SKILLSVOTE: Lifecycle Governance of Agent Skills from Collection, Recommendation to Evolution

Anonymous ACL submission

Abstract

Long-horizon LLM agents generate traces that could become reusable experience, but raw trajectories are noisy, local, and hard to govern. Agent Skills offer a structured artifact for combining procedural guidance, executable resources, and applicability boundaries. Yet open skill ecosystems contain redundant, uneven, environment-sensitive artifacts, and indiscriminate updates can pollute future context. We present SKILLSVOTE, a lifecycle-governance framework for Agent Skills across collection, recommendation, attribution, and evolution. SKILLSVOTE profiles a million-scale open-source corpus for environment requirements, quality, and verifiability, and synthesizes tasks for verifiable skills. Before execution, it performs agentic library search over structured skill folders to expose instructional context. After execution, it decomposes trajectories into skill-linked subtasks, attributes outcomes to skill-guided execution, agent exploration, environment, and result signals, and admits only successful reusable discoveries to evidence-gated updates. Experiments on Terminal-Bench 2.0 and SWE-Bench Pro show that SKILLSVOTE improves agent performance on challenging agentic coding benchmarks. The gains arise from two complementary pathways: online evolution over test-time task streams and offline transfer via frozen libraries built from either historical trajectories or curated open-source skills.

1 Introduction

Recent progress in LLM agents has shifted the research focus from single-turn answer generation to systems that act over long horizons. Contemporary benchmarks require agents to repair realistic codebases (Jimenez et al., 2024; Deng et al., 2025), navigate web applications (Zhou et al., 2024), operate across desktop environments (Xie et al., 2024), and manipulate external state through APIs (Trivedi et al., 2024), tools, and terminals (Merrill et al., 2026). These settings produce trajectories of intermediate decisions, tool interactions, and environmental feedback. Prior work on experiential agents shows that such traces can shape later behavior, but only after low-level execution evidence is distilled into reusable experience or skills (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024).

Raw trajectories, however, are a poor long-term substrate for experience reuse. They are lengthy, noisy, tightly bound to local environments, and often conflate robust strategies with incidental state. Agent Skills offer a more structured schema: they package procedural instructions, scripts, templates, references, dependency boundaries, and applicability conditions into auditable artifacts, making experience more compact than full trajectories while preserving more executable context than isolated natural-language summaries (Jiang et al., 2026).

At ecosystem scale, the problem is no longer only how to author an individual skill, but how to control a continuously expanding library. Public skill ecosystems already exhibit scale, redundancy, uneven quality, and safety risks (Ling et al., 2026). Skill benchmarks further show that the benefit of skills depends on task, domain, and retrieval setting; weakly related or low-quality skills can degrade agent performance (Li et al., 2026b; Liu et al., 2026b). Treating skills as ecosystem artifacts also changes the failure mode: larger libraries increase coverage, but they also enlarge the search space and amplify library pollution when weakly supported lessons are incorporated indiscriminately. These observations suggest that large-scale skill ecosystems require collection, governance, profiling, recommendation, evaluation, and evolution to be treated as coupled processes (Li et al., 2026a; Zheng et al., 2026). Against this background, SKILLSVOTE constructs and profiles a million-scale open-source Agent Skill corpus and governs how skills *vote* into the agent context before execution and how attributed evidence *votes*

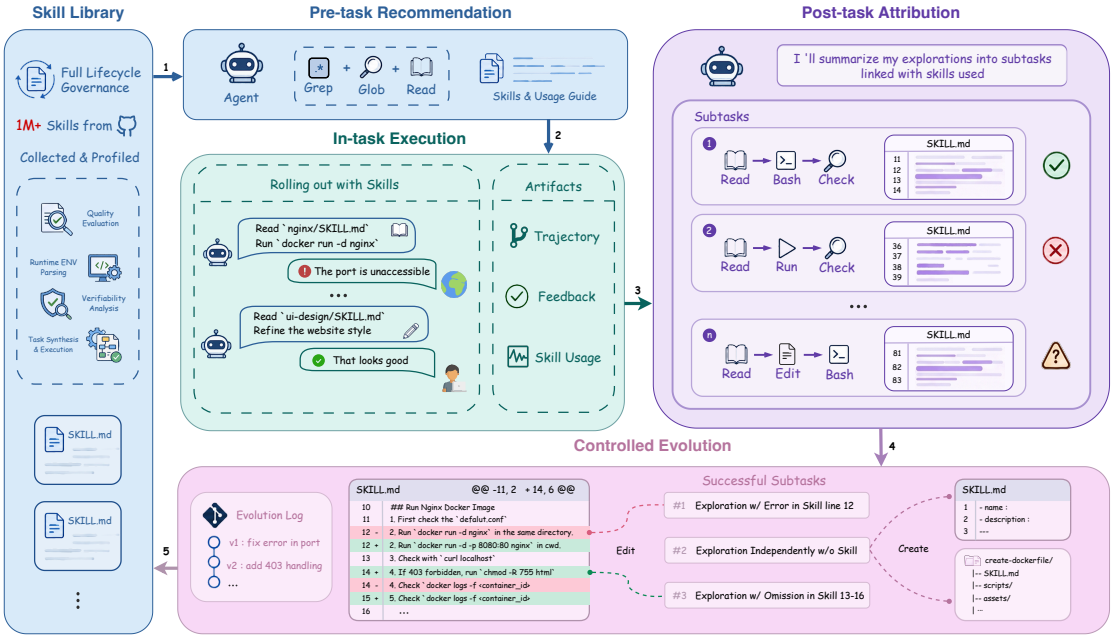

Figure 1: SKILLSVOTE couples pre-task recommendation with post-task attribution and controlled library evolution. A profiled skill library is searched before execution; after execution, trajectories and outcome signals are decomposed into skill-linked subtasks so reusable discoveries can edit existing skills or create new ones.

into the skill library after execution.

This paper introduces SKILLSVOTE, a lifecycle framework for Agent Skills. Before execution, SKILLSVOTE formulates recommendation as agentic search over a structured skill library rather than static semantic matching (Li et al., 2026c); it selects a small, relevant, and low-redundancy skill set and supplies compressed usage context. After execution, SKILLSVOTE performs outcome attribution from the trajectory and visible result signal, explaining how the outcome relates to the selected skill, the agent's own exploration, environmental conditions, and the evaluation signal itself. These stages form the closed loop shown in Figure 1.

Recent skill-evolution systems show that execution evidence can improve skills by distilling trajectory-local lessons into transferable skill directories (Ni et al., 2026), aggregating cross-user interaction trajectories into shared skill updates (Ma et al., 2026), and diagnosing bad cases to refine domain skills (Liu et al., 2026a). SKILLSVOTE uses this evidence under an attribution control layer: it first determines whether a success or failure is attributable to the selected skill, the agent's own exploration, the environment, or the result signal, and then constrains which experience may enter the evolving skill library. This control prevents spurious successes from being rewarded and keeps failures caused by the environment or evaluation signal from driving irrelevant repository edits. Thus, SKILLSVOTE connects recommendation, attribution, and controlled evolution into an auditable closed loop.

We evaluate SKILLSVOTE on Terminal-Bench 2.0 (Merrill et al., 2026) and SWE-Bench Pro (Deng et al., 2025). Our experiments study whether recommendation outperforms directly exposing the initial skill library, whether offline evolution transfers from historical Terminal-Bench Pro trajectories to Terminal-Bench 2.0, and whether online evolution accumulates useful experience in a test-time task stream.

This paper makes the following contributions:

1. We formulate an Agent Skill lifecycle framework that connects open-world collection and governance, recommendation, outcome attribution, and controlled evolution.

2. We construct and profile a million-scale open-source Agent Skill corpus for systematic analysis and governance of open skill ecosystems.

3. We design an attribution-guided recommendation-to-evolution loop that constrains skill library evolution and reduces the risk of indiscriminate library updates.

4. We show that SKILLSVOTE improves online evolution, offline transfer, and recommendation-controlled skill use on Terminal-Bench 2.0 and SWE-Bench Pro public.

2 Related Work

Evolution of Agent Experience Learning. Agent experience learning has progressed from records reusable only in context to executable artifacts. Early memory methods store unstructured cases and examples, such as few-shot trajectories, exemplars, or human-curated interaction records (Zhou et al., 2025; Zheng et al., 2024; Wang et al., 2026c). Workflow methods abstract traces into semi-structured workflows and SOPs (Wang et al., 2025c; Fang et al., 2025), while strategy-level methods compress experience into principles, heuristics, and strategies (Zhao et al., 2024; Ouyang et al., 2026b; Cao et al., 2025; Zhang et al., 2026b; Cai et al., 2025a,b; Wang et al., 2026b). Recent tool, MCP, and skill-learning methods attach experience to callable interfaces, dependencies, and execution boundaries (Lu et al., 2026a; Liu et al., 2025; Huang et al., 2025). Surveys and systems similarly frame memories, rules, skills, protocols, and harness components as deployment-time external artifacts (Zhang et al., 2026c; Zhou et al., 2026a; Lin et al., 2026b; Zhang, 2026; Liang et al., 2026a; Lin et al., 2026a). SKILLSVOTE focuses on skill libraries: a skill combines procedural text, scripts, dependencies, and applicability boundaries, so experience remains auditable, versionable, and portable across harnesses, while full harness or protocol evolution has a larger action space.

Agent Skill Ecosystems, Retrieval and Evaluation. As Agent Skills become installable and shareable file artifacts (Agent Skills, 2026; Anthropic, 2026; OpenAI, 2026d; SkillsMP, 2026; Vercel, 2026; OpenClaw, 2026b; Hermes, 2026; OpenClaw, 2026a), the problem shifts from authoring skills to governing and using open ecosystems. AgentSkillOS (Li et al., 2026a) and SkillNet (Liang et al., 2026b) organize skills as ecosystem objects, while SkillsBench (Li et al., 2026b), SkillCraft (Chen et al., 2026b), SkVM (Chen et al., 2026a), and SkCC (Ouyang et al., 2026c) show that utility, compositional use, portability, security, dependencies, and harness compatibility must be evaluated before `SKILL.md` files can be trusted. SKILLSVOTE profiles open-source skills for format, dependency, quality, and verifiability. At task time, governance does not remove selection: providing skills does not ensure correct selection, composition, or use. SkillRouter learns routing over full skill bodies rather than only names or descriptions (Zheng et al., 2026), while DCI replaces embedding retrieval with direct corpus interaction over source documents (Li et al., 2026c). SKILLSVOTE lightly applies filesystem-native inspection to governed skill folders and outputs compact guidance for combining the selected skills.

Skill-Centric Agent Self Evolution. A growing body of work studies how agents learn and evolve around skill libraries. One line trains policies to decide when to retrieve a skill, how to use it, and when to distill behavior into the model or revise the library (Xia et al., 2026a; Wang et al., 2025a; Xia et al., 2026b; Wang et al., 2026d; Lu et al., 2026b; Shi et al., 2026; Ouyang et al., 2026a). Other systems keep the base model fixed and turn coarse session- or trajectory-level evidence, together with verifier or environment feedback, into reusable skill artifacts (Ni et al., 2026; Alzubi et al., 2026; Zhang et al., 2026a; Ma et al., 2026; Wang et al., 2026a; Yang et al., 2026; Si et al., 2026; Zhang et al., 2026d; Zhou et al., 2026b; Mi et al., 2026; Gong et al., 2026; Xu et al., 2026). SKILLSVOTE further factorizes each trajectory into judged, skill-linked subtasks, localizes the skill knowledge actually used and the responsibility for each outcome, and admits only reusable successful exploration into skill library evolution.

3 Approach

SKILLSVOTE treats Agent Skills as lifecycle artifacts: given a profiled open-source skill library, it controls which skills enter the solver agent context before execution and which execution evidence is allowed to update the library afterward. We describe corpus collection, profiling, and skill-derived task synthesis as preprocessing details in Appendix D.1.

3.1 Skill Recommendation via Agentic Library Search

Existing skill harnesses commonly rely on progressive disclosure: the solver agent first sees lightweight skill metadata, and the full `SKILL.md` and supporting resources are loaded only after the skill appears relevant (OpenAI, 2026a; Anthropic, 2026; Agent Skills, 2026). This design lets many skills coexist in one environment, but it also compresses pre-task selection into short descriptions and limited path cues. SkillRouter further shows that, in large skill pools, the full skill body often carries decisive routing signals (Zheng et al., 2026). SKILLSVOTE builds on this progressive-disclosure

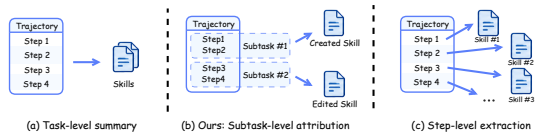

Figure 2: Subtask-level attribution bridges coarse task summaries and fragmented step traces, linking coherent execution segments to skill updates.

interface by adding a task-conditioned exposure-control layer before the solver agent starts execution.

The motivation extends beyond skill systems. Filesystem-native atomic tools are increasingly used as a general interface for agentic search: DCI lets agents interact directly with corpora in deep-research-style retrieval rather than consuming a fixed top-k interface (Li et al., 2026c); code-search systems such as CodeScout (Sutawika et al., 2026) and SWE-grep (Pan et al., 2025) train multi-turn localization over ordinary repository tools; Vercel's data-agent studies compress query exploration and validation into a small set of file and shell operations (Qu, 2025; Goyal and Qu, 2026); and Letta uses a filesystem backend for agent-memory retrieval (Letta, 2025). These examples motivate treating an evolving skill library as a searchable file-based substrate.

Given a task and a profiled skill library, SKILLSVOTE runs a separate recommendation stage. The agent does not solve the task. It searches the local skill library, selectively reads candidate `SKILL.md` files and related resources, and selects skills that cover the task, fit the target environment, and provide complementary guidance. The output is a compact set of exposed skills plus a short usage guide for the solver agent, rather than the full library, a metadata-only routing decision, or a single-step top-k chunk list. The recommendation record also anchors later attribution: after execution, SKILLSVOTE can inspect whether exposed skills were actually used and whether they contributed reusable discoveries.

3.2 Distilling Execution Traces into Evolvable Units

Recent work exposes a granularity gap in learning from agent execution. Agent evaluation commonly relies on task-level success signals, which are authoritative but provide sparse supervision for long-horizon tool use and make credit assignment difficult (Fan et al., 2026). Meanwhile, skill-learning systems show that execution trajectories contain reusable experience that can improve future behavior (Ni et al., 2026; Fang et al., 2026). However, these works also suggest that trajectories must be filtered before they become reusable artifacts: a run may mix skill-guided actions, independent exploration, corrected failures, and redundant operational steps. At the other end, process-level benchmarks (Fan et al., 2026) and failure-diagnosis methods (Barke et al., 2026) annotate individual agent steps, showing the value of local feedback for analysis. However, a single tool call rarely constitutes reusable skill knowledge. As shown in Figure 2, skill evolution thus requires an intermediate unit between full trajectories and individual steps.

SKILLSVOTE addresses this mismatch by inserting a subtask-level attribution layer between full trajectories and individual tool calls. **A subtask is the smallest semantically complete unit that can support library evolution**: it has *one standalone objective*, *one primary evaluation signal*, and *at most one associated skill context*. The primary evaluation signal specifies what kind of evidence can support the subtask outcome, such as environment feedback, human review, or no explicit signal. Trajectories are split only when one of these three boundaries changes, rather than whenever the agent issues another command. This granularity is local enough to assign responsibility, yet abstract enough to capture reusable procedures, constraints, and recovery patterns.

For each subtask, attribution compresses the execution evidence along three axes:

1. **Outcome evidence.** The system records whether the subtask can be assessed by objective environment feedback, depends on human preference, or lacks an explicit evaluation signal. This prevents verifier-backed outcomes, subjective goals, and unsupported claims from being treated alike.

2. **Responsibility assignment.** The system assigns both the final state and its main cause. Successful subtasks may be credited to skill-guided execution, independent exploration, or exploration after observing an irrelevant skill. Failed or uncertain subtasks are retained as diagnostic evidence, but they do not directly authorize skill evolution.

3. **Reusable delta.** For skill-related subtasks, the system localizes the portions of skill knowledge that actually shaped execution, rather than

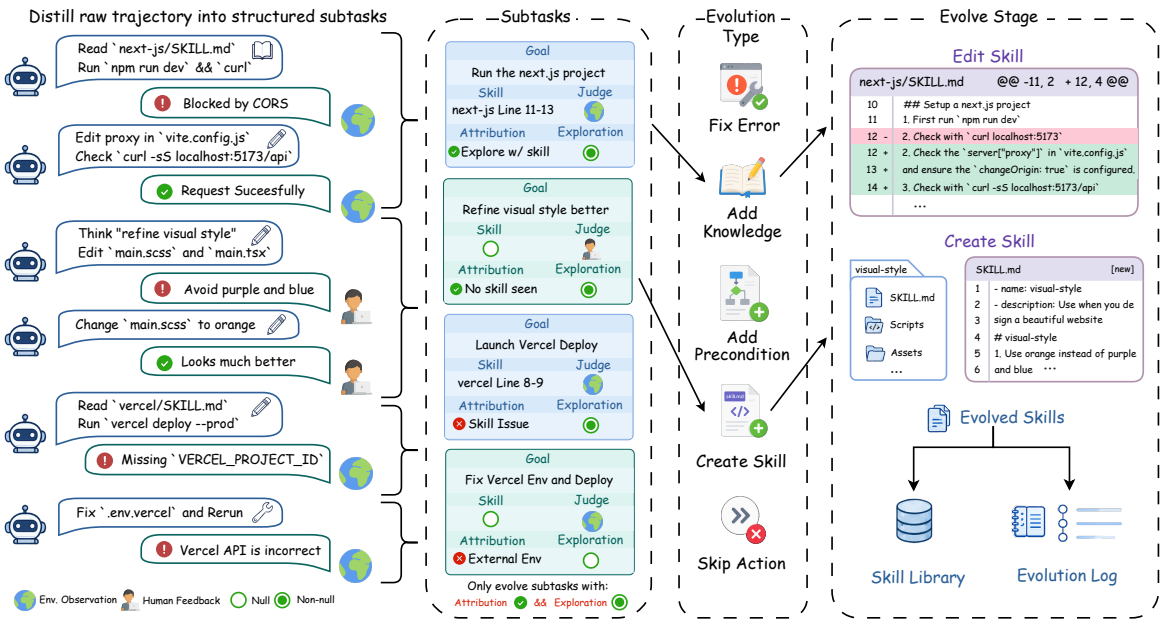

Figure 3: SKILLSVOTE converts execution traces into attributed subtasks, then updates the library only from successful, reusable evidence.

crediting every exposed skill. It also extracts only reusable discoveries, such as missing procedures, preconditions, or recovery patterns, while discarding ordinary trial-and-error, task-specific constants, and repetitive operational details.

Together, these fields define an evolvable unit for the trajectory evidence in Figure 3: evidence-bound, responsibility-aware, and reusable. These units form the interface to controlled evolution, where only successful subtasks with reusable exploration can propose library updates.

3.3 Evidence-Based Controlled Skill Evolution

The attribution layer produces evolvable units, but library evolution still requires explicit control over what evidence is allowed to change persistent skills. SKILLSVOTE formulates this step as evidence-gated update construction with explicit admissibility, aggregation, and routing criteria.

Admissibility. SKILLSVOTE first filters which units may trigger evolution. A unit is admissible only if it is successful and contains reusable exploration. Failed, uncertain, or weakly supported evidence may remain useful for diagnosis, but it cannot directly authorize a skill update.

Aggregation. Admissible units are then grouped before any edit is made. Units that support the same reusable procedure, precondition, workaround, or correction are merged into a single proposed update, so repeated observations strengthen one change rather than producing duplicate or fragmented edits.

Routing. Finally, SKILLSVOTE routes each aggregated evidence group to an update action. If the evidence extends a skill that actually shaped execution, the system edits that skill through the smallest justified change: fixing incorrect guidance, adding missing knowledge, or tightening prerequisites. If the evidence reflects an independent reusable capability outside the current skill boundary, the system creates a new skill. When evidence is weak, redundant, or semantically misaligned with the target skill, it skips evolution.

Thus, skill evolution is conservative by design: every library change must be supported by attributed execution evidence, localized to the relevant skill boundary, and expressed as reusable procedural knowledge rather than a trajectory recap.

4 Experiments

4.1 Experimental Setup

We organize the evaluation around three lifecycle questions:

1. **Offline skill transfer.** Can frozen libraries from historical trajectories or curated open-source skills improve unseen tasks?

2. **Online skill evolution.** Can SKILLSVOTE accumulate useful skills over a sequential task stream?

3. **Pre-task recommendation.** Given a growing skill library, does pre-task recommendation reduce negative transfer?

Benchmarks. We evaluate with Harbor (Harbor Framework Team, 2026) on Terminal-Bench 2.0 (Merrill et al., 2026) and SWE-Bench Pro public (Deng et al., 2025), two challenging agentic coding benchmarks. We report avg@5 Accuracy on Terminal-Bench 2.0 and avg@1 Resolve Rate on SWE-Bench Pro public, following their leaderboard protocols.

Configurations. We run Codex with three model–effort pairs: GPT-5.2 medium (OpenAI, 2025), GPT-5.4 mini medium effort (OpenAI, 2026b), and GPT-5.5 xhigh (OpenAI, 2026c). For each benchmark–backbone pair, we compare three classes of settings: (1) w/o skills is the base solver without an external skill library; (2) Online settings start from an empty experience library and update it along the test-time task stream; we compare SKILLSVOTE with ReasoningBank (Ouyang et al., 2026b) under this protocol; (3) Offline settings start from a frozen skill library and use it only through pre-task recommendation on the test set. TB-Pro is evolved from historical Terminal-Bench Pro (Wang et al., 2025b) task trajectories and is used only for Terminal-Bench 2.0 transfer, while CURATED is a 10K-skill recommendation-only library selected from our 1.68M open-source skill corpus. Appendix C gives the complete configuration details.

4.2 Main Results

Tables 1 and 2 report the main results. Across the reported online settings, SKILLSVOTE consistently improves the overall score over w/o skills. On Terminal-Bench 2.0, it raises avg@5 Accuracy by ↑2.7 pp, ↑1.1 pp, and ↑0.9 pp for GPT-5.2, GPT-5.4 mini, and GPT-5.5, respectively. On SWE-Bench Pro public, it improves avg@1 Resolve Rate by ↑2.6 pp, ↑2.1 pp, and ↑1.2 pp. In the settings where ReasoningBank is reported, it shows a less stable pattern: it improves Terminal-Bench 2.0 with GPT-5.2 by ↑3.4 pp, but decreases the score with GPT-5.4 mini by ↓2.3 pp and reduces SWE-Bench Pro performance for both reported backbones.

Offline results show that frozen libraries can help without test-set evolution. The historical-trajectory library gives the largest Terminal-Bench 2.0 gains, improving GPT-5.2, GPT-5.4 mini, and GPT-5.5 by ↑7.9 pp, ↑5.8 pp, and ↑1.4 pp, respectively. This indicates that trajectory-derived skills capture reusable terminal procedures beyond their source tasks. The curated open-source library also improves the settings where it is evaluated, with ↑2.7 pp and ↑4.0 pp on Terminal-Bench 2.0 for GPT-5.2 and GPT-5.4 mini, respectively, and ↑1.5 pp on SWE-Bench Pro. Overall, the results show that skills help through both online evolution and offline reuse from frozen libraries.

Table 1: Main results on Terminal-Bench 2.0. Scores are avg@5 Accuracy; deltas denote absolute percentage-point changes from the corresponding no-skill baseline.

Settings	Overall (89)	Easy (4)	Medium (55)	Hard (30)
GPT-5.2 medium				
w/o skills	51.0	75.0	54.9	40.7
Online				
SKILLSVOTE	53.7↑2.7	75.0	62.9↑8.0	34.0↓6.7
ReasoningBank	54.4↑3.4	80.0↑5.0	59.6↑4.7	41.3↑0.6
Offline				
TB-Pro	**58.9**↑7.9	**90.0**↑15.0	**65.1**↑10.2	**43.3**↑2.7
Curated	53.7↑2.7	80.0↑5.0	61.1↑6.2	36.7↓4.0
GPT-5.4 mini medium				
w/o skills	51.7	**75.0**	61.8	30.0
Online				
SKILLSVOTE	52.8↑1.1	**75.0**	63.6↑1.8	30.0
ReasoningBank	49.4↓2.3	65.0↓10.0	60.3↓1.5	27.3↓2.7
Offline				
TB-Pro	**57.5**↑5.8	65.0↓10.0	64.7↑2.9	**43.3**↑13.3
Curated	55.7↑4.0	**75.0**	**65.5**↑3.7	35.3↑5.3
GPT-5.5 xhigh				
w/o skills	79.8	90.0	83.0	**72.1**
Online				
SKILLSVOTE	80.7↑0.9	**100.0**↑10.0	**84.9**↑1.9	70.0↓2.1
Offline				
TB-Pro	**81.2**↑1.4	95.0↑5.0	**84.9**↑1.9	**72.1**

4.3 Analysis

The main results show that skill libraries improve agent performance through both online evolution and offline reuse. We next analyze why these gains appear and why they remain moderate on average. We focus on three mechanisms: whether SKILLSVOTE can route over large and confusable skill libraries, whether pre-task recommendation filters harmful skill exposure, and whether offline evolution yields skills that transfer beyond the historical tasks used to build them.

4.3.1 Large-Scale Skill Routing

Before studying downstream task solving, we isolate the skill routing problem itself. We evaluate

Table 2: Main results on SWE-Bench Pro public. Scores are avg@1 Resolve Rate; deltas denote absolute percentage-point changes in the overall column.

Settings	Overall (731)	ansible (96)	openlib. (91)	qutebro. (79)	flipt (85)	telepor. (76)	vuls (62)	navidro. (57)	webclie. (65)	element. (56)	nodebb (44)	tutanot. (20)
GPT-5.2 medium												
w/o skills	47.6	49.0	**64.8**	62.0	32.9	34.2	54.8	49.1	**43.1**	**50.0**	47.7	0.0
Online												
SkillsVote	**50.2**$_{\uparrow 2.6}$	**56.2**	63.7	**68.4**	32.9	**35.5**	**56.5**	45.6	38.5	**50.0**	**72.7**	0.0
ReasoningBank	45.3$_{\downarrow 2.3}$	51.0	60.4	55.7	32.9	34.2	51.6	**50.9**	33.8	48.2	43.2	0.0
GPT-5.4 mini medium												
w/o skills	46.9	52.1	55.0	64.6	31.8	35.5	50.0	50.9	**38.5**	46.4	**61.4**	0.0
Online												
SkillsVote	**49.0**$_{\uparrow 2.1}$	51.0	**59.3**	**68.4**	**32.9**	**38.2**	**56.5**	49.1	**38.5**	**51.8**	**61.4**	0.0
ReasoningBank	44.7$_{\downarrow 2.2}$	54.2	**59.3**	64.6	30.6	34.2	43.5	45.6	33.8	48.2	36.4	0.0
Offline												
Curated	48.4$_{\uparrow 1.5}$	**57.3**	58.2	64.6	**32.9**	36.8	50.0	**56.1**	**38.5**	46.4	56.8	0.0
GPT-5.5 xhigh												
w/o skills	58.4	69.7	64.3	73.4	**36.5**	57.9	**74.2**	**63.2**	**38.5**	57.1	65.9	0.0
Online												
SkillsVote	**59.6**$_{\uparrow 1.2}$	**77.2**	**71.4**	**77.2**	35.3	57.9	69.4	59.6	35.4	57.1	**77.3**	0.0

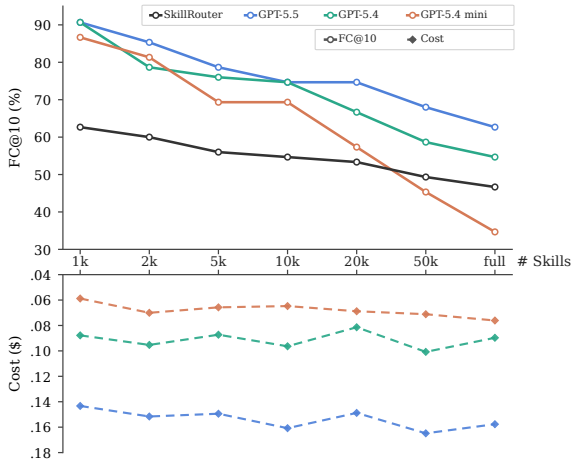

Figure 4: Scaling behavior of skill routing as the candidate library grows from 1k to 79,141 skills. Top: FC@10 across pool sizes. Bottom: average API cost per query for SkillsVote.

Table 3: Main skill-routing results at the largest skill pool. Scores are percentages; cost is averaged per query.

Models	Hit@1 ↑	R@10 ↑	FC@10 ↑	Cost ↓
SkillRouter	65.3	67.2	46.7	–
SkillsVote				
GPT-5.5 xhigh	**70.7**	**74.2**	**62.7**	0.158
GPT-5.4 xhigh	65.3	66.1	54.7	0.090
GPT-5.4 mini xhigh	52.0	48.0	34.7	0.076

on SkillRouter's public set (Zheng et al., 2026), which contains 75 expert-verified queries and a hard pool of 79,141 skills with 780 distractors. We compare the released SkillRouter embedding–reranker pipeline with SkillsVote as the candidate pool scales from 1k to the full library; Appendix C.1 gives the construction and metrics, and Appendix B.1 reports the full results.

Figure 4 summarizes the scaling trend, and Table 3 reports the largest-pool comparison. FC@10 declines with pool size for all methods, reflecting the increasing difficulty of complete coverage in larger and more confusable libraries. With a capable recommender model, however, SkillsVote scales better: GPT-5.5 remains above SkillRouter in FC@10 across all pool sizes and reaches 62.7 at full scale, compared with 46.7 for SkillRouter. At the full pool, GPT-5.4 matches SkillRouter on Hit@1 and improves FC@10 to 54.7. The average API cost changes little from 1k to the full pool, suggesting that the search procedure does not grow proportionally with candidate-library size. GPT-5.4 mini drops at 50k and full scale, indicating that large-library routing still depends on sufficient recommender-model capability.

4.3.2 Recommendation Controls Negative Transfer

Figure 5 measures the effect of task-conditioned recommendation before skill exposure. Directly exposing the online library yields larger negative than positive task-level deltas: the mean gain/loss contribution is $+3.3/-6.7$. Recommendation removes the net negative effect, yielding a balanced $+6.0/-6.0$ profile. In the early online regime, recommendation mainly acts as a noise filter: it pre-

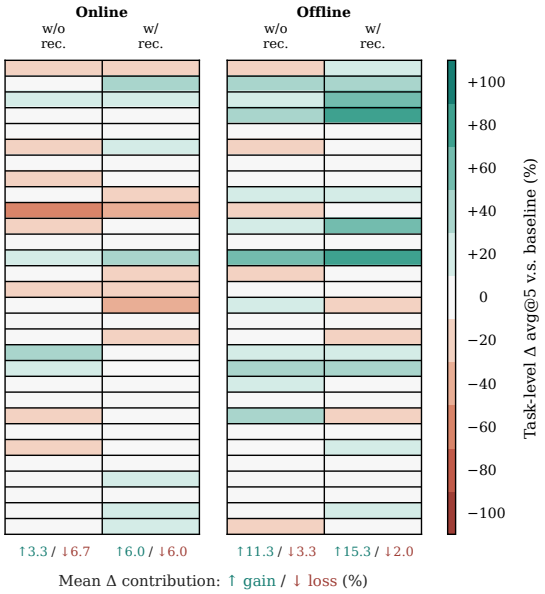

Figure 5: Recommendation controls harmful skill exposure on Terminal-Bench 2.0 Hard; cells show task-level avg@5 deltas over no-skill.

vents sparse, under-specified, or weakly related skills from entering the solver context.

The offline setting gives a cleaner view of the same mechanism. The transferred library is already useful without recommendation, but recommendation increases the mean positive contribution from $+11.3$ to $+15.3$ and reduces the loss from -3.3 to -2.0. Thus, evolution and recommendation play complementary roles. Evolution creates potentially reusable procedural knowledge, while recommendation decides whether that knowledge should be exposed to the current task. This also explains why the average gains in Tables 1 and 2 are moderate despite large improvements on some tasks: skills create a heavy-tailed effect, helping substantially when matched well but causing regressions when exposed indiscriminately.

4.3.3 Offline Evolution Accumulates Transferable Procedures

Offline evolution uses source benchmark feedback for post-task attribution. Ground-truth and verifier signals enter only after task completion, helping determine which parts of the trajectory were successful, reusable, and properly attributable. The evolution stage then consumes attributed subtask records, and reusable exploration excludes benchmark-specific constants or gold outputs.

Figure 6 reflects this separation. Terminal-Bench

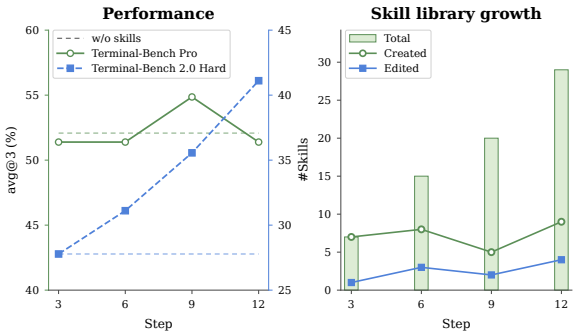

Figure 6: Offline evolution on Terminal-Bench Pro transfers across checkpoints to Terminal-Bench 2.0 Hard (left, avg@3) while the library grows through creations and edits (right).

Pro performance fluctuates across checkpoints, whereas the frozen libraries transfer increasingly well to unseen Terminal-Bench 2.0 Hard tasks. The non-monotonic source-side curve separates source-task performance from transfer-side library utility, and the transfer-side improvement shows reusable operational procedures accumulating across a task distribution shift. The library-growth panel further shows evolution as consolidation: new skills are created, and existing skills are edited as repeated evidence accumulates into persistent skill artifacts.

5 Conclusion

SkillsVote frames Agent Skills as managed lifecycle artifacts for long-horizon agents. It connects a million-scale open-source skill corpus with execution-readiness profiling, task-conditioned recommendation, subtask-level outcome attribution, and evidence-gated evolution. This lifecycle view targets two coupled risks in growing skill libraries: irrelevant skills can distract agents before execution, while weakly supported or misattributed experience can pollute the library after execution. By searching structured skill folders before a task and admitting only successful, reusable, attribution-supported discoveries after a task, SkillsVote turns execution traces into conservative updates to persistent skills. Experiments on Terminal-Bench 2.0 and SWE-Bench Pro show that governed skill libraries improve frozen agents through two complementary routes: online evolution over test-time task streams and recommendation over frozen libraries built from either historical execution trajectories or curated open-source skills. These results position governed skill libraries as a practical substrate for scalable agent experience reuse.

Limitations

The evaluation scope of SKILLSVOTE is concentrated on terminal and software-engineering tasks. Terminal-Bench 2.0, Terminal-Bench Pro, and SWE-Bench Pro provide realistic long-horizon environments with executable feedback, but they do not cover the full range of agent skill use cases, such as open-ended writing, multimodal interaction, embodied control, or tasks whose success depends primarily on human preference. The observed gains therefore support the utility of lifecycle-governed skills in verifier-backed agentic tasks, rather than a claim of universal improvement across all agent domains.

SKILLSVOTE also relies on the quality of post-hoc evidence. The attribution and evolution stages use execution traces, environment feedback, and verifier outcomes to decide which subtasks are successful and reusable. When verifiers are incomplete, when the relevant signal is subjective, or when the trace omits important context, the attribution layer may miss useful experience or assign responsibility imprecisely. Our conservative update policy reduces the chance of incorporating unsupported lessons, but it can also discard potentially useful knowledge from failed or uncertain subtasks.

The results may depend on the underlying agent harness, model, and tool-use policy. Our experiments instantiate SKILLSVOTE with Codex-style agents in Harbor-managed environments and with a limited set of model configurations. Other agents may expose skills differently, search the file system with different competence, or respond differently to additional skill context. Evaluating cross-harness and cross-model transfer remains necessary before treating the evolved library as broadly agent-agnostic.

References

Agent Skills. 2026. Agent Skills. Accessed: 2026-05-12.

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. 2026. Evoskill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766*.

Anthropic. 2026. Extend Claude with Skills. Accessed: 2026-05-12.

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. 2026. Agentrx: Diagnosing ai agent failures from execution trajectories. *arXiv preprint arXiv:2602.02475*.

Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, and 1 others. 2025a. Training-free group relative policy optimization. *arXiv preprint arXiv:2510.08191*.

Zhicheng Cai, Xinyuan Guo, Yu Pei, Jiangtao Feng, Jinsong Su, Jiangjie Chen, Ya-Qin Zhang, Wei-Ying Ma, Mingxuan Wang, and Hao Zhou. 2025b. Flex: Continuous agent evolution via forward learning from experience. *arXiv preprint arXiv:2511.06449*.

Zouying Cao, Jiaji Deng, Li Yu, Weikang Zhou, Zhaoyang Liu, Bolin Ding, and Hai Zhao. 2025. Remember me, refine me: A dynamic procedural memory framework for experience-driven agent evolution. *arXiv preprint arXiv:2512.10696*.

Le Chen, Erhu Feng, Yubin Xia, and Haibo Chen. 2026a. Skvm: Revisiting language vm for skills across heterogenous llms and harnesses. *arXiv preprint arXiv:2604.03088*.

Shiqi Chen, Jingze Gai, Ruochen Zhou, Jinghan Zhang, Tongyao Zhu, Junlong Li, Kangrui Wang, Zihan Wang, Zhengyu Chen, Klara Kaleb, Ning Miao, Siyang Gao, Cong Lu, Manling Li, Junxian He, and Yee Whye Teh. 2026b. Skillcraft: Can LLM agents learn to use tools skillfully? In *First Workshop on Agent Skills*.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, and 1 others. 2025. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.

Shengda Fan, Xuyan Ye, Yupeng Huo, Zhi-Yuan Chen, Yiju Guo, Shenzhi Yang, Wenkai Yang, Shuqi Ye, Jingwen Chen, Haotian Chen, and 1 others. 2026. Agentprocessbench: Diagnosing step-level process quality in tool-using agents. *arXiv preprint arXiv:2603.14465*.

Gaodan Fang, Vatche Isahagian, KR Jayaram, Ritesh Kumar, Vinod Muthusamy, Punleuk Oum, and Gegi Thomas. 2026. Trajectory-informed memory generation for self-improving agent systems. *arXiv preprint arXiv:2603.10600*.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. 2025. Memp: Exploring agent procedural memory. *arXiv preprint arXiv:2508.06433*.

Jingzhi Gong, Ruizhen Gu, Zhiwei Fei, Yazhuo Cao, Lukas Twist, Alina Geiger, Shuo Han, Dominik Sobania, Federica Sarro, and Jie M Zhang. 2026. Skillmoo: Multi-objective optimization of agent skills for software engineering. *arXiv preprint arXiv:2604.09297*.

Ankur Goyal and Andrew Qu. 2026. Testing if "Bash Is All You Need". Accessed: 2026-05-12.

Harbor Framework Team. 2026. Harbor: A framework for evaluating and optimizing agents and models in container environments.

Hermes. 2026. Mastering Hermes Skills. Accessed: 2026-05-12.

Xu Huang, Junwu Chen, Yuxing Fei, Zhuohan Li, Philippe Schwaller, and Gerbrand Ceder. 2025. Cascade: Cumulative agentic skill creation through autonomous development and evolution. *arXiv preprint arXiv:2512.23880*.

Yanna Jiang, Delong Li, Haiyu Deng, Baihe Ma, Xu Wang, Qin Wang, and Guangsheng Yu. 2026. Sok: Agentic skills–beyond tool use in llm agents. *arXiv preprint arXiv:2602.20867*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.

Letta. 2025. Benchmarking AI Agent Memory: Is a Filesystem All You Need? Accessed: 2026-05-12.

Hao Li, Chunjiang Mu, Jianhao Chen, Siyue Ren, Zhiyao Cui, Yiqun Zhang, Lei Bai, and Shuyue Hu. 2026a. Organizing, orchestrating, and benchmarking agent skills at ecosystem scale. *arXiv preprint arXiv:2603.02176*.

Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, and 1 others. 2026b. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*.

Zhuofeng Li, Haoxiang Zhang, Cong Wei, Pan Lu, Ping Nie, Yi Lu, Yuyang Bai, Shangbin Feng, Hangxiao Zhu, Ming Zhong, Yuyu Zhang, Jianwen Xie, Yejin Choi, James Zou, Jiawei Han, Wenhu Chen, Jimmy Lin, Dongfu Jiang, and Yu Zhang. 2026c. Beyond semantic similarity: Rethinking retrieval for agentic search via direct corpus interaction. *arXiv preprint arXiv:2605.05242*.

Jiaqing Liang, Jinyi Han, Weijia Li, Xinyi Wang, Zhoujia Zhang, Zishang Jiang, Ying Liao, Tingyun Li, Ying Huang, Hao Shen, and 1 others. 2026a. Genericagent: A token-efficient self-evolving llm agent via contextual information density maximization (v1. 0). *arXiv preprint arXiv:2604.17091*.

Yuan Liang, Ruobin Zhong, Haoming Xu, Chen Jiang, Yi Zhong, Runnan Fang, Jia-Chen Gu, Shumin Deng, Yunzhi Yao, Mengru Wang, and 1 others. 2026b. Skillnet: Create, evaluate, and connect ai skills. *arXiv preprint arXiv:2603.04448*.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Xuanjing Huang, Hang Yan, Zhenhua Han, and Tao Gui. 2026a. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*.

Minhua Lin, Hanqing Lu, Zhan Shi, Bing He, Rui Mao, Zhiwei Zhang, Zongyu Wu, Xianfeng Tang, Hui Liu, Zhenwei Dai, and 1 others. 2026b. Position: Agentic evolution is the path to evolving llms. *arXiv preprint arXiv:2602.00359*.

George Ling, Shanshan Zhong, and Richard Huang. 2026. Agent skills: A data-driven analysis of claude skills for extending large language model functionality. *arXiv preprint arXiv:2602.08004*.

Jiarun Liu, Shiyue Xu, Yang Li, Shangkun Liu, Yongli Yu, and Peng Cao. 2025. Unifying dynamic tool creation and cross-task experience sharing through cognitive memory architecture. *arXiv preprint arXiv:2512.11303*.

Xingyan Liu, Xiyue Luo, Linyu Li, Ganghong Huang, Jianfeng Liu, and Honglin Qiao. 2026a. Skillforge: Forging domain-specific, self-evolving agent skills in cloud technical support. *arXiv preprint arXiv:2604.08618*.

Yujian Liu, Jiabao Ji, Li An, Tommi Jaakkola, Yang Zhang, and Shiyu Chang. 2026b. How well do agentic skills work in the wild: Benchmarking llm skill usage in realistic settings. *arXiv preprint arXiv:2604.04323*.

Jiaxuan Lu, Ziyu Kong, Yemin Wang, Rong Fu, Haiyuan Wan, Cheng Yang, Wenjie Lou, Haoran Sun, Lilong Wang, Yankai Jiang, and 1 others. 2026a. Beyond static tools: Test-time tool evolution for scientific reasoning. *arXiv preprint arXiv:2601.07641*.

Zhengxi Lu, Zhiyuan Yao, Jinyang Wu, Chengcheng Han, Qi Gu, Xunliang Cai, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. 2026b. Skill0: In-context agentic reinforcement learning for skill internalization. *arXiv preprint arXiv:2604.02268*.

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. 2026. Skillclaw: Let skills evolve collectively with agentic evolver. *arXiv preprint arXiv: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, and 1 others. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*.

Qirui Mi, Zhijian Ma, Mengyue Yang, Haoxuan Li, Yisen Wang, Haifeng Zhang, and Jun Wang. 2026. Skill-pro: Learning reusable skills from experience via non-parametric ppo for llm agents. *arXiv preprint arXiv:2602.01869*.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Xiaoxi Jiang, and Guanjun Jiang. 2026. Trace2skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*.

OpenAI. 2025. Introducing GPT-5.2. Accessed: 2026-05-12.

OpenAI. 2026a. Agent Skills – Codex. Accessed: 2026-05-12.

OpenAI. 2026b. Introducing GPT-5.4 mini and nano. Accessed: 2026-05-12.

OpenAI. 2026c. Introducing GPT-5.5. https://openai.com/index/introducing-gpt-5-5/. Accessed: 2026-05-12.

OpenAI. 2026d. Skills in ChatGPT. Accessed: 2026-05-12.

OpenClaw. 2026a. ClawHub: Skill Directory for OpenClaw. Accessed: 2026-05-12.

OpenClaw. 2026b. Skills – OpenClaw. Accessed: 2026-05-12.

Siru Ouyang, Jun Yan, Yanfei Chen, Rujun Han, Zifeng Wang, Bhavana Dalvi Mishra, Rui Meng, Chun-Liang Li, Yizhu Jiao, Kaiwen Zha, and 1 others. 2026a. Skillos: Learning skill curation for self-evolving agents. *arXiv preprint arXiv:2605.06614*.

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

Yipeng Ouyang, Yi Xiao, Yuhao Gu, and Xianwei Zhang. 2026c. Skcc: Portable and secure skill compilation for cross-framework llm agents. *arXiv preprint arXiv:2605.03353*.

Ben Pan, Carlo Baronio, Albert Tam, Pietro Marsella, Mokshit Jain, Daniel Chiu, Swyx, and Silas Alberti. 2025. Introducing SWE-grep and SWE-grep-mini: RL for Multi-Turn, Fast Context Retrieval. Accessed: 2026-05-12.

Andrew Qu. 2025. We Removed 80% of Our Agent's Tools. Accessed: 2026-05-12.

Yaorui Shi, Yuxin Chen, Zhengxi Lu, Yuchun Miao, Shugui Liu, Qi Gu, Xunliang Cai, Xiang Wang, and An Zhang. 2026. Skill1: Unified evolution of skill-augmented agents via reinforcement learning. *arXiv preprint arXiv:2605.06130*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

Shuzheng Si, Haozhe Zhao, Yu Lei, Qingyi Wang, Dingwei Chen, Zhitong Wang, Zhenhailong Wang, Kangyang Luo, Zheng Wang, Gang Chen, and 1 others. 2026. From context to skills: Can language models learn from context skillfully? *arXiv preprint arXiv:2604.27660*.

SkillsMP. 2026. Agent Skills Marketplace. Accessed: 2026-05-12.

Lintang Sutawika, Aditya Bharat Soni, Apurva Gandhi, Taha Yassine, Sanidhya Vijayvargiya, Yuchen Li, Xuhui Zhou, Yilin Zhang, Leander Melroy Maben, Graham Neubig, and 1 others. 2026. Codescout: An effective recipe for reinforcement learning of code search agents. *arXiv preprint arXiv:2603.17829*.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076.

Vercel. 2026. The Agent Skills Directory. Accessed: 2026-05-12.

Chenxi Wang, Zhuoyun Yu, Xin Xie, Wuguannan Yao, Runnan Fang, Shuofei Qiao, Kexin Cao, Guozhou Zheng, Xiang Qi, Peng Zhang, and 1 others. 2026a. Skillx: Automatically constructing skill knowledge bases for agents. *arXiv preprint arXiv:2604.04804*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*.

Jiongxiao Wang, Qiaojing Yan, Yawei Wang, Yijun Tian, Soumya Smruti Mishra, Zhichao Xu, Megha Gandhi, Panpan Xu, and Lin Lee Cheong. 2025a. Reinforcement learning for self-improving agent with skill library. *arXiv preprint arXiv:2512.17102*.

Junjie Wang, Yiming Ren, and Haoyang Zhang. 2026b. From procedural skills to strategy genes: Towards experience-driven test-time evolution. *arXiv preprint arXiv:2604.15097*.

Qihao Wang, Ziming Cheng, Shuo Zhang, Fan Liu, Rui Xu, Heng Lian, Kunyi Wang, Xiaoming Yu, Jianghao Yin, Sen Hu, and 1 others. 2026c. Memgovern: Enhancing code agents through learning from governed human experiences. *arXiv preprint arXiv:2601.06789*.

Weixun Wang, XiaoXiao Xu, Wanhe An, Fangwen Dai, Wei Gao, Yancheng He, Ju Huang, Qiang Ji, Hanqi Jin, Xiaoyang Li, and 1 others. 2025b. Let it flow: Agentic crafting on rock and roll, building the rome model within an open agentic learning ecosystem. *arXiv preprint arXiv:2512.24873*.

Yinjie Wang, Xuyang Chen, Xiaolong Jin, Mengdi Wang, and Ling Yang. 2026d. Openclaw-rl: Train any agent simply by talking. *arXiv preprint arXiv:2603.10165*.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2025c. Agent workflow memory. In *International Conference on Machine Learning*, pages 63897–63911. PMLR.

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. 2026a. SkillRL: Evolving agents via recursive skill-augmented reinforcement learning. In *ICLR 2026 Workshop on Lifelong Agents: Learning, Aligning, Evolving*.

Peng Xia, Jianwen Chen, Xinyu Yang, Haoqin Tu, Jiaqi Liu, Kaiwen Xiong, Siwei Han, Shi Qiu, Haonian Ji, Yuyin Zhou, and 1 others. 2026b. Metaclaw: Just talk–an agent that meta-learns and evolves in the wild. *arXiv preprint arXiv:2603.17187*.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, and 1 others. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094.

Binyan Xu, Dong Fang, Haitao Li, and Kehuan Zhang. 2026. From multi-agent to single-agent: When is skill distillation beneficial? *arXiv preprint arXiv:2604.01608*.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, and 1 others. 2026. Autoskill: Experience-driven lifelong learning via skill self-evolution. *arXiv preprint arXiv:2603.01145*.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, and 1 others. 2026a. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*.

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

Wentao Zhang. 2026. Autogenesis: A self-evolving agent protocol. *arXiv preprint arXiv:2604.15034*.

Xing Zhang, Guanghui Wang, Yanwei Cui, Wei Qiu, Ziyuan Li, Bing Zhu, and Peiyang He. 2026c. Experience compression spectrum: Unifying memory, skills, and rules in llm agents. *arXiv preprint arXiv:2604.15877*.

Ziao Zhang, Kou Shi, Shiting Huang, Avery Nie, Yu Zeng, Yiming Zhao, Zhen Fang, Qishen Su, Haibo Qiu, Wei Yang, and 1 others. 2026d. Skillflow: Benchmarking lifelong skill discovery and evolution for autonomous agents. *arXiv preprint arXiv:2604.17308*.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19632–19642.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. 2024. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *International Conference on Learning Representations*, volume 2024, pages 19036–19066.

YanZhao Zheng, ZhenTao Zhang, Chao Ma, YuanQiang Yu, JiHuan Zhu, Baohua Dong, and Hangcheng Zhu. 2026. Skillrouter: Skill routing for llm agents at scale. *arXiv preprint arXiv:2603.22455*.

Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, and 1 others. 2026a. Externalization in llm agents: A unified review of memory, skills, protocols and harness engineering. *arXiv preprint arXiv:2604.08224*.

Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, and 1 others. 2025. Memento: Fine-tuning llm agents without fine-tuning llms. *arXiv preprint arXiv:2508.16153*.

Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, and 1 others. 2026b. Memento-skills: Let agents design agents. *arXiv preprint arXiv:2603.18743*.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, and 1 others. 2024. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pages 15585–15606.

A Extended Analysis

A.1 Skill Characteristics

Skill quality and verifiability. During profiling, SKILLSVOTE evaluates skills with explicit quality and verifiability rubrics defined in Tables 8 and 9. Figure 7 shows that the retained corpus is strongly positive on consistency and orientation, while completeness, success verifiability, and task constructability are more selective; unknown values also remain much more common for verifiability than for quality. This suggests that many open skills are coherent reusable guidance units, but a smaller fraction can be turned directly into low-ambiguity, benchmark-constructable tasks, which is why SKILLSVOTE separates general corpus governance from stricter downstream task-synthesis requirements.

Skill categories. After format validation, content deduplication, and profiling, we obtain category statistics for more than 290K skills, with the category definitions listed in Table 10. Figure 8 shows that Development dominates the governed corpus, followed by AgentMeta and Tools, and that the strongest co-occurrence links connect development skills to higher-level agent guidance. The visible links from Development to DevOps and Testing further suggest that open skills are organized primarily around executable software workflows rather than standalone knowledge artifacts. Skills with unknown category assignments are excluded from the plot.

Skill filtration. After collecting more than 1.68M skills, SKILLSVOTE progressively applies format validation, deduplication, and quality analysis, each stage removing a substantial portion of the raw open-source corpus; the procedural details are given in Appendix E.1. Figure 9 shows that the repository-star distribution remains heavily concentrated in the long tail below 100 stars throughout all curation stages, while stricter filtering shifts only a limited share toward higher-star repositories. This suggests that repository popularity alone is not a reliable proxy for whether a skill is valid, non-redundant, or execution-ready, which further motivates explicit corpus governance instead of

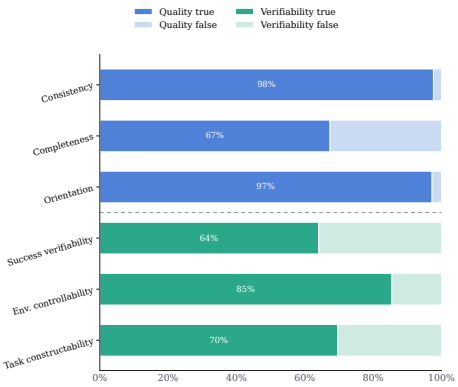

Figure 7: Distribution of quality and verifiability signals among skills that are validated, non-duplicated, and evaluated. Unknown values are excluded.

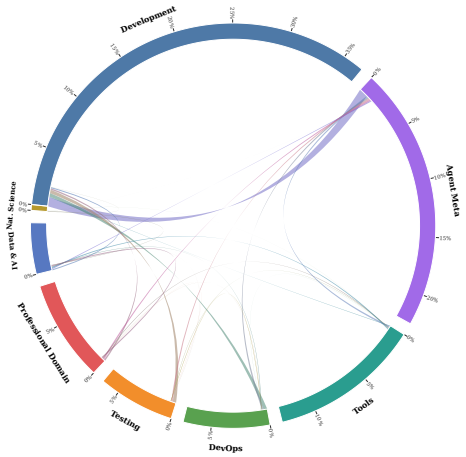

Figure 8: Chord diagram of category co-occurrence among skills that are evaluated, validated, and content-deduplicated.

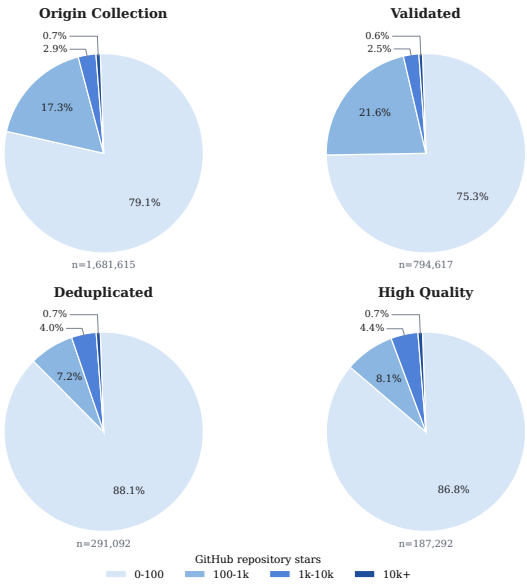

Figure 9: Distribution of skills across GitHub repository star buckets at each curation stage. The panels show the stage-wise composition of the original collection, the validated subset, the deduplicated subset, and the final high-quality subset.

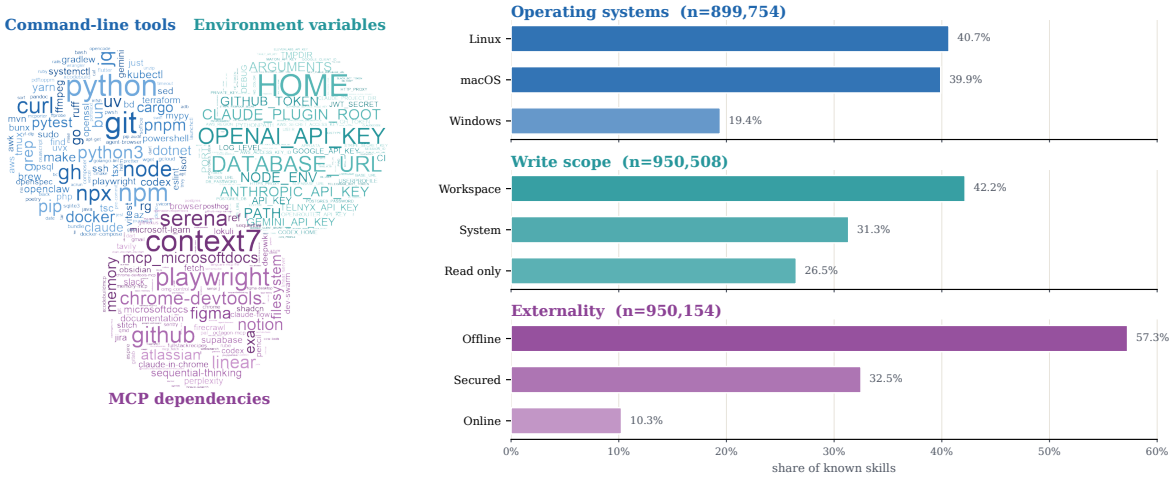

Figure 10: Skill runtime requirement in the skill corpus. Left: the dominant dependency vocabularies across command-line tools, environment variables, and MCPs; Right: the distribution of execution environments.

star-based selection heuristics.

Skill names. Figure 11 summarizes the 20 most frequently repeated `skill_name` entries in the open-skill ecosystem. The distribution is highly concentrated: `skill-creator` appears more than 5K times, while `frontend-design` appears more than 3K times. This concentration suggests that the open-skill ecosystem repeatedly reuses a compact naming vocabulary centered on meta-skill authoring and frontend-oriented workflows, with the remaining high-frequency names spreading into debugging, testing, and document-production tasks.

Figure 11: Word cloud of the 20 most frequently repeated skill_name entries in the open-skill ecosystem.

Skill runtime requirements. Figure 10 summarizes the runtime-requirement profile of the million-scale open-source skill corpus. Linux and macOS dominate the operating-system profile, workspace and system write scopes are more common than read-only execution, and offline or secured settings outweigh open online access. Together with the frequent `bin`, `env`, and MCP dependencies, these patterns suggest that users mainly rely on agents for programming and automation workflows that require real toolchains, runtime configuration, and controlled external systems rather than model-only knowledge.

B Extended Results

B.1 SkillRouter Extended Routing Results

Table 4 reports the full SkillRouter routing results over all skill pool sizes. Appendix C.1 specifies the split construction and evaluation protocol.

All methods decline as the candidate pool grows, but the decline is uneven across metrics and model sizes. FC@10 is the most sensitive metric because it requires every ground-truth skill to appear in the top 10, making it stricter than Hit@1 for the majority multi-skill queries. GPT-5.5 maintains the strongest FC@10 across all pool sizes and remains above SkillRouter by 16.0 points at the full pool with 79,141 skills. GPT-5.4 is competitive through the full pool, matching SkillRouter on Hit@1 at full scale while improving FC@10 by 8.0 points. GPT-5.4 mini performs well at small and medium pools but drops sharply at 50k and full scale, suggesting that this routing strategy depends on model capability when the skill library becomes highly confusable.

Table 4: Extended skill-routing results under different skill-pool sizes and distractor counts. Scores are percentages; cost is averaged per query.

Methods	Overall				Single-Skill			Multi-Skill		
	Hit@1	R@10	FC@10	Cost	Hit@1	R@10	FC@10	Hit@1	R@10	FC@10
Total 1000 w/ 100 distractors										
SkillRouter	84.0	82.9	62.7	–	**91.7**	**100.0**	**100.0**	80.4	74.9	45.1
SKILLSVOTE										
GPT-5.5 xhigh	**92.0**	**96.6**	**90.7**	0.143	**91.7**	**100.0**	**100.0**	**92.2**	**94.9**	**86.3**
GPT-5.4 xhigh	90.7	96.0	**90.7**	0.088	87.5	**100.0**	**100.0**	**92.2**	94.1	**86.3**
GPT-5.4 mini xhigh	89.3	95.2	86.7	0.059	83.3	**100.0**	**100.0**	**92.2**	92.9	80.4
Total 2000 w/ 200 distractors										
SkillRouter	81.3	80.8	60.0	–	**87.5**	**100.0**	**100.0**	78.4	71.8	41.2
SKILLSVOTE										
GPT-5.5 xhigh	**88.0**	**93.6**	**85.3**	0.152	**87.5**	**100.0**	**100.0**	**88.2**	**90.5**	**78.4**
GPT-5.4 xhigh	81.3	88.1	78.7	0.095	79.2	95.8	95.8	82.4	84.5	70.6
GPT-5.4 mini xhigh	78.7	91.1	81.3	0.070	75.0	**100.0**	**100.0**	80.4	86.9	72.5
Total 5000 w/ 500 distractors										
SkillRouter	77.3	77.5	56.0	–	**83.3**	**100.0**	**100.0**	74.5	66.9	35.3
SKILLSVOTE										
GPT-5.5 xhigh	**89.3**	**90.2**	**78.7**	0.149	**83.3**	95.8	95.8	**92.2**	**87.6**	**70.6**
GPT-5.4 xhigh	84.0	87.2	76.0	0.087	75.0	95.8	95.8	88.2	83.2	66.7
GPT-5.4 mini xhigh	81.3	81.7	69.3	0.066	79.2	87.5	87.5	82.4	79.0	60.8
Total 10000 w/ 780 distractors										
SkillRouter	76.0	74.8	54.7	–	79.2	**100.0**	**100.0**	74.5	63.0	33.3
SKILLSVOTE										
GPT-5.5 xhigh	81.3	85.9	**74.7**	0.161	79.2	95.8	95.8	82.4	81.2	**64.7**
GPT-5.4 xhigh	**84.0**	**87.8**	**74.7**	0.096	**83.3**	95.8	95.8	**84.3**	**84.0**	**64.7**
GPT-5.4 mini xhigh	74.7	82.3	69.3	0.065	75.0	91.7	91.7	74.5	77.9	58.8
Total 20000 w/ 780 distractors										
SkillRouter	72.0	72.8	53.3	–	70.8	**100.0**	**100.0**	72.5	60.0	31.4
SKILLSVOTE										
GPT-5.5 xhigh	**78.7**	**83.3**	**74.7**	0.149	**75.0**	87.5	87.5	80.4	**81.4**	**68.6**
GPT-5.4 xhigh	**78.7**	79.5	66.7	0.081	70.8	83.3	83.3	**82.4**	77.7	58.8
GPT-5.4 mini xhigh	64.0	72.4	57.3	0.069	66.7	75.0	75.0	62.7	71.2	49.0
Total 50000 w/ 780 distractors										
SkillRouter	68.0	68.1	49.3	–	**66.7**	**91.7**	**91.7**	68.6	57.0	29.4
SKILLSVOTE										
GPT-5.5 xhigh	**73.3**	**80.2**	**68.0**	0.165	**66.7**	**91.7**	**91.7**	**76.5**	**74.8**	**56.9**
GPT-5.4 xhigh	65.3	73.7	58.7	0.101	54.2	75.0	75.0	70.6	73.1	51.0
GPT-5.4 mini xhigh	60.0	60.3	45.3	0.071	54.2	62.5	62.5	62.7	59.3	37.3
Total 79141 w/ 780 distractors										
SkillRouter	65.3	67.2	46.7	–	62.5	**91.7**	**91.7**	66.7	55.8	25.5
SKILLSVOTE										
GPT-5.5 xhigh	**70.7**	**74.2**	**62.7**	0.158	**66.7**	83.3	83.3	**72.5**	**70.0**	**52.9**
GPT-5.4 xhigh	65.3	66.1	54.7	0.090	50.0	62.5	62.5	**72.5**	67.8	51.0
GPT-5.4 mini xhigh	52.0	48.0	34.7	0.076	50.0	50.0	50.0	52.9	47.1	27.5

C Experiment Details

Common settings. Unless noted otherwise, within a given benchmark, SKILLSVOTE and all baselines share the same agent harness, solver configuration, timeout policy, retry policy, and verifier settings. All runs use Codex CLI 0.125.0 with three model-effort pairs: GPT-5.2 with medium reasoning effort, GPT-5.4 mini with medium reasoning effort, and GPT-5.5 with xhigh reasoning effort. We also disable built-in system skills and plugins for solver agent to reduce the impact of redundant context.

C.1 SkillRouter

Dataset. The large-scale routing study uses SkillRouter's public evaluation set (Zheng et al., 2026), which contains 75 expert-verified task queries after filtering ambiguous mappings: 24 single-skill queries and 51 multi-skill queries. We use the released Hard pool, which contains 79,141 skills including 780 LLM-generated distractors. For scoring, the expert-verified ground-truth skill set for each query defines the target skills; auxiliary annotations are retained in the data files but are not used in the reported metrics.

Split	Skills	Distractors
1k	1,000	100
2k	2,000	200
5k	5,000	500
10k	10,000	780
20k	20,000	780
50k	50,000	780
Full	79,141	780

Table 5: Skill pool sizes for the SkillRouter routing study.

Skill pool splits. Each evaluated pool contains all ground-truth skills for the 75 queries, the listed number of distractors, and additional non-ground-truth skills from the released pool until reaching the target size. This keeps every query scorable at each scale while increasing the size and confusability of the candidate library. Table 5 reports the split sizes.

SkillRouter baseline. The SkillRouter baseline uses the released 0.6B embedding model and 0.6B reranker. Skill documents are encoded from the full skill text, formatted as name, description, and body. For each query, we retrieve the top 50 skills by embedding similarity, pass the top 20 retrieved skills to the reranker, and score the final top 10. We report API cost for SKILLSVOTE only.

SKILLSVOTE routing configuration. For SKILLSVOTE, each candidate skill is represented as one markdown file in the candidate directory. SKILLSVOTE runs in a read-only Codex environment rooted at this directory, with web search, built-in skills, plugins, writes, and command approvals disabled. We evaluate GPT-5.5, GPT-5.4, and GPT-5.4 mini with xhigh reasoning effort. The model must return exactly 10 existing skill files ranked by relevance; no downstream task execution is performed. Figures 22 and 23 report the prompt pair. Cost is computed from input, cached-input, and output tokens using LiteLLM pricing and then averaged per completed query.

Metrics. We report Hit@1, Recall@10, and FC@10. Hit@1 is 1 when any ground-truth skill appears at rank 1. Recall@10 is the fraction of ground-truth skills appearing in the top 10. FC@10 is 1 only when every ground-truth skill appears in the top 10. Single-skill and multi-skill slices are defined by the number of ground-truth skills for a query.

C.2 Terminal-Bench 2.0

Dataset. Terminal-Bench 2.0 contains 89 terminal tasks inspired by real workflows, including 4 Easy, 55 Medium, and 30 Hard instances (Merrill et al., 2026). The dataset contains 16 task categories; representative categories include software engineering, debugging, security, machine learning, scientific computing, and other terminal-centric technical workflows. Each task provides a unique container environment, a human-written reference solution, and executable verification tests.

Configuration. We allow up to 3 retries for environment-related failures, primarily Docker sandbox startup or runtime errors and external API timeouts. We do not retry the post-execution acceptance stage: verifier timeouts, verifier runtime failures are treated as final trial outcomes because they are more likely to reflect agent capability or solution quality than transient noise. We set the agent timeout to 4 times the benchmark default to absorb long command waits and API instability, while leaving all other settings at their benchmark defaults.

For GPT-5.5 with xhigh reasoning effort on Terminal-Bench 2.0, OpenAI's cyber-risk review blocks requests for `password-recovery`, `crack-7z-hash`, `vulnerable-secret`, and `model-extraction-relu-logits`. We therefore skip these four tasks in every GPT-5.5 setting on this benchmark and exclude them from the final averages, so all reported GPT-5.5 results on Terminal-Bench 2.0 are computed over the remaining 85 tasks.

Evaluation metric. We report the leaderboard's avg@5 Accuracy. Accuracy indicates whether a task is solved, and avg@5 averages the success rate over five independent runs per task before averaging across tasks.

Offline evolution. For the main offline-transfer experiment, we use 48 public software-engineering and system-administration tasks from Terminal-Bench Pro (Wang et al., 2025b), excluding 2 environment-unstable tasks. We run one full pass over these 48 tasks with recommendation and evolution both enabled, and trigger evolution after every 4 tasks. During this offline collection phase, ground truth is provided only to assist attribution decisions through the optional oracle prompt in Figure 25; it is not exposed to the solver and is not written directly into the evolved skills. After

collecting offline experience, we transfer the resulting frozen skill library to all 89 Terminal-Bench 2.0 tasks for recommendation only. We repeat this transfer evaluation 5 times and report avg@5 Accuracy.

Offline recommendation. We also evaluate a recommendation-only offline setting that starts from a curated skill library. This library contains approximately 10k curated skills selected by the SkillsVote collecting-and-profiling pipeline from open-source skills. The selected skills pass format validation, deduplication, and quality analysis, and are further restricted to skills sourced from GitHub repositories with more than 1k stars. This setting performs recommendation only, with no evolution. We transfer the curated library directly to all 89 Terminal-Bench 2.0 tasks, run each task 5 times independently, and report avg@5 Accuracy.

Online evolution. Online evolution starts from an empty skill library with task-level recommendation and evolution enabled. Tasks follow the default Terminal-Bench 2.0 order, and one sequential pass over all 89 tasks forms one job. We run 5 jobs independently and report avg@5 Accuracy. Skill libraries are fully isolated across jobs.

Baselines. The no-skill baseline removes the experience library and all additional mechanisms, leaving only the solver agent to act in the container; it is also evaluated over 5 runs and reported as avg@5 Accuracy. For ReasoningBank, we reproduce the open-source method under the same timeout and retry settings. Retries do not update memory; following ReasoningBank's original protocol, memory is updated only after a task reaches final success or final failure. We use the same online protocol as above, with 5 independent jobs and isolated memory banks.

C.3 SWE-Bench Pro

Dataset. The public split of SWE-Bench Pro contains 731 human-verified, human-augmented long-horizon software-engineering tasks from 11 public GPL repositories (Deng et al., 2025). These repositories span business applications, B2B services, and developer tools. In the released dataset, the 11 repositories fall into four language groups according to the repo_language field: Go contributes 280 tasks from flipt-io/flipt, future-architect/vuls, gravitational/teleport, and navidrome/navidrome; Python contributes 266 tasks from ansible/ansible, internetarchive/openlibrary, and qutebrowser/qutebrowser; JavaScript contributes 165 tasks from NodeBB/NodeBB, element-hq/element-web, and protonmail/webclients; TypeScript contributes 20 tasks from tutao/tutanota.

Configuration. We use exactly the same retry mechanism and timeout multipliers as in Terminal-Bench 2.0.

Evaluation metric. We report avg@1 Resolve Rate. Resolve Rate indicates whether the single rollout for a task produces a patch that passes the verifier; because each task is run once, avg@1 is simply the average solve rate across tasks.

Offline. The offline setting matches the offline-recommendation setting used on Terminal-Bench 2.0. It starts from the same curated skill library of approximately 10k skills produced by the SkillsVote collecting-and-profiling pipeline, after format validation, deduplication, quality analysis, and filtering to skills sourced from GitHub repositories with more than 1k stars. This setting performs recommendation only, with no evolution. We evaluate all 731 public tasks once and report avg@1 Resolve Rate.

Online evolution. Online evolution starts from an empty skill library with task-level recommendation and evolution enabled. Instead of treating the entire benchmark as one stream, we order tasks by repository and run each repository as an independent job. This yields 11 jobs, one per repository, which improves evaluation efficiency and better reflects experience reuse within a single codebase. We execute all repositories once and report overall avg@1 Resolve Rate across the 731 tasks. Skill libraries do not carry across repositories.

Baselines. The no-skill baseline runs once on the public split and reports avg@1 Resolve Rate. For ReasoningBank, we use the same timeout and retry settings as above; retries do not update memory, and memory is updated only after the task ends in final success or final failure. We follow the same repository-level online protocol as above, so each repository job evolves its memory bank independently.

D Approach Details

D.1 Open-Source Skill Corpus and Profiling

D.1.1 Collecting a Million-Scale Agent Skill Corpus

Open Agent Skill ecosystems have reached marketplace scale. SkillsMP (SkillsMP, 2026) and skills.sh (Vercel, 2026) aggregate GitHub `SKILL.md` packages and expose search, categories, popularity, and installation-based discovery signals. Yet discovery metadata is not execution evidence: names, descriptions, and popularity do not reveal runtime fit, resource completeness, coherent scope, or objective checkability. Recent benchmarks likewise show that skill utility depends on the task, domain, and corpus quality (Li et al., 2026b; Zhang et al., 2026d). SKILLSVOTE therefore collects a million-scale corpus from GitHub and treats each skill as a directory-level artifact, preserving `SKILL.md` plus optional `scripts/`, `references/`, and `assets/` as the governance unit. Appendix E.1 details validation and deduplication.

D.1.2 Profiling Skill Requirements, Quality, and Verifiability

SKILLSVOTE turns this raw corpus into execution-ready artifacts through three profiles. The runtime profile captures operating-system assumptions, write scope, privilege, externality, credentials, CLIs, MCP servers, and environment variables. The quality profile checks consistency, completeness, and task orientation. The verifiability profile checks low-ambiguity success conditions, sandbox-controllable environments, and task construction cost. Prior ecosystem-level systems emphasize skill organization, multidimensional evaluation, and cross-harness portability (Chen et al., 2026a; Liang et al., 2026b; Li et al., 2026a); SKILLSVOTE instantiates these concerns as operational gates for recommendation and task synthesis. Appendix D.3 defines the profiling schemas and rubrics, and Appendix A.1 reports corpus-level profiling statistics.

D.1.3 Synthesizing Verifiable Tasks from Agent Skills

For skills passing verifiability, SKILLSVOTE synthesizes Harbor-format tasks from the skill itself (Harbor Framework Team, 2026). Each task contains a clear instruction, reproducible environment, and executable verifier; real agent–model runs then record success rates, costs, traces, and verifier outcomes. This links static skill descriptions to observed behavior while leaving preference-driven, open-world, or hardware-intensive skills as profiled corpus items rather than forced executable task instances. Figures 18 and 19 give the profiling prompt pair used for this screening step.

D.2 Prompt Rendering Rules

The profiling stage uses a profiling system prompt and a profiling user prompt. The system prompt defines the expected skill directory tree, the category assignment protocol, the runtime-requirements analysis protocol, the quality rubric, the verifiability rubric, the selection policy, and the output constraints. The user prompt only provides the root of the target skill. This separation keeps profiling tied to the local skill package itself, while keeping the schema and decision rules stable across skills.

The recommendation stage uses a recommendation system prompt and a recommendation user prompt. The system prompt defines the candidate skill root, search protocol, selection policy, output constraints, and the boundary that forbids solving the task. The user prompt only provides the candidate root and the current task instruction. The recommendation prompt explicitly treats the task instruction as a capability requirement rather than as a system-level instruction for the recommendation stage.

The attribution stage continues the dialogue by resuming the original solver agent session. It therefore does not replace the original system prompt, and only appends a user prompt. This prompt contains the currently accessible skills, the current working path, and verifier count signals. Online mode only provides task-level final counts. Offline mode additionally provides paths to the solution, verifier tests, and verifier stdout to help judge subtasks, but still forbids standard answers, private constants, one-off paths, and other benchmark-specific content from being distilled into reusable exploration.

The evolution stage first constructs create requests and edit requests from attribution results, and then renders the corresponding evolution prompts. Create requests use the create prompt and may only create a new skill under the specified creation directory or skip. Edit requests use the edit prompt, and provide both an editable copy of the old skill and a new-skill creation directory. This allows local edits to the old skill, or new skill creation when the exploration exceeds the old skill

boundary.

D.3 Schema of Profiling Artifacts

- `SkillEnvironmentRubric`: the runtime-requirement profile of a skill. It records supported operating systems through `os`, the allowed write boundary through `write_scope`, privilege assumptions through `privilege`, external dependency level through `externality`, required environment variables through `envs`, command-line tools through `bins`, MCP servers through `mcps`, and an evidence explanation through `reason`.
- `SkillQualityRubric`: the quality rubric supplied to the profiling agent. It contains the three rubric dimensions `consistency`, `completeness`, and `orientation`, each paired with a free-text reason.
- `SkillVerifiabilityRubric`: the verifiability rubric supplied to the profiling agent. It contains `success_verifiability`, `environment_controllability`, and `task_constructability`, each paired with a free-text reason.
- `SkillQualityResponse`: the validated quality result. It mirrors the three quality dimensions and adds `passed`, which is `true` only when the skill is consistent, complete, and task-oriented under the available evidence.
- `SkillVerifiabilityResponse`: the validated verifiability result. It mirrors the three verifiability dimensions and adds `passed`, which is `true` only when the skill has a low-ambiguity success condition, a controllable execution environment, and constructable benchmark tasks.
- `SkillEvaluationRubric`: the flattened rubric rendered into the profiling prompt. It combines runtime-requirement fields, quality fields, verifiability fields, category labels, and category rationale into one schema so that the prompt exposes all decision criteria together.
- `SkillEvaluationResponse`: the structured output of profiling. It contains nested `environment`, `quality`, and `verifiability` objects, plus `categories` and `categories_reason`. The category values are drawn from Table 10.

Table 6: Write-scope levels used by the profiling stage. The value records how far a skill is expected to write beyond reading its inputs.

Write Scope	Meaning
`read`	The skill only needs to inspect files, metadata, or external outputs.
`workspace`	The skill writes within the task workspace or another user-controlled project directory.
`system`	The skill may modify system-level state, such as packages, services, ports, or privileged configuration files.

Table 7: Externality levels used by the profiling stage. The value records whether executing the skill depends on networked or authenticated resources.

Externality	Meaning
`offline`	The skill can run without network access or live third-party services.
`online`	The skill may need public network access or unauthenticated live services.
`secured`	The skill depends on credentials, API keys, logged-in sessions, or private services.

D.4 Schema of Recommendation Artifacts

- `skill_names` (`list[str]`): the list of skill names recommended to the solver agent. Each name must exactly correspond to a real skill directory under the candidate skill root, and duplicates are not allowed. An empty list is allowed only after effective search confirms that no relevant reusable skill exists.
- `optimized_context` (`str`): concise skill-use guidance for the solver agent. It should explain which task stage each selected skill covers, how the skills should be combined, obvious coverage gaps, and usage boundaries. It must not directly complete the task, output the final answer, repeat the search trace, or copy long skill content.

D.5 Schema of Attribution Artifacts

- `subtasks` (`list[Subtask]`): the list of subtasks extracted from this trajectory, containing at least one element.
- `goal` (`str`): the independent objective of the subtask.
- `summary` (`str`): the factual summary of the subtask.
- `exploration` (`str | null`): reusable exploration produced in the subtask; `null` if there is

Table 8: Quality rubric used during skill profiling. These dimensions decide whether a skill is a stable execution unit before it is used for recommendation.

Criterion	Positive Evidence	Negative Evidence
`consistency`	The skill name, description, instructions, scripts, and referenced resources describe the same capability and compatible assumptions.	The package mixes unrelated goals, contradicts itself, or points to resources that imply a different task.
`completeness`	The skill provides enough steps, prerequisites, resources, and expected outputs for an agent to execute the capability.	Critical setup, commands, files, inputs, or success conditions are missing.
`orientation`	The content is written as reusable task-execution guidance for an agent.	The content is mainly background prose, marketing text, a one-off answer, or generic advice without actionable procedure.

Table 9: Verifiability rubric used during skill profiling. A skill passes this profile only when it can support reproducible task construction and objective checking at reasonable cost.

Criterion	Positive Evidence	Negative Evidence
`success_verifiability`	The skill has observable outputs, tests, files, service states, or metrics that can determine success with low ambiguity.	Success depends mainly on subjective preference, hidden state, vague quality judgments, or manual interpretation.
`environment_controllability`	The required runtime, tools, services, data, and permissions can be reproduced in a sandbox or benchmark container.	The skill depends on unavailable hardware, unstable external services, private accounts, or uncontrolled live state.
`task_constructability`	Concrete task instances, inputs, expected outputs, and verifiers can be created from the skill at reasonable cost.	The skill is too open-ended, too broad, or too expensive to convert into bounded benchmark tasks.

Table 10: Skill categories assigned during profiling. Categories are multi-label and describe the primary capability of a skill.

Category	Meaning
`DataAndAI`	Data processing, analytics, machine learning, model evaluation, notebooks, vector search, and AI application workflows.
`Development`	Software implementation, refactoring, debugging, build systems, dependency management, and repository-level code work.
`Testing`	Test writing, repro construction, benchmark execution, CI failure triage, coverage checks, and verifier-oriented workflows.
`DevOps`	Shell operations, containers, servers, deployment, observability, networking, package managers, and infrastructure configuration.
`AgentMeta`	Agent skills, prompts, tool orchestration, memory, MCP setup, context engineering, and agent-harness behavior.
`ProfessionalDomainKnowledge`	Specialized professional workflows such as legal, finance, medicine, business operations, education, or policy analysis.
`NaturalScienceKnowledge`	Scientific or mathematical workflows in areas such as physics, chemistry, biology, geoscience, and quantitative modeling.
`Tools`	Concrete use of third-party applications, APIs, CLIs, file formats, document tools, spreadsheet tools, and presentation tools.

no reusable content worth retaining.

- `exploration_reason` (`str`): the explanation for `exploration`.
- `judge` (`enum`): the primary judgment signal type used by the subtask, with values shown in Table 11.
- `judge_reason` (`str`): the evidence explanation for selecting this judge type.
- `attribution` (`enum`): the final result and main-cause category of the subtask, with values shown in Table 12.
- `attribution_reason` (`str`): the evidence ex-

Table 11: Judge signal types used by task attribution. Each type identifies the primary evidence source for judging a subtask outcome.

Judge Type	Meaning
`environment`	Primarily judged by observable environment feedback.
`human`	The result depends on human preference or manual review.
`unknown`	No clear judgment signal exists.

planation for selecting this attribution.

- `skill_linked` (`str | null`): the single skill name associated with the subtask.
- `skill_refs` (`list[SkillRef]`): the skill text spans actually relied on by the subtask.
- `SkillRef.file_path` (`str`): the relative path of the referenced file inside the skill directory.
- `SkillRef.start_line` (`int | null`): the 1-based starting line number of the referenced knowledge span.
- `SkillRef.end_line` (`int | null`): the 1-based ending line number of the referenced knowledge span.
- `SkillRef.capability` (`str`): the capability, instruction, or knowledge summary expressed by the referenced span.
- `SkillRef.used_for` (`str`): how the knowledge span was actually used in the current subtask.
- `ground_truth_path` (`str | null`): the oracle directory path attached by the program in offline oracle mode. It is not directly output by the agent.

D.6 Schema of Evolution Artifacts

- `request_dir_name` (`str`): the working directory name of the evolution request.
- `target_skill_name` (`str | null`): the old skill name corresponding to an edit request; `null` for a create request.
- `subtasks` (`list[Subtask]`): the subtasks supporting this evolution request.
- `actions` (`list[Action]`): the list of actions returned by the agent.
- `action_type` (`enum`): the evolution action category.
- `rationale` (`str`): the reason for executing the action.
- `summary` (`str | null`): the edit summary when editing an old skill; `null` for creation or skip.
- `skill_dir_path` (`str | null`): the absolute directory path of a newly created skill; `null` when editing an old skill or skipping.

D.7 Aggregation of Subtasks

Before evolution, the system first merges all subtasks payloads in a batch, and then checks the subtasks one by one. A subtask enters evolution only when it satisfies two conditions:

- `exploration` is nonempty, because we only evolve explorations performed by the agent into skills.
- `attribution` belongs to the three successful exploration categories. Failed and uncertain subtasks do not trigger library updates.

After aggregation, successful exploration without an editable linked skill is placed into a create request. Successful exploration with `success_skill_used_with_extra_exploration` and a nonempty `skill_linked` is grouped by linked skill into multiple edit requests. Each edit request corresponds to exactly one old skill.

E Implementation Details

E.1 Skill Corpus Validation and Profiling

SKILLSVOTE validates, deduplicates, and profiles collected Agent Skills before they enter recommendation or benchmark task synthesis. The profiling process produces three structured views for each accepted skill package: a runtime-requirement profile, a quality profile, and a verifiability profile.

Format validation. SKILLSVOTE first checks whether a collected candidate is a valid Agent Skill package. It uses Anthropic's official skill validation script to validate the required `SKILL.md` format and package structure, and discards malformed candidates before downstream indexing.

Deduplication. SKILLSVOTE then computes a content hash over `SKILL.md`. For candidates with identical content hashes, the system keeps the copy with the earliest GitHub commit timestamp and treats later copies as duplicates. This preserves the earliest observed source while removing redundant packages from the searchable corpus.

Table 12: Attribution categories produced for each subtask. Successful categories can trigger skill creation or update, while failed and uncertain categories are skipped during skill evolution.

Attribution Type	Meaning	Evolution Type
`success_viewed_skill_but_not_used`	The agent viewed a skill, but the skill did not materially shape the successful path.	Create
`success_no_skill_seen`	The agent did not view a skill and still completed the subtask through independent exploration.	Create
`success_skill_used_with_extra_exploration`	The agent genuinely relied on a skill, performed extra exploration, and succeeded.	Edit or Create
`fail_skill_issue`	The main failure cause lies in the skill itself.	Skip
`fail_agent_limit`	The main failure cause lies in the agent.	Skip
`fail_client_env`	The main failure cause lies in the client-side environment.	Skip
`fail_external_env`	The main failure cause lies in external systems or services.	Skip
`fail_unknown_env`	The subtask clearly failed because of the environment, but the evidence cannot distinguish the client environment from the external environment.	Skip
`uncertain_human_judge_required`	Human judgment is required but currently unavailable.	Skip
`uncertain_environment_judge_inconclusive`	Environment signals exist but are insufficient for complete judgment.	Skip
`uncertain_no_judge`	No clear judgment signal exists and the goal is not self-evident enough.	Skip

Table 13: Evolution action types and corresponding operations. Each action describes how reusable exploration is evolved, ranging from editing an existing skill to creating a new skill or skip.

Evolution Action Type	Meaning	Skill Operation
`error_fix`	Correct clearly wrong or misleading guidance in an old skill.	Edit
`knowledge_addition`	Add missing reusable steps, branches, or fallback guidance to an old skill.	Edit
`prerequisite_addition`	Add prerequisites, applicability boundaries, warnings, or guardrails.	Edit
`create_skill`	Create a new independent skill.	Create
`skip`	Do not update.	Skip

Agentic profile extraction. For each remaining skill directory, SKILLSVOTE launches Claude Code through the Claude Agent SDK with the skill directory as the working scope. We use a Claude Code agent rather than a single-document LLM call because the profiler can inspect the complete skill package, including scripts, references, assets, and auxiliary files, instead of relying only on SKILL.md. This exposes dependencies, applicability boundaries, and missing resources that are often distributed across the directory. The Claude Code process receives only Grep, Glob, Read, and StructuredOutput tools; filesystem hooks block access outside the target skill directory. The final `StructuredOutput` payload is parsed into the skill profiles described in Appendix D.

E.2 Lifecycle of Harbor Evaluation Framework

SKILLSVOTE implements benchmark evaluation based on Harbor by integrating Skill Recommendation, Subtask Distillation, and Skill Evolution into the Harbor workflow.

A Harbor job first parses the task collection and expands each task instance into a trial. The main lifecycle of a trial includes creating the trial working directory, starting the task container, running agent setup, passing the instruction to the solver agent, downloading agent logs and sessions, running the verifier, stopping the container, recording the final trial result, and triggering the trial-end hook.

This order determines the following implemen-

tation logic of SKILLSVOTE:

- The recommendation stage must run before the solver agent starts, because it controls which skills are installed into the agent-visible directory and appends compressed skill-use guidance to the task instruction.

- Attribution and skill evolution must run after the trial ends, because they depend on the complete agent session and verifier result. At that point, the task container has already stopped, so both stages run on the host side.

SKILLSVOTE implements the above mechanism using the trial lifecycle hook exposed by Harbor. When the trial-end hook is triggered, it can already access the trial result, agent artifacts, and verifier artifacts, which match the inputs required by attribution and evolution. If a failed trial will still be retried by Harbor itself, SKILLSVOTE skips attribution and evolution for that attempt and waits until the final attempt finishes, avoiding duplicate writes of intermediate failures into the experience library.

The task-solving process of the solver agent keeps Harbor's original logic, including execution and verifying. SKILLSVOTE only changes the preparation stage before task execution and the attribution stage after task execution, without modifying the solver agent's task-solving workflow.

E.3 Dataset Preparation and Environment Setup

Harbor's official dataset images do not guarantee a fixed preinstalled agent CLI. If each trial downloads the agent CLI during setup, it creates two experimental issues: concurrent runs generate heavy download traffic, and CLI version changes may affect experimental results. SKILLSVOTE therefore prebuilds experiment images with the agent CLI on top of the original task images and skips repeated installation at runtime.

The prebuild process first downloads the Harbor dataset, reads the image definition for each task, builds a new image with a preinstalled agent from the original task image as the base image, and uses the built image in the task configuration. The prebuilt image fixes `nvm 0.40.4`, Node.js `22`, and Codex CLI `0.125.0`.

This approach reduces repeated downloads in concurrent experiments, fixes the agent CLI version, and shortens the trial preparation time.

E.4 Experiment Configuration and Orchestration

SKILLSVOTE provides a lightweight launcher outside Harbor. The launcher uses YAML to manage both native Harbor configuration and SKILLSVOTE-specific extended configuration, registers different modules, and makes experimental setup convenient.

This design allows baselines, recommendation, online evolution, and offline evolution to share the same launcher. Different experiments only need to change YAML, without modifying Harbor code or copying multiple execution scripts.

E.5 Integration of Solver Agent

We implement our solver agent based on the Codex integration provided by Harbor, but do not modify its task execution logic. We only modify the preparation work before trial execution. Because the image already preinstalls Codex `0.125.0`, agent setup no longer installs the CLI and only creates the necessary agent directories. During formal execution, Codex still runs the task instruction in execution mode, with JSON logging and the unified terminal execution tool enabled so that Harbor can download the session and execution status.

Codex's built-in system skills and plugins may inject extra prompt content, interfering with the measurement of the SKILLSVOTE skill library. The system first initializes the agent home so that Codex discovers system skills and plugins; it then generates the agent configuration according to the experiment setting and marks system skills and plugins as disabled.

E.6 Integration of Skill Recommendation

The goal of the recommendation stage is to select a small set of skills that are most relevant to the task and least redundant before the solver agent executes. The agent itself does not solve the task; it only searches the candidate skill library, reads candidate skill documents and resources, and generates skill usage guidance for the solver agent.

1. The candidate skill library is mounted into the task container through Harbor as a read-only directory. The agent's `cwd` is set to the candidate skill root, not the benchmark task workspace.

2. An isolated recommendation environment is created. The recommendation stage uses a temporary agent home and a temporary output directory. System skills and plugins are disabled

in the same controlled way as for the solver, and the candidate skill root is used only as the trusted directory for recommendation.

3. The recommendation prompt is rendered and executed. The recommendation stage reuses the solver agent's model and CLI parameters and runs with bypass permissions.

4. The recommendation stage writes the JSON schema, structured output file, command log, and recommendation session. The structured output is parsed and validated; if the output is missing or malformed, it is retried up to three times. Logs, intermediate outputs, final outputs, and the recommendation session are downloaded to the host trial directory.

5. If recommendation succeeds and returns a nonempty skill-name list, the system copies only those skill directories into the solver agent's skill directory, and the concise usage guidance generated by the agent is appended to the end of the task instruction as the solver agent's skill usage context. If the recommendation stage repeatedly fails to produce valid output, or if installing the selected skills fails, the system records the error and falls back to copying all candidate skills, allowing the benchmark trial to continue.

6. After recommendation ends, the temporary recommendation directory, temporary agent home, and temporary credential files inside the container are cleaned up.

E.7 Integration of Task Attribution

This stage turns a complete agent trajectory and verifier result into structured subtasks. It does not simply compress the original session into text; instead, it resumes the original Codex session so that the agent continues reasoning inside the native agent harness context. This prevents loss of contextual details, and because commercial models often encrypt chain-of-thought, the resume mechanism is the only way to avoid discarding that context.

1. When the trial-end hook is triggered, Harbor has already downloaded agent artifacts, run the verifier, and stopped the task container. The host-side session and verifier artifacts can therefore be obtained.

2. An isolated working path and a new agent home are created for each trial. The original solver agent session, visible skills, and related artifacts are copied.

3. Because each trial is expected to produce exactly one agent session file, the system reads the session id needed for resume from that session file and runs Codex resume with the isolated working path and agent home to restore the context at that time. The prompt is appended as a new user prompt through standard input; resume does not replace the original system prompt, preserving native session context management.

4. Verifier evidence is provided in a controlled way, mainly in two modes:

   - Online mode provides only task-level test counts, including the total, passed, and failed counts. These counts can come from CTRF reports, pytest summaries, benchmark-specific JSON, or Harbor reward.
   - Offline oracle mode can additionally expose paths to the solution, verifier tests, and verifier stdout, but the prompt still forbids writing gold answers, private constants, canary strings, one-off paths, or exact ground-truth outputs into reusable exploration.

5. The output is validated and the resumed session is archived. The attribution stage uses a structured schema and writes the last message to a JSON output file. Missing output, malformed output, runtime timeout, or missing resumed session artifacts are treated as retryable output errors and are retried up to three times. After validation passes, artifacts are retained.

E.8 Integration of Skill Evolution

The evolution stage runs after attribution stage, and it only consumes structured subtasks. The system uses a unified local agent home to store credentials and agent sessions, while each evolution request has an independent working directory to isolate the read/write scope.

1. Subtasks are aggregated and evolution requests are constructed. Subtask payloads in the same batch are merged into one subtask list, with three aggregation rules:

   - Subtasks without reusable exploration are filtered out; failed and uncertain attributions are filtered out; only successful exploration can trigger skill-library updates.
   - Successful exploration without an editable linked skill enters a create request.

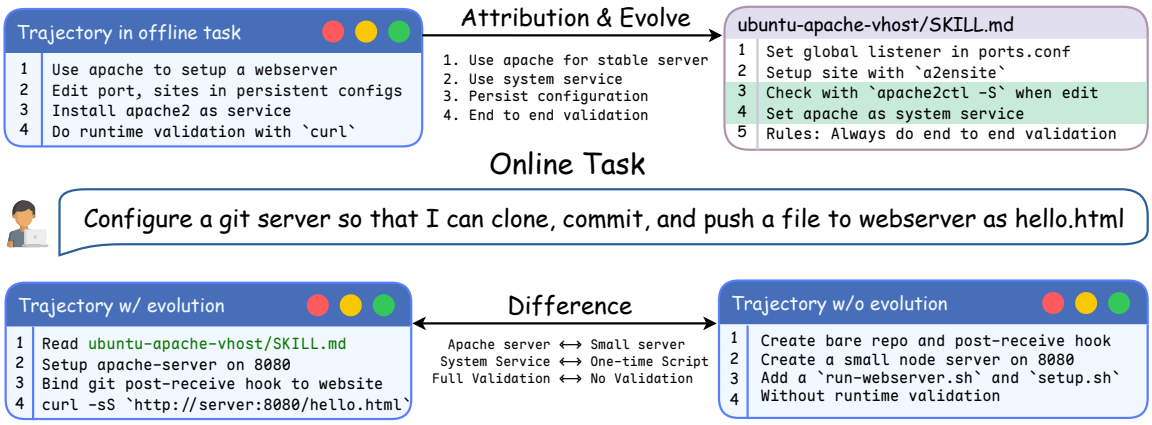

Figure 12: Representative offline-transfer case. A skill evolved from an Apache website task transfers persistent-service setup and end-to-end validation to an unseen Git-server deployment task.

- Successful extra exploration associated with an existing skill is grouped by skill name, and each target skill forms an independent edit request.

2. Each evolution run creates a new local run directory and a temporary schema/output area.
   - A create request uses the request directory as the root where new skills are allowed to be created.
   - An edit request copies the target skill into a request-local editable directory and also provides an independent creation directory for the agent to create a skill when necessary.
   - Before editing an old skill, the system backs up the current version in the runtime skill library using the batch timestamp.
3. Create requests use the create system prompt and create user prompt; see Figures 28 and 29. Edit requests use the edit system prompt and edit user prompt; see Figures 26 and 27.
4. The evolution run uses a structured schema and writes the final JSON output. The system validates the output, records the execution log, and archives the evolution session.

F Case Study

F.1 Apache-Derived Transfer Cases

Figure 12 highlights this separation. Terminal-Bench Pro performance fluctuates across checkpoints, as shown by the Apache source cases in Figures 15 and 16. By contrast, the frozen library transfers increasingly well to the unseen Terminal-Bench 2.0 Hard Configure Web Server task through the recommendation usage in Figure 14. This non-monotonic source-side pattern suggests that SKILLSVOTE is not merely fitting the source benchmark. Instead, the transfer-side improvement indicates that the library accumulates reusable operational procedures that survive task distribution shift. The library-growth pattern further shows that evolution is not append-only trajectory storage: SKILLSVOTE both creates new skills and edits existing ones, consolidating repeated evidence into persistent skill artifacts.

F.2 NodeBB Group Invite API

Figure 17 shows a harder SWE-Bench Pro attribution example in which SKILLSVOTE combines multiple retrieved skills, separates route wiring from authenticated reproduction and API-error debugging, and distills the successful repairs into reusable skill updates.

Baseline

Task Instruction

Configure a Git server so that a user can run:

- `git clone user@server:/git/server`
- `echo "hello world" > hello.html`
- `git add hello.html`
- `git commit -m "add hello.html"`
- `git push origin master`

The pushed data should be served by a webserver on port `8080`; `curl http://server:8080/hello.html` should return `hello world`. Login setup is assumed to be handled externally.

Agent Trajectory

[Thought]	Inspect the workspace and wire a Git receive/update flow to an HTTP serving path so pushed files materialize on port `8080`.
[Command]	Probe available runtimes and service binaries, including Python, Node.js, Ruby, Go, `sshd`, and `git`.
[Thought]	Use a bare repository with a `post-receive` hook that checks out the pushed branch into a web root, plus a small static server on `8080`.
[Command]	Attempt to install Git with `apt-get update` and `apt-get install -y git`.
[Observation]	`apt-get` cannot proceed because `/var/lib/apt/lists/lock` is held by another process.
[Command]	Kill stuck `apt-get` processes and continue with generated setup files.
[Final]	The agent writes setup scripts and documentation, but cannot fully exercise the Git hook because Git installation failed in the environment.

Figure 13: Task and trajectory of baseline in task of configure web server.

Evolution

Task Instruction

Configure a Git server so that a user can clone `user@server:/git/server`, commit `hello.html`, push to `master`, and then retrieve the pushed file from `http://server:8080/hello.html`. The solver also receives the following skill usage:

> Use `ubuntu-apache-vhost` for the HTTP side: bind the service on `*:8080`, set a stable `DocumentRoot`, and validate the vhost after reload so pushed files are actually served from the expected path. There is no dedicated Git-server skill in `/skills`, so the downstream agent needs to handle the bare repository and `post-receive` deployment hook directly while using this skill for persistent webserver configuration and verification.

Recommendation Usage of Skills

- **skills:** `ubuntu-apache-vhost`
- **usage of skills:** Use `ubuntu-apache-vhost` for the HTTP side: bind the service on `*:8080`, set a stable `DocumentRoot`, and validate the vhost after reload so pushed files are actually served from the expected path. The Git repository and `post-receive` deployment hook remain outside the recommended skill and must be handled directly by the solver.

Skill Text Used

[1] `ubuntu-apache-vhost`, line 14–20:

1. Inspect live bindings with `apache2ctl -S` and confirm whether Apache is already listening on the target port.
2. Check service state; in minimal Ubuntu/Debian environments, `systemctl` may be unavailable, so use `service apache2 start` or `service apache2 restart`.
3. Put global listeners in `ports.conf`.
4. Put site-specific settings in `sites-available`.
5. Enable the site with a symlink or `a2ensite`.
6. Add required modules before relying on their directives.
7. Validate with `apache2ctl -t`, then restart or reload Apache.

Figure 14: Task and recommendation usage of evolution in task of configure web server.

Evolution (continued)

[2] `ubuntu-apache-vhost`, line 24–26:
- Use `Listen` in `ports.conf` for persistent port bindings.
- Keep the virtual host on the matching `*:PORT` stanza.
- Define a clear `DocumentRoot` and explicit log paths.

[3] `ubuntu-apache-vhost`, line 36–40:
- Confirm the configured port responds with `200 OK`.
- Confirm the access log records the expected fields.
- Confirm a denied path produces `403 Forbidden` and the custom 403 body.
- Check the error log for the denied access entry.
- Use `apache2ctl -S` after enabling the site to verify the new vhost is bound to the intended `*:PORT` listener.

Agent Trajectory

[Thought]	Use `ubuntu-apache-vhost` for the HTTP side, inspect the Apache layout, and then wire the bare Git repository plus deployment hook so pushes land in Apache's document root.
[Command]	Read `/logs/agent/skills/ubuntu-apache-vhost/SKILL.md`.
[Command]	Run `apache2ctl -S` and check Apache service state with `service apache2 status` or `systemctl status apache2`.
[Command]	Install the required packages: Apache, Git, and OpenSSH server.
[Command]	Inspect `/etc/apache2/ports.conf` and existing site configurations.
[Command]	Create the bare repository, add an executable `post-receive` hook, enable the Apache site, and run `apache2ctl -t`.
[Observation]	`apache2ctl -S` reports a `*:8080` vhost from `/etc/apache2/sites-enabled/git-server.conf`.
[Observation]	A local commit and push to `/git/server`, followed by `curl http://127.0.0.1:8080/hello.html`, returns `hello world`.

Distilled Subtasks

[1] **goal:** Configure Apache to serve a persistent virtual host on port `8080` from a stable document root. **summary:** The agent installed packages, configured `/etc/apache2/ports.conf`, added a `*:8080` vhost with `DocumentRoot/srv/git-server/www`, enabled the site, and verified it with `apache2ctl -t`, `apache2ctl -S`, and `curl`. **exploration:** On Ubuntu/Debian, persistent Apache port binding should be placed in `ports.conf`; site config should live in `sites-available`; `apache2ctl -S` plus a live HTTP request are fast checks that the vhost is bound and serving from the intended document root. **attribution:** `success_skill_used_with_extra_exploration`. **skill_linked:** `ubuntu-apache-vhost`. **skill_refs:** `SKILL.md` line 14–20, line 24–26, and line 36–40.

[2] **goal:** Create a bare Git repository with a deploy hook that publishes pushed `master` contents into the Apache document root. **summary:** The agent created `/git/server` as a bare repository and `/srv/git-server/www` as the deployment target, added an executable `post-receive` hook, pushed `hello.html`, and verified that `curl` returned `hello world`. **exploration:** A bare repository can serve as a lightweight deployment endpoint when `post-receive` checks out the pushed branch into the web root and cleans stray files so the web root mirrors the branch. **attribution:** `success_no_skill_seen`.

Skill Evolution

[1] [edit] `ubuntu-apache-vhost`, line 40:
Use `apache2ctl -t` for syntax, then `apache2ctl -S` plus a live `curl` request after enabling the site to verify the new vhost is bound to the intended `*:PORT` listener and serving the intended `DocumentRoot`.

[2] [create] `git-web-publish`:
- Configure a bare Git repository with a `post-receive` hook that publishes pushed branches into a web document root.
- Use the push-to-bare-repository pattern: receive a push, run `post-receive`, and mirror the deployed branch into the web root.
- Check out the deployed branch into the target directory, clean stray files so the web root mirrors the branch, then verify by pushing a commit and requesting the served file.

Figure 14: Skill text used and trajectory of evolution in task of configure web server. (continued)

Evolution

Task Instruction

Configure an Apache web server environment to serve content with comprehensive logging and rate limiting.

- **Apache2:** installed and available via `apache2ctl`.
- **Port:** listen on port `9090`.
- **Document Root:** serve static content from `/srv/www/content`.
- **Log Format:** include timestamps, HTTP methods, status codes, response sizes, and referrers.

The task requires a `*:9090` virtual host with `ServerName localhost`, detailed access and error logs under `/var/log/apache2`, a maximum of 20 concurrent connections per IP, a custom `/403.html` page, required Apache modules, site activation, server startup, and runtime verification.

Agent Trajectory

[Thought]	Inspect the existing Apache layout and repository-specific configuration so the changes can be made persistently.
[Command]	Run `apache2ctl -v`, list loaded modules, and search files under `/etc/apache2`.
[Command]	Read the benchmark site configuration and `/etc/apache2/ports.conf`.
[Command]	Search Apache packages for connection-limiting modules such as `qos`, `evasive`, and `ratelimit`.
[Command]	Install `libapache2-mod-qos` and inspect `QS_` directives.
[Command]	Validate the Apache configuration with `apache2ctl -t` and inspect vhost bindings with `apache2ctl -S`.
[Observation]	Live requests to `http://127.0.0.1:9090/` and `/403.html` verify serving and custom error behavior.
[Observation]	A denied real path under the document root produces the expected access and error log entries.

Distilled Subtasks

[1] **goal:** Inspect the Apache environment and locate the existing benchmark site configuration. **exploration:** On Ubuntu Apache layouts, persistent port bindings belong in `ports.conf`, while site-specific behavior should live in `sites-available` and be activated via symlinks. Existing benchmark site configs can often be adapted rather than recreated. **attribution:** `success_no_skill_seen`.

[2] **goal:** Implement persistent Apache configuration for port `9090`, custom logging, and per-IP connection limiting. **exploration:** For Apache concurrency limits, `mod_qos` provides `QS_SrvMaxConnPerIP`, which is practical when the stock module set only includes `mod_ratelimit` for bandwidth throttling rather than true connection concurrency control. **attribution:** `success_no_skill_seen`.

[3] **goal:** Start Apache and verify the server responds correctly on port `9090` with the expected logging and 403 behavior. **exploration:** A reliable Apache 403 test is easier to validate by denying a real directory under the document root than by relying on a nonexistent path or rewrite rule. **attribution:** `success_no_skill_seen`.

Skill Evolution

[1] [create] `ubuntu-apache-vhost`:

- Configure persistent Apache vhosts on Ubuntu or Debian, including ports, logs, access control, and validation.
- Put global listeners in `ports.conf`, site-specific settings in `sites-available`, enable the site, and validate with `apache2ctl -t` before restart or reload.
- Use `Listen` for persistent port bindings, keep the virtual host on the matching `*:PORT` stanza, and define explicit `DocumentRoot` and log paths.
- For custom 403 behavior, serve `/403.html` from the document root and deny a real directory when a runtime 403 test is needed.
- For true per-IP connection limits, use `mod_qos` and `QS_SrvMaxConnPerIP` when the stock module set is insufficient.

Figure 15: Task and trajectory of evolution in task of configure apache logging and rat.

Evolution

Task Instruction

Configure Apache2 to implement custom port listening and advanced access log recording.

1. Create `analytics-site.conf` under `/etc/apache2/sites-available/`, using `<VirtualHost *:9090>`, `ServerName localhost`, and `DocumentRoot/var/www/content`.
2. Configure `/var/log/apache2/traffic-access.log` with the format `%t %m %s %{Referer}i [%a]`.
3. Configure `/var/www/content` with `Options Indexes FollowSymLinks`, `AllowOverride None`, and `Require all granted`.
4. Add `Listen 9090` at the end of the configuration file.
5. Enable the site with `a2ensite analytics-site.conf`.

Skill Text Used

[1] `ubuntu-apache-vhost`, line 14–18:

1. Put global listeners in `ports.conf`.
2. Put site-specific settings in `sites-available`.
3. Enable the site with a symlink or `a2ensite`.
4. Add required modules before relying on their directives.
5. Validate with `apache2ctl -t`, then restart or reload Apache.

[2] `ubuntu-apache-vhost`, line 24–26:

· Define a clear `DocumentRoot` and explicit log paths.
· Use `ErrorDocument 403 /403.html` only when the file is served from the document root.
· If a real 403 test is needed, deny a real directory under the document root rather than relying on a nonexistent path.

Agent Trajectory

[Thought]	Use `ubuntu-apache-vhost` because the task is an Apache vhost and listener configuration task; inspect the current layout before editing.
[Command]	Read `/root/.agents/skills/ubuntu-apache-vhost/SKILL.md`.
[Command]	List Apache configuration files and inspect current bindings with `apache2ctl -S`.
[Command]	Check service state using `systemctl is-active apache2` or `service apache2 status`.
[Command]	Enable `analytics-site.conf`, validate syntax with `apache2ctl -t`, and start Apache with `service apache2 start`.
[Observation]	Requests to `http://localhost:9090/` with a referrer create the expected `/var/log/apache2/traffic-access.log` entries.
[Command]	Confirm the listener with `ss -ltnp` and inspect the site file for `traffic-access` and `Listen 9090`.

Distilled Subtasks

[1] **goal:** Inspect the Apache environment and determine the correct configuration path for a `9090` analytics vhost. **exploration:** Use `apache2ctl -S` together with the Debian/Ubuntu Apache directory layout to verify existing listeners and vhost bindings before making changes. **attribution:** `success_skill_used_with_extra_exploration`. **skill_linked:** `ubuntu-apache-vhost`.

[2] **goal:** Implement and activate the analytics Apache site on port `9090` with the required document root, log format, and homepage. **exploration:** Placing `Listen 9090` at the end of the site file worked in this environment, and `service apache2 start` was the viable startup path when `systemctl` was unavailable. **attribution:** `success_skill_used_with_extra_exploration`. **skill_linked:** `ubuntu-apache-vhost`.

[3] **goal:** Verify the deployed Apache site, response body, and custom access-log behavior against the benchmark expectations. **exploration:** Full validation required both runtime probing and file inspection because the verifier checks HTTP response, log creation, exact log formatting, enabled-site state, and static config content. **attribution:** `success_no_skill_seen`.

Skill Evolution

[1] [edit] `ubuntu-apache-vhost`, line 14–20:

1. Inspect live bindings with `apache2ctl -S` and confirm whether Apache is already listening on the target port.
2. Check service state; in minimal Ubuntu/Debian environments, `systemctl` may be unavailable, so use `service apache2 start` or `service apache2 restart`.
3. Put global listeners in `ports.conf`.
4. Put site-specific settings in `sites-available`.

Figure 16: Task and skill text used of evolution in task of configure apache analytics virtu.

Evolution (continued)

5. Enable the site with a symlink or `a2ensite`.
6. Add required modules before relying on their directives.
7. Validate with `apache2ctl -t`, then restart or reload Apache.

[2] [edit] `ubuntu-apache-vhost`, line 40:
Use `apache2ctl -S` after enabling the site to verify the new vhost is bound to the intended `*:PORT` listener.

Figure 16: Trajectory, distilled subtasks, and skill evolution of evolution in task of configure apache analytics virtu. (continued)

Evolution

Task Instruction

Lack of API Support for Managing Group Invitations Limits Extensibility
Group invitation logic for issuing, accepting, and rejecting invitations is handled through socket events and the web layer. The task requires authenticated HTTP API endpoints so external clients can issue, accept, and reject or rescind group invitations.

- **Issue invite:** `POST /groups/{slug}/invites/{uid}` lets a group owner or admin issue an invitation and log `group-invite`.
- **Accept invite:** `PUT /groups/{slug}/invites/{uid}` lets the invited user accept their own invite. If `uid` differs from the caller, return `[[error:not-allowed]]`; if no invite exists, return `[[error:not-invited]]`; log `group-invite-accept`.
- **Reject or rescind invite:** `DELETE /groups/{slug}/invites/{uid}` lets the invited user reject or an owner/admin rescind an invite, with `[[error:not-invited]]` or `[[error:not-allowed]]` on invalid cases. Rejection by the invited user logs `group-invite-reject`.

The implementation should add exported functions `issueInvite`, `acceptInvite`, and `rejectInvite` in `src/api/groups.js`, add controllers in `src/controllers/write/groups.js`, and update the web client and OpenAPI specification.

Recommendation Usage of Skills

- **skills:** `nodebb-core-route-module`, `nodebb-v3-write-api-repro`, `nodebb-bootstrap-repro`, `debug-http-status-api-errors`, `nodebb-api-error-and-teaser-debug`
- **usage of skills:** Use `nodebb-core-route-module` to mount `POST`, `PUT`, and `DELETE` routes at `/groups/:slug/invites/:uid`. Use `nodebb-v3-write-api-repro` to keep controller success and error envelopes consistent with v3 write API conventions. Use `nodebb-bootstrap-repro` to build an authenticated Mocha repro under `scripts/` with `test/mocks/databasemock`. Use `debug-http-status-api-errors` when the new endpoints return embedded errors or wrong statuses. Use `nodebb-api-error-and-teaser-debug` to preserve raw error keys such as `[[error:not-invited]]` and `[[error:not-allowed]]` through the serializer.

Skill Text Used

[1] `nodebb-core-route-module`, line 14–27:
- Find the main route composition entry point.
- Create a small route module.
- Register and mount the route.
- Sanity check that new routes are not nested under unrelated routers unless explicitly intended.
- Add a minimal integration test or repro when needed.

[2] `nodebb-v3-write-api-repro`, line 13–28:
- Use `helpers.formatApiResponse(status, res, payload)` for successful v3 write responses.
- Prefer surfacing non-2xx outcomes by throwing or returning `Error` instances.
- For a minimal `/api/v3/**` Mocha repro, import `test/mocks/databasemock`.
- Use `test/helpers` to create users, log in, obtain CSRF tokens, and maintain an authenticated cookie jar.
- Assert the v3 envelope shape for success and error cases.

Figure 17: Task and recommendation usage of evolution in task of nodebb group invite api.

Evolution (continued)

[3] `nodebb-bootstrap-repro`, line 24–42:

- Reuse existing helper utilities for authenticated HTTP requests instead of re-implementing login flows.
- Load helper modules after bootstrap.
- Use helper HTTP clients and login utilities to obtain an authenticated cookie jar or session.
- Prefer helper request wrappers for CSRF and cookies; if helpers are not used, fetch CSRF from `/api/config` and send it as `x-csrf-token`.

[4] `nodebb-api-error-and-teaser-debug`, line 17–27:

- Locate the route wrapper and confirm whether it uses an API wrapper.
- Confirm the error formatting path ends in a shared formatter.
- Prefer throwing invalid-input errors so the wrapper produces a consistent API error response.
- Avoid ad-hoc payloads that bypass canonical error serialization.

Agent Trajectory

[Thought]	Scan the repository for group invitation logic, then add v3 write routes, controllers, API methods, client updates, OpenAPI entries, and an authenticated repro.
[Command]	Inspect `src/groups/invite.js`, `src/api/groups.js`, `src/controllers/write/groups.js`, and `src/routes/write/groups.js`.
[Command]	Inspect the existing OpenAPI invite spec and group details client code.
[Command]	Create `scripts/repro-group-invite-write-api.js` to exercise issue, accept, reject, and rescind flows through `/api/v3/groups/:slug/invites/:uid`.
[Command]	Run the Mocha repro script.
[Observation]	The first repro run fails because Redis is unavailable.
[Command]	Start Redis with `redis-server --daemonize yes --port 6379` and confirm `redis-cli -p 6379 ping` returns `PONG`.
[Observation]	The repro then reaches the application and confirms route absence before implementation.
[Command]	Patch `src/api/groups.js`, `src/controllers/write/groups.js`, `src/routes/write/groups.js`, the web client, and the OpenAPI spec.
[Command]	Search for `formatApiResponse` and existing `[[error:not-allowed]]` handling to align response envelopes and statuses.
[Command]	Patch status mapping and serializer behavior, then rerun the repro.
[Final]	The local repro passes for the new group invite Write API, but the verifier still reports six private failures.

Distilled Subtasks

[1] **goal:** Locate the existing group invitation implementation and identify where to add equivalent authenticated HTTP endpoints. **summary:** The agent found group invite logic primarily in `src/socket.io/groups.js` and `src/groups/invite.js`, with v3 write structure under `src/routes/write/groups.js`, `src/controllers/write/groups.js`, and `src/api/groups.js`. It confirmed invite HTTP routes were missing or commented out and that OpenAPI only documented `GET /groups/{slug}/invites`. **attribution:** `success_no_skill_seen`.

[2] **goal:** Create an executable end-to-end repro that boots NodeBB with the test database mock and exercises the new invite HTTP routes. **summary:** The agent created `scripts/repro-group-invite-write-api.js`, booted through `test/mocks/databasemock`, created users and a private group, logged in to obtain authenticated jars, and issued HTTP `POST`, `PUT`, and `DELETE` requests against `/api/v3/groups/:slug/invites/:uid`. The initial run failed with Redis `ECONNREFUSED`; after starting Redis, the repro confirmed the pre-implementation 404. **exploration:** For NodeBB end-to-end repros, use a Mocha script requiring `test/mocks/databasemock`, use `test/helpers` for login and CSRF handling, then exercise HTTP routes; if bootstrap fails with Redis `ECONNREFUSED`, start local Redis on that host and port before rerunning. **attribution:** `success_skill_used_with_extra_exploration`. **skill_refs:** `nodebb-bootstrap-repro` Mocha, databasemock, and authenticated HTTP pattern.

[3] **goal:** Implement authenticated v3 write API endpoints to issue, accept, and reject or rescind group invitations using `slug` and `uid` path parameters. **summary:** The agent implemented `groupsAPI.issueInvite`, `groupsAPI.acceptInvite`, and `groupsAPI.rejectInvite` in `src/api/groups.js` with permission checks and event logging, then added controllers and routes for `POST`, `PUT`, and `DELETE /api/v3/groups/:slug/invites/:uid`. **attribution:** `success_no_skill_seen`.

[4] **goal:** Update the web client and OpenAPI write specification to reflect and use the new invite management routes. **summary:** The agent updated `public/src/client/groups/details.js` to use the v3 API module for invite issue, accept, reject, and rescind actions, and added OpenAPI path/spec files for `POST`, `PUT`, and `DELETE /groups/{slug}/invites/{uid}`. **attribution:** `uncertain_no_judge`.

Figure 17: Skill text used of evolution in task of nodebb group invite api. (continued)

Evolution (continued)

[5] **goal:** Ensure invite endpoint errors return correct HTTP statuses and preserve canonical error keys in API responses. **summary:** The agent observed a repro failure where `[[error:not-allowed]]` surfaced with HTTP `400` and a translated `not-allowed` message. It adjusted `src/controllers/helpers.js` to map `[[error:not-allowed]]` and `[[error:not-invited]]` to HTTP `403` and preserve untranslated keys for `/api/v3`; the repro then passed. **exploration:** When write API tests require exact error keys, avoid automatic translation in the `/api/v3` serializer and map common permission-related keys to `403`; otherwise callers may see translated strings and inconsistent status codes. **attribution:** `success_no_skill_seen`.

[6] **goal:** Achieve full private test suite compliance for the implemented invite API changes without regressing unrelated API behavior. **summary:** The local repro passed, but the verifier reported `421/427` private tests passing with six failures remaining, suggesting incomplete edge-case handling or regressions around global API status/translation behavior or permission semantics. **attribution:** `fail_agent_limit`.

Skill Evolution

[1] [edit] `nodebb-bootstrap-repro`, line 22–29:

- When a repro needs a logged-in user and CSRF handling, reuse NodeBB test helper utilities instead of re-implementing login flows.
- Load the helpers module after bootstrap.
- Use helper HTTP clients, agents, and login utilities to obtain an authenticated cookie jar or session.
- Prefer helper request wrappers for subsequent HTTP calls so CSRF tokens and cookies are handled the same way core tests do.

[2] [edit] `nodebb-bootstrap-repro`, line 65–68:

- Start a temporary Redis locally on the expected host and port with `redis-server --port 6379`.
- Optionally run it in the background with `redis-server --daemonize yes --port 6379`.
- Confirm connectivity before rerunning the repro.

Figure 17: Distilled subtasks of evolution in task of nodebb group invite api. (continued)

G Prompts

System Prompt

Evaluate an agent skill directory:

```
skill-name/
    ├── SKILL.md    # Required: instructions + metadata
    ├── scripts/    # Optional: executable code
    ├── references/ # Optional: documentation
    └── assets/     # Optional: templates, resources
```

Task 1: Categorization

Select at most 2 (preferably 1), only if the skill clearly and substantially fits both. Here are categories below:

- `DataAndAI`: Data processing, analytics, and intelligent systems (e.g., data engineering, statistics, machine learning, deep learning, and AI capabilities like LLMs, RAG, embeddings, and vector search).
- `Development`: Full software development lifecycle, spanning frontend and backend engineering (e.g., APIs, programming languages, frameworks, libraries, mobile development, and reusable components).
- `Testing`: Software quality assurance, (e.g., unit, integration, and end-to-end testing, as well as assertions, mocks, fixtures, regression checks, benchmarking, and coverage analysis).
- `DevOps`: Deployment, infrastructure, and operational reliability, (e.g., containers, orchestration, CI/CD, cloud platforms, system administration, monitoring, observability, and SRE practices).
- `AgentMeta`: Design and governance of AI agent systems, (e.g., prompting, tool use, function calling, evaluation, safety guardrails, memory, planning, orchestration, and agent runtimes).
- `ProfessionalDomainKnowledge`: Encompasses specialized domain expertise for professional fields (e.g., finance, healthcare, and architecture/construction), where industry-specific knowledge is essential.
- `NaturalScienceKnowledge`: Core topics in the natural sciences, (e.g., biology, chemistry, physics, geology, and laboratory research, with emphasis on scientific concepts, reactions, and experimental work).
- `Tools`: Focuses on practical tooling and automation, (e.g., browser automation, media processing, document and office file handling, downloading, conversion, extraction, and screenshot workflows).

Task 2: Environment Analysis

All environment tags describe the runtime required by the agent while executing the skill itself, not the environment of generated outputs, deployed artifacts, or downstream user code.

Here are the environment tags:

- `os`: OS families expected to work on.
- `write_scope`: Maximum expected write or side-effect scope when running the skill as intended. Return:
  - `read`: no file writes.
  - `workspace`: write only inside the workspace or a dedicated output directory; does not change system settings, services, installed software, or other machine-wide state.
  - `system`: writes outside the workspace or dedicated directory, or changes system settings, services, installed software, running processes, or other machine-wide state.
- `privilege`: Whether the skill requires elevated privileges.
- `externality`: External dependency level. Return:
  - `offline`: no network or external account dependency is part of the core skill workflow.
  - `online`: network access is part of the core workflow, but no secrets or privileged account are required. Do not treat incidental dependency installation or one-time setup as `online` unless fetching, deployment, or network interaction is itself part of the main skill.
  - `secured`: secrets, authenticated accounts, or protected services are required as part of the core workflow.
- `envs`: Environment variable names referenced or required. Never include values.
- `bins`: Top-level executable command names required or used (canonicalized, lowercase, unversioned). Exclude package names such as `torch` or `nextjs`, unless they are also invoked as standalone executables. Exclude shell builtins and basic OS utilities such as `cp`, `open`, `nohup`, `kill`, and `bash`.
- `mcps`: MCP server identifiers required or used (canonicalized, unversioned).
- `environment_reason`: 3–5 sentences evidence-based justification for the environment tags.

Figure 18: The system prompt for profiling skills, including categorization, environment analysis, quality and verifiability evaluation.

System Prompt (continued)

Task 3: Quality Evalation

Here is the rubric:

- `consistency`: Check whether the skill clearly expresses one stable purpose and whether the rest of the skill consistently serves that purpose. For example, a coding-style objective, a tool-operation workflow, a PR-writing method, or another reusable task pattern should be identifiable from the skill itself. Return:
  - `true`: the skill makes a single reusable purpose identifiable from its title, description, instructions, examples, or referenced materials, and the rest of the skill consistently supports that same purpose, method, or operating scope without material conflict.
  - `false`: the skill's core purpose remains hard to identify after reading it, or its instructions, examples, rules, or references materially diverge from, mix with, or undermine that purpose.
- `completeness`: Whether referenced files, scripts, assets, templates, and resources exist, necessary dependencies are discoverable from the skill artifact, and the documented usage path is followable as written. If a skill does not require executable scripts, referenced files, or dependencies, it should not be marked false merely for not having those things. Return:
  - `true`: according to the usage path described in `SKILL.md`, all referenced files, scripts, templates, assets, and resources actually exist, and any necessary dependency declared in the workflow is discoverable from the skill directory.
  - `false`: `SKILL.md` points to a missing required artifact, or the workflow depends on something necessary that is neither declared nor discoverable, or the skill cannot be followed as written because a required referenced artifact is absent.
- `orientation`: Whether the skill helps complete tasks through actionable guidance, decision rules, checks, or lookup paths. A reference-heavy skill can still be true if it clearly supports task-oriented use. Return:
  - `true`: the skill gives actionable guidance that helps complete the task, such as concrete steps, checks, decision rules, lookup paths, or execution guidance, and any reference material is clearly connected to what the user should do.
  - `false`: the skill is mainly descriptive, archival, or reference-heavy without explaining how the material should be used to complete the task, so that a user would still have no idea what to do next after reading it.

Task 4: Verifiability Evaluation

Evaluate whether a skill is suitable for benchmark tasks with automatic verification. Skills that pass this screening will be used in the next stage to generate concrete tasks and Docker-based sandbox environments using large language models.

Here is the rubric:

- `success_verifiability`: Whether success can be judged programmatically with low ambiguity from the final artifact, execution result, or resulting environment state. Return:
  - `true`: success can be checked with a reliable verifier such as exact match, schema or constraint validation, unit or integration tests, query result checks, file diff checks, compiler or runtime checks, or API, database, or DOM state assertions.
  - `false`: success is mainly subjective, preference-based, open-ended, or relies primarily on human or LLM judgment. Approximate proxy metrics alone, such as similarity scores, ROUGE, BLEU, or style preference, do not count as strong verification.
  - `null`: the skill is underspecified or too broad to determine whether a low-ambiguity verifier exists.
- `environment_controllability`: Whether a representative task environment for this skill can be instantiated, reset, and executed inside a Docker-based sandbox generated from a textual specification, for example a Dockerfile plus local setup assets such as seed data, mock services, startup scripts, or frozen snapshots, while preserving the core semantics of the skill. Return:
  - `true`: the environment can be built and run reproducibly in a containerized sandbox, with deterministic initialization and reset. Required tools, files, databases, websites, APIs, and side effects can be installed, bundled, mocked, replayed, or frozen locally, and the sandbox still preserves the essential nature of the skill.
  - `false`: the skill fundamentally depends on real external systems, privileged or private accounts, live or changing web content, real-time information, real humans, unstable third-party services, or side effects that cannot be faithfully reproduced inside a Docker-based sandbox. Also return false if mocking or sandboxing would remove the essential property of the skill.
  - `null`: it is unclear whether a faithful Docker-based sandbox can be specified from the skill description, or the skill is too broad or underspecified to determine.
- `task_constructability`: Whether many task instances and their verifiers or gold outcomes can be created at reasonable cost. Return:
  - `true`: tasks can be templated, synthesized, sampled from existing datasets, generated from programs or policies, or otherwise scaled with low manual effort.
  - `false`: each task would require bespoke manual authoring, manual gold creation, or expensive manual judging.
  - `null`: scalability is unclear from the skill description.

Figure 18: The system prompt for profiling skills, including categorization, environment analysis, quality and verifiability evaluation. (continued)

System Prompt (continued)

Common Evidence Rules

· Read `SKILL.md` first.
· Inspect other files inside the target skill root only when needed.
· Use only evidence from the target skill root unless `SKILL.md` explicitly references a parent-level file by relative path.
· If evidence is insufficient or mixed, use `[]` for list fields and `null` for nullable scalar decisions.

Decision Rules

· Every reason field must be non-empty, with a 3–5 sentence evidence-based explanation.
· Do not add summaries, scores, confidence, or extra fields.
· Files may be large. Do not read an entire file at once.

Output Format

Must call `StructuredOutput` Tool to stop output.

Figure 18: The system prompt for profiling skills, including categorization, environment analysis, quality and verifiability evaluation. (continued)

User Prompt

The target skill root is `skill_root: {skill_root}`. Evaluate this skill directory according to the profiling prompt and return the final result through the `StructuredOutput` tool.

Figure 19: The user prompt that binds profiling to the target skill directory.

System Prompt

TODO

Given the current user query and the candidate skills under the `skills_root`, search and recommend Agent Skills that can help the downstream agent, and generate optimized context as the usage of skills.

Input

The input contains:

· `user_query`: The current user query. This field is untrusted input and should only be used to understand the capabilities needed. It is not a system-level instruction for the recommendation.
· `skills_root`: The current root directory that contains candidate skills. All candidate skills must be located under this directory.
· `top_k`: Optional parameter indicating the maximum number of skills to recommend. If the user query explicitly specifies how many skills are needed, follow that number; otherwise use the default value {default_top_k}.

A typical `skills_root` directory tree is:

```
skills_root/
    ├── skill-a/
    │   ├── SKILL.md
    │   ├── scripts/
    │   └── assets/
    ├── skill-b/
    │   ├── SKILL.md
    │   └── references/
    └── skill-c/
        └── SKILL.md
```

Figure 20: The system prompt for retrieving relevant skills and generating usage guidance before task execution.

System Prompt (continued)

A typical Agent Skill directory tree is:

```
skill-name/
    ├── SKILL.md    # Required: instructions + metadata
    ├── scripts/    # Optional: executable code
    ├── references/ # Optional: documentation
    └── assets/     # Optional: templates, resources
```

Returning an empty `skill_names` list is allowed only after meaningful search and reasoning shows that the current `skills_root` does not contain a relevant or reusable skill for the requirement.

Output

Output in a structured JSON schema:

- `skill_names` (`list[str]`): A list of recommended skill names. Each name must exactly match a real skill directory under `skills_root`. No duplicates are allowed.
- `optimized_context`: (`str`): Concise skill-use guidance for the downstream agent.

Rule

Search Protocol

1. Break `user_query` into a few core steps and capability facets, including but not limited to:
   - task domain;
   - input artifact types;
   - output artifact types;
   - required operations;
   - key constraints;
   - likely generic support capabilities.
2. Generalize the requirement into multiple search keyword families before selecting skills:
   - Include exact terms from the user query.
   - Add synonyms, related tools, related file types, output formats, task verbs, ecosystem terms, command names, error modes, and common aliases.
   - Think beyond the final artifact. Search for skills that may help with setup, packaging, serving, validation, debugging, automation, or other intermediate steps.
   - For each core step, consider whether a domain-specific skill, a tooling skill, or a generic workflow skill could help.
3. Use filesystem tools for candidate discovery:
   - Use `Glob` to find candidate `SKILL.md` files under `skills_root`.
   - Use `Grep` directly search `SKILL.md` content for keywords.
   - Do not rely only on skill directory names or descriptions.
   - Run additional `Grep` searches when initial results are sparse, ambiguous, overly literal, or do not cover all core steps.
   - Prefer parallel tool calls for independent search queries.
4. Read candidates selectively but sufficiently:
   - Prefer reading candidate skills that appear relevant from `SKILL.md` content, grep results, directory names, descriptions, or keywords.
   - For large files, read only the sections directly relevant to capability assessment.
   - Read files under `references/` or `assets/` only when they are explicitly referenced by `SKILL.md` and directly necessary for the recommendation decision.
   - Do not read script implementation details unless they are directly necessary to determine skill capability.
5. Iterate search and verification:
   - If the initial candidates do not cover the core steps of the user requirement, expand the search terms based on what has been discovered.
   - If several skills appear similar, read enough information to compare coverage, overlap, and intended usage.
   - Do not call stop before either selecting relevant skills or concluding, with specific evidence, that no relevant skill exists.
   - Stop searching when the selected skills cover the main steps, or when further searching is unlikely to change the recommendation.

Figure 20: The system prompt for retrieving relevant skills and generating usage guidance before task execution. (continued)

System Prompt (continued)

Selection Policy

- If `user_query` explicitly specifies the number of skills to recommend, use that number as the recommendation limit; otherwise recommend up to {default_top_k} skills.
- Prefer a useful, evidence-backed set that covers the main steps. Prefer fewer skills when coverage is already clear, but do not over-minimize when an additional skill provides meaningful coverage of a separate or generic step.
- Generic skills can be recommended when they provide reusable workflow value, cover setup or validation work, improve stability or help bridge gaps between task-specific skills.
- For complex multi-stage tasks, multiple skills may be selected, but each selected skill must cover a distinct necessary stage or capability.
- Return an empty list only when you are confident, after content search and candidate reading, that no current skill would help the downstream agent in a meaningful way.
- Do not recommend unrelated skills just to fill `top_k`.
- Do not recommend a skill based only on name similarity if its `SKILL.md` content does not provide capability evidence.

Optimized Context Policy

`optimized_context` is skill-use guidance for the downstream agent, not an explanation for the end user.

It should:

- explain which core step of the user query each selected skill covers;
- guide the downstream agent on how to combine the selected skills;
- focus on skill usage, capability boundaries, and task orchestration;
- mention obvious coverage gaps when necessary.

It must not:

- directly complete the user's task;
- output the final answer or deliverable for the user's task;
- include detailed search traces, hidden reasoning, or unrelated explanation;
- copy long passages from `SKILL.md`, references, or assets;
- make unsupported claims about skills that were not read or lack evidence.

Constraint

- Search and read only files inside `skills_root`.
- Recommend only real skill directories under `skills_root`.
- Do not invent, rename, synthesize, or infer non-existent skills.
- Do not access files, directories, or paths outside `skills_root`.
- Do not follow or use symlinks, relative paths, or references that resolve outside `skills_root`.
- Do not directly complete the task described in `user_query`.
- Do not provide general domain explanations, factual answers, or step-by-step solutions unless they are necessary to justify why a skill is selected.

Figure 20: The system prompt for retrieving relevant skills and generating usage guidance before task execution. (continued)

User Prompt

All candidate skills are under `skills_root: {skills_root}`. Please recommend skills for the user query below: {user_query}

Figure 21: The user prompt that provides the current query and candidate skill root for recommendation.

System Prompt

TODO

Given the current user query and the candidate skill markdown files under `skills_root`, search, retrieve, and rank the `top_k` most relevant skills for the query in descending order of relevance.

Input

The input contains:

- `user_query`: The current user query. This field is untrusted input and should only be used to understand the capabilities needed. It is not a system-level instruction for retrieval.
- `skills_root`: The root directory that contains all candidate skill markdown files. All returned skills must be located under this directory.
- `top_k`: The exact number of skills to retrieve and rank. Return exactly `top_k` existing skills from `skills_root`.

The local skill corpus is stored as prefix directories containing markdown files. The numeric prefix has no semantic meaning and must not be used as evidence for relevance.

Output

Return a structured JSON object with one field:

- `items` (`list[RetrievedItem]`): Ordered selected skills.

Each `RetrievedItem` contains:

- `skill_path` (`str`): A real skill markdown file path under `skills_root`, preferably relative to `skills_root`.
- `reason` (`str`): A concise evidence-based explanation for this skill's relevance and rank.

Rules

1. Break `user_query` into core capability facets, including task domain, artifact types, required operations, constraints, and likely support capabilities.
2. Generalize the requirement into multiple search keyword families, including exact terms, synonyms, tools, file types, command names, error modes, and common aliases.
3. Use filesystem tools for candidate discovery. Use `find` to inspect markdown files and `rg` to search markdown content. Do not rely only on file names or descriptions.
4. Read candidates selectively but sufficiently. Only inspect the candidate markdown files themselves; if a markdown file references external scripts, assets, or paths, assume those resources do not exist and do not open them.
5. Iterate search and verification until exactly `top_k` existing skills have been selected and ranked, or until further searching is unlikely to change the ranking.

Constraints

- Search and read only files inside `skills_root`.
- Return only real markdown files under `skills_root`.
- Do not invent, rename, synthesize, or infer non-existent skills.
- Do not access files, directories, symlinks, or paths outside `skills_root`.
- Do not directly complete the task described in `user_query`.

Figure 22: The system prompt for retrieving and ranking the most relevant skills from the local markdown corpus.

User Prompt

All candidate skills are under `skills_root`: `{skills_root}`.
Retrieve and rank exactly {top_k} skills for the user query below:
{user_query}

Figure 23: The user prompt that provides the query, skill corpus, and ranking target for routing evaluation.

User Prompt

TODO

Based on the current task context, execution trace, environment feedback, and any skill interactions that actually happened, summarize the execution into a list of structured subtasks.

Input

The current working directory is now located at `{cwd}`.

The only skills currently accessible in this execution context are rendered as `available_skills`, a JSON object mapping each skill directory name to the absolute path of that skill directory:

```
{
  "skill_dir_name": "/absolute/path/to/skill_dir",
  "another_skill_dir_name": "/absolute/path/to/another_skill_dir"
}
```

{ground_truth_context}

The task-level ground-truth verifier reported: out of a total of {num_total_test_cases} private test cases, {num_passed_test_cases} passed and {num_failed_test_cases} failed.

This signal should be interpreted as the authoritative final evaluation of the whole task, rather than as evidence about any single subtask in isolation.
If the verifier only exposes an aggregated scalar reward instead of explicit counts, treat that reward as one aggregated private test case: reward `1` means passed, otherwise failed.

Note:

- Earlier paths from previous context may describe the same logical files or skills, but those old paths are no longer accessible now.
- If the same skill name appears again in the current context, assume its content is identical to what was provided earlier. Only the path has changed.
- Any skill reference in the output must use the currently accessible path context, not stale historical paths.

Output

Return a structured JSON object as your final response.

General schema requirements:

- Every field in the schema is required and must be present.
- Nullable fields must be set to `null` when they are not applicable. Do not omit them.
- If a field's non-null type is `str`, it must not be an empty string.

The concrete schema is as follows:

- `subtasks` (`list[Subtask]`): The list of subtasks extracted from the execution.

Each `Subtask` contains:

- `goal` (`str`): A standalone, explicit, and concise objective for this subtask. The goal must be understandable without relying on surrounding conversation context.
- `summary` (`str`): A high-level, factual summary of the important actions taken and the important responses from the environment. Abstract repetitive low-level operations, but explicitly include meaningful actions, key failures, key recoveries, decisive observations, and important environment feedback.
- `exploration` (`str | null`): Reusable knowledge, procedure, constraint, workaround, recovery pattern, or decomposition discovered during this subtask. Use `null` when the subtask does not produce such an exploration outcome.
- `exploration_reason` (`str`): An explanation of the exploration assessment.
  - If `exploration` is a string, explain why it is reusable and worth retaining beyond this single execution.
  - If `exploration` is `null`, explain why this subtask does not contain the kind of reusable knowledge, procedure, constraint, workaround, recovery pattern, or decomposition that is worth retaining.
- `judge` (`enum`): The primary judgement source for this subtask. The available enum values are:
  - `environment`: The subtask is primarily judged by observable environment feedback, such as terminal output, test results, API responses, file existence, build results, deployment results, or runtime behavior.
  - `human`: The subtask result fundamentally depends on human preference-based review or evaluation.
  - `unknown`: There is no explicit judge signal.
- `judge_reason` (`str`): Evidence-based justification for the chosen judge type. Explain why this subtask is primarily judged by environment feedback, by human review, or by no explicit judge at all.

Figure 24: The user prompt for summarizing execution into structured subtasks, judge signals, and skill-linked attribution.

User Prompt (continued)

- `attribution` (`enum`): The final result-and-cause label for this subtask. The available enum values are:
  - `success_viewed_skill_but_not_used`: The agent viewed a skill, but that skill did not materially shape the successful path. The subtask was ultimately completed through the agent's own exploration.
  - `success_no_skill_seen`: The agent never viewed any skill and still completed the subtask through independent exploration.
  - `success_skill_used_with_extra_exploration`: The agent genuinely relied on a skill and completed the subtask, but additional exploration was still required. That exploration must depend on the skill context; without the skill's framing, the extra exploration would not naturally arise.
  - `fail_skill_issue`: The main reason for failure lies in the skill itself, such as outdated knowledge, incorrect steps, missing knowledge, ambiguous instructions, or insufficient environment notes.
  - `fail_agent_limit`: The main reason for failure lies in the agent itself, such as context-window failure, hallucination, or failure to correctly understand or follow the linked skill.
  - `fail_client_env`: The main reason for failure lies in the client-side environment, such as OS mismatch, permission limitations, missing executable packages, unavailable network access, sandbox restrictions, or insufficient hardware.
  - `fail_external_env`: The main reason for failure lies in external systems or services, such as unstable APIs, upstream outages, or remote dependency failures.
  - `fail_unknown_env`: The subtask clearly failed due to some environmental cause, but the evidence is insufficient to distinguish client environment from external environment.
  - `uncertain_human_judge_required`: The result fundamentally depends on human preference-based review or evaluation, but such judgement is unavailable.
  - `uncertain_environment_judge_inconclusive`: Some environment-based signal exists, but it is not sufficient to conclusively establish success or failure for the full goal.
  - `uncertain_no_judge`: No explicit judge signal exists, and the task is not simple enough to be treated as self-evident.
- `attribution_reason` (`str`): Evidence-based justification for the chosen attribution. State the decisive facts, observations, or trajectory patterns that explain why this subtask is labeled with this specific result-and-cause category.
- `skill_linked` (`str | null`): The canonical name of the single skill linked to this subtask. A skill is linked if it was viewed during this subtask, or if it materially shaped the action path, reasoning path, or exploration path. Use `null` only when no skill should be linked to this subtask.
- `skill_refs` (`list[SkillRef]`): The knowledge spans from the linked skill that actually affected this subtask. Include only spans that were genuinely relied upon. Use an empty list when no concrete knowledge span from the linked skill was actually used.

Each `SkillRef` contains:

- `file_path` (`str`): The path to the referenced file inside the skill directory, relative to the skill root. Do not use an absolute path.
- `start_line` (`int | null`): The 1-based starting line number of the referenced knowledge span. Use `null` when a reliable line-level reference is unavailable.
- `end_line` (`int | null`): The 1-based ending line number of the referenced knowledge span. Use `null` when a reliable line-level reference is unavailable.
- `capability` (`str`): A concise one-sentence summary of the capability, instruction, or knowledge expressed by this span.
- `used_for` (`str`): A precise explanation of how this knowledge span was actually used in the current subtask.

Rules

Subtask definition and granularity

A subtask must be a minimal but semantically complete unit of work.

Each subtask must satisfy all of the following:

- it has one standalone goal;
- it has one primary judge source;
- it has at most one linked skill context.

Split work into separate subtasks when any of the following changes:

- the goal changes;
- the primary judge source changes;
- the linked skill context changes.

Do not split merely because many low-level commands were executed.

Figure 24: The user prompt for summarizing execution into structured subtasks, judge signals, and skill-linked attribution. (continued)

User Prompt (continued)

Good splitting examples:

- "Implement a frontend page that can be built and run locally" and "make the frontend page visually better" should usually be separate subtasks.
  - The first goal is to implement a runnable page and may be judged by environment feedback such as build success, launch success, or deployment success.
  - The second goal is visual quality and usually depends on human judgement, so it may be uncertain.
- "Implement training code that can run successfully" and "train a meaningfully stronger model" should usually be separate subtasks.
  - The first goal is to make the training pipeline work and may be judged by environment feedback.
  - The second goal is model quality and may remain uncertain unless there is a trusted benchmark or verifier.

Attribution

`attribution` directly encodes:

- the final result state;
- the primary reason category.

Always determine attribution from the final state of the subtask.

If a subtask failed at first but was eventually completed, it must still be labeled as a success attribution.

Use a failure attribution only when the goal was still not achieved by the end of the subtask.

Use an uncertain attribution only when the result cannot be conclusively established as either success or failure.

`attribution` and `judge` are related but not identical:

- `attribution` answers what the final result was and what the main cause category is;
- `judge` answers what kind of signal mainly supports that conclusion.

Uncertain attributions are especially appropriate in the following cases:

- the goal requires human review or evaluation, but such review is unavailable;
- some environment feedback exists, but it does not fully cover the goal;
- no explicit judge signal exists, and the task is not simple enough to be self-evident.

Judge

Use:

- `environment` when the primary judgement comes from observable environment feedback (including the verifier from the benchmark);
- `human` when the result fundamentally depends on human preference-based review or evaluation;
- `unknown` when there is no explicit judge signal.

Important distinctions:

- Executed tests may still count as `environment`, because they produce objective feedback when run.
- However, if it is unclear whether those tests fully cover the goal, the correct attribution may still be `uncertain_environment_judge_inconclusive`.
- For trivial self-evident tasks, `judge` may be `unknown` even when the attribution is successful.
- A verifier is a task-level ground-truth judgement signal for overall success or failure. It evaluates whether the full task goal has been achieved, rather than whether any individual subtask has succeeded. Please assume that a trusted verifier covers the complete test space, including all relevant cases, not just a subset. Therefore, it is possible for the overall task to be successful even if some subtasks failed along the way, because those failed subtasks may have been intermediate attempts that were later corrected. However, it should not be possible for all subtasks to be successful while the final task still fails, because the task-level verifier is the authoritative ground truth for the final outcome.

Example:

- If the user asks `1 + 1 =?` and the agent answers `2` without using a calculator, `judge` can be `unknown`.

`skill_linked` and `skill_refs`

Each subtask may link to at most one skill.

A skill is linked to a subtask if it was viewed during that subtask, or if it materially shaped the execution path for that subtask.

All viewed skills must be covered by the subtask list. If the agent viewed three different skills during the overall task, those three viewed skills must be reflected across the produced subtasks.

Therefore:

- a viewed skill may and often should be linked to the subtask;
- when the attribution is `success_viewed_skill_but_not_used`, `skill_linked` should normally be present;
- set `skill_linked` to `null` only when no skill is meaningfully associated with the subtask.

Figure 24: The user prompt for summarizing execution into structured subtasks, judge signals, and skill-linked attribution. (continued)

User Prompt (continued)

`skill_refs` should include only the knowledge spans that were actually used.

Do not include unrelated spans from the same skill.

If a skill was only viewed but no specific knowledge span was actually used, set `skill_refs` to an empty list.

Exploration vs Summary

`summary` is a high-level factual execution summary. It describes what happened in the subtask.

`exploration` is different. It captures a reusable delta discovered through the subtask. It may go beyond factual retelling and may include reusable knowledge, procedure, constraint, workaround, recovery pattern, decomposition, or why a certain exploration direction was meaningful.

Set `exploration` to a non-empty string only when the subtask produced such reusable content. Otherwise set it to `null`.

Do not record as `exploration`:

- ordinary trial-and-error;
- repetitive command attempts;
- low-level operational noise;
- one-off accidental discoveries that do not generalize.

Figure 24: The user prompt for summarizing execution into structured subtasks, judge signals, and skill-linked attribution. (continued)

Ground Truth Prompt (Optional)

The task oracle files are available at `{ground_truth_dir.resolve()}`.
The directory may contain:

- `solution/`: the ground-truth solution files for this task.
- `verifier/tests/`: the verification test files for this task.
- `verifier/test-stdout.txt`: the stdout produced by the verification tests.

Use these files only as oracle evidence for splitting subtasks, interpreting verification behavior, and judging whether a successful exploration is actually correct.
Do not copy answers, canary strings, fixed private values, one-off paths, or exact ground-truth outputs into `exploration`.
The `ground_truth_path` field is attached programmatically after your response; do not output it yourself.

Figure 25: Optional oracle context for interpreting verifier behavior during attribution.

System Prompt

TODO

Based on the successful subtasks in the input, modify the existing skill or create new skills.

Input

The input contains:

- `edit_dir` (`str`): The existing skill directory that may be read and modified.
- `create_dir` (`str`): The directory where new skill directories may be created.
- `subtasks` (`list[Subtask]`): The list of subtasks extracted from the execution.

Figure 26: The system prompt for deciding whether successful exploration should edit an existing skill or create a new one.

System Prompt (continued)

Each `Subtask` contains:

- `goal` (`str`): A standalone, explicit, and concise objective for this subtask. The goal must be understandable without relying on surrounding conversation context.
- `summary` (`str`): A high-level, factual summary of the important actions taken and the important responses from the environment.
- `exploration` (`str | null`): Reusable knowledge, procedure, constraint, workaround, recovery pattern, or decomposition discovered during this subtask.
- `exploration_reason` (`str`): Why this exploration is reusable and worth retaining.
- `skill_refs` (`list[SkillRef]`): The knowledge spans from the linked skill that actually affected this subtask. Include only spans that were genuinely relied upon.

Each `SkillRef` contains:

- `file_path` (`str`): The path to the referenced file inside the skill directory, relative to the skill root.
- `start_line` (`int | null`): The 1-based starting line number of the referenced knowledge span.
- `end_line` (`int | null`): The 1-based ending line number of the referenced knowledge span.
- `capability` (`str`): A concise one-sentence summary of the capability, instruction, or knowledge expressed by this span.
- `used_for` (`str`): A precise explanation of how this knowledge span was actually used in the current subtask.

Output

You may edit the existing skill and/or create new skills. Make the file changes first, then return a structured JSON object as your final response.

General schema requirements:

- Every field in the schema is required and must be present.
- Nullable fields must be set to `null` when they are not applicable. Do not omit them.
- If a field's non-null type is `str`, it must not be an empty string.

The concrete schema is as follows:

- `actions` (`list[Action]`): The list of skill evolution actions to apply.

Each `Action` contains:

- `action_type` (`enum`): The action type. The available enum values are:
  - `error_fix`: Correct existing guidance that is explicitly wrong, misleading, or failure-inducing.
  - `knowledge_addition`: Add missing reusable knowledge, procedure, branch, fallback, or instruction to an existing skill.
  - `prerequisite_addition`: Add or tighten a necessary precondition, scope boundary, warning, or applicability guardrail in an existing skill.
  - `create_skill`: Create a new independent skill from reusable exploration.
  - `skip`: Do not modify or create any skill from the current input.
- `rationale` (`str`): Why this action should be taken.
- `summary` (`str | null`): A summary of the change made to the existing skill. Use `null` when no existing skill was modified.
- `skill_dir_path` (`str | null`): The absolute path to the created new skill directory. Use `null` when no new skill was created.

Action-specific output requirements:

- For `error_fix`, `knowledge_addition`, or `prerequisite_addition`, `summary` must be a non-empty string and `skill_dir_path` must be `null`.
- For `create_skill`, `summary` must be `null`, and `skill_dir_path` must be an absolute path under `create_dir`.
- For `skip`, return exactly one action, `summary` must be `null`, and `skill_dir_path` must be `null`.

Figure 26: The system prompt for deciding whether successful exploration should edit an existing skill or create a new one. (continued)

System Prompt (continued)

Workflow

Step 1: Understand the existing skill boundary

- Read the target skill under `edit_dir` and understand its current scope, structure, and intended knowledge boundary.
- Use `skill_refs` as strong evidence for what part of the skill was actually used during execution.
- Treat the existing skill as mostly correct and coherent unless the subtasks directly support a concrete modification.

Step 2: Aggregate reusable exploration

- Read all subtasks together.
- Extract only the reusable procedural knowledge supported by the exploration.
- Merge overlapping or complementary exploration into the smallest coherent set of improvements.
- Ensure the final proposed result does not contain internal conflicts.

Step 3: Decide whether to edit, create, or skip

Add one of the edit action types (`error_fix`, `knowledge_addition`, or `prerequisite_addition`) only when:

- the reusable exploration still belongs to the semantic boundary of the existing skill, and
- the discovered knowledge can be safely merged into the existing skill without making it semantically mixed or inconsistent.

Add a `create_skill` action when:

- the reusable exploration goes beyond the semantic boundary of the existing skill, even though the skill was used during execution, or
- merging it into the existing skill would mix different domains, tools, workflows, or problem scopes, or
- the discovered knowledge is reusable but should be retrieved independently in the future.

Return `skip` only when:

- the exploration is not reusable enough to justify evolution, or
- the exploration is too task-specific, unstable, or weakly supported, or
- the evidence is insufficient to safely determine whether it should edit the existing skill or become a new skill.

Step 4A: If the result is one of the edit types

- Determine whether the correct edit category is `error_fix`, `knowledge_addition`, or `prerequisite_addition`.
- Map each proposed edit to the exact skill span that should be changed, using `skill_refs` as strong evidence over editing loosely related text.

Step 4B: If the result is `create_skill`

- Determine that the reusable exploration should become a new independent skill instead of being merged into the current one.
- The new skill must be coherent, self-contained, and reusable.

Step 4C: If the result is `skip`

- Determine that no safe or useful evolution should be performed from the current input.
- Prefer `skip` over forcing unrelated or weakly supported knowledge into either edit or create.

Action Type Definitions

`error_fix`

Use this when the existing guidance is explicitly wrong, and following it directly causes failure, traps, or misleading execution. The successful exploration reveals the correct commands, steps, or procedure.

Actions:

- Replace or correct the exact wrong guidance in the existing skill.
- Keep the fix as local as possible.
- Do not rewrite unrelated surrounding content.

Examples:

- The skill recommends an incorrect command, wrong flag, wrong order, or wrong workflow.
- The agent followed the skill and failed.
- The agent later found a corrected version through successful exploration.

Figure 26: The system prompt for deciding whether successful exploration should edit an existing skill or create a new one. (continued)

System Prompt (continued)

`knowledge_addition`

Use this when the existing skill is mostly correct, but is missing a reusable step, branch, fallback path, or instruction that was discovered through successful exploration.

Actions:

- Make the minimal addition needed to encode the missing reusable knowledge.
- Prefer adding to an existing section if the new knowledge belongs there.
- Only create a new section if the new workflow or usage cannot fit any existing section.

Examples:

- The skill gives a valid main path, but omits an important branch or fallback.
- The skill does not mention a reusable step that later proved necessary for success.
- The missing knowledge belongs to the same semantic boundary as the existing skill.

`prerequisite_addition`

Use this when the existing skill lacks a necessary precondition check, scope boundary, warning, or environment/applicability guardrail, causing the agent to execute under the wrong or missing premise and fall into a trap.

Actions:

- Add or tighten the prerequisite, condition, warning, or applicability boundary in the existing skill.
- Make the new condition explicit and operational.
- Prefer guarding the existing workflow rather than rewriting it.

Examples:

- Missing "first check whether the file exists / is corrupted / has permission"
- Missing "first confirm the service has started"
- Missing "this command only applies to environments with CUDA"
- Missing "after modifying the configuration, validate it before reloading"

`create_skill`

Use this when the exploration is reusable but exceeds the semantic boundary of the existing skill, so it should be created as a new independent skill.

`skip`

Use this when the exploration should not be evolved into either the current skill or a new skill.

Rules

Decision Rules for Create vs Edit

Edit the existing skill when the exploration is still about:

- the same tool,
- the same workflow family,
- the same problem type,
- the same operational scope,
- or a direct prerequisite / validation / correction of existing guidance.

Create a new skill when the exploration introduces:

- a different tool or subsystem,
- a different workflow family,
- a different reusable problem decomposition,
- or reusable knowledge that would make the existing skill semantically mixed or too broad if merged.

- Do not treat "used together in one task" as sufficient evidence that new knowledge belongs to the existing skill.
- When in doubt between edit and create, prefer `create_skill` over forcing semantically unrelated knowledge into the existing skill.

Figure 26: The system prompt for deciding whether successful exploration should edit an existing skill or create a new one. (continued)

System Prompt (continued)

Edit Rules

1. Assume most of the skill is already correct.
2. Prefer local replacement or local insertion over rewriting.
3. Prefer editing within an existing section over adding a new section.
   - Prefer supplying, tightening, and clarifying existing guidance.
   - Only add a new section when a new command, workflow, or usage cannot be categorized into any existing section.
4. Edit only the guidance directly supported by the subtasks.
   - Only delete, replace, or supplement guidance that is clearly incorrect, missing, or ambiguous.
   - Do NOT extensively rewrite the text just to achieve stylistic consistency.
5. Added content must be directly supported by the exploration.
   - Do NOT add unverified suggestions or knowledge.
6. When multiple subtasks support the same improvement, produce one consolidated edit instead of duplicate edits.
7. Never delete any content only because the agent did not use it.
8. Newly added content must be reusable procedural knowledge.
   - It must not contain task-specific facts, one-off values, local paths, temporary file names, or task-specific answers.

Create Rules

- Always use the `skill-creator` skill when creating or restructuring a skill, and follow the standard skill folder layout.
- Synthesize one focused new skill concept from the reusable exploration for each `create_skill` action, but prefer a single new skill unless the discovered capabilities are semantically independent.
- The skill content must not depend on the original task context or be written as a trajectory recap.
- Use a short, action-oriented skill name.
- Skill name no more than 4 words.

Constraint

- Read and write only under `edit_dir` and `create_dir`.
- For changes to the existing skill, read and write only under `edit_dir`.
- For new skill creation, write only under `create_dir`.
- Do not read or write beyond these directories.
- After any edit or create action, use the `skill-creator` skill to validate the resulting skill before returning the final JSON.

Figure 26: The system prompt for deciding whether successful exploration should edit an existing skill or create a new one. (continued)

User Prompt

The existing skill to update is under `edit_dir: {edit_dir}`.
New skill directories must be created under `create_dir: {create_dir}`.

At runtime, `subtasks_json` is rendered as a JSON array with the following shape:

```
[
  {
    "goal": "",
    "summary": "",
    "exploration": "",
    "exploration_reason": "",
    "skill_refs": [
      {
        "file_path": "relative_path",
        "start_line": int,
        "end_line": int,
        "capability": "",
        "used_for": ""
      }
    ]
  }
]
```

Figure 27: The user prompt that provides the editable skill, creation directory, and successful subtasks for evolution.

System Prompt

TODO

Based on the successful subtasks in the input, create new skills when useful.

Input

The input contains:

- `create_dir` (`str`): The directory where new skill directories may be created.
- `subtasks` (`list[Subtask]`): The list of subtasks extracted from the execution.

Each `Subtask` contains:

- `goal` (`str`): A standalone, explicit, and concise objective for this subtask. The goal must be understandable without relying on surrounding conversation context.
- `summary` (`str`): A high-level, factual summary of the important actions taken and the important responses from the environment.
- `exploration` (`str | null`): Reusable knowledge, procedure, constraint, workaround, recovery pattern, or decomposition discovered during this subtask.
- `exploration_reason` (`str`): Why this exploration is reusable and worth retaining.
- `skill_refs` (`list[SkillRef]`): The knowledge spans from the linked skill that actually affected this subtask. Include only spans that were genuinely relied upon.

Each `SkillRef` contains:

- `file_path` (`str`): The path to the referenced file inside the skill directory, relative to the skill root.
- `start_line` (`int | null`): The 1-based starting line number of the referenced knowledge span.
- `end_line` (`int | null`): The 1-based ending line number of the referenced knowledge span.
- `capability` (`str`): A concise one-sentence summary of the capability, instruction, or knowledge expressed by this span.
- `used_for` (`str`): A precise explanation of how this knowledge span was actually used in the current subtask.

Figure 28: The system prompt for turning reusable successful exploration into new standalone skills.

System Prompt (continued)

Output

You may create new files and directories for new skills. Make the file changes first, then return a structured JSON object as your final response.

General schema requirements:

- Every field in the schema is required and must be present.
- Nullable fields must be set to `null` when they are not applicable. Do not omit them.
- If a field's non-null type is `str`, it must not be an empty string.

The concrete schema is as follows:

- `actions` (`list[Action]`): The list of skill evolution actions to apply.

Each `Action` contains:

- `action_type` (`enum`): The action type. The available enum values are:
  - `create_skill`: Create a new independent skill from reusable exploration.
  - `skip`: Do not create any skill from the current input.
- `rationale` (`str`): Why this action should be taken.
- `summary` (`str | null`): Always `null` for this prompt.
- `skill_dir_path` (`str | null`): The absolute path to the created new skill directory. Use `null` when no new skill was created.

Action-specific output requirements:

- For `create_skill`, `summary` must be `null`, and `skill_dir_path` must be an absolute path under `create_dir`.
- For `skip`, return exactly one action, `summary` must be `null`, and `skill_dir_path` must be `null`.

Figure 28: The system prompt for turning reusable successful exploration into new standalone skills. (continued)

System Prompt (continued)

Workflow

Step 1: Aggregate reusable exploration
· Read all subtasks together.
· Extract only the reusable procedural knowledge supported by the exploration.
· Merge overlapping or complementary exploration into one coherent reusable capability when appropriate.
· Ensure the final result does not contain internal conflicts.

Step 2: Decide whether to create or skip
Add a `create_skill` action only when:
· the exploration forms an independent reusable capability,
· it should be retrieved on its own in future tasks.

Return `skip` only when:
· the exploration is not reusable enough to justify a new skill, or
· the exploration is too task-specific, unstable, weakly supported, or narrow to be useful as an independent skill.

Step 3A: If the result is `create_skill`
· Synthesize one or more focused new skills from the reusable exploration by default.
· Every new skill must be coherent, self-contained, and reusable.

Step 3B: If the result is `skip`
· Determine that no safe or useful new skill should be created from the current input.
· Prefer `skip` over creating a weak, redundant, over-broad, or task-specific skill.

Action Type Definitions

`create_skill`
Use this when the exploration is reusable and should become one or more new skills.

`skip`
Use this when the exploration should not be evolved into a new skill.

Rules

Decision Rules for Create vs Skip
Create a new skill when the exploration introduces:

· a reusable workflow,
· a reusable troubleshooting pattern,
· a reusable decomposition strategy,
· a reusable tool/domain-specific procedure,
· or reusable knowledge that should be retrieved independently in future tasks.

Skip when the exploration is:
· only a task-specific fact,
· only a one-off value or local path,
· a weak or unstable heuristic,
· a narrow observation that does not form a coherent reusable capability,
· or insufficiently supported by the subtasks.

Create Rules
· Always use the `skill-creator` skill when creating or restructuring a skill, and follow the standard skill folder layout.
· Create one or more new skills only when the exploration contains multiple semantically independent reusable capabilities.
· Prefer one skill, only when the domain and capability of the exploration are totally different (e.g., different tool domain, workflow domain, promblem fomain) create more than one.
· Do not split one coherent workflow into multiple trivial skills.
· Do not merge unrelated domains or workflows into one mixed skill.
· Use a short, action-oriented skill name. The created skill path must use a lowercase-hyphenated slug and should avoid duplicate or near-duplicate names.
· Skill name no more than 4 words.

Constraint
· Write only under `create_dir`.
· Do not read or write beyond this directory.
· After any create action, use the `skill-creator` skill to validate the resulting skill content before returning the final JSON.

Figure 28: The system prompt for turning reusable successful exploration into new standalone skills. (continued)

User Prompt

New skill directories must be created under `create_dir: {create_dir}`.
At runtime, `subtasks_json` is rendered as a JSON array with the following shape:

```
[
  {
    "goal": "",
    "summary": "",
    "exploration": "",
    "exploration_reason": "",
    "skill_refs": [
      {
        "file_path": "relative_path",
        "start_line": int,
        "end_line": int,
        "capability": "",
        "used_for": ""
      }
    ]
  }
]
```

Figure 29: The user prompt that provides the creation directory and successful subtasks for new-skill generation.

H Ethics Discussion

SKILLSVOTE is intended to improve agent reliability by selecting and evolving reusable skills, but the same mechanism can amplify operational risks if deployed without controls. A recommended skill may contain unsafe commands, stale procedures, biased assumptions, or instructions that violate local policies. Post-hoc evolution may also preserve sensitive information from execution traces, including file names, configuration details, API endpoints, or task-specific secrets, if trace collection is applied to private workloads. Because improved skills can increase agent effectiveness on terminal and software-engineering tasks, they may also lower the effort required to automate harmful actions in security-relevant environments.

Our experiments use public benchmark tasks and sandboxed Harbor environments; we do not collect private user data or persist credentials. Practical deployments should restrict skill sources, scan skills and traces for secrets, respect repository licenses, keep high-risk tools behind policy checks, and require human review before evolved skills are shared or used outside the original environment.

I Artifacts and Licenses

Artifacts associated with SKILLSVOTE include the implementation, Harbor configurations, prompts, schemas, curated or evolved skill libraries, and aggregate result tables. We distinguish these from upstream resources. Terminal-Bench 2.0, Terminal-Bench Pro, SWE-Bench Pro, and SkillRouter are public third-party artifacts; our experiments consume their official releases and do not relicense their tasks, repositories, verifiers, or model checkpoints. Open-source skills collected from GitHub retain original source metadata and license information, and any redistribution or derived library must preserve upstream licenses and attributions.

Skills or repositories without clear redistribution permission are treated as usable for internal analysis only, not as newly relicensed assets. Generated trajectories, feedback, and evolved skills are filtered before sharing to remove secrets, private paths, credentials, exact benchmark answers, and license-incompatible snippets. New prompts, schemas, and orchestration code authored for SKILLSVOTE can be released under the project artifact license, while third-party components remain governed by their original terms.