CODESKILL: LEARNING SELF-EVOLVING SKILLS FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents produce rich trajectories while solving software-engineering tasks. To enable agent self-evolution, these trajectories can be distilled into reusable procedural skills that compactly encode experience to guide future behavior. However, existing skill construction and maintenance methods often rely on fixed prompts and heuristic update rules, leaving it unclear how knowledge should be selected, abstracted, and maintained to best serve downstream agents. We propose CODESKILL, an LLM-based framework that reformulates skill extraction and skill-bank maintenance as a learnable management policy. CODESKILL extracts multi-granularity procedural skills from coding-agent trajectories, evolves skills with new experience, and maintains a compact skill bank for future task solving. We train CODESKILL with reinforcement learning, using a hybrid reward that combines dense rubric-based skill-quality feedback with sparse verifiable execution feedback from the frozen downstream agent. Experiments on EnvBench, SWE-Bench Verified, and Terminal-Bench 2 show that CODESKILL improves average pass rate by 11.03 over the no-skill baseline and by 5.10 over the strongest prompt-based or memory baseline, while maintaining a compact skill bank.

1 INTRODUCTION

Large language model (LLM) agents have shown strong capabilities in complex interactive tasks, such as coding agents that resolve realistic software issues. These long-horizon interactions produce rich trajectories with reusable experience beyond individual tasks. To reuse such experience, recent agent systems explore memory mechanisms that store prior episodes, retrieve relevant cases, or summarize past trajectories (Chen et al., 2025; Tang et al., 2025; Zhu et al., 2026; Shen et al., 2026). Beyond raw episodic memory, higher-level procedural knowledge such as *skills* provides a more compact and reusable way to specify when a recurring situation arises and what actions should be taken. Skill-based agents and skill libraries have been shown effective in long-horizon and tool-use settings, making skills an important form of reusable agent knowledge for improving future behavior (Wang et al., 2024; Ling et al., 2026; Liang et al., 2026).

Inspired by the effectiveness of skills, recent work has begun to automate the distillation of skills and other reusable knowledge from past trajectories to support self-evolving agents. Across general agents and coding agents, these methods use LLMs to analyze prior successes and failures, extract reusable lessons, workflows, or reasoning patterns, and update memory over time for future task solving (Ni et al., 2026; Xia et al., 2026; Yang et al., 2026; Chen et al., 2025; Ouyang et al., 2026).

Despite this progress, existing approaches still rely heavily on fixed prompts and heuristic criteria (Xia et al., 2026; Yang et al., 2026; Shen et al., 2026), which predefine what to extract and how to update memory rather than adapting these decisions to the downstream agent. This limitation is especially salient for software-engineering (SWE) agents, where long-horizon, tool-driven trajectories contain dense and heterogeneous feedback, making it difficult to distinguish reusable procedural knowledge from task-specific or accidental details. As a result, it remains unclear what evidence should be extracted, how abstract a skill should be, and how the skill bank should be revised so that the resulting skills are both reusable across tasks and actionable for concrete agent decisions.

To this end, we propose CODESKILL, an LLM-based framework optimized to generate reusable skills from coding-agent trajectories and maintain a skill bank for augmenting future software-

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.

engineering tasks. Given past trajectories and an existing skill bank, CODESKILL extracts reusable procedural knowledge as skills, evolves existing skills based on new or failed experience, and maintains the skill bank by adding useful candidates, merging redundant ones, or dropping unhelpful skills. To train CODESKILL, we first warm-start it with supervised data constructed from coding-agent trajectories and teacher-generated skill operations, and then optimize it with Group Relative Policy Optimization (GRPO) using feedback from a frozen downstream coding agent (Shao et al., 2024). The reward combines verifiable task feedback on whether the skill helps the downstream agent solve the task with rubric-based judgments of skill quality and behavior-skill alignment (Zheng et al., 2023). This enables CODESKILL to learn a management policy for extracting, refining, and updating skills that remain reusable and actionable.

We validate CODESKILL by instantiating it with Qwen3.5-4B and evaluating it on EnvBench, SWE-Bench Verified, and Terminal-Bench 2. Across these benchmarks, CODESKILL improves average pass rate by 11.03 over the no-skill baseline and by 5.10 over the strongest prompt-based skill extraction or memory-construction baseline, corresponding to relative gains of about 37.6% and 14.5%, respectively. The gains hold across both in-domain and out-of-distribution software-engineering tasks.

Our contributions are summarized as follows.

- We propose CODESKILL, an LLM-based framework that analyzes coding-agent trajectories across diverse tasks, extracts reusable skills, and maintains an evolving skill bank for software-engineering tasks.
- We reformulate skill management as a learnable management policy, and train CODESKILL with reinforcement learning. Our reward combines sparse verifiable feedback with dense rubric-based judgments, balancing executable outcomes with informative supervision when task success is sparse.
- We validate CODESKILL on EnvBench, SWE-Bench Verified, and Terminal-Bench 2, showing that it achieves state-of-the-art performance among memory-based methods.

2 Related Work

2.1 Self-evolving Long-term Memory

Long-term memory enables LLM agents to retain experience beyond a single interaction and improve over time. Early memory-augmented agents store reflections, past experiences, or persistent user information for later retrieval (Shinn et al., 2023; Zhao et al., 2024; Chhikara et al., 2025; Zhou et al., 2025). Recent self-evolving agents further learn utility estimates for episodic memories, distill trajectories into reusable principles, or construct reasoning and workflow memories from successes and failures (Zhang et al., 2026b; Wu et al., 2025; Ouyang et al., 2026; Wang et al., 2025b). Procedural memory summarizes agent behaviors into reusable action knowledge (Fang et al., 2025). Beyond general agent settings, several works augment software-engineering agents with memories distilled from past repair trajectories or shared across agent frameworks (Chen et al., 2025; Tang et al., 2025). Another line studies how coding agents can retrieve and reuse related issue contexts or subtask-level histories (Zhu et al., 2026; Shen et al., 2026). These works show that coding agents can benefit from past experience, but they mainly retrieve or summarize prior cases. CODESKILL instead learns to extract, evolve, and maintain procedural memories using downstream feedback.

2.2 Agent Skills & Automatic Skill Management

Agent skills have emerged as a procedural form of memory that augments agents with reusable workflows, instructions, scripts, or tool-use knowledge. Unlike episodic memories that mainly recall past interactions, skills are designed to provide actionable guidance when similar situations arise (Wang et al., 2024; Ling et al., 2026; Liang et al., 2026). Recent work has therefore studied automatic skill construction from trajectories and interaction histories, where agents distill reusable lessons, workflows, or failure patterns into skill banks that can be reused and updated over time (Ni et al., 2026; Yang et al., 2026; Zheng et al., 2025; Alzubi et al., 2026; Zhang et al., 2026a; Ma et al., 2026; Zhang et al., 2026c). Several methods further combine skills with reinforcement learning or agent self-improvement loops (Xia et al., 2026; Wang et al., 2025a; Tu et al., 2026; Li et al., 2026). These

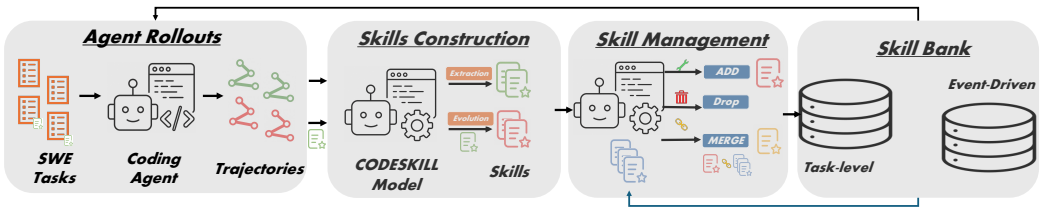


Figure 1: Overview of the CODESKILL pipeline.

works typically improve the task-solving policy with skills, train agents to use skill libraries, or co-evolve skills with the solver. In contrast, CODESKILL focuses on skill management itself, learning how to generate and maintain skills that are adapted to a frozen downstream coding agent. Recent studies on software-engineering skills also show that static skill injection does not always improve coding agents, and that skill usefulness depends strongly on task fit and context (Han et al., 2026). This further motivates adaptive skill management rather than relying on a fixed skill management strategy.

3 CODESKILL

3.1 Problem Formalization

We consider a frozen downstream coding policy π that solves a software-engineering task $x \in \mathcal{X}$ by interacting with a repository or terminal environment. A rollout produces a trajectory $\tau = (o_1, a_1, \ldots, o_T, a_T, y)$, where o_t is an observation, a_t is an action such as a shell command or file edit, and y is the task outcome. Our goal is to learn from such trajectories without updating π.

CODESKILL learns a policy M_θ for managing a skill bank $\mathcal{B} = \{s_i\}_{i=1}^{N}$, where each skill s_i contains reusable procedural instructions that provide actionable guidance. Given trajectory evidence τ and relevant skill-bank context $\mathcal{C} \subseteq \mathcal{B}$, M_θ outputs an operation $u = (a, z)$, where $a \in \mathcal{A}$ denotes the operation type, such as generation, evolution, or maintenance, and z denotes the operation content, such as a generated skill or a maintenance decision. We provide the full list of operation types in Appendix Table 3. Applying u produces an updated skill bank $\mathcal{B}' = \mathrm{Update}(\mathcal{B}, u)$.

The objective is to optimize M_θ so that the updated skill bank improves the downstream performance of the frozen policy π when used as prior knowledge. We write this objective as

$$\max_{\theta} \quad \mathbb{E}_{\tau, x'}[R_{\text{task}}(\pi(x' \mid \mathcal{B}'))]\,,$$
$$u = M_\theta(\tau, \mathcal{C}), \quad \mathcal{B}' = \mathrm{Update}(\mathcal{B}, u).$$

where R_{task} denotes downstream task performance on future task x'.

3.2 Skill Management Loop

Figure 1 provides an overview of the skill-management framework of CODESKILL. In this section, we describe how CODESKILL organizes the skill bank and manages it through three major components, including skill extraction, skill evolution, and skill-bank maintenance, together with their corresponding operations.

Multi-granularity Skill Bank. CODESKILL maintains a skill bank $\mathcal{B}$ composed of reusable procedural skills. Following prior work (Ni et al., 2026; Xia et al., 2026; Yang et al., 2026; Han et al., 2026), each skill is represented as a markdown instruction file with a title, a trigger condition, and actionable instructions for the agent. Detailed skill representations and examples are provided in Appendix D. To cover both task-level procedural knowledge and reusable guidance for local execution events, we organize the bank with two granularities. Task-level skills capture high-level strategies for a task or a family of related tasks, such as how to inspect the repository, localize the issue, or validate a fix. They are often distilled from trajectories that solve related queries. Event-driven skills

instead provide local guidance for recurring execution events, such as command failures, error messages, test-output patterns, or repeated failure modes after specific actions. They specify how the agent should react when the corresponding event is triggered, and can transfer across tasks because similar execution events recur across software-engineering problems.

Skill Bank Construction. For skill extraction, the input is trajectory evidence ranging from a single trajectory to a small set of related trajectories, and the output is either a new task-level skill, a new event-driven skill, or `skip` when the evidence does not support reusable knowledge. Task-level extraction abstracts high-level procedures from related task trajectories, while event-driven extraction captures local execution patterns from a specific trajectory. For skill evolution, the input is an existing skill together with new or failed trajectory evidence, and the output is a revised candidate skill or `skip`. This operation updates the applicability condition or procedural guidance of a skill when new evidence reveals missing cases, more effective procedures, or failure modes.

Skill Bank Maintenance. To keep the skill bank compact while allowing it to accumulate useful procedural knowledge over time, each newly extracted or evolved candidate skill is further passed to a maintenance stage. Similar skills are retrieved from the current bank and provided to M_θ together with the candidate. Based on this context, M_θ outputs a maintenance operation that either adds the candidate, merges it with an existing skill, or drops it. The `add` operation inserts the candidate as a new skill when it provides useful knowledge not covered by existing skills. The `merge` operation combines the candidate with an existing skill when they overlap but contain complementary guidance. The `drop` operation rejects candidates that are redundant, weakly grounded, overly specific, or unlikely to transfer.

Skill Usage. For downstream task solving, relevant skills are retrieved from the updated skill bank and inserted into the prompt of the frozen coding policy as prior knowledge. Retrieval is based on dense semantic similarity, matching task-level skills to the task goal and event-driven skills to the agent's current reasoning and observation through their trigger conditions. The original task prompt and policy parameters remain unchanged.

3.3 Model Training

Figure 2 summarizes the training procedure of `CODESKILL`. We first warm up M_θ on teacher-generated supervised data for skill extraction, evolution, and maintenance, and then optimize it with reinforcement learning using feedback from both rubric-based LLM-as-judge rewards and a frozen downstream coding policy.

3.3.1 Supervised Warmup

We first warm up M_θ with supervised data generated by teacher models. Given normalized coding-agent trajectories, optional existing skills, and retrieved skill-bank context, the teacher produces structured targets $u^* = (a^*, z^*)$ for skill extraction, evolution, and maintenance. Each example is written as (q, u^*), where $q = (\tau, \tilde{s}, \mathcal{C})$ contains trajectory evidence τ, an optional candidate or existing skill $\tilde{s}$, and retrieved similar skills $\mathcal{C} \subseteq \mathcal{B}$. We fine-tune the base model with

$$\mathcal{L}_{\mathrm{SFT}} = -\mathbb{E}_{(q,u^*)} \log M_\theta(u^* \mid q).$$

This produces an initial policy that follows the operation schema before reinforcement learning.

3.3.2 RL for CODESKILL

Attribution-Aware Hybrid Reward Design. To train M_θ with RL, we first introduce a rubric-based LLM-as-judge quality reward $R_Q \in [0, 1]$ to score each generated operation and its content. For skills, the rubric evaluates grounding, reusability, specificity, format, and actionability, with action-specific criteria detailed in Appendix B. However, quality reward alone is not execution-grounded, since a skill may look plausible but remain unused or ineffective.

We therefore define an execution reward by comparing the frozen coding policy's performance with and without the generated skill. Given a sampled operation u that generates a valid skill s_u, we select a task instance from the RL training pool by reverse retrieval, $x_u \sim \mathrm{TopK}(s_u, \mathcal{D}_{\mathrm{task}})$, where the

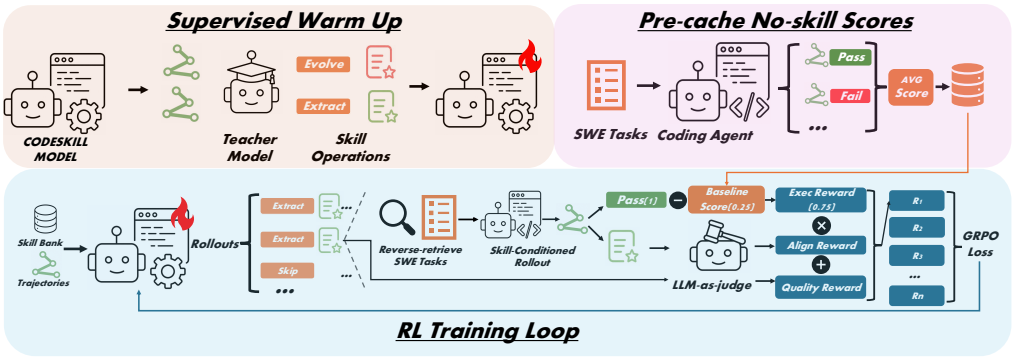


Figure 2: Training pipeline of CODESKILL.

skill content retrieves task goals or baseline trajectories and x_u is sampled from the top K results. The no-skill baseline is

$$b_\pi(x_u) = \frac{1}{n}\sum_{i=1}^{n} V(\tau^0_{\pi,i}), \quad \tau^0_{\pi,i} \sim \pi(\cdot \mid x_u).$$

We then run a skill-conditioned rollout $\tau^u_\pi \sim \pi(\cdot \mid x_u, s_u)$ and define the execution reward as the verifier improvement over the no-skill baseline, $R_E(u; x_u, \pi) = V(\tau^u_\pi) - b_\pi(x_u)$. Here $\tau^0_{\pi,i}$ is the i-th no-skill rollout, τ^u_π is the rollout conditioned on the skill produced by u, and $V(\tau) \in [0,1]$ is the task verifier score. Execution improvement alone does not establish whether the generated skill contributed to the outcome: the coding policy may succeed without following the skill. We therefore use an alignment factor $R_A(u; \tau^u_\pi) \in [0,1]$ to judge whether the rollout actually matches the skill trigger condition and follows its workflow or constraints, with LLM-as-judge rubrics detailed in Appendix B. The final reward for skill-output operations is

$$R(u; q) = \lambda R_Q(u; q) + R_A(u; \tau^u_\pi) R_E(u; x_u, \pi)$$

This attribution-aware design combines dense quality supervision with baseline-relative execution feedback, gated by behavioral evidence of skill use. It provides a practical proxy for skill-level credit assignment by discounting execution feedback when the rollout does not reflect the skill's trigger conditions or prescribed behavior.

Policy Optimization Loop. Algorithm 1 summarizes the RL loop. For each prompt q, we sample a group of G operations $\{u_j\}_{j=1}^{G}$ from the old policy $M_{\theta_{\text{old}}}$. Each operation u_j is parsed and assigned a quality reward $R_Q(u_j; q)$ by the LLM-as-judge. For operations that produce an injectable skill s_j, including extracted, evolved, and merged skills, we reverse-retrieve a task instance x_j from the RL training pool and run the frozen coding policy to obtain a skill-conditioned trajectory $\tau_j \sim \pi(\cdot \mid x_j, s_j)$. We compute the execution reward $R_E = V(\tau_j) - b_\pi(x_j)$ relative to the pre-computed no-skill baseline, and the alignment factor $R_A(u_j; \tau_j)$ from evidence of skill use in the trajectory. The final reward for each skill-output operation is then $R_j = \lambda R_Q(u_j; q) + R_A(u_j; \tau_j) R_E$. For `skip`, `add`, and `drop`, we use only the quality reward with a higher weight. After assigning rewards to all operations in the group, we normalize them into advantages and update M_θ with the GRPO objective

$$\mathcal{L}_{\text{GRPO}} = -\frac{1}{G}\sum_{j=1}^{G}\left[\min(\rho_j \hat{A}_j, \bar{\rho}_j \hat{A}_j) - \beta D_j\right].$$

Here $\rho_j = M_\theta(u_j \mid q)/M_{\theta_{\text{old}}}(u_j \mid q)$, $\bar{\rho}_j = \text{clip}(\rho_j, 1-\epsilon, 1+\epsilon)$, $\hat{A}_j = (R_j - \mu_R)/(\sigma_R + \epsilon_A)$ is the group-normalized advantage, μ_R and σ_R are the mean and standard deviation of rewards within the group, and D_j is the KL penalty to the reference model (Shao et al., 2024).

Three-stage Curriculum Training. We train CODESKILL with a three-stage curriculum that follows the life cycle of a skill bank. Skill management forms a natural data loop, extracted skills become candidates for later evolution, both extracted and evolved skills become candidates for maintenance, and maintenance decisions determine what remains in the bank for future updates. Since CODESKILL must learn multiple operation types under sparse verifiable feedback, we introduce

Algorithm 1 GRPO Training for CODESKILL

Require: prompts $\mathcal{Q}$, task pool D, downstream policy π, verifier V, old policy M_o

$\tau_i^0(x) \sim \pi(\cdot \mid x), \quad x \in D,\ i = 1, \ldots, n$ ▷ baseline rollouts

$b_\pi(x) \leftarrow n^{-1} \sum_{i=1}^{n} V(\tau_i^0(x)), \quad x \in D$ ▷ pre-cache scores

for $q \in \mathcal{Q}$ **do**

  $\{u_j\}_{j=1}^{G} \sim M_o(\cdot \mid q)$ ▷ CODESKILL rollout

  for $j = 1, \ldots, G$ **do**

    $R_Q \leftarrow R_Q(u_j; q); \quad s_j \leftarrow \mathrm{Skill}(u_j)$ ▷ compute quality score; extract skill

    $x_j \sim \mathrm{TopK}(s_j, D); \quad \tau_j \sim \pi(\cdot \mid x_j, s_j)$ ▷ reverse retrieval; downstream policy execution

    $R_E \leftarrow V(\tau_j) - b_\pi(x_j); \quad R_A \leftarrow R_A(u_j; \tau_j)$ ▷ execution reward; alignment reward

    $R_j \leftarrow \lambda R_Q + R_A R_E$ ▷ final hybrid reward

  end for

  $\hat{A}_j \leftarrow (R_j - \mu_R)/(\sigma_R + \epsilon_A),\ j = 1, \ldots, G$; update M_θ with $\mathcal{L}_{\mathrm{GRPO}}$ ▷ advantage; policy update

end for

them progressively so that earlier-stage skill outputs can be reused as training context for later-stage evolution and maintenance. ❶ trains skill extraction, teaching M_θ to generate task-level and event-driven skills from trajectory evidence. ❷ adds skill evolution, where existing skills are revised using new or failed trajectory evidence. ❸ trains the full pipeline by adding skill-bank maintenance, so that extracted and evolved candidates can be added, merged, or dropped according to the current bank context. This curriculum aligns the full skill-management loop, and the implementation details are provided in Appendix A.

4 Experiments

4.1 Experimental Setup

Implementation Details. We instantiate CODESKILL by initializing M_θ from Qwen3.5-4B (Qwen Team, 2025). For both supervised warmup and RL, we collect coding-agent trajectories from SWE-Bench Verified and SWE-smith using mini-SWE-agent, and from EnvBench using a ReAct-style bash agent (Jimenez et al., 2024; OpenAI, 2024; Yang et al., 2025; SWE-agent Team, 2025; Yang et al., 2024; Eliseeva et al., 2025; Yao et al., 2023). GPT-5.4-mini is used as the teacher model to generate supervised targets from trajectory evidence and skill-bank context for skill extraction and skill-bank maintenance (OpenAI, 2026). Skill evolution data is synthesized by pairing existing skills with related trajectory evidence, avoiding an additional skill-conditioned rollout for revised-skill supervision. During RL, the frozen downstream coding policy π and no-skill baseline rollouts use Qwen3.5-35B-A3B. GPT-5.4-mini is also used as the LLM-as-judge for quality and alignment rewards. More implementation details are provided in Appendix A.

Benchmarks. We evaluate CODESKILL on three software-engineering benchmarks. EnvBench tests environment setup and dependency repair, SWE-Bench Verified tests repository-level issue resolution, and Terminal-Bench 2 tests terminal-based problem solving (Eliseeva et al., 2025; Jimenez et al., 2024; OpenAI, 2024; Merrill et al., 2026). Since EnvBench and SWE-Bench Verified are also used for training, we use disjoint held-out splits for evaluation. For SWE-Bench Verified, we randomly sample 150 of the 500 Verified instances for evaluation and use the remaining 350 for RL training. For EnvBench, we sample 150 of 994 repositories for evaluation, covering 107 JVM and 43 Python repositories, and split the remaining 844 repositories evenly for SFT trajectory collection and RL training. Terminal-Bench 2 is not used during training, and serves as an out-of-distribution benchmark for testing whether the learned skill-management policy generalizes to new coding-agent tasks. We use mini-SWE-agent for SWE-Bench Verified and Terminal-Bench 2, and a ReAct-style agent for EnvBench (Yang et al., 2024; Yao et al., 2023). Although the benchmarks and agent wrappers differ, all agents interact with the environment through bash-command-based actions. We therefore normalize their trajectories into the same reasoning-action-observation format, allowing a single CODESKILL model to manage skills across different SWE tasks. We report task pass rate on all benchmarks. For EnvBench, we also report average issue count, and for all benchmarks we report the average number of reasoning steps on the common subset of instances solved by all compared methods within each benchmark and downstream-policy setting.

Table 1: Main results on SWE-related benchmarks. Success is task pass rate. Issues denotes the average issue count on EnvBench, and Steps denotes the average reasoning steps on solved instances. Avg. reports macro-average success and reasoning steps across benchmarks.

Method	Skill Backbone	EnvBench-Python			EnvBench-Java			SWE-Bench-Verified		Terminal-Bench 2		Average	
		Succ. ↑	Issues ↓	Steps ↓	Succ. ↑	Issues ↓	Steps ↓	Succ. ↑	Steps ↓	Succ. ↑	Steps ↓	Succ. ↑	Steps ↓
Frozen coding policy: Qwen3.5-35B-A3B													
No-skill baseline	–	6.98	81.70	30.00	27.10	6.10	14.03	57.33	88.41	25.88	44.05	29.32	44.12
Subtask Memory	GPT-5.4-mini	9.30	86.33	26.50	32.71	**4.25**	13.34	61.33	81.31	30.59	37.88	33.48	39.76
Prompt Skill Mgmt.	Qwen3.5-4B	4.65	91.95	26.80	30.84	4.67	**13.09**	58.67	76.69	24.71	39.73	29.72	39.08
Prompt Skill Mgmt.	GPT-5.4-mini	11.63	**31.58**	25.00	36.45	5.49	13.72	64.67	75.97	28.24	33.38	35.25	37.02
SFT-CODESKILL	Qwen3.5-4B	13.95	88.60	25.25	35.51	7.85	13.65	64.00	78.65	24.71	**34.62**	34.54	38.04
CODESKILL	Qwen3.5-4B	**18.60**	62.74	**24.33**	**38.32**	5.35	14.00	**68.00**	**70.94**	**36.47**	34.69	**40.35**	**35.99**
Frozen coding policy: GPT-5.4-mini													
No-skill baseline	-	4.65	70.72	13.00	15.89	3.62	9.76	46.67	7.56	20.00	8.23	21.80	9.64
Subtask Memory	GPT-5.4-mini	9.30	66.77	11.81	25.23	3.49	9.64	52.67	6.73	22.35	**6.06**	27.39	8.56
Prompt Skill Mgmt.	Qwen3.5-4B	6.98	31.19	10.67	22.43	2.56	9.25	50.67	**6.32**	23.53	6.25	25.90	8.12
Prompt Skill Mgmt.	GPT-5.4-mini	9.30	79.60	10.50	24.30	2.27	9.35	**56.67**	6.53	21.18	**6.06**	27.86	8.11
SFT-CODESKILL	Qwen3.5-4B	6.98	70.77	10.50	23.36	2.21	**8.25**	54.67	6.66	23.53	6.50	27.14	**7.98**
CODESKILL	Qwen3.5-4B	**13.95**	**25.05**	**9.00**	**27.10**	**1.79**	9.10	56.00	6.43	**29.41**	7.58	**31.62**	8.03

Baselines. We compare CODESKILL with representative memory and skill-management baselines. First, we evaluate prompt-based skill-management pipelines that use the same operation space as CODESKILL but replace the learned policy with fixed prompts and heuristic decisions. This baseline follows the common design of prior automatic skill-bank management methods, such as SkillRL and AutoSkill (Yang et al., 2026; Xia et al., 2026), while keeping the downstream coding policy frozen. We also compare with reasoning-oriented subtask-level memory, a strong coding-agent memory baseline that decomposes long software-engineering trajectories into subtasks and distills reusable memories from the resulting sub-trajectories (Shen et al., 2026).

Test-Time Skill Bank Construction. We construct the skill bank through two preparatory passes over the test set before final evaluation. In the first pass, CODESKILL extracts skills from no-skill rollout trajectories. In the second pass, it evolves skills using trajectories from skill-conditioned rollouts. Every extracted or evolved candidate undergoes maintenance through add, merge, or drop decisions. Together, these passes simulate a skill lifecycle of experience accumulation and refinement across tasks. We track the source instances of each skill, preserving their provenance through evolution and merging. During final evaluation, we exclude any skill whose provenance includes the current test instance to prevent access to knowledge derived from that instance. Task-level and event-driven skills are retrieved using task goals and local execution signals, respectively. Detailed retrieval procedures are provided in Appendix C.

4.2 Main Results

Table 1 reports the main evaluation results across EnvBench, SWE-Bench Verified, and Terminal-Bench 2. We compare CODESKILL with no-skill execution, subtask-level memory, and prompt-based skill management using different skill backbones, including open-source Qwen3.5-4B and closed-source GPT-5.4-mini. The table also reports results under two frozen downstream coding policies, Qwen3.5-35B-A3B and GPT-5.4-mini.

Comparison with Baselines. We first compare CODESKILL with baselines under Qwen3.5-35B-A3B as the downstream coding policy. Compared with the no-skill baseline, CODESKILL improves average pass rate from 29.32 to 40.35, showing the benefit of managed procedural skills for downstream coding agents. Prompt-based skill management with the small Qwen3.5-4B backbone brings little improvement over no-skill, indicating that a weak model is insufficient for the same skill-management pipeline. Stronger baselines are more effective, with subtask-level memory and GPT-5.4-mini prompt-based skill management improving over no-skill by 4.16 and 5.93 average pass rate, respectively. However, they still underperform CODESKILL by 6.87 and 5.10. SFT-CODESKILL improves average pass rate over no-skill, but remains weaker than prompt-based management with a stronger closed-source model. After RL training, CODESKILL achieves the best pass rate on all benchmark groups, improving over the strongest baseline by 14.5% relative gain on average. Over-

Table 2: Ablation study of the skill-bank lifecycle with Qwen3.5-35B-A3B as the frozen coding policy. Skill Num. denotes the number of skills in the resulting skill bank.

Variant	EnvBench-Python			EnvBench-Java			SWE-Bench Verified		Terminal-Bench 2		Skill Num.
	Succ. ↑	Issues ↓	Steps ↓	Succ. ↑	Issues ↓	Steps ↓	Succ. ↑	Steps ↓	Succ. ↑	Steps ↓	
No-skill baseline	6.98	81.70	30.00	27.10	6.10	14.03	57.33	88.41	25.88	44.05	0
Event-driven only	16.28	53.98	26.57	32.71	5.65	**13.57**	62.00	76.99	29.41	33.83	916
Task-level only	11.63	78.33	26.60	30.84	**3.28**	14.57	58.67	88.25	28.24	46.25	336
Extraction only	13.95	58.12	25.09	**41.12**	5.79	14.15	64.67	78.05	31.76	**30.63**	1252
Extraction + Evolution	**18.60**	**43.00**	24.74	39.25	4.16	13.90	67.33	77.28	32.94	32.17	1252
Full lifecycle	**18.60**	62.74	**24.33**	38.32	5.35	14.00	**68.00**	**70.94**	**36.47**	34.69	676

all, the results suggest that downstream-feedback optimization learns a skill-management policy better suited to the frozen coding agent than fixed-prompt skill distillation methods.

Efficiency. We also compare the average number of reasoning steps on the common subset of instances successfully solved by all compared methods, ensuring that all averages are computed over the same tasks. Skill-based methods generally reduce the reasoning cost compared with the no-skill baseline, suggesting that retrieved procedural guidance can help the coding agent reach successful solutions more directly. Under the Qwen3.5-35B-A3B policy, CODESKILL achieves the lowest average reasoning steps, reducing the average from 44.12 for no-skill to 35.99. This indicates that learned skill management improves not only task success but also the efficiency of successful problem solving.

Generalization across Tasks. We first examine whether CODESKILL generalizes beyond the task distributions used for training. CODESKILL is trained with trajectories from EnvBench and SWE-Bench Verified, while Terminal-Bench 2 is held out and used only for evaluation. Although Terminal-Bench 2 shares the same bash-command interaction interface, it contains different terminal-based problem-solving tasks and therefore tests out-of-distribution generalization within software engineering. Under the Qwen3.5-35B-A3B coding policy, CODESKILL improves Terminal-Bench 2 pass rate from 25.88 to 36.47 over no-skill, and also outperforms the strongest baseline on this benchmark, subtask-level memory with GPT-5.4-mini, by 5.88 pass rate. In contrast, SFT-CODESKILL does not improve over no-skill on Terminal-Bench 2. This suggests that supervised warmup alone may not generalize reliably to unseen SWE task types, while RL with downstream feedback helps CODESKILL learn more transferable skill-management behavior.

Generalization across Policies. We further evaluate whether the learned skill-management policy transfers to a different downstream coding agent. Although CODESKILL is optimized with reward rollouts from Qwen3.5-35B-A3B, we also use GPT-5.4-mini as the frozen coding policy while keeping the same skill-management and retrieval protocol. As shown in Table 1, CODESKILL again achieves the best average pass rate, improving over the no-skill baseline by 9.82 and over the strongest prompt-based or memory baseline by 3.76, corresponding to relative gains of 45.0% and 13.5%, respectively. This suggests that CODESKILL does not merely overfit to the downstream policy used during training, but learns skill-management behavior that can benefit different coding models.

4.3 Ablation & Analysis

Lifecycle Ablation. To understand how each component of the skill-bank lifecycle affects downstream coding agents, we conduct the ablation study in Table 2. We fix Qwen3.5-35B-A3B as the downstream coding policy and compare variants that progressively enable different skill granularities and lifecycle operations. The event-driven-only and task-level-only variants use a single skill granularity, while extraction only combines both. Event-driven and task-level skills each improve over the no-skill baseline on different tasks, and their combination further improves average performance, suggesting that local execution guidance and high-level task strategies capture complementary knowledge. Adding skill evolution improves the average pass rate from 37.88 to 39.53 over extraction only. Full lifecycle maintenance further improves the average pass rate to 40.35 while shrinking the skill bank from 1252 to 676 skills, a reduction of approximately 46%. These results

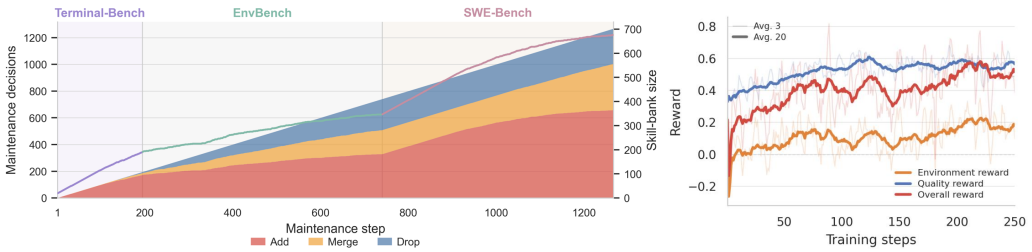


Figure 3: Analysis of skill-bank maintenance and RL training dynamics. The left panel shows cumulative add, merge, and drop decisions together with the resulting skill-bank size during skill-bank maintenance. The right panel shows the reward number over the stage-3 training process.

suggest that maintenance improves the compactness of the skill bank without sacrificing aggregate performance. The additional performance gain may partly arise from reducing redundant or conflicting skills, making retrieved guidance less repetitive and more consistent. Finally, all skill-based variants reduce average reasoning steps over the no-skill baseline, suggesting that procedural skill guidance contributes to more efficient successful task execution.

Skill-Bank Maintenance Dynamics. Figure 3 visualizes skill-bank maintenance during test-time skill construction. As CODESKILL processes evaluation instances, each newly extracted or evolved candidate skill enters the maintenance stage, where it can be added, merged with an existing skill, or dropped. The figure records cumulative maintenance decisions and the resulting skill-bank size. Add decisions dominate early, when the bank is small and most candidates provide uncovered knowledge. As the bank grows, merge and drop decisions become more frequent, indicating that CODESKILL identifies overlapping, redundant, or low-value candidates rather than adding every candidate to the bank. The benchmark-wise dynamics illustrate how maintenance behavior varies with the amount of task experience within each benchmark. Terminal-Bench has fewer test instances and candidate skills, and add decisions remain prominent in its segment. EnvBench and SWE-Bench provide more opportunities to accumulate related skills, with merge and drop decisions consolidating overlapping knowledge. We hypothesize that, under a relatively stable task distribution, continued evolution could drive the bank toward a steady size as reusable patterns become increasingly covered. Meanwhile, add decisions remain active across benchmark segments, consistent with the need to incorporate new procedural knowledge when the task distribution changes.

Reward Optimization Dynamics. The right panel of Figure 3 shows reward dynamics during phase-3 RL training. Unlike directly optimizing a task-solving policy, CODESKILL affects outcomes only after its skills are retrieved, injected, and followed in long-horizon software tasks, making the environment reward noisy. Still, its 20-step trend increases from 0.004 in steps 1–20 to 0.158 in steps 180–200, suggesting that CODESKILL gradually learns management decisions that improve the downstream agent over its no-skill baseline. The quality reward, provided by rubric-based LLM-as-judge feedback, is more stable and rises before stabilizing after roughly 100 steps. The overall reward follows the same upward pattern, indicating that the hybrid objective improves both executable utility and skill quality during RL training.

5 Conclusion

We presented CODESKILL, an LLM-based framework that learns to extract, evolve, and maintain reusable skills from coding-agent trajectories. CODESKILL formulates skill management as a learnable policy and optimizes it with verifiable downstream feedback and rubric-based skill-quality judgments by reinforcement learning. Experiments on EnvBench, SWE-Bench Verified, and Terminal-Bench 2 show that learned skill management improves over no-skill execution, prompt-based skill management, and memory-based baselines. These results highlight learned procedural-memory management as a promising direction for coding agents that accumulate and reuse long-horizon software-engineering experience. These gains are achieved without updating the downstream coding policy, emphasizing the value of learning how experience is organized and reused. Future work could extend this framework to strictly streaming task settings and evaluate the end-to-end cost of skill acquisition, maintenance, and use.

AI USE STATEMENT

We used generative AI tools to polish the language and improve the readability of the manuscript. As part of the research pipeline, we used LLMs to generate supervised skill-management targets and to provide rubric-based quality and behavior-alignment judgments, as detailed in Section 3.3 and Appendices A–B. The authors reviewed and edited the AI-assisted text and take full responsibility for the final content, claims, and reported results.

REFERENCES

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. EvoSkill: Automated skill discovery for multi-agent systems, 2026. URL https://arxiv.org/abs/2603.02766.

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, 2025. URL https://arxiv.org/abs/2507.23361.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory, 2025. URL https://arxiv.org/abs/2504.19413.

Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin, Egor Bogomolov, and Yaroslav Zharov. EnvBench: A benchmark for automated environment setup. In *ICLR 2025 Workshop on Deep Learning for Code*, 2025. URL https://openreview.net/forum?id=izy1oaAOeX.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. MemP: Exploring agent procedural memory, 2025. URL https://arxiv.org/abs/2508.06433.

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?, 2026. URL https://arxiv.org/abs/2603.15401.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

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://openreview.net/forum?id=VTF8yNQM66.

Yu Li, Rui Miao, Zhengling Qi, and Tian Lan. ARISE: Agent reasoning with intrinsic skill evolution in hierarchical reinforcement learning, 2026. URL https://arxiv.org/abs/2603.16060.

Yuan Liang, Ruobin Zhong, Haoming Xu, Chen Jiang, Yi Zhong, Runnan Fang, Jia-Chen Gu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, Xin Xu, Tongtong Wu, Kun Wang, Yang Liu, Zhen Bi, Jungang Lou, Yuchen Eleanor Jiang, Hangcheng Zhu, Gang Yu, Haiwen Hong, Longtao Huang, Hui Xue, Chenxi Wang, Yijun Wang, Zifei Shan, Xi Chen, Zhaopeng Tu, Feiyu Xiong, Xin Xie, Peng Zhang, Zhengke Gui, Lei Liang, Jun Zhou, Chiyu Wu, Jin Shang, Yu Gong, Junyu Lin, Changliang Xu, Hongjie Deng, Wen Zhang, Keyan Ding, Qiang Zhang, Fei Huang, Ningyu Zhang, Jeff Z. Pan, Guilin Qi, Haofen Wang, and Huajun Chen. SkillNet: Create, evaluate, and connect AI skills, 2026. URL https://arxiv.org/abs/2603.04448.

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

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. SkillClaw: Let skills evolve collectively with agentic evolver, 2026. URL https://arxiv.org/abs/2604.08377.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026. URL https://arxiv.org/abs/2601.11868.

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, 2026. URL https://arxiv.org/abs/2603.25158.

OpenAI. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/, 2024. Accessed 2026-05-23.

OpenAI. Introducing GPT-5.4 mini and nano. https://openai.com/index/introducing-gpt-5-4-mini-and-nano/, March 2026. Accessed 2026-05-23.

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://openreview.net/forum?id=jL7fwchScm.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pp. 3982–3992, 2019. doi: 10.18653/v1/D19-1410. URL https://aclanthology.org/D19-1410.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Kangning Shen, Jingyuan Zhang, Chenxi Sun, Wencong Zeng, and Yang Yue. Structurally aligned subtask-level memory for software engineering agents, 2026. URL https://arxiv.org/abs/2602.21611.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

SWE-agent Team. mini-SWE-agent: A 100-line software engineering agent. https://github.com/SWE-agent/mini-swe-agent, 2025. Software repository.

Xiangru Tang, Tianrui Qin, Tianhao Peng, Ziyang Zhou, Yanjun Shao, Tingting Du, Xinming Wei, He Zhu, Ge Zhang, Jiaheng Liu, Xingyao Wang, Sirui Hong, Chenglin Wu, and Wangchunshu Zhou. AGENT KB: A hierarchical memory framework for cross-domain agentic problem solving. In *ICML 2025 Workshop on Collaborative and Federated Agentic Workflows*, 2025. URL https://openreview.net/forum?id=ohXoWHlrn8. Oral.

Songjun Tu, Chengdong Xu, Qichao Zhang, Yaocheng Zhang, Xiangyuan Lan, Linjing Li, and Dongbin Zhao. Dynamic dual-granularity skill bank for agentic RL, 2026. URL https://arxiv.org/abs/2603.28716.

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. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=ehfRiF0R3a.

Jiongxiao Wang, Qiaojing Yan, Yawei Wang, Yijun Tian, Soumya Smruti Mishra, Zhichao Xu, Megha Gandhi, Panpan Xu, and Lin Lee Cheong. Reinforcement learning for self-improving agent with skill library, 2025a. URL https://arxiv.org/abs/2512.17102.

Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=NTAhi2JEEE.

Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, and Botian Shi. EvolveR: Self-evolving LLM agents through an experience-driven lifecycle, 2025. URL https://arxiv.org/abs/2510.16079.

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. In *ICLR 2026 Workshop on Memory Agents*, 2026. URL https://openreview.net/forum?id=By7Pj576U3. Oral.

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 *Advances in Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=5As8S9dyaD.

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. *arXiv preprint arXiv:2504.21798*, 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, 2026. URL https://arxiv.org/abs/2603.01145.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

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, 2026a. URL https://arxiv.org/abs/2604.01687.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Yutao Qi, Bo Tang, and Muning Wen. MemRL: Self-evolving agents via runtime reinforcement learning on episodic memory, 2026b. URL https://arxiv.org/abs/2601.03192.

Ziao Zhang, Kou Shi, Shiting Huang, Avery Nie, Yu Zeng, Yiming Zhao, Zhen Fang, Qisheng Su, Haibo Qiu, Wei Yang, Qingnan Ren, Shun Zou, Wenxuan Huang, Lin Chen, Zehui Chen, and Feng Zhao. SkillFlow: Benchmarking lifelong skill discovery and evolution for autonomous agents, 2026c. URL https://arxiv.org/abs/2604.17308.

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.v38i17.29950. URL https://ojs.aaai.org/index.php/AAAI/article/view/29950.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. SkillWeaver: Web agents can self-improve by discovering and honing skills, 2025. URL https://arxiv.org/abs/2504.07079.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2306.05685.

Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and Jun Wang. Memento: Fine-tuning LLM agents without fine-tuning LLMs, 2025. URL https://arxiv.org/abs/2508.16153.

Jiayuan Zhu, Junde Wu, Minhao Hu, Shengda Zhu, Jiazhen Pan, Weixiang Shen, Yijun Yang, Fenglin Liu, Jianye Hao, Yueming Jin, Qirong Ho, and Min Xu. SWE Context Bench: A benchmark for context learning in coding, 2026. URL https://arxiv.org/abs/2602.08316.

A TRAINING DETAILS

This appendix provides additional details on the data construction and training configuration of CODESKILL. We organize the discussion by training stage. Supervised fine-tuning first teaches the model the action schema and basic skill-management behavior, while reinforcement learning further optimizes the policy with downstream feedback. All training experiments are conducted on four H100 80GB GPUs.

A.1 SUPERVISED FINE-TUNING

Trajectory Sources. We construct SFT data from coding-agent trajectories collected across software-engineering tasks. The initial SWE-style trajectory pool is downloaded from publicly available mini-SWE-agent rollouts on SWE-smith, where Qwen3-30B is used as the coding agent. We additionally collect EnvBench trajectories using a ReAct-style agent. Since different sources use different logging formats, we normalize all trajectories into a unified representation containing task context, interleaved reasoning/action/observation steps, and final outcome information.

Table 3: SFT data statistics by output action.

Action	SFT examples
task-level `generate`	2419
event-driven `generate`	4485
`skip`	1344
`evolve`	1718
`merge`	1500
`add`	790
`drop`	600
Total	12856

Table 4: Training settings for CODESKILL.

Setting	Value
General settings	
Backbone model	Qwen3.5-4B-Instruct
Training method	LoRA fine-tuning
Prompt template	`qwen3_nothink`
Max sequence length	14k tokens
Precision	bf16
Hardware	4×H100 80GB
SFT settings	
Epochs	2
batch size	4
Gradient accumulation	8
Learning rate	1×10^{-4}
Scheduler	cosine
Training time	∼20 hours
RL settings	
Group size	6 generations per prompt
Gradient accumulation	2
Quality reward weight λ	0.25
KL coefficient	0.02
Learning rate	2×10^{-6}
Training steps	500
Rollout temperature	0.7
Judge model	GPT-5.4-mini
Frozen coding policy	Qwen3.5-35B-A3B
Training time	∼210 hours

Table 5: RL training data statistics. Instances are counted within each phase, while the all-phases row reports global unique instance coverage. Prompt counts are grouped by action type.

Phase	Instances	Task-level	Event-driven	Evolve	Maintain
Phase 1	335	335	506	0	0
Phase 2	513	199	302	487	0
Phase 3	632	222	223	384	500
All phases	660	756	1031	871	500

Teacher Data Construction. We use teacher models to generate supervised targets for the skill-management action space. For task-level and event-driven extraction, the teacher reads normalized trajectory evidence and outputs either a reusable skill or a `skip` decision. For skill evolution, we synthesize training instances by retrieving an existing skill whose applicability condition or rules match a new trajectory, especially a failed or partially successful one. The paired trajectory is selected to expose missing conditions, better procedures, or failure modes not fully covered by the existing skill, and the teacher is asked to revise the skill or skip when no meaningful update is supported. This avoids collecting an additional round of skill-conditioned rollouts. For skill-bank maintenance, each candidate skill is paired with retrieved similar skills from the bank, and the teacher outputs `add`, `merge`, or `drop`.

SFT Configuration. We fine-tune Qwen3.5-4B-Instruct with LoRA using the `qwen3_nothink` template(Hu et al., 2022). The model is trained only on completion tokens. This stage teaches `CODESKILL` to follow the action schema and produce valid extraction, evolution, and maintenance outputs before reinforcement learning. The main SFT hyperparameters are summarized in Table 4.

A.2 Reinforcement Learning

3-stage RL Training Since `CODESKILL` learns multiple operation types and verifiable task feedback is sparse, we use a three-stage curriculum that follows the lifecycle of skill management and enables data reuse across stages. Phase 1 focuses on skill extraction from trajectory evidence. During this stage, generated skills are injected into the frozen coding policy to compute execution rewards, and the resulting skill-conditioned trajectories provide evidence for constructing phase-2 evolution prompts. Phase 2 trains extraction and evolution together, allowing `CODESKILL` to revise existing skills with new trajectory evidence. Skills produced by the first two stages are then reused as candidate skills and bank entries for phase-3 maintenance prompts. Phase 3 contains the full action space, including extraction, evolution, and maintenance.

Table 5 summarizes the RL prompt data. Across all phases, the RL data covers 660 unique task instances. The curriculum starts with extraction-only prompts, then adds evolution prompts, and finally introduces maintenance prompts, aligning the training data with the skill-bank lifecycle.

RL Configuration. We optimize `CODESKILL` with GRPO starting from the SFT checkpoint. The RL curriculum runs for 500 update steps in total, with 130, 120, and 250 steps for phases 1, 2, and 3, respectively. The frozen downstream coding policy used for reward-side rollouts is Qwen3.5-35B-A3B, and rubric-based quality and alignment judgments use GPT-5.4-mini. The main RL hyperparameters are summarized in Table 4.

B Reward Design

To compute the execution reward, we first pre-compute a no-skill baseline for every RL task instance. Specifically, we run the frozen Qwen3.5-35B-A3B coding policy four times on each instance without injected skills, and use the average verifier score as the baseline score. The execution reward of a generated skill is then computed as the improvement over this instance-specific baseline.

For rubric-based rewards, we design action-specific judge prompts for task-level extraction, event-driven extraction, skill evolution, skill-bank maintenance, and behavior alignment. Representative judge templates are provided in Figures 10–14. Each rubric consists of multiple dimensions, and

each dimension is further decomposed into binary yes/no questions. The final rubric reward is computed as the number of satisfied yes/no questions divided by the total number of yes/no questions in the rubric. Thus, the relative weight of each dimension is determined by the number of questions assigned to it, while the final reward remains normalized to $[0, 1]$.

C Experimental Details

Skill Retrieval. The frozen downstream coding policy solves each task with retrieved skills injected as prior knowledge. We build separate retrieval indexes for task-level and event-driven skills, so that the two granularities are matched with different query signals. For each skill, the retrieval document is constructed from its `title`, `when_to_apply`, and `rules`. We encode retrieval documents with `sentence-transformers/all-MiniLM-L6-v2` and build dense indexes separately for each benchmark and skill type (Reimers & Gurevych, 2019). Retrieval is scoped to the same benchmark and same granularity, and skills generated from the same evaluation instance are filtered out to avoid same-instance leakage.

For task-level skills, the query is constructed from the task goal, problem statement, and available repository or benchmark context. Task-level skills are retrieved once before task solving and appended to the initial downstream policy user prompt as prior knowledge. For event-driven skills, the query is constructed online from local execution signals, including the current task context, recent reasoning, executed actions, observations, error messages, command outputs, and test-output snippets when available. Each instance is solved with retrieved skills through skill-conditioned generation, and every resulting skill-conditioned trajectory is prompted to `CODESKILL` as evidence for skill evolution. Each newly extracted or evolved skill is then passed to the maintenance stage, where it may be added, merged, or dropped to update the current skill bank.

Baselines. We compare `CODESKILL` with no-skill execution, prompt-based skill management, and subtask-level memory. The no-skill baseline uses the same frozen downstream coding policy without any prior-knowledge section. The prompt-based skill-management baseline is designed as an adaptation of recent automatic skill-bank methods to software-engineering tasks. Many such methods follow an extraction, evolution, and maintenance pipeline, or use a subset of these operations, but do not provide a software-engineering implementation. We therefore keep the same pipeline, action space, prompts, retrieval procedure, downstream policy, and decoding settings as `CODESKILL`, while replacing the learned skill-management policy with fixed prompt-based decisions.

We also compare with a subtask-level memory baseline (Shen et al., 2026). This baseline is relevant to software-engineering tasks because long-horizon coding trajectories naturally decompose into smaller reasoning and action segments, and prior work has shown strong performance for reasoning-oriented memory in SWE settings. Since the original implementation is not publicly available, we implement a high-level reproduction. We segment each trajectory into subtasks using rule-based patterns based on agent action types and execution boundaries. Each sub-trajectory is then summarized into a memory item using a reasoning-oriented prompt, and relevant memory items are retrieved and injected into the downstream policy prompt in the same format used by other baselines.

D Skill Examples

In general, an agent skill can be represented as a structured folder that contains reusable instructions, metadata, external resources, executable scripts, APIs, or tool-specific routines for guiding future agent behavior. In this work, since the coding-agent environments already provide fixed tools and execution interfaces, we follow prior work on instruction-level agent skills and focus on the natural-language instruction file rather than generating new scripts, APIs, or tools. Each skill file contains a short title, a granularity label, an applicability condition `when_to_apply`, and a set of actionable rules. `CODESKILL` maintains two types of skills. Task-level skills encode high-level procedures for a task or a family of related tasks, while event-driven skills encode local guidance for recurring execution signals. Figures 4 and 5 show representative examples from the maintained skill bank.

Task-Level Skill Example

Title
Diagnose Maven build failures via environment, config, and dependency resolution.

Granularity
`general`

When to Apply
When a Java/Maven project fails to build and the cause may involve toolchain or wrapper configuration, repository settings, missing or stale dependencies, or build-phase ordering rather than source code errors.

Instructions

- Start by identifying the project type and build instructions from repository metadata such as the main build file and README before making changes.
- Run the build with the project's Maven wrapper first, then capture both early and later output to distinguish dependency download progress from the actual failure.
- Use the first concrete failure message to distinguish network or repository issues, missing artifacts, and compilation errors before changing code.
- If the build fails in the enforcer or toolchain phase, use verbose output to identify the required Java and Maven versions before changing anything.
- Check the active Java and Maven toolchain versions in the shell and align them with the project's documented requirements when needed.
- When a dependency cannot be resolved, check whether the missing artifact is generated later in the build, and verify whether the build phase order or plugin lifecycle is appropriate.
- Inspect local Maven settings and repository configuration before changing code; misconfigured mirrors or missing settings files can mask the real build problem.
- After adjusting toolchain, wrapper settings, repository configuration, or dependency versions, rerun the build and re-check the logs to confirm whether the failure mode has changed.

Provenance
This skill is an example from the maintained skill bank and was produced by a `merge` operation.

Figure 4: Example task-level skill from the maintained skill bank.

E PROMPTS

We present the prompt template of task-level extraction in 6, event-level extraction in 7, skill evolution in 8, and skill management in 9.

Event-Driven Skill Example

Title
Handle missing Maven test-jar dependencies.

Granularity
`event-driven`

When to Apply
When a Maven build fails with a dependency resolution error for a test-jar or similar scoped artifact that has not yet been produced or installed.

Instructions

- Inspect the failing module's POM to confirm the dependency type and scope, for example a test-jar with compile scope.
- Check the producing module's POM to see whether it already defines an execution for the required artifact via a plugin configuration.
- If the test jar is only generated when tests run, either run the tests or explicitly invoke the Maven test-jar goal to create the test artifact before rebuilding.
- If the artifact is missing, add or configure the appropriate plugin execution in the producing module's POM to generate the required artifact, and place the new plugin configuration inside the correct parent element so the POM remains well formed.
- Do not simply retry the same Maven install command with tests skipped; skipping tests often prevents the test jar from being created.
- After generating or configuring the artifact, run a full install/build command so the artifact is generated and installed into the local repository; a compile-only run may not install the needed artifact.
- Verify that the artifact is actually present in the local Maven repository before retrying the original failing command.
- After the artifact is installed, rerun the original failing command to confirm that the dependency resolution error is resolved.

Provenance
This skill is an example from the maintained skill bank and was produced by a `merge` operation.

Figure 5: Example event-driven skill from the maintained skill bank.

Task-Level Skill Extraction Prompt

System Prompt

You are an expert at extracting reusable memory from bash-agent trajectories. Extract one reusable `general` skill from multiple related trajectories of a bash-based agent. Use only the provided evidence.

Input

The user prompt provides:

- light task context
- 2–3 related trajectories
- optional result summaries

Task

Choose exactly one action:

- `generate`: create one bootstrap skill
- `skip`: create no skill

Rules

1. A bootstrap skill is a task-level reusable pattern supported by multiple trajectories. It must be broader than a single local event.
2. Generate only if the pattern is reusable across similar tasks. Keep `when_to_apply` high-level and write transferable, actionable `rules`.
3. Ground every part of the skill in repeated evidence from the trajectories and outcomes. Do not invent unsupported steps, checks, or guidance. If failed trajectories reveal reusable cautions, include them as cautionary rules.
4. Skip if the evidence is weak, contradictory, accidental, too local, or collapses into an event-level reaction instead of a task-level pattern.
5. Do not include repository names, issue descriptions, exact task goals, variable names, function names, class names, module names, exact file paths, or one-off literals.

Output Schema

`generate`

```
{
  "action": "generate",
  "skill": {
    "title": "short reusable skill name",
    "granularity": "general",
    "when_to_apply": "high-level task situation where this skill should be used",
    "rules": ["reusable rule 1", "reusable rule 2"]
  }
}
```

`skip`

```
{
  "action": "skip",
  "reason": "short reason"
}
```

Figure 6: Prompt template for task-level skill extraction.

Event-Driven Skill Extraction Prompt

System Prompt

You are an expert at extracting reusable memory from bash-agent trajectories. Extract one reusable `event-driven` skill from a full trajectory of a bash-based agent. Use only the provided evidence.

Input

The user prompt provides:

- light task context
- one full trajectory
- optional result summary

Task

Choose exactly one action:

- `generate`: create one event-driven skill
- `skip`: create no skill

Rules

1. An event-driven skill is a local trigger-response pattern. It must focus on one important event inside the trajectory and stay narrower than a whole-task workflow.
2. Generate only if the event yields a reusable lesson beyond this exact task. Keep `when_to_apply` transferable and write local actionable `rules`.
3. Ground the trigger and guidance in the trajectory and result. Do not invent an event that is not clearly present. Failure-derived cautions are valid if they are clearly supported.
4. If multiple candidate events exist, choose the single most reusable one.
5. Skip if there is no strong reusable local event, or if the lesson is too task-specific or workflow-level.
6. Do not include repository names, issue descriptions, exact task goals, variable names, function names, class names, module names, exact file paths, or one-off literals.

Output Schema

`generate`

```
{
  "action": "generate",
  "skill": {
    "title": "short reusable skill name",
    "granularity": "event-driven",
    "when_to_apply": "high-level local signal or situation where this skill should be used
     ",
    "rules": ["reusable rule 1", "reusable rule 2"]
  }
}
```

`skip`

```
{
  "action": "skip",
  "reason": "short reason"
}
```

Figure 7: Prompt template for event-driven skill extraction.

Skill Evolution Prompt

System Prompt

You are an expert at extracting and revising reusable memory from bash-agent trajectories. You revise one existing reusable skill using new trajectory evidence from a bash-based agent. Use only the provided evidence.

Input

The user prompt provides:

- one or more relevant skills
- one trajectory
- optional result summary

Task

Choose exactly one action:

- `evolve`: revise exactly one existing skill
- `skip`: make no revision

Rules

1. **Alignment and need for revision.** Choose `evolve` only when one existing skill is clearly aligned with the current trajectory context and trigger pattern, and the current trajectory result shows that this aligned skill should be revised. If multiple skills are aligned and all are plausible revision targets, choose the single one that is most worth revising based on the strength and reusability of the evidence. The revision should be motivated by the observed outcome, such as a missing case, missing check, bad ordering, misleading guidance, or reusable caution.
2. **Reusability and rule writing.** Revise the skill only if the trajectory reveals a reusable missing case, missing check, better decision rule, improved ordering, or reusable caution. `rules` should contain the revised reusable core of the skill as actionable guidance.
3. **Applicability.** `when_to_apply` should remain at a high transferable level rather than concrete task text.
4. **Grounding.** The revision must be directly supported by the provided trajectory and result. Do not add unsupported refinements.
5. **Identity preservation.** Keep the same capability identity. Do not drift into a different skill.
6. **When to skip.** Choose `skip` if the evidence is weak, contradictory, too local, or if the trajectory is better described as a brand-new skill rather than a revision.
7. **Do not include task-specific details.** Do not include repository names, issue descriptions, exact task goals, variable names, function names, class names, module names, exact file paths, or one-off literals that only make sense for one instance.

Output Schema

`evolve`

```
{
  "action": "evolve",
  "target_skill_id": "id of the single skill you chose to revise",
  "reason": "short reason",
  "skill": {
    "title": "short reusable skill name; same capability identity as target skill",
    "granularity": "general | event-driven",
    "when_to_apply": "high-level condition where this revised skill should be used",
    "rules": ["revised rule 1", "revised rule 2"]
  }
}
```

`skip`

```
{
  "action": "skip",
  "reason": "short reason"
}
```

Figure 8: Prompt Template for skill evolution.

Skill-Bank Maintenance Prompt

System Prompt

You are an expert at maintaining reusable memory for bash-based agents. You decide how a candidate reusable skill should enter the skill bank for a bash-based agent. Use only the provided candidate skill and retrieved similar skills.

Input

The user prompt provides:

- 1 candidate skill
- multiple retrieved similar skills

Task

Choose exactly one action:

- `add`: accept the candidate as a new skill
- `merge`: merge the candidate with exactly one retrieved skill
- `drop`: reject the candidate

Rules

1. **Add.** Choose `add` when the candidate is reusable, coherent, and distinct from the retrieved skills.
2. **Merge.** Choose `merge` when the candidate and one retrieved skill share the same capability identity and can be combined into one stronger reusable skill. The merged skill should have clearer applicability and cleaner rules with less duplication.
3. **Drop.** Choose `drop` when the candidate is already covered by stronger retrieved skills, redundant, weakly evidenced, unsafe, too local, or too task-specific.
4. **Do not include task-specific details.** No resulting skill may include repository names, issue descriptions, exact task goals, variable names, function names, class names, module names, exact file paths, or one-off literals that only make sense for one instance.

Output Schema

`add`

```
{
  "action": "add",
  "reason": "short reason"
}
```

`drop`

```
{
  "action": "drop",
  "reason": "short reason"
}
```

`merge`

```
{
  "action": "merge",
  "merge_target_skill_id": "id of the retrieved skill selected for merge",
  "reason": "short reason",
  "skill": {
    "title": "short reusable skill name",
    "granularity": "general | event-driven",
    "when_to_apply": "high-level condition where this skill should be used",
    "rules": ["merged rule 1", "merged rule 2"]
  }
}
```

Schema requirements for `merge.skill`:

- `title`: short reusable name; not task-specific.
- `when_to_apply`: transferable applicability, not concrete task text.
- `rules`: a non-empty list of reusable actionable rules.

Figure 9: Prompt template for skill-bank maintenance.

Task-Level Skill Quality Judge

System Prompt

You are an expert judge for skill-manager outputs. You are strict and critical by default. When in doubt, give a lower score.

Task

Judge a proposed `bootstrap` prior-knowledge output. The output should be a task-level reusable item with `when_to_apply` and `rules`, rather than a local event-trigger rule, one-step trick, or instance-specific note.

Evaluation Input

{{TRAJECTORY_EVIDENCE}} and {{PROPOSED_OUTPUT}}.

Scoring Rule

For each dimension, answer the listed sub-questions with yes or no. The score is the number of yes answers.

Dimensions

- **groundedness, 0–3.** Q1: every rule is directly traceable to observable trajectory behavior or outcome. Q2: `when_to_apply` is directly supported by trajectory signals. Q3: no rule is contradicted, absent, or weakly hinted.
- **reusability, 0–3.** Q1: rules avoid repository names, file paths, versions, or identifiers. Q2: rules transfer to at least two distinct project types. Q3: applicability is broad enough for unseen repositories but not nearly every coding task.
- **specificity, 0–3.** Q1: each rule gives a concrete executable action or check. Q2: rules add value beyond common engineering practice. Q3: rules would not equally apply to most unrelated coding tasks.
- **format_validity, 0–1.** The output is valid and complete.
- **when_to_apply_quality, 0–3.** Q1: applicability contains discriminating conditions. Q2: it would not trigger on clearly different tasks. Q3: it is narrow enough to avoid applying to most SWE tasks.
- **task_level_granularity, 0–3.** Q1: the output is a multi-step workflow-level pattern. Q2: it can guide a new agent from the start of a similar task. Q3: it is not primarily a local reactive rule.

Output Format

```
{
  "groundedness": 0,
  "reusability": 0,
  "specificity": 0,
  "format_validity": 0,
  "when_to_apply_quality": 0,
  "task_level_granularity": 0,
  "reason": "short explanation citing specific evidence or lack thereof"
}
```

Figure 10: Rubric template for task-level skill quality judgment.

Event-Driven Skill Quality Judge

System Prompt

You are an expert judge for skill-manager outputs. You are strict and critical by default. When in doubt, give a lower score.

Task

Judge a proposed `event-driven` prior-knowledge output. The output should capture one reusable local event and its trigger-response knowledge, rather than a whole-task workflow or instance-specific note.

Evaluation Input

{{TRAJECTORY_EVIDENCE}} and {{PROPOSED_OUTPUT}}.

Scoring Rule

For each dimension, answer the listed sub-questions with yes or no. The score is the number of yes answers.

Dimensions

- **groundedness, 0–3.** Q1: every rule is traceable to observable behavior or outcome. Q2: the trigger appears explicitly in the trajectory. Q3: no rule is contradicted, absent, or weakly hinted.
- **reusability, 0–3.** Q1: rules avoid repository names, file paths, exact error text, or other non-transferable identifiers. Q2: rules apply to the same event type in distinct projects. Q3: the trigger can recur in unseen projects.
- **specificity, 0–3.** Q1: each rule gives a concrete action or check. Q2: rules add event-specific value beyond generic debugging. Q3: rules would not apply to most unrelated local events.
- **format_validity, 0–1.** The output is valid and complete.
- **when_to_apply_quality, 0–3.** Q1: the trigger is specific and recognizable. Q2: it would not fire on unrelated local events. Q3: it is narrow enough to avoid frequent false triggers.
- **event_level_granularity, 0–3.** Q1: the output focuses on one local trigger-response pattern. Q2: it is local and reactive rather than workflow-level. Q3: the trigger is distinguishable from other common events.

Output Format

```
{
  "groundedness": 0,
  "reusability": 0,
  "specificity": 0,
  "format_validity": 0,
  "when_to_apply_quality": 0,
  "event_level_granularity": 0,
  "reason": "short explanation citing specific evidence or lack thereof"
}
```

Figure 11: Rubric template for event-driven skill quality judgment.

Skill Evolution Quality Judge

System Prompt

You are an expert judge for skill-manager outputs. You are strict and critical by default. When in doubt, give a lower score.

Task

Judge a proposed `evolve` update. The output should refine existing prior knowledge using new trajectory evidence, rather than paraphrase the old skill, drift to a new skill, or produce an instance-specific note.

Evaluation Input

{{SYSTEM_PROMPT}}, {{EXISTING_PRIOR_KNOWLEDGE}}, {{TRAJECTORY_EVIDENCE}}, and {{PROPOSED_OUTPUT}}.

Scoring Rule

For each dimension, answer the listed sub-questions with yes or no. The score is the number of yes answers.

Dimensions

- **groundedness, 0–3.** Q1: every changed rule is supported by new evidence. Q2: changed applicability is supported without unjustified broadening. Q3: no new claim is contradicted, absent, or weakly hinted.
- **reusability, 0–3.** Q1: the update avoids one-off identifiers. Q2: changed guidance applies to distinct similar future situations. Q3: applicability lets a future agent decide when not to use the skill.
- **specificity, 0–3.** Q1: changed rules give concrete actions, checks, orderings, or decision criteria. Q2: the update adds value beyond universal debugging. Q3: the guidance is domain-specific enough that unrelated domains would not fit.
- **format_validity, 0–1.** The output is valid and complete.
- **failure_responsiveness, 0–3.** Q1: new evidence reveals a failure, limitation, contradiction, missing case, or caution. Q2: the update directly addresses it. Q3: the update would change a future concrete decision.
- **update_quality, 0–3.** Q1: the update preserves skill identity. Q2: it adds, corrects, narrows, or sharpens non-redundant content. Q3: the final skill is coherent and focused.

Output Format

```
{
  "groundedness": 0,
  "reusability": 0,
  "specificity": 0,
  "format_validity": 0,
  "failure_responsiveness": 0,
  "update_quality": 0,
  "reason": "short explanation"
}
```

Figure 12: Rubric template for skill evolution quality judgment.

Skill-Bank Maintenance Merge Judge

System Prompt

You are an expert judge for skill-manager `maintain` outputs. You are strict and critical by default. When in doubt, give a lower score.
The action under review is `merge`. The model decided to merge the candidate prior knowledge into one specific existing entry, identified by `merge_target_skill_id`, producing a single merged `skill` that replaces the target.
Note: maintain decisions are made without a trajectory. The input is the candidate prior-knowledge item and the existing prior-knowledge entries. Judge the merge purely on the textual evidence of the two sides and the merged result.

Task

Judge a proposed `merge` decision in skill-bank maintenance. The goal is to merge a candidate with one substantially overlapping existing entry, producing a stronger entry that preserves both sides' useful content, avoids becoming an over-general umbrella, and remains high-quality reusable prior knowledge.

Evaluation Input

- {{CANDIDATE_PRIOR_KNOWLEDGE}}
- {{EXISTING_PRIOR_KNOWLEDGE}}
- {{PROPOSED_OUTPUT}}

Scoring Rule

For each dimension, answer the listed sub-questions with yes or no. The score is the number of yes answers. A yes requires clear evidence. When in doubt, answer no.

Dimensions

- **target_choice_correct, 0–3.** Q1: the chosen target's `when_to_apply` directly matches the candidate's trigger condition. Q2: the chosen target has the closest underlying capability among all existing entries. Q3: no other existing entry would be equally good or better.
- **target_overlap_substantive, 0–3.** Q1: candidate and target have at least 30% conceptual rule overlap. Q2: their `when_to_apply` fields describe truly overlapping situations. Q3: keeping both separately would harm the bank through duplicate retrieval, conflict, or noise.
- **merge_integration_quality, 0–3.** Q1: overlapping rules are deduplicated and consolidated. Q2: the merged `when_to_apply` covers both original trigger conditions. Q3: the merged skill is internally consistent.
- **identity_preserved, 0–3.** Q1: the merged title preserves the target's broad capability. Q2: the merged granularity matches or sensibly subsumes the target's granularity. Q3: the merged skill would be retrieved in roughly the same situations as the original target.
- **no_critical_loss_target, 0–3.** Q1: every actionable rule from the original target is preserved, absorbed, or correctly deduplicated. Q2: target-specific orderings, exceptions, or cautions remain represented. Q3: target edge cases and trigger qualifiers remain intact.
- **no_critical_loss_candidate, 0–3.** Q1: every actionable rule from the candidate is preserved, absorbed, or deduplicated. Q2: the candidate's distinctive insights are preserved or strengthened. Q3: candidate-specific trigger conditions are reflected in the merged result.
- **merged_when_to_apply_tightness, 0–3.** Q1: the merged applicability is no broader than the union of candidate and target triggers. Q2: it retains enough cues for a future agent to decide when not to retrieve it. Q3: it contains no over-generalized trigger such as "any build issue" or "any failure with logs".
- **format_validity, 0–1.** The output is structurally valid for a `merge` decision, including a valid `merge_target_skill_id`, a present merged `skill`, and required fields `title`, `granularity`, `when_to_apply`, and non-empty `rules`.

Output Format

```
{
  "target_choice_correct": 0,
  "target_overlap_substantive": 0,
  "merge_integration_quality": 0,
  "identity_preserved": 0,
  "no_critical_loss_target": 0,
  "no_critical_loss_candidate": 0,
  "merged_when_to_apply_tightness": 0,
  "format_validity": 0,
  "reason": "short explanation"
}
```

Figure 13: Rubric template for judging merge decisions in skill-bank maintenance.

Behavior Alignment Judge

System Prompt

You are a strict judge evaluating whether an AI agent's behavior reflects provided prior knowledge. Be conservative when in doubt. Prior knowledge that only restates general best practices is hard to credit.

Task

Judge whether a policy-model rollout actually reflects the provided prior knowledge. Alignment means the agent's concrete actions and reasoning show that the prior knowledge specifically guided behavior, not merely that the agent was competent.

Evaluation Input

{{TASK_CONTEXT}}, {{USER_PROMPT}}, {{PRIOR_KNOWLEDGE}}, {{TRAJECTORY_EVIDENCE}}, and {{RESULT_SUMMARY}}.

Scoring Dimensions

- **when_to_apply_match, 0–2.** A: the task context falls within the stated condition. B: concrete trajectory evidence shows the applicable situation is present.
- **rule_specificity, 0–3.** A: a specific non-trivial rule maps to a concrete action. B: the action is plausibly caused by the prior knowledge rather than generic competence. C: the agent follows the intended spirit of the rule.
- **trajectory_evidence, 0–3.** A: reasoning explicitly reflects the prior knowledge. B: at least two distinct steps align with distinct rules. C: alignment is sustained rather than coincidental.

Important Rules

Do not reward task success, trajectory length, or apparent competence. Generic exploration does not earn credit unless the prior knowledge is specific and distinctive.

Output Format

```
{
  "when_to_apply_match": 0,
  "rule_specificity": 0,
  "trajectory_evidence": 0,
  "reason": "concise summary of the key factors driving your scores"
}
```

Figure 14: Rubric template for behavior alignment judgment.