.

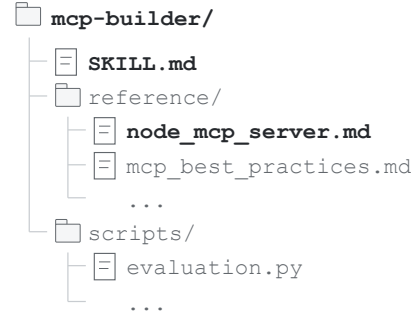


Figure 1: Structure of the `mcp-builder` skill.

Workflow of Code Agents. In general, a code agent accesses its skill set through progressive disclosure. At startup, only the name and description of each installed skill are included in the system prompt. When a task t arrives, the agent uses these descriptions to select relevant skills and loads their `SKILL.md` documents into context, and it reads supporting artifacts as needed during execution. At each step, the agent selects an action (such as reading a file, editing code, or running tests) based on the loaded instructions and accumulated context, executes it, and receives the resulting observation. This cycle continues until the task is completed or a resource limit is reached, yielding the execution trajectory

$$\tau = \langle \xi_1, \xi_2, \dots, \xi_T \rangle, \quad \xi_k = \langle a_k, o_k \rangle, \quad (1)$$

where a_k is an action (e.g., a tool invocation) and o_k is the observation obtained after executing it. A task-specific evaluation function Ω_t then maps the trajectory to an outcome $y = \Omega_t(\tau) \in \mathcal{Y}_t$, such as a pass/fail result or a structured record of functional correctness or performance.

As shown in the workflow, the skills affect the outcome through the actions that the agent selects under their guidance. As a result, a deficiency in the skill cannot be observed from its content alone since the agent implicitly decides whether and how to follow each instruction. Likewise, the final outcome summarizes an execution without necessarily identifying which actions caused a failure, and successful executions may include erroneous actions from which the agent later recovered. Therefore, a deficiency only manifests as problematic actions in trajectories, which tend to recur across the related tasks that the skill set serves. This makes actions the natural intermediate between outcomes and skill contents, and motivates evolving skills by first localizing suspicious actions and then tracing them back to the skill contents that guided them.

2.2 Skill Evolution Problem

Quality of a Skill Set. In this paper, we study how to iteratively evolve a skill set to improve the effectiveness of a code agent. Following this goal, we assess a skill set extrinsically, by the behavior it induces in the code agent that reads it. Specifically, let $\mathcal{A}$ denote a code agent equipped with a skill set $\mathcal{S}$, and let D denote a set of tasks of the class that $\mathcal{S}$ is intended to support. An execution of $\mathcal{A}$ on a task $t \in D$ under $\mathcal{S}$ produces a record (τ, y) , as stated in Equation 1, where τ is the execution trajectory and $y = \Omega_t(\tau)$ is the outcome of τ . Let $\mathcal{Y}_t^+ \subseteq \mathcal{Y}_t$ denote the outcomes in which task t is solved. The quality of $\mathcal{S}$ over D is then defined as the expected proportion of tasks solved,

$$J_D(\mathcal{S}) = \frac{1}{|D|} \sum_{t \in D} \mathbb{E}_{\tau \sim \mathcal{A}(t, \mathcal{S})} [\mathbf{1}\{\Omega_t(\tau) \in \mathcal{Y}_t^+\}]. \quad (2)$$

Skill Evolution. Given a code agent $\mathcal{A}$ whose model and harness are fixed, an initial skill set $\mathcal{S}_0$, an evolution task set D_e , and a loop budget L . In loop i , executing the tasks in D_e under $\mathcal{S}_i$ yields a set of execution records $\mathcal{E}_i = \{(\tau, y)\}$, from which the next skill set is produced. Skill evolution aims to construct this revision procedure so that the skill set obtained after L loops is as effective as possible on unseen tasks:

$$\max J_{D_v}(\mathcal{S}_L) \quad \text{s.t.} \quad \mathcal{S}_{i+1} = \text{Evolve}(\mathcal{S}_i, \mathcal{E}_i), \quad 0 \leq i < L, \quad (3)$$

where each $\mathcal{S}_{i+1}$ differs from $\mathcal{S}_i$ only in the contents of the artifacts, and D_v is a held-out set of tasks of the same class as D_e that is never accessed during evolution, i.e., $D_e \cap D_v = \emptyset$.

Equations 2 and 3 motivate three design considerations for the function `Evolve`. First, `Evolve` should distinguish persistent skill deficiencies from incidental agent behavior. Equation 2 defines J_D as an expectation over executions averaged across related tasks, thus a failure in one execution or task need not indicate a persistent deficiency. Moreover, `Evolve` observes executions only on D_e , whereas its objective is evaluated on D_v . Thus, revisions that fit incidental behavior on D_e can improve J_{D_e} without improving J_{D_v} , motivating the identification of deficiencies that persist across executions and tasks. Second, `Evolve` should retain and reuse historical evidence to preserve gains across evolution loops. Since $\mathcal{S}_L$, rather than the most effective intermediate skill set, is returned, a revision that disrupts earlier effective guidance leaves a regression carried to $\mathcal{S}_L$. Linking each revision to its motivating behavioral evidence and subsequent executions supports reassessing earlier edits and detecting these regressions. Third, `Evolve` should support coordinated revisions across multiple skills. A task may involve several skills whose contents jointly guide the agent’s actions. Therefore, producing an improved $\mathcal{S}_{i+1}$ may require revising multiple skills, since an isolated edit can leave erroneous or conflicting guidance elsewhere. Together, these considerations motivate revising the skill contents responsible for persistent deficiencies, potentially across multiple skills, using behavioral evidence accumulated across executions, evolution loops, and tasks.

3 SKILLMORPH in Action

We use the `mcp-builder` skill to show how SKILLMORPH turns failures on different tasks into one coordinated revision across skill artifacts. The skill guides agents in building Model Context Protocol (MCP) servers that expose application functions as tools; as Figure 1 shows, it consists of a workflow in `SKILL.md`, implementation references such as `reference/node_mcp_server.md`, and testing scripts.

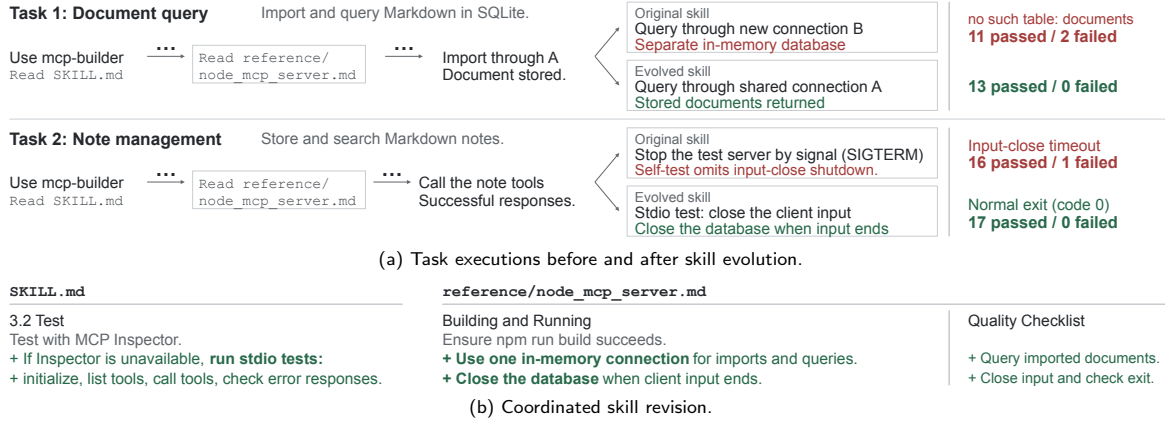


Figure 2: An example of SKILLMORPH evolving the mcp-builder skill.

For illustration, we use document query (Task 1) and note management (Task 2) for evolution, and hold out Task 3, which indexes Markdown documents for search and retrieval.

Failures under the original skill. Figure 2(a) compares key execution fragments and verification results before and after evolution. In Task 1, the agent imports a document through connection A but queries through a second in-memory connection B. Since an in-memory SQLite database is private to its connection, B cannot see the imported data, and verification fails with `no such table: documents` (11/13 checks pass). In Task 2, the agent’s self-test succeeds and ends by stopping the server with `SIGTERM`, never checking shutdown on input close. However, verification does: the server keeps running after the client’s input closes, and the check times out (16/17 checks pass).

Localization. In both tasks, the problematic action (opening connection B; stopping the server by `SIGTERM`) follows a read of `reference/node_mcp_server.md`, thus SKILLMORPH traces both to this artifact, whose *Building and Running* and *Quality Checklist* sections have no rule for sharing in-memory connections or cleaning up on input close. Additionally, the testing gap in Task 2 further leads SKILLMORPH to the *3.2 Test* section of `SKILL.md`, which recommends MCP Inspector but gives no fallback.

Coordinated revision. As Figure 2(b) shows, SKILLMORPH revises these three sections together: *Building and Running* gains two runtime rules (use one in-memory connection; close the database when input ends), *Quality Checklist* gains the matching checks, and *3.2 Test* gains a stdio testing procedure (initialize, list tools, call tools, check error responses) that keeps the checks runnable without Inspector. Revising either artifact alone would leave the guidance incomplete: the checks would lack a procedure to run them, or the procedure would lack rules and checks to apply.

Outcome. With the evolved skill, the agent queries through the shared connection A in Task 1 and closes the database when input ends in Task 2, passing all checks. To check that the revision repairs the guidance rather than fitting these two tasks, we evaluate on the held-out Task 3. Task 3 exhibits the same query and shutdown failures under the original skill; with the evolved skill, retrieval succeeds and the server terminates normally.

4 Approach

Figure 3 illustrates the workflow of SKILLMORPH. Given an initial skill set $\mathcal{S}_0$ and a task set D_e , SKILLMORPH evolves the skills over L loops, each comprising three stages. First, *Execution and Trajectory Abstraction* repeatedly executes the tasks with the current skill set and abstracts the resulting trajectories into a state-transition representation, enabling comparison across executions. Using these abstractions, *Suspicious Actions Localization* first compares the failure and success coverage of actions across repeated executions of each task. Then, it adjusts this evidence according to changes since the preceding loop, and finally aggregates it across tasks to score and select suspicious actions. These suspicious actions guide *Edit-Site Localization and Revision*, which locates the relevant skills, artifacts, and contents, generates and evaluates revision proposals, and applies a patch that consolidates the selected proposals to evolve the current skill set. In addition, an evolution memory preserves execu-

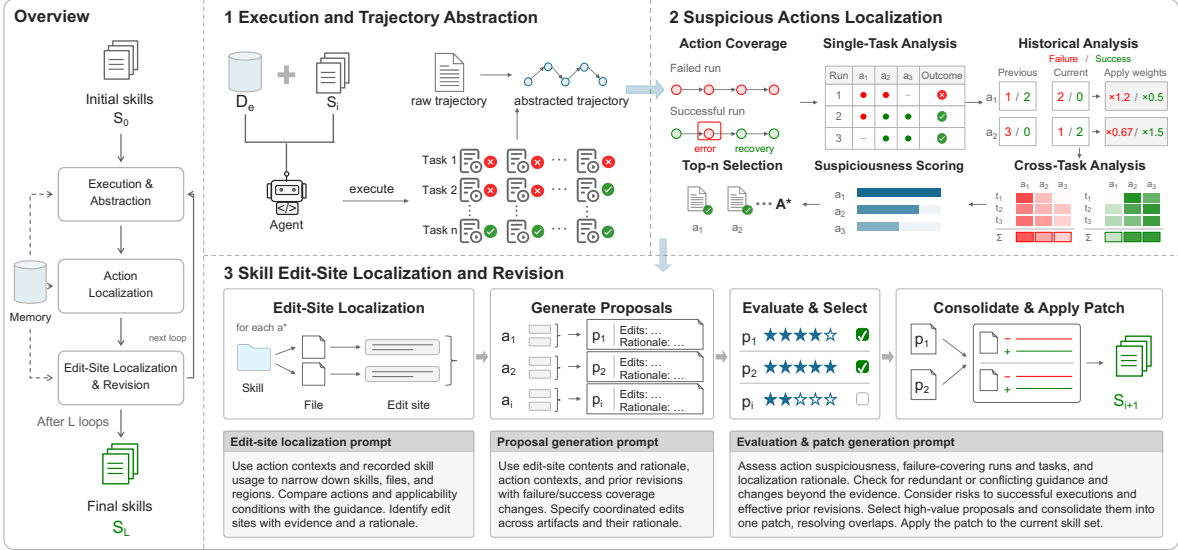


Figure 3: Overview of SKILLMORPH.

tion evidence, analyses, and revision history to support these analyses. After L loops, SKILLMORPH returns the resulting skill set S_L .

4.1 Execution and Trajectory Abstraction

This stage collects execution evidence under the current skill set to support subsequent edit-site localization and revision. SKILLMORPH first executes each evolution task multiple times and retains all runs, regardless of their outcomes. It then abstracts each raw trajectory into a uniform state-transition representation that captures actions and their observed effects, enabling analysis within individual runs and comparison across executions. We detail these two steps below.

Task Execution To obtain a set of execution trajectories, we independently run the code agent multiple times on each evolution task under the current skill set. Repeated execution is important to SKILLMORPH because agent behavior is stochastic: a single run cannot separate a deficiency induced by the skills from an incidental sampling artifact [22, 25, 51]. Formally, in evolution loop i , the fixed code agent $\mathcal{A}$ executes each task $t \in D_e$ independently m times under the current skill set S_i , producing $\mathcal{T}_i(t) = \{(\tau_{i,t}^{(j)}, y_{i,t}^{(j)})\}_{j=1}^m$, where $\tau_{i,t}^{(j)}$ and $y_{i,t}^{(j)}$ denote the trajectory and final outcome of the j -th run, respectively. Moreover, SKILLMORPH retains all trajectories, regardless of their outcomes, because successful and failed runs provide complementary evidence. A successful run may involve repeated unsuccessful attempts before recovery, revealing potential weaknesses in skill guidance that the final outcome alone does not expose. Conversely, a failed run may contain effective intermediate actions that remain useful despite the overall failure.

Trajectory Abstraction Given the collected execution trajectories $\mathcal{T}_i(t)$, a straightforward approach is to provide them directly to an LLM for failure attribution or revision generation, as done in prior work [17, 46, 1]. However, raw trajectories present two challenges for analysis. First, record boundaries do not necessarily correspond to meaningful behaviors. A single agent step may contain several tool calls whose responses must be examined together, while individual records may provide little information about the behavior being analyzed. The analysis therefore requires units that capture meaningful operations and their effects. Second, raw trajectories are lengthy and verbose, and similar behaviors may differ in wording, ordering, and number of steps across runs, making them difficult to compare.

To combat these two challenges, we abstract each raw trajectory into a uniform state-transition representation, drawing on prior work on action-state mapping and execution-log normalization [11, 18]. More specifically, for each trajectory $\tau_{i,t}^{(j)} \in \mathcal{T}_i(t)$, we follow prior work [11, 18] to use an LLM equipped with a hierarchical clustering algorithm to semantically parse and normalize the trajectory $\tau_{i,t}^{(j)}$ into an ordered sequence of transitions (μ_{k-1}, a_k, μ_k) . Here, a_k describes the action and affected object, while μ_{k-1} and μ_k describe the observed states before and after the action, respectively. This

yields the abstracted trajectory set $\hat{\mathcal{T}}_i(t) = \{(\hat{\tau}_{i,t}^{(j)}, y_{i,t}^{(j)})\}_{j=1}^m$, where $\hat{\tau}_{i,t}^{(j)} = \langle \mu_0, a_1, \mu_1, \dots, a_K, \mu_K \rangle$ and K denotes the number of abstracted operations. Finally, by aggregating all trajectories from different tasks, we obtain the execution trajectory set $\hat{\mathcal{T}}_i = \bigcup_{t \in D_e} \hat{\mathcal{T}}_i(t)$ for the current skill set $\mathcal{S}_i$.

4.2 Suspicious Actions Localization

This stage aims to identify the suspicious actions from the execution trajectory set $\hat{\mathcal{T}}_i$ that are more likely to reflect deficiencies in the current skill set $\mathcal{S}_i$. The key premise is that a deficiency is a property of the skills themselves, which every execution under the skill set shares, thus the actions it induces tend to recur across independent runs and tasks. However, routine actions, such as reading files or running tests, also recur during normal execution. Therefore, recurrence alone is insufficient; we must compare how actions occur in favorable and unfavorable execution contexts. This follows the intuition of spectrum-based fault localization (SBFL), which ranks program entities by comparing their coverage in failing and passing tests, namely entities consistently covered by failing tests but seldom by passing tests warrant closer inspection [38]. We adapt this principle by treating actions as the entities to be localized and individual runs as executions. Note that FAMAS adopts a related approach to failure attribution in multi-agent systems by comparing repeated executions of the same task [11]. However, skill evolution additionally requires identifying problems shared across tasks and tracking how behavioral evidence changes as skills are revised. Therefore, we adjust each task’s spectrum counts according to their changes since the preceding loop and aggregate the adjusted evidence across tasks.

For a task $t \in D_e$ in loop i , let $\hat{\mathcal{T}}_i(t)$ denote the set of its m runs, each comprising an abstracted trajectory $\hat{\tau}$ and its outcome y . We say that a run covers an action a if its trajectory contains at least one occurrence of a , i.e., $a_k = a$ for some step k . We then count the runs that provide failure and success coverage of a according to the following rules. If y indicates a failure, every action occurring in the run directly receives failure coverage, as a statement executed by a failing test does in SBFL. If y indicates a success, we further inspect whether an action introduced an error that the agent later recovered from during the execution. This inspection is necessary because LLM-based agents typically repair their own errors, so a successful run may still contain errors that an action introduced and the agent later recovered from. However, such recovery varies across runs: the same error that is repaired in one run may remain unresolved in another, causing the run to fail. Counting every action of a successful run as success coverage would therefore exonerate the very actions that cause failures elsewhere. Moreover, even when the recovery succeeds, an action that introduces an error still reveals a deficiency, since the skill should ideally guide the agent to avoid both the error and the cost of recovering from it. We thus inspect each occurrence of an action at step k in a successful run. Specifically, we adopt an LLM to examine the window of h steps before and after k , i.e., the states and actions from μ_{k-h-1} to μ_{k+h} , and thus to decide whether the action introduced an error state (an unresolved error or a failed validation) rather than merely revealing an existing one. If so, the occurrence contributes failure coverage; otherwise, it contributes success coverage. Note that this LLM-based judgment is deliberately narrow: it asks only whether the states around an action show an error that the action introduced, not which action caused a failed outcome, a question that requires reasoning over the whole trajectory and remains difficult even for strong LLMs [50]. Finally, each run contributes at most once to each coverage count of a , so that the resulting spectrum measures how many independent executions exhibit the action rather than how often it repeats within a single run.

Single-Task Analysis. We first measure the failure and success coverage of each action across the m runs of task t , reducing sensitivity to stochastic variation in agent execution. Within this comparison, a suspicious action should meet two conditions. First, an action should occur across runs providing failure evidence, giving stronger support than an isolated occurrence. Second, its failure coverage should be considered relative to its success coverage, helping distinguish problematic actions from routine behavior. Using only the number of failure-covering runs favors routine actions, such as reading files or running tests, which appear in nearly every run regardless of the outcome. Conversely, using only the share of failure coverage among all coverage of an action can overemphasize an action observed in a single failed run. This is analogous to the Ochiai form [27] widely used in SBFL. We therefore follow the Ochiai form to compute the spectrum matrix. Formally, let $\chi^-(\hat{\tau}, y; a)$ and $\chi^+(\hat{\tau}, y; a)$ indicate whether a run provides failure and success coverage of a , respectively. Let $\eta^-(\hat{\tau}, y)$

indicate whether the run provides failure coverage of at least one action. The task-level counts are:

$$N_{i,t}^\sigma(a) = \sum_{(\hat{\tau}, y) \in \widehat{\mathcal{T}}_i(t)} \chi^\sigma(\hat{\tau}, y; a), \quad F_{i,t} = \sum_{(\hat{\tau}, y) \in \widehat{\mathcal{T}}_i(t)} \eta^-(\hat{\tau}, y), \quad \sigma \in \{-, +\}. \quad (4)$$

Here, $N_{i,t}^-(a)$ and $N_{i,t}^+(a)$ count the runs providing failure and success coverage of a , respectively, while $F_{i,t}$ counts all runs providing failure coverage of at least one action.

Historical Analysis. The counts above describe how an action occurs in the current loop, but not how this evidence has evolved. The same counts can arise from a deficiency that the preceding revision has partly resolved, one that it failed to fix, or a regression that it introduced, yet these cases call for different priorities. To address this, we compare each task’s current coverage counts with those of the preceding loop and adjust them accordingly. The key insight is that, between two adjacent loops, only the skill set differs, owing to the revision made in the preceding loop. The change in coverage counts can thus be read as an effect of that revision. Specifically, we adjust failure and success coverages according to the direction of their change. For failure coverage, a decrease suggests that the revision has taken effect, so we downweight the current failure evidence to avoid repeatedly revisiting problems that may already be improving. An increase suggests that the revision was ineffective or misdirected, thus we upweight the evidence to highlight actions increasingly associated with failure. This includes newly observed actions, whose failure coverage rises from zero and may indicate a regression introduced by the revision. Success coverage serves as contrasting evidence in the suspiciousness calculation, and its change carries the reverse implication. An action whose success coverage grows may be an effective practice that the revision has promoted, thus we upweight its success evidence to shield it from suspicion, whereas a decrease weakens this contrasting evidence. When coverage remains unchanged or no comparable history is available, the adjustment is neutral, so a persistent deficiency retains its full weight while improving ones are downweighted. Together, these adjustments account for both the current evidence and its direction of change across evolution loops.

Formally, for each action a , we retrieve the preceding loop’s records of the same task from the evolution memory and compare the coverage counts:

$$r_{i,t}^\sigma(a) = \frac{N_{i,t}^\sigma(a) + 1}{N_{i-1,t}^\sigma(a) + 1}, \quad \sigma \in \{-, +\}. \quad (5)$$

Adding 1 to each count accommodates zero coverage and softens the ratio when the counts are small. Based on $r_{i,t}^\sigma(a)$, we then borrow the bounded transformation used in Yule’s Q [35] and shift it to obtain a multiplicative weight with a neutral value of 1:

$$w_{i,t}^\sigma(a) = 1 + \frac{r_{i,t}^\sigma(a) - 1}{r_{i,t}^\sigma(a) + 1} = \frac{2(N_{i,t}^\sigma(a) + 1)}{N_{i,t}^\sigma(a) + N_{i-1,t}^\sigma(a) + 2}. \quad \sigma \in \{-, +\}. \quad (6)$$

Finally, we separately adjust failure and success coverage counts obtained from single-task analysis:

$$\tilde{N}_{i,t}^-(a) = w_{i,t}^-(a)N_{i,t}^-(a), \quad \tilde{N}_{i,t}^+(a) = w_{i,t}^+(a)N_{i,t}^+(a). \quad (7)$$

Note that such adjusted coverage counts are used only for identifying suspicious actions in the current loop and are not accumulated across loops.

Cross-Task Analysis. As discussed in §2, a skill set typically supports a class of related tasks, indicating that their revisions should generalize to unseen tasks of the same type. However, the adjusted coverage counts $\tilde{N}_{i,t}^-(a)$ and $\tilde{N}_{i,t}^+(a)$ are specific to a single task t , and the actions they flag are likely bound to that task’s conditions, thus providing limited signals for skill evolution. Symmetrically, an action that stably appears in the successful runs of other tasks is harmless in itself, so its failures in this task more likely stem from the task’s context. Therefore, we aggregate the coverage counts after historical adjustment across tasks:

$$\bar{N}_i^-(a) = \sum_{t \in D_e} \tilde{N}_{i,t}^-(a), \quad \bar{N}_i^+(a) = \sum_{t \in D_e} \tilde{N}_{i,t}^+(a). \quad (8)$$

It is important to note that we keep the historical adjustment to each task before aggregation because aggregating first would mask the different trends across tasks, e.g., a decrease in failure coverage on

one task may offset an increase on another, leaving the total unchanged. In contrast, adjusting first instead lets each task’s change shape its own contribution to the combined evidence.

Suspicious Action Identification. Now we state how to identify suspicious actions based on the coverage counts. We keep every action with failure coverage in at least one task, i.e., $\sum_{t \in D_e} N_{i,t}^-(a) > 0$, as a candidate. For each candidate, we measure its suspicious score as follows:

$$s_i(a) = \underbrace{\frac{\overline{N}_i^-(a)}{F_i}}_{\text{coverage ratio}} \cdot \underbrace{\frac{\overline{N}_i^-(a)}{\overline{N}_i^-(a) + \overline{N}_i^+(a)}}_{\text{specificity ratio}} = \frac{\overline{N}_i^-(a)}{\sqrt{F_i (\overline{N}_i^-(a) + \overline{N}_i^+(a))}}, \quad F_i = \sum_{t \in D_e} F_{i,t}. \quad (9)$$

In Equation 9, the coverage ratio measures how widespread a is among the failure-covering runs across tasks, while the specificity ratio is the share of failure coverage in the total coverage of a , and measures how specific a is to failures. For example, an action that recurs in the failures of many tasks accumulates a large $\overline{N}_i^-(a)$, whereas one confined to a single task covers only a fraction of the failure-covering runs and receives a small coverage ratio. An action that also appears in the successful runs of other tasks accumulates a large $\overline{N}_i^+(a)$ and receives a low specificity ratio.

Finally, we rank all candidates by their suspicious scores and select the top- n actions as the suspicious actions, denoted as A^* for guiding the subsequent skill evolution.

4.3 Skill Edit-Site Localization and Revision

Based on the identified suspicious actions A^* , this stage first localizes the skill contents to be revised (i.e., edit sites) by linking each suspicious action to the skill content that guides it, and then generates revisions for these edit sites to evolve the skill set $\mathcal{S}_i$.

Edit-Site Localization. A skill set generally consists of natural-language instructions, together with reference documents, templates, and scripts spread over multiple skills and artifacts, which makes it hard to link an action to the contents that guide it by static rules. We address this challenge with an LLM-based agent that reads the skill set and maps each of the suspicious actions $a^* \in A^*$ to the edit sites in the current skill set.

For each a^* , the agent walks through the skill set $\mathcal{S}_i$ recursively, narrowing from skills to artifacts to contents, rather than processing it all at once, since the latter would bring irrelevant contents (e.g., skills unrelated to a^*) into its context and add noise to the localization. Specifically, it first examines the occurrences of a^* and their execution contexts in the trajectories retained during aggregation, alongside the skill descriptions and the recorded skill usage, to identify the skills that guide these actions. Then, it inspects the `SKILL.md` files of these skills and the artifacts they reference to find the instructions or script logic relevant to the actions. Next, it compares the prescribed actions and applicability conditions in these contents with the actions, revealing where the guidance prescribes unsuitable actions, omits necessary conditions, or conflicts with related contents, and accordingly these places are the edit sites. Finally, the agent outputs an edit-site for a^* , together with the supporting execution evidence and a rationale that links the skill contents to the action. In particular, an edit site specifies the set of skill contents to be revised, represented by their line ranges together with the corresponding skills and artifacts. Note that because a suspicious action may be associated with multiple skills or artifacts, a single edit site may span multiple contents across different skills or artifacts.

Revision Generation and Application Based on the identified edit-sites, SKILLMORPH next develops revision proposals and implements them as patches to evolve the skills. Specifically, we first prompt an LLM to generate a revision proposal for each edit site, specifying the intended revisions and their rationale. The proposal generation draws on three complementary sources of information. First, the edit site with its current contents and the localization rationale, which reveal what the existing skill contents prescribe and why they fail, and thus guide how to revise the site without introducing redundancy or conflicts. Second, the action a^* associated with the edit site and its execution contexts in the trajectories, which specify the problem that the revision should address. Third, the revision history, i.e., the earlier revisions to the relevant skills together with the changes in the failure and success coverage of a^* between adjacent loops (measured by Eq. 5), which show whether the earlier edits took effect, so that the new revision can retain, refine, or revert them instead of repeating ineffective changes. In addition, we explicitly specify the structure of each revision proposal, including the coordinated changes required across the involved skills or artifacts to maintain consistency.

The proposals of different edit sites can overlap, and not all of them are worth applying (e.g., some duplicate existing skill contents). Therefore, we evaluate the proposals and select a subset for developing the final patch, instead of applying them directly. Specifically, we design a chain-of-thought prompt that guides the LLM through three steps. First, the LLM evaluates the proposals in three aspects: (1) the evidence of the proposal’s action a^* , i.e., its suspiciousness score $s_i(a^*)$ and the runs and tasks in which it provides failure coverage, together with the localization rationale, which indicate how strongly the problem is established, how many failures the revision can address, and how certain the localization is; (2) the proposal and the current skill contents around its edit sites, which show whether the change duplicates or conflicts with the existing guidance and whether it extends beyond what the evidence supports; and (3) the successful cases and the revision history, i.e., the successful runs that rely on the edited contents and the observed effects of earlier revisions, which show whether the change may disrupt guidance supporting successful executions or undo an effective earlier revision. Based on the evaluation, the LLM then selects the high-value proposals. Finally, it consolidates the selected proposals into the final patch, resolving any overlap among them. Applying the patch evolves the current skill set $\mathcal{S}_i$ into $\mathcal{S}_{i+1}$, which is used in the next evolution loop or returned as the final skill set $\mathcal{S}_L$ when evolution completes.

5 Experiment

We evaluate SKILLMORPH on the following research questions:

- **RQ1:** How effective and efficient is SKILLMORPH in evolving agent skill sets?
- **RQ2:** How do skills evolved by SKILLMORPH compare in effectiveness with the original skills and those evolved by existing methods?
- **RQ3:** How do the configurations and key components of SKILLMORPH affect its effectiveness?

5.1 Experimental Setup

Benchmarks. We evaluate SKILLMORPH on skill sets that vary in structure and complexity: independent skills from SWE-Skills-Bench [12] and the coordinated multi-skill system of CannBot.

- *SWE-Skills-Bench.* It comprises 49 independent skills for software engineering, each paired with ten different task batches grounded in a real-world GitHub repository pinned to a fixed commit.
- *CannBot.* It comprises 12 coordinated skills for complex AscendC kernel generation in the CANN ecosystem [20]. We evaluate CannBot on tasks drawn from NPUIKernelBench [4].

Baselines. The baselines include the original skills and four existing skill-evolution methods.

- *Original Skills.* We use the carefully curated public skills from SWE-Skills-Bench and the official CannBot skill system designed for AscendC kernel generation as the unevolved baselines.
- *Skill-Evolution Approaches.* We select comparison methods based on three criteria: (1) Their source code is publicly available, enabling reproduction and adaptation. (2) They demonstrate strong performance in their original evaluation settings. (3) Their skill-evolution workflows can be adapted to our benchmarks by integrating the corresponding task-execution and verification procedures. Based on these criteria, we select four representative skill-evolution approaches. *SkillOpt* [44] directly optimizes a single skill document through rollout-guided textual edits and validation gating. *SkillAdaptor* [46] first localizes an actionable fault in a failed trajectory, then revises a responsible skill or generates a new one when skill coverage is missing. *EvoSkill* [1] evolves complete structured skill folders through failure-driven mutations and Pareto-frontier selection based on validation performance. *CAP-Evolve* [6] evolves agent prompts, skill packages, and tools through an iterative workflow of failure diagnosis, candidate generation, and validation-based selection.

Metrics. We use three metrics to evaluate effectiveness. A trial is considered successful only if the agent completes the task within the time limit and passes all tests or verification checks provided by the benchmark. Following prior work on code generation and agent evaluation [8, 45], we use *pass@3* and *pass*³ to measure the proportions of tasks solved in at least one of three independent trials and in all three trials, respectively. In addition, we use *trial-level accuracy* to measure the proportion of successful individual trials.

Implementation. We implement SKILLMORPH in Python 3.11 following its three-stage workflow. We use Claude Code [3] with DeepSeek-V4-Flash-0731 [9] as the code agent for task execution using skills, and the same model throughout evolution. To reduce wall-clock time, we parallelize task-level

Table 1: Pass@3, trial-level accuracy, and Pass³ (%) of the original skills, skill-evolution baselines, and SKILLMORPH on SWE-Skills-Bench and CannBot.

Method	SWE: A→B			SWE: B→A			SWE: Overall			CannBot		
	Pass@3	Trial Acc.	Pass ³	Pass@3	Trial Acc.	Pass ³	Pass@3	Trial Acc.	Pass ³	Pass@3	Trial Acc.	Pass ³
Base Skills	84.49	77.69	71.02	80.41	73.20	64.90	82.45	75.44	67.96	80.00	66.67	46.67
SkillOpt	93.47	89.93	85.71	84.49	79.86	75.10	88.98	84.90	80.41	80.00	71.11	53.33
EvoSkill	87.76	83.27	78.37	83.67	77.55	70.61	85.71	80.41	74.49	86.67	73.33	53.33
SkillAdaptor	87.76	83.95	80.41	85.71	80.14	73.88	86.73	82.04	77.14	86.67	71.11	60.00
CAP-Evolve	96.33	93.61	89.39	88.16	85.17	80.41	92.24	89.39	84.90	93.33	84.44	73.33
SKILLMORPH	97.55	95.24	92.65	93.47	89.25	84.08	95.51	92.24	88.37	93.33	93.33	93.33

analyses across independent tasks. Unless otherwise stated, each evolution task is independently executed $m = 3$ times per loop, with $L = 3$ loops for SWE-Skills-Bench and $L = 5$ for CannBot. Task execution follows SWE-Skills-Bench’s timeout settings, while CannBot uses a 9,000-second limit. Each loop selects the top-10 suspicious actions as inputs to edit-site localization.

To assess generalization to tasks unseen during evolution, the tasks are divided into an evolution set D_e , which provides execution evidence for skill evolution, and a held-out evaluation set D_v , which is never accessed during evolution. For each skill in SWE-Skills-Bench, we partition its ten tasks into two five-task folds, A and B, containing tasks with odd- and even-numbered original IDs, respectively. We conduct two complementary experiments, A→B and B→A, by swapping the folds’ roles as D_e and D_v , with both experiments starting from the same original skill. Because CannBot incurs substantially higher execution costs, we use a single fixed split with 15 tasks for evolution and 15 for held-out evaluation. For each set, an LLM selects 10 Level-1 and 5 Level-2 tasks from NPUKernelBench, prioritizing task diversity.

We reproduce the compared skill-evolution methods using their official implementations, with the same settings as SKILLMORPH: (1) code agent and task split; (2) execution results from the original skills for the first evolution loop; and (3) numbers of executions per task and evolution loops. We adapt their execution and verification interfaces while retaining their evolution workflows.

Environment. The SWE-Skills-Bench experiments run on an Ubuntu 22.04.4 machine with an Intel Xeon Gold 6426Y CPU and 512 GB RAM. The CannBot experiments run on a machine with eight Ascend 910B3 NPUs and 1.5 TB RAM, using CANN 9.1.0.

5.2 RQ1: Overall Performance of SKILLMORPH

Effectiveness. As shown in the last row of Table 1, SKILLMORPH achieves 95.51% (=468/490) pass@3 and 88.37% (=433/490) pass³ on the 490 SWE-Skills-Bench tasks. Across the 1,470 evaluation runs on these tasks, it achieves 92.24% (=1356/1470) trial-level accuracy. On CannBot, it succeeds on 14 of 15 tasks in every trial, yielding 93.33% on all three metrics. These results indicate that SKILLMORPH can evolve skills that enable the code agent to achieve high success rates on held-out tasks and execute reliably across repeated trials.

Efficiency. We examine the average time spent in an evolution loop. Each loop involves task execution by the code agent, as well as analysis and revision by the SKILLMORPH framework. Specifically, the latter includes trajectory abstraction, suspicious actions localization, and skill edit-site localization and revision. SKILLMORPH takes approximately 49 minutes per loop for a SWE-Skills-Bench skill and 127 minutes for CannBot. In particular, code-agent task execution takes 19.8 and 97.8 minutes, respectively. The three framework components take 3.1, 7.4, and 18.3 minutes on SWE-Skills-Bench, respectively, while the corresponding times on CannBot are 4.4, 5.6, and 19.0 minutes. Although code-agent execution takes nearly five times as long on CannBot as on SWE-Skills-Bench, the combined time spent on framework analysis and revision remains similar across the two benchmarks. Consequently, code-agent execution accounts for most of the difference in evolution time between the two benchmarks.

Generalizability. SKILLMORPH remains effective not only across different task splits within SWE-Skills-Bench but also across skill systems that differ in structure, scale, and task domain. Under the two complementary task splits described in Section 5.1, A→B achieves 97.55% (=239/245) pass@3, 95.24% (=700/735) trial-level accuracy, and 92.65% (=227/245) pass³, while B→A achieves 93.47% (=229/245), 89.25% (=656/735), and 84.08% (=206/245), respectively. Although performance differs

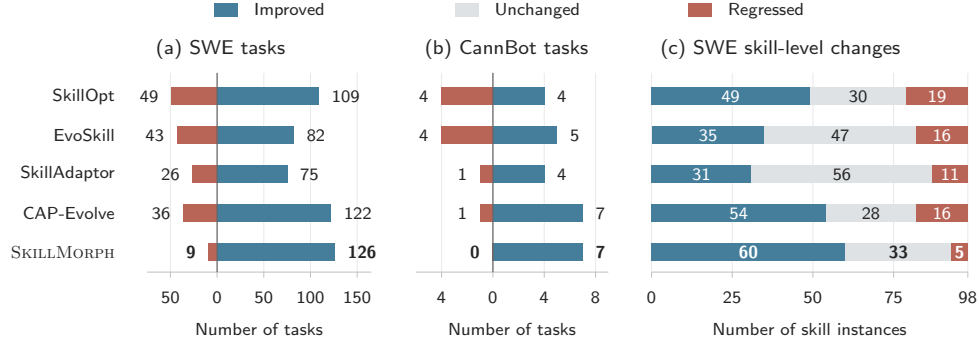


Figure 4: Fine-grained performance changes relative to the original skills.

between the two split directions, both achieve high held-out scores across all three metrics, indicating that the evolved skills generalize to tasks unseen during evolution. SWE-Skills-Bench evaluates compact, independent skills for repository-level software engineering, whereas CannBot involves larger, coordinated skills for AscendC kernel generation. The high scores across all three metrics on CannBot show that the same evolution workflow remains effective with more extensive guidance distributed across multiple files and skills.

Answer to RQ1. SKILLMORPH achieves high held-out performance and reliable execution under both SWE-Skills-Bench split directions and on CannBot’s larger, coordinated skill system. Its framework processing time remains comparable across the two benchmarks, with CannBot’s longer evolution time mainly attributable to code-agent task execution.

5.3 RQ2: Comparison against Baselines

Overall Effectiveness Comparison. Table 1 also compares the effectiveness of SKILLMORPH with that of the original skills and four existing skill-evolution methods. The results show that SKILLMORPH achieves the highest pass@3, trial-level accuracy, and pass³ on both benchmarks.

Compared with the original skills, SKILLMORPH increases pass@3 from 82.45% to 95.51% on SWE-Skills-Bench and from 80.00% to 93.33% on CannBot, corresponding to relative improvements of 15.84% and 16.67%, respectively. For trial-level accuracy, relative gains of 22.27% and 40.00% raise the scores from 75.44% to 92.24% and from 66.67% to 93.33%, respectively. More notably, pass³ increases from 67.96% to 88.37% on SWE-Skills-Bench, a 30.03% relative gain, and doubles from 46.67% to 93.33% on CannBot. These results show that all three metrics improve, with the largest relative gains in pass³ indicating more consistent success with the evolved skills.

Against existing methods on SWE-Skills-Bench, SKILLMORPH solves 16 more tasks than CAP-Evolve, the strongest baseline, and 48 more than EvoSkill, corresponding to pass@3 gains of 3.27 and 9.80 percentage points. More notably, the gain is larger for consistently solved tasks in every baseline comparison, ranging from 17 to 68 tasks (3.47–13.88 points in pass³). Consistent with these gains, SKILLMORPH records 42–174 additional successful trials out of 1,470, raising trial-level accuracy by 2.86–11.84 percentage points. The advantage in execution consistency is particularly clear on CannBot. Although CAP-Evolve matches SKILLMORPH in pass@3 (14/15), SKILLMORPH solves all 14 tasks in every trial, compared with 8–11 among the four baselines, giving a pass³ advantage of 20.00–40.00 percentage points. This greater consistency is also reflected at the trial level, where SKILLMORPH records 4–10 additional successes out of 45 trials and raises accuracy by 8.89–22.22 percentage points. These results indicate that SKILLMORPH’s advantage extends from covering additional tasks to supporting more consistent success across repeated executions.

Fine-Grained Comparison. We further compare improvements and regressions at the task and skill levels relative to the original skills to understand how the overall gains are distributed. At the task level, we classify a task as improved, unchanged, or regressed according to whether its number of successful trials out of three increases, remains the same, or decreases. At the skill level, we use the same categories, comparing pass@3 first and using trial-level accuracy to break ties. Figure 4 presents task-level changes on both benchmarks and skill-level changes on SWE-Skills-Bench.

On SWE-Skills-Bench, SKILLMORPH improves a similar number of tasks to CAP-Evolve (126 versus

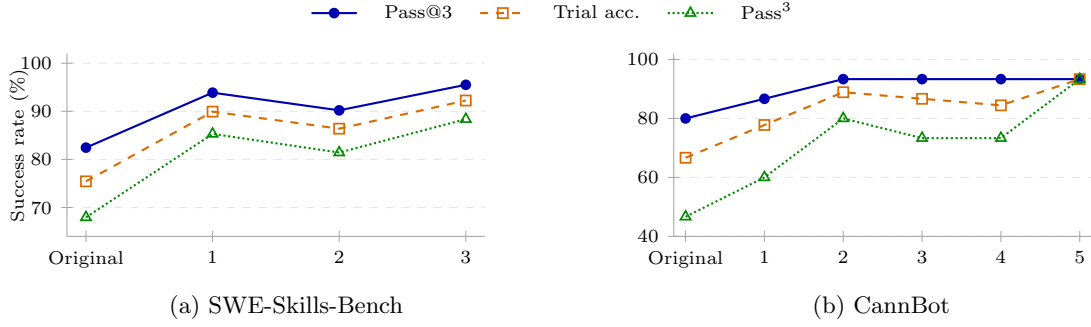


Figure 5: Effect of the Number of Evolution Loops on SKILLMORPH’s Performance.

122), but introduces substantially fewer regressions (9 versus 36). Specifically, among the 333 originally consistently solved tasks, SKILLMORPH retains consistent success on 327 (98.20%), compared with 309 (92.79%) under CAP-Evolve. This advantage extends to the other three baselines, which improve 75–109 tasks and regress on 26–49. These comparisons indicate that SKILLMORPH’s advantage reflects improvements on weaker tasks and better preservation of reliable behavior.

On CannBot, SKILLMORPH improves task performance while better preserving existing reliable behavior. Across the 15 tasks, it improves 7 without regression, whereas the four baselines improve 4–7 and regress on 1–4. Specifically, all 5 tasks initially solved inconsistently now succeed in every trial under SKILLMORPH, compared with 3–4 under the baselines. SKILLMORPH also retains consistent success on all 7 originally consistently solved tasks, whereas the baselines retain only 4–6. Together, these results highlight SKILLMORPH’s advantage in improving execution consistency while preserving existing capabilities in this more complex, coordinated skill setting.

Beyond individual tasks, we examine how these changes are distributed across the 98 skill instances formed by 49 SWE-Skills-Bench skills under the two complementary task splits. SKILLMORPH improves 60 instances, leaves 33 unchanged, and regresses on 5, whereas the four baselines improve 31–54 instances and regress on 11–19. Thus, the advantage observed at the task level also extends across more skill instances, with fewer regressions than under the existing methods.

Execution Time Comparison. We further compare the average agent execution time to examine whether the effectiveness gains come with increased execution costs. On SWE-Skills-Bench, the average time remains nearly unchanged, decreasing from 16.45 minutes with the original skills to 16.35 minutes with SKILLMORPH. On CannBot, however, SKILLMORPH reduces the average execution time from 141.55 to 87.07 minutes, a reduction of 38.49%, compared with 110.19–143.26 minutes under the four baselines. These results suggest that skill evolution can improve execution reliability without increasing average execution time, while also yielding substantial time savings in CannBot’s more complex, coordinated skill system.

Answer to RQ2. SKILLMORPH outperforms the original skills and four existing skill-evolution methods in overall effectiveness, improving more skill instances while introducing fewer regressions. These gains come with nearly unchanged execution time on SWE-Skills-Bench and substantial time savings on CannBot.

5.4 RQ3: Configurations and Components of SKILLMORPH

In this section, we evaluate how the number of evolution loops affects SKILLMORPH’s effectiveness and conduct an ablation study to assess the contributions of its key components.

Effect of Evolution Loops. We examine the original skills and skills produced after each of the three SWE-Skills-Bench loops and five CannBot loops. Each version is evaluated on the same held-out tasks under the protocol described in Section 5.1. As shown in Figure 5, the first loop produces large gains in all three metrics on both benchmarks. Although some scores decline in intermediate loops, all three metrics reach their highest values in the final loop. On SWE-Skills-Bench, the first loop raises pass³ from 67.96% to 85.31% and accounts for 85–88% of the final gain across the three metrics. From the first to the third loop, pass³ increases by 3.06 percentage points, compared with 1.63 points for pass@3, despite a decline in the second loop. On CannBot, the first loop increases pass@3 from 12 to 13 of 15 tasks and pass³ from 7 to 9. Pass@3 reaches 14 tasks by the second loop and remains there,

Table 2: Performance of SKILLMORPH variants with different component configurations (%).

Variant	SWE Overall			CannBot		
	Pass@3	Trial	Pass ³	Pass@3	Trial	Pass ³
Single-run	90.20	87.69	84.90	80.00	77.78	73.33
w/o History	93.88	89.93	85.31	93.33	88.89	80.00
w/o Localization	91.22	87.35	82.24	86.67	75.56	60.00
Full SKILLMORPH	95.51	92.24	88.37	93.33	93.33	93.33

whereas pass³ falls from 12 to 11 in the intermediate loops before reaching 14 in the fifth. Together, these results show that early loops produce most of the gains in task coverage, while later loops can improve execution consistency despite intermediate declines.

Effect of Components. We conduct an ablation study to assess the contributions of repeated execution, the use of history, and skill edit-site localization. We construct three variants of SKILLMORPH: *Single-run* executes each evolution task once per loop ($m = 1$) while retaining the same number of loops; *w/o History* omits the historical adjustment of action coverage counts and excludes historical information from proposal generation and evaluation; and *w/o Localization* skips Skill Edit-Site Localization and generates revisions directly from the selected suspicious actions and their execution contexts. Table 2 reports the performance of SKILLMORPH variants with different component configurations on both benchmarks, measured by pass@3, trial-level accuracy, and pass³. Single-run has the largest effect on pass@3, lowering it by 5.31 (=26/490) and 13.33 (=2/15) percentage points on SWE-Skills-Bench and CannBot, respectively. Removing localization has the largest effect on pass³, lowering it by 6.12 (=30/490) and 33.33 (=5/15) points, respectively. The w/o History variant has the smallest effect on both benchmarks, with CannBot pass@3 unchanged but lower trial-level accuracy and pass³. These results suggest that repeated execution contributes most clearly to task coverage, while explicit localization is particularly important for consistent success.

Answer to RQ3. (1) The first loop yields large gains, while later loops improve execution consistency, particularly after CannBot’s pass@3 levels off. (2) Single-run has the largest effect on pass@3, w/o Localization on pass³, and w/o History the smallest effect.

6 Practical Application

We apply SKILLMORPH in collaboration with a team that develops AI operators to evolve the skills used in automated kernel generation. The evaluation covers 63 AI operators of varying complexity, with test cases spanning multiple data types and tensor shapes. Execution evidence from these tasks guides skill revisions addressing numerical correctness, kernel performance, and the AI operators’ development and verification workflow. We have submitted these skill revisions in 6 pull requests, all of which have been accepted by the developers. The collaboration is ongoing, and the team continues to use and refine the revised skills.

For tasks that pass correctness checks with both the original and evolved skills, the average completion time decreases from 123.63 to 72.81 minutes, a reduction of 41.11%. These savings show that refining workflow guidance to reduce unnecessary operations, such as unfocused searches, can improve efficiency even for tasks that the original skills already solve.

For kernels that pass correctness checks, we measure speedup over the official reference implementation using device-side execution times. Across 1,788 case-level measurements, 64.26% and 54.14% achieve speedups of at least 0.6 $\times$ and 0.8 $\times$, respectively. In 44.18% of these cases, the generated kernels match or exceed the reference performance.

7 Discussion

Effect of the Number of Selected Actions The preceding experiments show that SKILLMORPH achieves strong performance when selecting the top-10 suspicious actions in each loop. To examine how performance depends on this number, we conduct evolution experiments on SWE-Skills-Bench

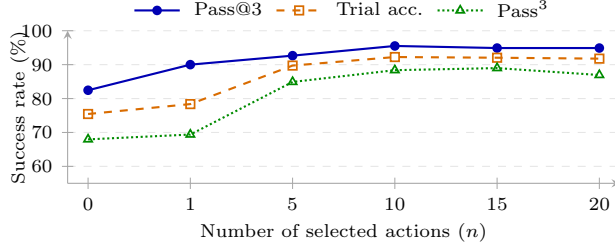


Figure 6: Effect of top- n action selection.

and vary n among 1, 5, 10, 15, and 20. All other experimental settings remain unchanged. Figure 6 presents the results, where $n = 0$ denotes the original skills before evolution.

At $n = 1$, pass@3 reaches 90.00%, but pass³ is only 69.39%, indicating high task coverage with limited execution consistency. As n increases to 5 and 10, all three metrics improve, with pass³ rising to 84.90% and 88.37%, respectively, narrowing the gap with pass@3. Manual inspection shows that revisions guided by the additional actions address skill deficiencies affecting execution reliability, improving consistency across trials. Performance remains relatively stable as n further increases to 15 and 20. This pattern suggests that selecting too few actions may leave important skill deficiencies affecting execution consistency unaddressed, whereas additional actions beyond a certain selection size yield limited further gains.

Lessons Learned Based on SKILLMORPH, we derive the following implications.

- *Skill Authoring.* We observe that instructions to test thoroughly can still leave verification blocked after the requested code changes are completed. In these cases, the skill lacks concrete guidance on dependency handling, verification environments, or re-verification after repairs. This suggests that skill authors should specify the applicability conditions, required execution order, and completion criteria of key operations, using actual trajectories to identify gaps in the guidance. This observation aligns with prior work emphasizing procedural guidance and verification feedback [14]. When such guidance spans multiple files, the main instructions should clearly reference the relevant materials, and related rules should remain consistent across documents and scripts.

- *Task-Specific Adaptation.* Experience accumulated through cross-task evolution may still leave difficulties specific to the current task unresolved. In our bidirectional cross-validation, 21 tasks failed in all three evaluation runs with both the original skill and the skill evolved using the opposite task fold. We examined the evolution records in which these tasks themselves belonged to the task set used for evolution. Relative to their initial executions in those records, five tasks subsequently succeeded in at least one run, while four others showed improvements in individual verification checks. These observations suggest that task-specific feedback may complement experience accumulated across tasks. Reflexion [30] uses feedback-driven reflection to guide subsequent attempts, while Voyager [32] stores verified programs for reuse. Building on these mechanisms, future work could explore combining task-level adaptation with SKILLMORPH to address immediate blockers and turn verified recovery operations and their applicability conditions into reusable skill guidance.

Threats to Validity The main threat to *internal* validity arises from stochastic execution and the adaptation of the compared approaches. To reduce this threat, we use their official implementations under the common experimental protocol described in Section 5.1 and evaluate each task in three independent trials. The main threat to *external* validity lies in the selection of skill systems and tasks. To mitigate this threat, we evaluate SKILLMORPH on held-out tasks from SWE-Skills-Bench and CannBot, which differ in task domain and skill structure. We also swap the tasks used for evolution and evaluation on SWE-Skills-Bench to reduce dependence on a particular task split. The main threat to *construct* validity lies in the metrics used to assess skill quality. To reduce this threat, we use pass@3, trial-level accuracy, and pass³ to provide complementary measures of task completion and consistency across repeated executions.

8 Related Work

Agent Skill Construction and Evolution. Agent skills package reusable procedural knowledge, while their utility depends on the compatibility among the skill, task, and execution context [19, 12]. Existing methods for improving agent skills can be broadly divided into skill construction and skill evolution. Skill construction derives new artifacts from execution experience, such as executable libraries, consolidated trajectory lessons, or skills authored by a meta-skill [32, 26, 48]. Skill evolution instead revises existing artifacts from execution feedback, either through optimization loops based on failure analysis, generated evaluation tasks, or validation-gated edits [1, 31, 44], or by diagnosing defects from trajectories [23, 46, 33]. SKILLMORPH links suspicious-action analysis across tasks and evolution loops to explicit edit-site localization and coordinated skill revisions.

Agent Trajectory Analysis and Utilization. Beyond skill evolution, agent execution trajectories have also been widely studied for behavior analysis [5, 29], failure diagnosis [7, 50, 11], and context management [21, 28, 39]. For example, Bouzenia and Pradel [5] characterize the behavioral differences between successful and unsuccessful executions of software engineering agents. FAMAS [11] adapts spectrum-based fault localization to multi-agent systems, contrasting successful and failed executions to attribute failures to the responsible agent actions. AgentDiet [39] removes redundant and expired information to reduce coding-agent inference cost. Unlike these works, which analyze or manage trajectories without revising persistent artifacts, SKILLMORPH links behavioral evidence to editable skill artifacts and tracks their revisions across evolution loops.

Feedback-Driven Software Improvement. Using execution feedback to improve software has a long history [16, 13, 38], among which automated program repair (APR) is a representative line of research [36, 37, 41, 40]. APR leverages tests throughout the repair loop: fault localization contrasts passing and failing executions to rank suspicious program entities [38], and candidate patches are searched for and validated against test results, as exemplified by GenProg [36]. Treating the skills as evolvable software, SKILLMORPH follows this localize-then-revise paradigm, using spectrum-guided execution feedback to localize and revise deficient skill artifacts across evolution loops.

9 Conclusion

In this paper, we propose SKILLMORPH, a skill-evolution approach for LLM-based agents based on trajectory-guided edit-site localization. SKILLMORPH explicitly determines the edit sites, i.e., the skill contents that should be revised, before generating any revision, and grounds this decision in behavioral evidence from execution trajectories. In particular, SKILLMORPH abstracts raw trajectories into sequences of state transitions, progressively aggregates evidence across repeated executions, evolution loops, and tasks to identify suspicious actions, and then traces each suspicious action back to edit sites that may span multiple skills, together with a rationale linking the skill content to the action. Experimental results on SWE-Skills-Bench and CannBot demonstrate the effectiveness of SKILLMORPH. Compared with the original skills, SKILLMORPH achieves relative improvements of 15.84% in pass@3 and 30.03% in pass³ on SWE-Skills-Bench, and 16.67% and 100% on CannBot, respectively, consistently outperforming the improvements achieved by all the four compared approaches. Moreover, in a deployment with our collaboration partner, SKILLMORPH produced six skill-revision pull requests that have been accepted for future AI operator development, further showing its practical value.

References

- [1] Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. Evoskill: Automated skill discovery for multi-agent systems. *CoRR*, abs/2603.02766, 2026.
- [2] Anthropic. Agent Skills specification, 2025.
- [3] Anthropic. Claude Code: Overview. Online documentation, 2026.
- [4] Ascend. Npukernelbench, 2026.
- [5] Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *40th IEEE/ACM International Conference on Automated*

- [6] cap-evolve: a skills-native, host-agnostic harness for honestly optimizing ai-agent capabilities, 2026. <https://github.com/skillberry-ai/cap-evolve>.
- [7] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya G. Parameswaran, Dan Klein, Kannan Ramchandran, Matei A. Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025.
- [8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.
- [9] DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. *CoRR*, abs/2606.19348, 2026.
- [10] Srishti Gautam, Arjun Radhakrishna, and Sumit Gulwani. Skillaxe: Sharpening llm-authored agent skills through evaluation-guided self-refinement. *CoRR*, abs/2606.10546, 2026.
- [11] Yu Ge, Linna Xie, Zhong Li, Yu Pei, and Tian Zhang. Spectrum-based failure attribution for multi-agent systems. *Proc. ACM Softw. Eng.*, 3(FSE), June 2026.
- [12] Tingxu Han, Yi Zhang, Wei Song, Chunrong Fang, Zhenyu Chen, Youcheng Sun, and Lijie Hu. Swe-skills-bench: Do agent skills actually help in real-world software engineering? *CoRR*, abs/2603.15401, 2026.
- [13] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.
- [14] David Boram Hong, Aaron Imani, and Iftekhar Ahmed. From anatomy to smells: An empirical study of skill.md in agent skills, 2026.
- [15] Wei-Jung Huang. Do LLM-authored agent skills help? a component-level ablation in data analysis tasks. OpenReview, 2026.
- [16] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
- [17] Gaoyuan Li, Meihao Fan, Yizhe Liu, Shaolei Zhang, Ju Fan, Siyi Wang, Jiaheng Hou, Xudong Weng, Honghan Tian, and Zang Li. Skilladam: Stable and efficient skill evolution for agents. *arXiv preprint arXiv:2609.08944*, 2026.
- [18] Han Li, Yifan Yao, Letian Zhu, Rili Feng, Hongyi Ye, Jiaming Wang, Yancheng He, Pengyu Zou, Lehan Zhang, Xinping Lei, Haoyang Huang, Ken Deng, Ming Sun, Zhaoxiang Zhang, He Ye, and Jiaheng Liu. Codetracer: Towards traceable agent states. *CoRR*, abs/2604.11641, 2026.

- [19] Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, Shuyi Wang, Qunhong Zeng, Di Wang, Xuandong Zhao, Yuanli Wang, Roey Ben Chaim, Zonglin Di, Yipeng Gao, Junwei He, Yizhuo He, Liqiang Jing, Luyang Kong, Xin Lan, Jiachen Li, Songlin Li, Yijiang Li, Yueqian Lin, Xinyi Liu, Xuanqing Liu, Haoran Lyu, Ze Ma, Bowei Wang, Runhui Wang, Tianfu Wang, Wengao Ye, Yue Zhang, Hanwen Xing, Yiqi Xue, Steven Dillmann, and Han-Chung Lee. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *CoRR*, abs/2602.12670, 2026.
- [20] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing : Industry track paper. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 789–801. IEEE, 2021.
- [21] Shukai Liu, Bo Jiang, Jian Yang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens, editors, *Findings of the Association for Computational Linguistics, ACL 2026, San Diego, California, United States, July 2-7, 2026*, pages 20604–20617. Association for Computational Linguistics, 2026.
- [22] Shuyang Liu, Yang Chen, Rahul Krishna, Saurabh Sinha, Jatin Ganhotra, and Reyhaneh Jabbarvand. Process-centric analysis of agentic software systems. *Proc. ACM Program. Lang.*, 10(OOP-SLA1), April 2026.
- [23] Yuxuan Liu, Zhaochen Su, Lingyun Xie, Yuhao Zhang, Qing Zong, Jiahe Guo, Zhongwei Xie, Yiyang Ji, Yauwai Yim, Hongyu Luo, Xiyu Ren, Ruan Chenyu, Haoran Li, and Yangqiu Song. Skillrevise: Improving llm-authored agent skills via trace-conditioned skill revision. *CoRR*, abs/2606.01139, 2026.
- [24] Zijian Lu, Yiping Zuo, Yupeng Nie, Xin He, Weibei Fan, Lianyong Qi, and Shi Jin. Contractskill: Repairable contract-based skills for multimodal web agents. *CoRR*, abs/2603.20340, 2026.
- [25] Oorja Majgaonkar, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. Understanding code agent behaviour: An empirical study of success and failure trajectories. *CoRR*, abs/2511.00197, 2025.
- [26] Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjuan Jiang. Trace2skill: Distill trajectory-local lessons into transferable agent skills. *CoRR*, abs/2603.25158, 2026.
- [27] Akira Ochiai. Zoogeographical studies on the soleoid fishes found in japan and its neighbouring regions-i. *Nippon Suisan Gakkaishi*, 22(9):522–525, 1957.
- [28] Jincheng Ren, Siwei Wu, Yizhi Li, Kang Zhu, Shu Xu, Boyu Feng, Ruibin Yuan, Wei Zhang, Riza Batista-Navarro, Jian Yang, and Chenchua Lin. A self-evolving framework for efficient terminal agents via observational context compression. *CoRR*, abs/2604.19572, 2026.
- [29] Benjamin Rombaut, Sogol Masoumzadeh, Kirill Vasilevski, Dayi Lin, and Ahmed E. Hassan. Watson: A cognitive observability framework for the reasoning of llm-powered agents. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*, pages 739–751. IEEE, 2025.
- [30] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *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*, 2023.
- [31] Yu Tian, Jiawei Chen, Lifan Zheng, Mingxiang Tao, Xinyi Zeng, Zhaoxia Yin, Hang Su, and Xian Sun. Skills-coach: A self-evolving skill optimizer via training-free GRPO. *CoRR*, abs/2604.27488, 2026.

- [32] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Trans. Mach. Learn. Res.*, 2024, 2024.
- [33] Hanyu Wang, Yifan Lan, Bochuan Cao, Lu Lin, and Jinghui Chen. Skillgrad: Optimizing agent skills like gradient descent. *CoRR*, abs/2605.27760, 2026.
- [34] 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, 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.
- [35] Matthijs J. Warrens. On association coefficients for 2×2 tables and properties that do not depend on the marginal distributions. *Psychometrika*, 73(4):777–789, 2008.
- [36] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 364–374. IEEE, 2009.
- [37] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1–11. ACM, 2018.
- [38] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Trans. Software Eng.*, 42(8):707–740, 2016.
- [39] Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Reducing cost of llm agents with trajectory reduction. *Proc. ACM Softw. Eng.*, 3(FSE), June 2026.
- [40] Linna Xie, Zhong Li, Yu Pei, Zhongzhen Wen, Kui Liu, Tian Zhang, and Xuandong Li. Premm: Llm-based program repair for multi-method bugs via divide and conquer. *Proc. ACM Program. Lang.*, 9(OOPSLA2):1316–1344, 2025.
- [41] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 789–799. ACM, 2018.
- [42] Renjun Xu and Yang Yan. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *CoRR*, abs/2602.12430, 2026.
- [43] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024.
- [44] Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. Skillopt: Executive strategy for self-evolving agent skills. *CoRR*, abs/2605.23904, 2026.
- [45] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. τ -bench: A benchmark for Tool-Agent-User interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*, 2025.
- [46] Zhuoyun Yu, Xin Xie, Wuguannan Yao, Chenxi Wang, Lei Liang, Xiang Qi, and Shumin Deng. Skilladaptor: Self-adapting skills for LLM agents from trajectories. *CoRR*, abs/2606.01311, 2026.

- [47] Barry Zhang, Keith Lazuka, and Mahesh Murag. Equipping agents for the real world with Agent Skills. *Anthropic Engineering*, October 2025.
- [48] Genrui Zhang, Erle Zhu, Jinfeng Zhou, Caiyan Jia, and Hongning Wang. Skillevolver: Skill learning as a meta-skill. *CoRR*, abs/2605.10500, 2026.
- [49] Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, Xue Liu, Xiaoxiao Li, and Philip S. Yu. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *CoRR*, abs/2604.01687, 2026.
- [50] Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025.
- [51] Xiangxin Zhao, Han Li, Shuaiting Li, Tianyi Zhao, Earl T. Barr, Federica Sarro, and He Ye. Failure as a process: An anatomy of cli coding agent trajectories, 2026.