SKILL2TASK: SELF-EVOLVING AGENTIC TASK SYNTHESIS VIA SKILL GRAPH AND PROGRESSIVE VALIDATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Tool-using language agents are increasingly expected to complete long-horizon workflows in realistic computing environments, and training them at scale requires a large and diverse supply of executable, verifiable tasks. Yet existing pipelines generate this data with a fixed, one-shot synthesizer built from a hand-designed harness that cannot keep improving from the failures surfaced during construction. We present Skill2Task, a self-evolving framework that turns reusable community skills into executable general-agent tasks and makes the synthesis harness itself the object of optimization. It composes skills through a skill-state graph, progressively validates each environment, instruction, and verifier before use, and feeds construction and rollout diagnostics back to revise the harness each round, which raises the success rate and efficiency of task construction over successive rounds. This loop yields 10,916 verifiable tasks and 10,880 curated trajectories, and supervised fine-tuning on them consistently improves two open backbones of different scales across four agent benchmarks, raising their average benchmark score by up to 18% relative, with the largest gains on skill-composition tasks, where accuracy more than doubles. These results show that evolving the task-synthesis harness itself, rather than freezing it, yields broader and more reliable training data than one-shot synthesis.

1 INTRODUCTION

Large language model agents are moving from passive text generation toward systems that perceive, plan, and act through tools in real computing environments (Wang et al., 2023; Jimenez et al., 2024; Merrill et al., 2026). As agents enter everyday workflows, they must reliably edit files, inspect data, call APIs, browse the web, run code, and coordinate intermediate artifacts over long horizons (Yang et al., 2025; Li et al., 2026b). Progress on such agents is now limited less by model capacity than by the supply of executable, verifiable tasks at the scale and diversity needed to train and evaluate long-horizon tool use.

Progress toward such a supply has been steady but incomplete. Hand-curated, execution-verified benchmarks ground correctness in real environments but are capped at a few thousand instances by manual authoring (Jimenez et al., 2024; Merrill et al., 2026). Automatic pipelines lift this ceiling by synthesizing executable environments and tasks at scale, though early efforts produce mostly isolated single-command or single-bug tasks (Yang et al., 2025; Zhu et al., 2026; Lin et al., 2026); more recent systems such as SkillSynth compose reusable skills so that a single task exercises several capabilities (Fan et al., 2026). Yet even these compositional pipelines share two limitations. First, a synthesized task couples three fragile components: an executable environment, a well-posed instruction, and a reliable verifier, and a defect in any one of them either aborts construction or yields a flawed task, because synthesis lacks a decoupled mechanism that attributes and repairs each component separately. Second, the harness that generates the tasks is hand-built and hand-tuned, and stays fixed once written. Recent execution-feedback methods might seem to address the latter, but they improve the trained policy, its runtime harness, or the RL environment (Shinn et al., 2023; Dong et al., 2026; Zhang et al., 2026), while the pipeline that synthesizes the data itself remains a one-shot generator, so the failures surfaced during construction are never fed back to improve how tasks are generated.

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

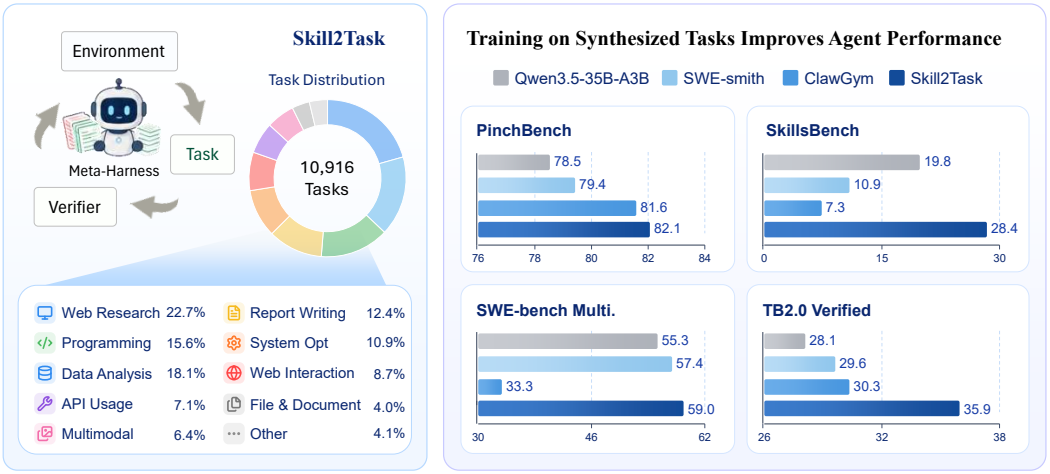


Figure 1: Teaser of Skill2Task. Community skills are composed into an executable multi-artifact task, and training on verified trajectories enables the resulting model to complete the full workflow rather than stopping at an intermediate artifact. The bottom panels compare the Qwen3.5-35B-A3B base model with Skill2Task-SFT across four benchmarks.

We propose Skill2Task, which turns this fixed generator into one that keeps improving from its own construction failures. Rather than one-shot generation, Skill2Task treats task synthesis as a self-evolving loop that attacks both limitations above: it decouples construction so that each task component is validated and repaired on its own, and it feeds the resulting diagnostics back to revise the synthesis harness round after round. Our core contribution is thus to make the task-synthesis harness, rather than the trained policy, the object of self-evolution.

Concretely, a realistic task chains several compatible skills, yet skills sampled independently rarely fit together, since one skill's output seldom matches the next skill's expected input. Skill2Task therefore composes skills through a scenario-level state-transition graph whose edges connect capabilities with compatible input and output scenarios, so sampled paths form coherent multi-step workflows that exercise several capabilities instead of one isolated command. It then validates the environment, instruction, and verifier as separate staged loops: because each defect is localized to one component, most failures are repaired in place rather than by resampling the whole skill path, which recovers bundles a single end-of-pipeline filter would discard and raises the yield of valid tasks to 91.5%. Finally, an evolution operator aggregates construction and rollout diagnostics into edits of the synthesis harness itself, lifting the first-attempt construction quality of a weak static harness from 42.3% toward this repaired level with fewer repair iterations per task, and the same diagnostics drive outcome- and step-level curation of the collected trajectories.

Our key contributions are as follows:

- We propose Skill2Task, a self-evolving synthesis framework that continually improves the task-synthesis harness itself from construction and rollout diagnostics, raising the yield of valid task bundles with fewer repair iterations per task.
- We develop a progressive environment–task–verifier generation and repair pipeline that composes reusable skills along a skill-state graph and validates and repairs each component before rollout, attributing every detected failure to the component that caused it and recovering bundles that a single end-of-pipeline filter would discard.
- Beyond outcome-level filtering, we introduce step-level trajectory masking that excludes flawed intermediate actions from the imitation loss while preserving their context, and show it yields further gains.
- We construct 10,916 verifiable tasks that span diverse domains and chain multiple skills into complex, realistic multi-step workflows, together with 10,880 curated training trajectories, and supervised fine-tuning on them consistently improves two open base models across four agent benchmarks.

2 Related Work

2.1 Executable and Verifiable Task Synthesis

Grounding agent tasks in real, executable environments makes their correctness checkable rather than judged by a model. Early benchmarks of this kind are hand-authored, pairing real issues or terminal tasks with per-task environments and reference tests (Jimenez et al., 2024; Merrill et al., 2026; Lin et al., 2018); execution grounds correctness, but manual authoring caps coverage at a few thousand instances. To move past this ceiling, automatic pipelines synthesize the environment first and then generate tasks inside it, whether by injecting test-breaking bugs at repository scale, by inverting healthy environments into buggy states, or by building Dockerized environments, tasks, and executable checks end to end (Yang et al., 2025; Zhu et al., 2026; Lin et al., 2026; Wu et al., 2026; Pi et al., 2026; Gandhi et al., 2026). Extending this idea to general agents, ClawGym synthesizes heterogeneous computer-use tasks and collects large trajectory volumes for supervised training (Bai et al., 2026). These pipelines scale the number of instances, but a synthesized task typically exercises a single bug or command and is admitted through one final pass/fail filter, so a faulty environment or an exploitable verifier surfaces only after its errors have already entangled with the instruction and check. Skill2Task instead composes each task from multiple skills and validates the environment, instruction, and verifier as separate staged loops before rollout, attributing every detected failure to the component that produced it.

2.2 Skill Organization and Skill-Centric Composition

A complementary body of work treats reusable "skills"—packaged instructions, tools, and resources—as first-class assets to be accumulated, organized, and composed. Voyager introduced a self-verified skill library for a single embodied domain (Wang et al., 2023), and registries such as ClawHub and SkillsMP now index large community skill collections (OpenClaw, 2026; SkillsMP, 2026). To make these collections usable, AgentSkillOS organizes skills into a hierarchical capability tree with graph-based orchestration and SkillNet models them as nodes in a relational graph (Li et al., 2026a; Liang et al., 2026), while SkillSynth builds a scenario-mediated skill graph and samples compositional paths that a multi-agent harness instantiates into executable terminal tasks (Fan et al., 2026). These systems show that structured composition unlocks the value of individual skills, but they treat synthesis as organization followed by one-shot instantiation, leaving the task generator fixed once built. Skill2Task organizes skills through abstract scenario states, so sampled paths are semantically coherent to chain and expand into multi-artifact workflows over files, documents, data, and APIs rather than single terminal commands, and it validates each component through staged repair loops. Crucially, rather than fixing the generator, Skill2Task makes the synthesis harness itself the object of self-evolution, revising its synthesis skills, prompts, and validators across rounds from execution diagnostics. This also separates Skill2Task from feedback-driven systems that improve the trained policy, its runtime harness, or the RL environment while leaving the data-synthesis pipeline untouched (Shinn et al., 2023; Zhang et al., 2026; Lee et al., 2026; Dong et al., 2026).

3 Method

3.1 Overview

Skill2Task synthesizes executable general-agent tasks from community skills through a self-evolving data loop. We first describe how skills are organized into a graph and sampled as compositional paths, then introduce the progressive environment-task-verifier validation pipeline, the feedback-driven self-evolution mechanism, and the trajectory curation procedure used for SFT.

3.2 Skill Graph Construction and Sampling

Skill filtering. Community skills provide broad coverage of real workflows, but raw collections also contain capabilities that cannot be safely or reproducibly instantiated in our sandbox. We assemble our raw pool from two public community skill sources, SkillsMP (SkillsMP, 2026) and ClawHub (OpenClaw, 2026); we retain only skills released under licenses that permit research use and record each skill's source, version, and license for reproducibility. We first normalize each skill

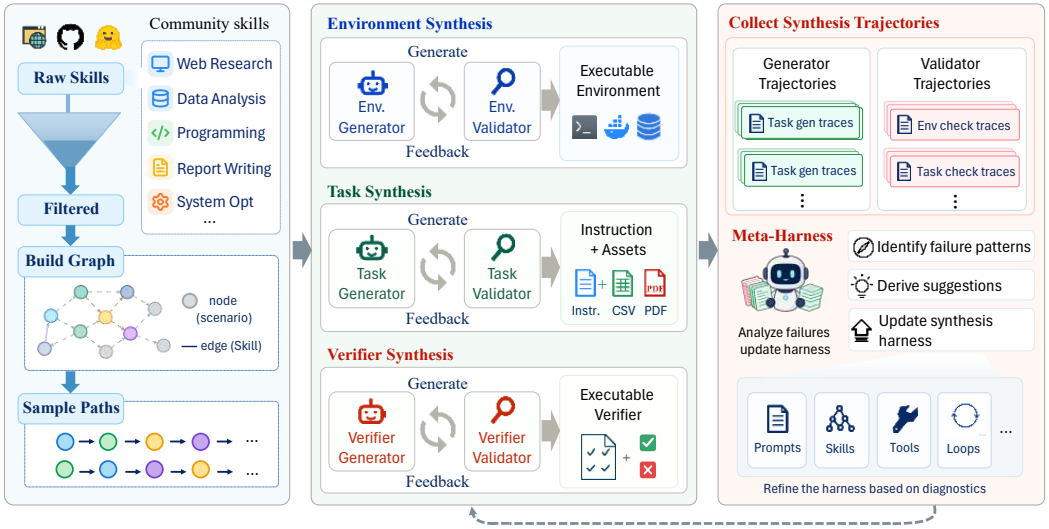


Figure 2: Overview of Skill2Task. The system forms a closed data loop that filters community skills, composes semantically coherent capabilities through a skill-state graph, progressively validates environment-task-verifier bundles, collects curated training trajectories, and updates the next synthesis distribution from execution feedback.

into

$$\kappa = (n, d, p, q, u, r, v), \tag{1}$$

where n is the name, d the domain, p and q the precondition and postcondition, u the required tool interface, and r, v risk and verification metadata. We then use `gpt-oss-120b` (Agarwal et al., 2025) to identify six exclusion conditions: UI interaction, login or authentication, API-key dependence, malicious behavior, real transactions, and GPU requirements. Skills matching any condition are removed, together with low-quality and duplicate entries; we also apply a strict multi-level decontamination filter against all evaluation benchmarks to remove retained skills that overlap with any evaluation task (Appendix A.7). This filtering removes 64,842 of 149,570 raw skills and retains 84,728.

Graph construction. A general-agent task often chains several skills whose input and output scenarios fit together, so we construct a skill-state graph

$$G = (\Omega, E), \quad e = (\sigma_i, \kappa, \sigma_j), \tag{2}$$

where Ω contains abstract scenario states and each skill κ forms an edge from its precondition state σ_i to its postcondition state σ_j. Each skill carries a textual precondition p and postcondition q describing these scenarios, not an executable specification. We embed them with Qwen3-Embedding-8B (Zhang et al., 2025) and consolidate near-equivalent scenarios into shared nodes by two-stage clustering that merges paraphrases while keeping distinct states apart (details in Appendix A.2). Each skill is inserted as a labeled edge between its merged nodes, discarding edges with inconsistent tool interfaces, yielding 10,457 scenario nodes and 83,264 skill-labeled edges. Since a node captures a skill's operating scenario rather than the concrete files, schemas, or data types it consumes, an edge sequence indicates only that skills are *semantically coherent* to chain; the concrete execution requirements are established downstream by the validation stages (Section 3.3).

Graph sampling. The graph provides compositional structure for constructing tasks that require more than one isolated capability. We sample a path

$$P = (\sigma_0, e_1, \sigma_1, e_2, \ldots, e_L, \sigma_L). \tag{3}$$

Starting from an initial scenario, each next edge is selected from the outgoing edges of the current scenario node. The path length L controls task horizon and complexity, while tool-availability and repeated-state constraints avoid clearly infeasible or cyclic compositions. The sampled path is then passed to the progressive task-bundle pipeline below, which turns a semantically coherent skill sequence into a concrete, executable task.

3.3 PROGRESSIVE TASK BUNDLE VALIDATION

Each sampled path is converted into an executable task bundle through three progressive generation-validation-repair loops—environment, task, and verifier—each building only on an already-validated upstream component, so that a detected failure is attributed to the stage that produced it rather than hidden behind a single final filter. We use GLM-5.2 (Z.ai, 2026) as the generator LLM for all three construction stages.

Environment loop. The first loop constructs only the runtime environment required by the sampled skill path. An environment constructor produces a Dockerfile containing system and Python dependencies, runtime configuration, and local mock services when external services are unavailable; it does not create task-specific files or other user assets. The environment validator checks dependency resolution, image build and startup success, service health, and sandbox compatibility. Failures trigger stage-local repair, and only a validated runtime environment is passed downstream.

Task loop. The second loop constructs the actual agent task on top of the validated runtime environment. The task authoring agent receives the environment and sampled skill path, writes the user-facing instruction, and creates task-specific initial assets such as spreadsheets, PDFs, datasets, configuration files, and required directories. The task validator checks consistency between the instruction, assets, environment, and skill path, rejecting tasks that are ambiguous, already solved, impossible, underspecified, unrelated to the sampled capability sequence, or dependent on unsupported external resources. When a mismatch is found, the validator returns a rubric-based critique and rule-based failure code so the task authoring agent can repair the task without changing the runtime environment.

Verifier loop. The third loop constructs and validates the executable verifier. The verifier constructor writes checks that evaluate the final environment state after an agent rollout, such as inspecting files, running tests, comparing structured outputs, querying local services, validating documents, or checking databases. The verifier validator then probes the verifier against reference correct final states, unchanged initial states, deliberately incorrect states, and shortcut attempts, and Skill2Task accepts the verifier only when it accepts the correct states, rejects all the others, and avoids artifacts the agent can trivially manipulate. A failed verifier is repaired using counterexamples from the validator.

3.4 SELF-EVOLVING TASK SYNTHESIS

After each round, Skill2Task turns the failures observed during construction into an update of the *synthesis harness* rather than the trained policy. Let θ_t denote the harness at round t, a tuple of editable artifacts $\theta_t = (\mathcal{S}_t, \mathcal{T}_t, \mathcal{V}_t)$ owned by the three stages: synthesis-skill records and prompts $\mathcal{S}_t$, environment templates $\mathcal{T}_t$, and validators $\mathcal{V}_t$. Self-evolution collects a diagnostic record, aggregates it into failure patterns, and proposes an edit accepted only if it raises construction quality.

Diagnostic feedback. Every construction attempt on a sampled path emits a structured event

$$z = (\text{domain},\ L,\ s,\ \text{mode},\ a), \tag{4}$$

recording the skill domain, path length L, producing stage $s \in \{\text{env}, \text{task}, \text{verifier}\}$, failure or success mode, and implicated artifact a (the skill, template, or validator involved); a rejection also carries the validator critique that triggered it, tying each defect to the stage and artifact that caused it.

Failure aggregation. We collect the round-level event set $Z_t = \{z_i\}$ and, since recurring mechanisms matter more than isolated errors, group the rejected events by stage and failure mode into clusters $\{C_k\}$, scoring each by how often it recurs and how much it lowers the per-stage acceptance rate. The dominant clusters form a compact report

$$R_t = \big\{\ (C_k,\ a_k,\ \rho_k)\ \big\}_k, \tag{5}$$

where a_k is the most implicated artifact of C_k and ρ_k its rejection share. This separates generation from execution failures: recurrent dependency errors concentrate on environment templates, whereas failures on coherent paths concentrate on the responsible skills, prompts, or validators.

Update and acceptance. An update operator $\mathcal{U}$, an evolution LLM instantiated with GLM-5.2 (Z.ai, 2026), proposes a revised harness $\theta'_{t+1} = \mathcal{U}(\theta_t, R_t)$ by editing only the implicated artifacts $\{a_k\}$, with no human editing. Proposals are evaluated *only* on construction quality: we re-run construction with θ'_{t+1} on a fixed held-out path set disjoint from the feedback set and measure the held-out valid-bundle rate $Y(\theta)$, the fraction of paths whose full environment–task–verifier bundle passes validation. The update is accepted only if it does not lower this rate and two guardrails hold, preventing inflation by collapsing difficulty or diversity:

$$\theta_{t+1} = \begin{cases} \theta'_{t+1} & \text{if } Y(\theta'_{t+1}) \geq Y(\theta_t) \text{ and guardrails hold,} \\ \theta_t & \text{otherwise,} \end{cases} \tag{6}$$

where the guardrails require the target path-length distribution to be preserved and the teacher rollout solve rate to stay within a fixed band. Any change in downstream performance is thus an observed consequence, not a selection target (setup and evolution curve in Section 4.3).

3.5 Trajectory Curation and Learning

Outcome-level curation. We deploy each of the 10,916 validated task bundles with the default OpenClaw system prompt and use GLM-5.2 (Z.ai, 2026) as the rollout agent, independently sampling four candidate trajectories per task for 43,664 candidates in total, each rollout running in an isolated sandbox. A candidate is outcome-valid only when its final environment state satisfies the task's executable verifier, so the verifier judges whether a complete execution is suitable for training rather than individual intermediate actions. Of the 43,664 candidates, 10,880 are outcome-valid and retained for step-level curation.

Step-level curation. Each retained trajectory is segmented into interaction steps and annotated by behavior type using DeepSeek-V4-Pro (Xu et al., 2026) as the step-level labeler. The annotation scheme covers normal progress, tool-use errors, repetitive failed attempts, invalid recovery, premature termination, protocol violations, unsupported assumptions, and hallucinated observations. Erroneous steps remain in the interaction context but are marked as skip and excluded from the imitation loss, while the remaining responses serve as optimization targets. This selective masking avoids imitating flawed intermediate behavior without destroying the causal context needed to understand later actions. Because step-level curation masks individual steps but discards no trajectory, all 10,880 outcome-valid trajectories are used for supervised fine-tuning.

4 Experiments

4.1 Experimental Setup

Base models. We fine-tune the released Qwen3.5-35B-A3B and Qwen3.5-9B (Qwen Team, 2026) models. Throughout the paper, **Skill2Task-35B-A3B** and **Skill2Task-9B** denote these backbones trained on curated trajectories synthesized by our pipeline.

Evaluation. We evaluate on four complementary agent benchmarks: **PinchBench** (Kilo Code, 2026) for general computer-use tool competence, **SkillsBench** (Li et al., 2026b) for invoking and composing community skills, **SWE-bench Multilingual** (Yang et al., 2025) for repository-level issue resolution, and **Terminal-Bench 2.0 Verified** (Merrill et al., 2026) for long-horizon command-line tasks. We use the official 2.0 Verified split for Terminal-Bench and the with-skill setting for SkillsBench. All benchmarks report mean accuracy over independent runs—avg@8 for PinchBench, whose small task set benefits from more runs, and avg@4 for the others; per-run statistics are in Table A.5 and per-benchmark task counts and harnesses in Appendix A.5.

4.2 Main Results

For a fair comparison, we keep the teacher consistent by re-rolling out the baselines' original tasks and environments with the *same* GLM-5.2 policy used by Skill2Task and retaining only the trajectories that pass each task's own checker, yielding 17,734 ClawGym and 18,004 SWE-smith trajectories, versus 10,880 curated Skill2Task trajectories. Skill2Task improves both backbones on all

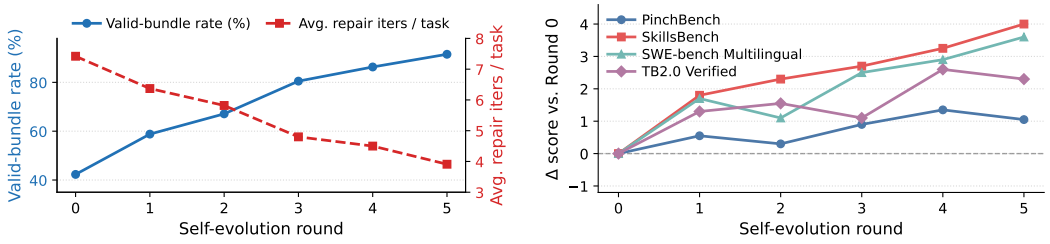
Table 1: Main benchmark results. Within each block, the first row is the SFT-free backbone and the rows below fine-tune it on ClawGym (Bai et al., 2026), SWE-smith (Yang et al., 2025), and Skill2Task data under a matched recipe; **Avg.** is the mean over the four benchmarks, the best score in each block is in **bold**, and subscripts are changes over the backbone (green for gains, red for drops).

Model (training data)	Avg.	PinchBench	SkillsBench	SWE-bench Multi.	TB2.0 Verified
Qwen3.5-9B	37.43	75.64	11.17	42.67	20.22
+ ClawGym	33.39 $_{-4.04}$	80.34 $_{+4.70}$	4.88 $_{-6.29}$	25.58 $_{-17.09}$	22.75 $_{+2.53}$
+ SWE-smith	38.03 $_{+0.60}$	79.18 $_{+3.54}$	5.63 $_{-5.54}$	44.00 $_{+1.33}$	23.31 $_{+3.09}$
+ Skill2Task (ours)	**44.21** $_{+6.78}$	**80.89** $_{+5.25}$	**22.45** $_{+11.28}$	**45.67** $_{+3.00}$	**27.81** $_{+7.59}$
Qwen3.5-35B-A3B	45.45	78.53	19.83	55.33	28.09
+ ClawGym	38.14 $_{-7.31}$	81.58 $_{+3.05}$	7.32 $_{-12.51}$	33.33 $_{-22.00}$	30.34 $_{+2.25}$
+ SWE-smith	44.30 $_{-1.15}$	79.42 $_{+0.89}$	10.86 $_{-8.97}$	57.42 $_{+2.09}$	29.49 $_{+1.40}$
+ Skill2Task (ours)	**51.35** $_{+5.90}$	**82.06** $_{+3.53}$	**28.39** $_{+8.56}$	**59.00** $_{+3.67}$	**35.96** $_{+7.87}$

four benchmarks and outperforms both baselines on every benchmark. The two baselines are also narrow: relative to the SFT-free 35B backbone, ClawGym's terminal-command trajectories degrade both SkillsBench and SWE-bench Multilingual, while SWE-smith's repository-repair trajectories help SWE-bench Multilingual but collapse SkillsBench (19.83→10.86), as neither exercises multi-skill composition. Using fewer training trajectories and fewer total tokens than either baseline (Appendix A.6), Skill2Task improves all four benchmarks with the largest gains on SkillsBench, so the improvement stems from data quality and coverage rather than sheer volume.

4.3 Self-Evolution of the Synthesis Harness

The self-evolved harness produces the task pool used in all other experiments, so we isolate its effect first. From a fixed probe set of 500 skill paths stratified over domains and path lengths, 300 *held-in* paths generate the diagnostic feedback Z_t and 200 disjoint *held-out* paths are used for evaluation. Each round the evolution LLM proposes a harness update from the held-in diagnostics, which we keep only if it does not lower the held-out valid-bundle rate while preserving the target path-length distribution and the teacher solve rate, so the gate cannot inflate the rate by collapsing difficulty. The four benchmarks are never consulted during selection, so the only changing factor is the accepted harness.



(a) Valid-bundle rate and repair cost per round.

(b) Per-benchmark score gains.

Figure 3: Self-evolution of the synthesis harness over five rounds; Round 0 is the static harness. (a) Overall held-out valid-bundle rate, the fraction of held-out paths passing full validation after repair (left axis), and average total repair iterations per successfully constructed task (right axis) at each round. (b) Score of a model trained on each round's output, as improvement over Round 0.

Figure 3 shows that on the held-out paths the valid-bundle rate rises from 42.3% under the static harness to 91.5% at Round 5, while the average repair iterations per constructed task fall from 7.42 to 3.91, so self-evolution makes construction both more reliable and more efficient. Fine-tuning a fresh Qwen3.5-35B-A3B on each round's output tests whether better construction yields better data; we hold the skill paths, task count, and teacher fixed across rounds, so the harness is the only

changing factor (Appendix A.9). As the benchmarks never enter the gate, their trend is independent evidence: under this matched budget the four-benchmark mean rises from 46.40 to 49.14, though not monotonically—it plateaus at Round 2, and benchmarks peak at different rounds since the gate never optimizes them. Because every round is capped well below the full pool, these scores isolate harness quality at fixed data volume; scaling the Round 5 harness to the full $10{,}916$-task pool then yields the 51.35 mean of Table 1.

4.4 Ablation Study

Holding the final self-evolved harness fixed, we isolate the data construction choices that directly affect SFT: whether skill sampling uses the skill-state graph rather than randomly combining skills with no path structure, and whether trajectories are filtered at the outcome and step levels. All settings use the same backbone, evaluation protocol, and matched task count, so differences reflect the data source and curation rather than dataset size.

Table 2: Data construction ablation on Qwen3.5-35B-A3B, reported as avg@8 on PinchBench and avg@4 on the other benchmarks. The table compares skill sampling and trajectory filtering components under the same SFT setting and matched task count.

Setting	Graph	Outcome	Step	PinchBench	SkillsBench	SWE-bench Multi.	TB2.0 Verified
Random skills	–	✓	✓	79.21	25.08	54.33	31.74
No filtering	✓	–	–	79.68	23.62	54.00	31.18
Outcome only	✓	✓	–	81.16	26.55	56.92	33.99
Full Skill2Task	✓	✓	✓	82.06	28.39	59.00	35.96

Table 2 shows that each component contributes to the final model. Replacing graph-guided sampling with random skill combinations and removing outcome- and step-level filtering both cause large drops relative to full Skill2Task, indicating that coherent skill paths and trajectory curation provide substantial and complementary supervision. Step-level filtering adds a further gain over outcome filtering alone by preserving context while excluding flawed intermediate actions.

5 Analysis

5.1 Scaling with Path Length and Task Count

We analyze how performance scales with two properties of the synthesized task distribution, skill path length and task count, fine-tuning Qwen3.5-35B-A3B and varying only the analyzed property. Skill path length controls how many connected capabilities a task requires. To isolate length from scale, every length uses the *same* matched budget of $2{,}000$ tasks drawn only from that length, so all path-length points use far fewer tasks than the full model and none reaches it. Under this budget, path length 1 to 7 raises the four-benchmark mean from 46.22 to 49.50 (+3.28) but not monotonically—SkillsBench gains most (+5.31) with a clear jump from length 1 to 2, while the other benchmarks improve mainly beyond length 3–4. The task-count axis instead fixes the full path-length mixture and grows the validated tasks from 10 up to the rightmost 10^4 tick on a $\log_{10}$ scale, which corresponds to the full released pool of 10,916 tasks. Here the mean rises from 44.87 to 51.35 (+6.48): at 10 and 100 tasks only SkillsBench improves while the others stay at or below the SFT-free backbone, 1,000 tasks give a small gain across all four, and the pronounced growth appears only over 1,000→5,000 and 5,000→10,000 (setup in Appendices A.10–A.12).

5.2 Progressive Validation

Progressive validation substantially improves the quality and yield of synthesized task bundles. Without validation, generated tasks often fail because key environment artifacts are missing, instructions are underspecified, or verifiers accept incomplete solutions and reject valid alternatives. The three local repair iterations correct these issues before rollout. Figure 5a plots the cumulative success rate after each repair iteration under the final self-evolved harness, showing that most failures are recovered through local repair rather than resampling the full skill path. For the complete

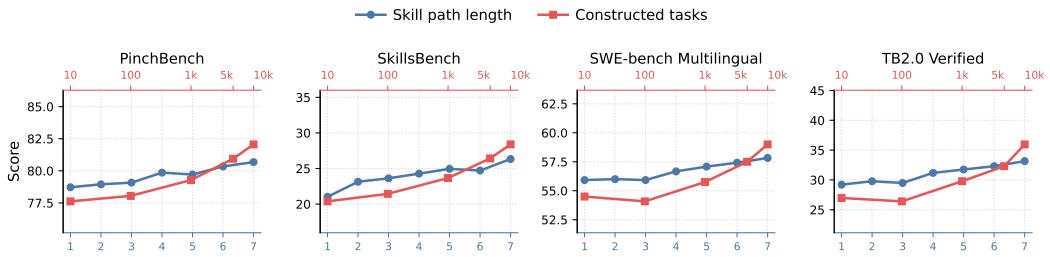
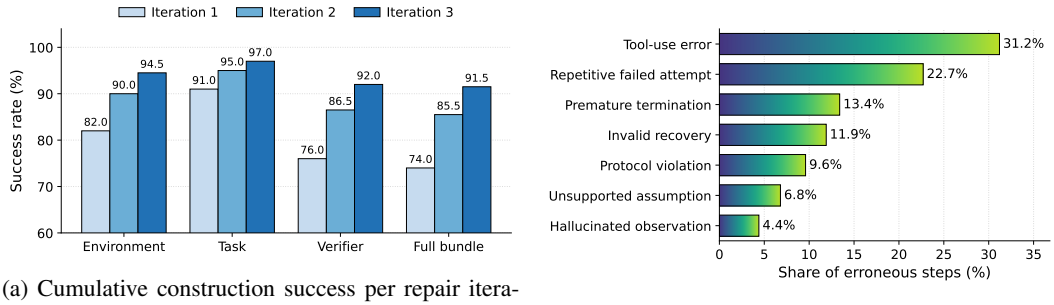


Figure 4: Benchmark scaling with sampled path length and constructed task count on Qwen3.5-35B-A3B. The blue curve (bottom axis) varies skill path length under a matched per-length task budget; the red curve (top axis) grows the validated task count on a $\log_{10}$ scale. Each axis is colored to match its curve.



(a) Cumulative construction success per repair iteration.

(b) Distribution of erroneous step categories.

Figure 5: Analysis of task construction and trajectory quality. (a) Cumulative construction success after each progressive repair iteration, for the environment, task, and verifier stages and the full task bundle. (b) Distribution of erroneous step categories identified by DeepSeek-V4-Pro on candidate trajectories, with percentages computed over steps annotated as non-normal.

environment–task–verifier bundle, cumulative success rises from 74.0% after the first repair iteration to 91.5% after the third. This within-round repair complements harness self-evolution, which raises the first-attempt construction quality across rounds (Section 4.3).

5.3 Step-Level Error Distribution

Step-level curation (Section 3.5) relies on annotating erroneous interaction steps so they can be excluded from the imitation loss. Across the 10,880 outcome-valid trajectories, DeepSeek-V4-Pro annotates roughly 137,000 interaction steps, of which about 9% are labeled non-normal and masked as skip while the rest remain as optimization targets. Figure 5b shows how these erroneous annotations are distributed. Tool-use errors and repetitive failed attempts dominate, indicating that fine-grained masking mostly acts on execution-level mistakes rather than higher-level protocol or hallucination issues.

6 Conclusion

We presented Skill2Task, a self-evolving system for synthesizing executable general-agent tasks from community skills. Skill2Task composes filtered skills through a skill-state graph and converts sampled capability paths into task bundles through progressive environment-task-verifier validation, with execution and verifier feedback continually updating the synthesis harness. It yields 10,916 verifiable tasks, and curating the rollouts at both the outcome and step level produces 10,880 training trajectories on which supervised fine-tuning consistently improves both Qwen3.5-9B and Qwen3.5-35B-A3B across four agent benchmarks.

REFERENCES

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Fei Bai, Huatong Song, Shuang Sun, Daixuan Cheng, Yike Yang, Chuan Hao, Renyuan Li, Feng Chang, Yuan Wei, Ran Tao, et al. Clawgym: A scalable framework for building effective claw agents. *arXiv preprint arXiv:2604.26904*, 2026.

Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, et al. Agent-world: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*, 2026.

Zhiyuan Fan, Tinghao Yu, Yuanjun Cai, Jiangtao Guan, Yun Yang, Dingxin Hu, Jiang Zhou, Xing Wu, Zhuo Han, Feng Zhang, et al. Toward scalable terminal task synthesis via skill graphs. *arXiv preprint arXiv:2604.25727*, 2026.

Kanishk Gandhi, Shivam Garg, Noah D Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling rl environments for terminal agents. *arXiv preprint arXiv:2601.16443*, 2026.

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

Stephen C Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, 1967.

Kilo Code. PinchBench: Benchmarking llm models as openclaw agents. `https://github.com/pinchbench/skill`, 2026. Accessed: 2026-07-20.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

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

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026b.

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

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the eleventh international conference on language resources and evaluation (LREC 2018)*, 2018.

Yusong Lin, Haiyang Wang, Shuzhe Wu, Lue Fan, Feiyang Pan, Sanyuan Zhao, and Dandan Tu. Cli-gym: Scalable cli task generation via agentic environment inversion. *arXiv preprint arXiv:2602.10999*, 2026.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

OpenClaw. ClawHub: Open skill registry for ai agents, 2026. URL `https://clawhub.ai`. Accessed: 2026-07-20.

Renjie Pi, Grace Lam, Mohammad Shoeybi, Pooya Jannaty, Bryan Catanzaro, and Wei Ping. On data engineering for scaling llm terminal capabilities. *arXiv preprint arXiv:2602.21193*, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

SkillsMP. SkillsMP: A community marketplace for agent skills, 2026. URL https://skillsmp.com. Accessed: 2026-07-20.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, et al. Large-scale terminal agentic trajectory generation from dockerized environments. *arXiv preprint arXiv:2602.01244*, 2026.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

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

Z.ai. GLM-5.2: Built for long-horizon tasks, June 2026. URL https://huggingface.co/zai-org/GLM-5.2. Model card.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

Kaijie Zhu, Yuzhou Nie, Yijiang Li, Yiming Huang, Jialian Wu, Jiang Liu, Ximeng Sun, Zhenfei Yin, Lun Wang, Zicheng Liu, et al. Termigen: High-fidelity environment and robust trajectory synthesis for terminal agents. *arXiv preprint arXiv:2602.07274*, 2026.

A Additional Results

This appendix provides implementation details and supplementary results that support the main paper. We include the synthesis algorithm, rollout inference and SFT training hyperparameters, main results with run-to-run variance, tabular values for scaling analyses, case studies of progressive repair and harness self-evolution, and a discussion of limitations and future work.

A.1 Progressive Synthesis Algorithm

Algorithm 1 details the progressive generation-validation-repair pipeline of Section 3.3. Each sampled skill path is expanded stage by stage—environment, then instruction, then verifier—with a stage-local repair loop attempted before falling back to path resampling, so failures are localized and a single stage error does not discard an otherwise valid path.

Algorithm 1 Skill2Task progressive task synthesis.

Filter and classify community skills
Build skill-state graph $G = (\Omega, E)$
for evolution round $t = 1, \ldots, 5$ **do**
  Sample a batch of compatible paths $\mathcal{P}_t$ from G
  for each $P \in \mathcal{P}_t$ **do**
    repeat
      Construct environment x_0 from P
      Validate structure, dependencies, and initial state
    until x_0 passes or retry budget is exhausted
    repeat
      Author task instruction y from (P, x_0)
      Validate semantic and cross-stage consistency
    until y passes or retry budget is exhausted
    repeat
      Construct verifier V for (P, x_0, y)
      Validate V with positive and negative cases
    until V passes or retry budget is exhausted
    Deploy (x_0, y, V) for rollout and SFT curation
  end for
  Aggregate validation and rollout diagnostics
  Automatically update synthesis skills, prompts, templates, and validators
end for

A.2 Skill-State Graph Construction Details

This appendix expands the scenario-node consolidation used in the skill-state graph construction of Section 3.2. We embed each skill's precondition and postcondition scenario description with Qwen3-Embedding-8B (Zhang et al., 2025) and ℓ_2-normalize the embeddings. To avoid the quadratic memory cost of global hierarchical clustering, we consolidate scenarios in two stages. First, we build a sparse semantic similarity graph by connecting each scenario to its nearest neighbors above a cosine similarity of 0.8 and apply Louvain community detection (Blondel et al., 2008) to obtain coarse buckets. Then, within each bucket, we run complete-linkage agglomerative clustering (Johnson, 1967) with cosine distance; complete linkage requires every pair within a cluster to be mutually close, which prevents chain-drift artifacts where transitively linked but semantically distant scenarios (A close to B, B close to C, but A unlike C) are merged into one node. We select the agglomerative distance threshold by sweeping on held-out scenario samples and inspecting the resulting clusters, balancing the merging of clear paraphrases and near-equivalent states against over-fragmentation, and take each final cluster as a single scenario node. After consolidation, each skill is inserted as a labeled edge between its merged input- and output-scenario nodes, and we discard edges whose declared tool interfaces are inconsistent, yielding 10,457 scenario nodes and 83,264 skill-labeled edges.

A.3 SFT TRAINING HYPERPARAMETERS

This subsection summarizes the supervised fine-tuning configuration used for all Skill2Task-SFT experiments. The settings are kept fixed across main results and ablations unless otherwise stated.

Table A.1: SFT hyperparameters used for Skill2Task-SFT training.

Hyperparameter	Value
Global batch size	64
Epochs	3
Pack max length	262,144 tokens
Learning rate	2×10^{-5}
Minimum learning rate	1×10^{-6}
Weight decay	0.05
Warmup ratio	0.1
Learning-rate schedule	cosine
Recompute ratio	1.0
Loss reduction	square
Training hardware	32 NVIDIA H200 GPUs

Optimization and recomputation. The global batch size counts packed sequences across all workers in one optimizer update. The learning rate is warmed up over the first 10% of optimizer steps to a peak value of 2×10^{-5} and then decayed with a cosine schedule to 1×10^{-6}. A recompute ratio of 1.0 applies activation checkpointing to all Transformer layers: intermediate activations are recomputed during the backward pass to reduce training memory at the cost of additional computation.

Sequence packing. We set both the per-sample tokenization limit and the packed-sequence limit to 262,144 tokens. The former bounds the length of an individual trajectory after tokenization, while the latter bounds the total number of tokens after multiple trajectories are grouped into one packed training sequence. Packing improves utilization for variable-length trajectories; attention boundaries and loss masks keep the packed trajectories logically separate. All SFT runs use 32 NVIDIA H200 GPUs.

Square loss reduction. Let n_i be the number of effective supervised tokens in trajectory i, excluding tokens masked with the ignore index, and let $\ell_{i,t}$ denote the token-level cross-entropy. Under `square` reduction, every effective token in trajectory i receives weight $n_i^{-1/2}$, followed by normalization over all effective token weights in the global batch:

$$\mathcal{L}_{\text{square}} = \frac{\sum_i n_i^{-1/2} \sum_{t=1}^{n_i} \ell_{i,t}}{\sum_i \sqrt{n_i}}. \tag{7}$$

Consequently, the aggregate contribution of a trajectory grows as $\sqrt{n_i}$ rather than linearly with its length. This choice is intermediate between token reduction, which gives longer trajectories proportionally more weight, and sample reduction, which assigns equal aggregate weight to every trajectory.

A.4 ROLLOUT INFERENCE HYPERPARAMETERS

We use GLM-5.2 to generate four independent rollout trajectories for each constructed task and initialize the agent with the default OpenClaw system prompt. Every trajectory is executed in an isolated sandbox with fixed compute and wall-clock limits. Table A.2 reports both the inference configuration and the sandbox limits used for trajectory collection.

A.5 BENCHMARK EVALUATION CONFIGURATION

The rollout configuration in Appendix A.4 governs *training-trajectory collection* on our synthesized tasks; it is distinct from the *evaluation* configuration used to score models on the four benchmarks,

Table A.2: Rollout inference hyperparameters used for trajectory collection.

Hyperparameter	Value
Rollout model	GLM-5.2
System prompt	OpenClaw default
Samples per task	4
`temperature`	0.7
`top_p`	0.9
`max_tokens`	200,000
Sandbox CPU limit	4 cores
Sandbox memory limit	16 GB
Wall-clock timeout	30 minutes

which we detail here. For each benchmark we follow its official protocol and use its recommended agent harness, so scores are comparable to prior reports. Table A.3 lists the number of tasks and the agent harness for each benchmark. As in the main text, PinchBench is evaluated with avg@8 and the other three benchmarks with avg@4 (independent runs with different rollout seeds), and no benchmark task is used for training.

Table A.3: Per-benchmark evaluation configuration. Task counts are the sizes of the official evaluation splits, and each benchmark is scored with its recommended agent harness.

Benchmark	Tasks	Agent harness
PinchBench	23	OpenClaw (2026.5.7)
SkillsBench	87	OpenClaw (2026.5.7)
SWE-bench Multilingual	300	mini-swe-agent (Yang et al., 2024)
Terminal-Bench 2.0 Verified	89	Terminus 2

A.6 Baseline Training Data

The ClawGym (Bai et al., 2026) and SWE-smith (Yang et al., 2025) baselines in Table 1 are trained under the *same* teacher as Skill2Task, so differences reflect the tasks and environments rather than the rollout model. We do not reuse either baseline's publicly released trajectories—produced by different teachers—but take their original tasks and executable environments and re-collect trajectories with the same GLM-5.2 policy (Appendix A.4) in a pass@4 setting, keeping only trajectories that the task's own checker scores with reward 1. This yields 17,734 checker-passing trajectories from ClawGym and 18,004 from SWE-smith. All three sources thus share the same rollout model, sampling budget, success-only selection rule, and SFT recipe (Appendix A.3); only the tasks and environments differ.

Table A.4 reports the resulting training budget for each source, in both trajectory count and total supervised-fine-tuning tokens. Skill2Task uses the fewest trajectories (10,880 vs. 17,734 and 18,004) and the fewest total tokens (288,223,349 vs. 381,217,005 and 579,109,652), i.e. roughly $0.76\times$ the ClawGym token budget and $0.50\times$ the SWE-smith token budget, yet achieves the best score on every benchmark in Table 1. This confirms that the gains come from the composition and curation of the tasks rather than from a larger training budget.

Table A.4: Training data budget for each data source under the teacher-consistent protocol, measured in curated trajectories and total supervised-fine-tuning tokens. Skill2Task uses the fewest of both.

Data source	Trajectories	SFT tokens
ClawGym (Bai et al., 2026)	17,734	381,217,005
SWE-smith (Yang et al., 2025)	18,004	579,109,652
Skill2Task (ours)	10,880	288,223,349

We note that SkillSynth (Fan et al., 2026) is a skill-composition pipeline (Section 2) that could in principle serve as a data baseline, but neither its code nor its synthesized tasks and environments are

publicly released. Because our teacher-consistent protocol requires re-rolling out each baseline's original executable tasks and environments, SkillSynth cannot be reproduced or fairly compared under this setting, so we restrict the data baselines to ClawGym and SWE-smith, whose tasks and environments are available.

A.7 Decontamination Protocol

Because Skill2Task's training skills are drawn from community registries (SkillsMP and ClawHub) that also supply skill-centric benchmarks, we apply a strict, multi-level decontamination procedure so that no training skill overlaps with any evaluation task. The exact name/description filter mentioned in Section 4 is only the first level; on its own it is a surface string filter and cannot rule out paraphrased or semantically equivalent overlap, so we augment it with two additional levels. We run all three levels against *every* benchmark used in this paper—PinchBench, SkillsBench, SWE-bench Multilingual, and Terminal-Bench 2.0 Verified—not only SkillsBench, and remove any training skill flagged at any level before synthesis.

For each candidate training skill we form a query from its name, description, and precondition/postcondition scenario text, and compare it against every evaluation task's specification at three increasingly semantic levels:

- **String (exact).** After normalizing case, whitespace, and punctuation, we drop any training skill whose name or description exactly matches that of an evaluation task.
- **n-gram (lexical).** We compute the word-level Jaccard similarity over $\{3, 4, 5\}$-grams and drop any training skill whose maximum n-gram overlap with an evaluation task exceeds a conservative Jaccard threshold of 0.25, which catches paraphrases, reordered instructions, and partial copies that evade exact matching.
- **Embedding (semantic).** We embed both the training-skill query and each evaluation task with Qwen3-Embedding-8B (Zhang et al., 2025) and drop any training skill whose cosine similarity to an evaluation task exceeds a conservative threshold of 0.75, which catches semantically equivalent tasks expressed with disjoint vocabulary.

Both thresholds are set aggressively—a low n-gram Jaccard cutoff and a moderate embedding cutoff—and calibrated by manually inspecting borderline pairs, so the filter errs toward discarding legitimate skills rather than admitting near-duplicates. A training skill is removed if flagged at *any* level against *any* evaluation task, and every task whose sampled path uses a removed skill is excluded. Because skill-level filtering cannot rule out overlap introduced during construction, we re-run the same three-level check on the *final* artifacts—each constructed task's instantiated instruction and environment, and every training trajectory—against all evaluation tasks, dropping anything flagged before SFT. After applying this at both levels, no remaining skill, task, or trajectory is flagged by any of the three checks against any evaluation task, so to the resolution of this multi-level filter the reported gains—including the large SkillsBench improvement—reflect genuine capability transfer rather than detectable leakage.

A.8 Main Results with Variance

Table 1 reports single-number scores for readability. To quantify run-to-run stability, Table A.5 re-reports every row of Table 1—the SFT-free backbones, the ClawGym (Bai et al., 2026) and SWE-smith (Yang et al., 2025) fine-tuned backbones, and the Skill2Task models—giving the mean and standard deviation of benchmark scores across independent evaluation runs with different rollout seeds (avg@8 for PinchBench, avg@4 for the others). Because every task is scored $0/1$, the deviation scales with the benchmark size: it is largest on PinchBench, whose 23 tasks make each flipped task worth $\approx$4.3 points, and shrinks on the larger benchmarks. On SkillsBench, SWE-bench Multilingual, and Terminal-Bench the Skill2Task gains are several times this deviation, so those improvements are not attributable to evaluation noise; on the small PinchBench set the deviation is comparable to the gain, and we treat its scores as indicative rather than decisive.

Table A.5: Main results with variance. Each entry reports the mean and standard deviation over independent evaluation runs (avg@8 for PinchBench, avg@4 for the others), expanding every row of Table 1; **Avg.** is the mean over the four benchmarks and the best score in each block is in **bold**. Deviations are larger on the smaller benchmarks (PinchBench has only 23 tasks, so a single flipped task shifts a run by ≈ 4.3 points), yet on SkillsBench, SWE-bench Multilingual, and Terminal-Bench the gains from Skill2Task remain several times the run-to-run deviation.

Model (training data)	Avg.	PinchBench	SkillsBench	SWE-bench Multi.	TB2.0 Verified
Qwen3.5-9B	37.43±0.94	75.64±3.38	11.17±1.02	42.67±0.48	20.22±1.24
+ ClawGym	33.39±0.85	80.34±3.02	4.88±0.74	25.58±0.42	22.75±1.30
+ SWE-smith	38.03±0.88	79.18±3.12	5.63±0.78	44.00±0.49	23.31±1.32
+ Skill2Task (ours)	**44.21**±0.91	**80.89**±2.98	**22.45**±1.46	**45.67**±0.50	**27.81**±1.42
Qwen3.5-35B-A3B	45.45±0.95	78.53±3.22	19.83±1.38	55.33±0.49	28.09±1.43
+ ClawGym	38.14±0.85	81.58±2.90	7.32±0.86	33.33±0.46	30.34±1.48
+ SWE-smith	44.30±0.90	79.42±3.10	10.86±1.00	57.42±0.48	29.49±1.46
+ Skill2Task (ours)	**51.35**±0.91	**82.06**±2.82	**28.39**±1.62	**59.00**±0.47	**35.96**±1.58

A.9 Per-Round Training Setup for Self-Evolution

Figure 3(b) trains one model per evolution round and reports its score as an improvement over Round 0, so the comparison is only meaningful if every round trains under matched conditions with the harness as the sole changing factor. The construction-quality gate reuses a single fixed split across rounds: the same 200 held-out skill paths (disjoint from the 300 held-in paths that generate the diagnostics) are re-evaluated every round, so the valid-bundle rate and the average repair iterations per task in Figure 3(a) are directly comparable across all five rounds. For the per-round *training* models we hold three further quantities fixed. First, the *skill paths* are identical: we sample one fixed set of paths and re-run construction on exactly these paths with each round's accepted harness, so a round differs only in how well the same paths become valid bundles. Second, the *training-task count* is matched: because a stronger harness validates a larger fraction of the same paths, we cap every round to a common budget of validated tasks—well below the full 10,916 pool—and subsample down to it when a round yields more, so no round is advantaged by more data. Third, the *teacher and curation* are unchanged: every round uses the same GLM-5.2 policy, four-sample budget, and outcome- and step-level curation (Appendices A.4 and A.3). Under these controls the only variable behind the Figure 3(b) curve is the harness itself; the released Skill2Task pool instead uses the full task budget from the accepted Round 5 harness.

A.10 Scaling Experiment Setup

This subsection details the two scaling studies in Section 5.1 (Figure 4 and Tables A.6 and A.7). Both vary a single property of the synthesized task distribution while holding everything else fixed: the base model (released Qwen3.5-35B-A3B), the optimization configuration (Appendix A.3), the rollout and curation pipeline, and the four-benchmark evaluation protocol are identical across all points. Each point trains a fresh copy of the backbone from the same initialization, so differences reflect only the analyzed property and not whether SFT is applied.

Two independent one-dimensional slices. The two curves are *orthogonal one-dimensional slices* of the synthesis distribution, and—unlike a single nested sweep—they are *not* constrained to meet at a common endpoint. Each axis isolates one property under a control designed for that property:

- **Path-length axis (Table A.6).** To compare lengths fairly, every length must use the *same* amount of data; otherwise a longer-path setting would be confounded with simply having more tasks. We therefore fix a matched budget of 2,000 tasks for each length $1, \ldots, 7$ and draw those tasks exclusively from paths of that length. Because $2{,}000 \ll 10{,}916$, *no* path-length point uses the full pool, and length 7 under this budget is a strict length-only slice that is markedly smaller than the released model. The path-length curve consequently sits below the full-pool score at every length and does not reach the released model at its right end.

- **Task-count axis (Table A.7).** We fix the path-length distribution to the full released mixture and vary only the number of constructed tasks, drawing nested subsets of sizes 10, 100, 1,000, 5,000, and the full 10,916. Only the largest point, 10,916 tasks, is the complete task pool and hence the released Skill2Task-35B-A3B model.

Because the path-length axis caps every setting at a small matched budget while only the task-count axis grows to the full pool, the two curves have *different* right endpoints: the task-count curve terminates at the released model, whereas the path-length curve terminates well below it. This is by design—the path-length study answers "holding data fixed, how much does longer composition help?", for which the full-pool point would be an unfair comparison, while the task-count study answers "holding composition fixed, how much does more data help?". Their lower ends also differ: length 1 trains on 2,000 single-skill tasks, whereas 10 tasks trains on the full path-length mixture over only 10 tasks.

A.11 Path-Length Ablation

Table A.6 isolates the effect of sampled skill path length while holding the training budget fixed at a matched 2,000 tasks per length (setup in Appendix A.10). Because the data budget is held constant and small, differences reflect the compositional signal available at each length rather than dataset size, and every point—including length 7—uses far fewer tasks than the full pool, so none of these points corresponds to the Skill2Task-35B-A3B model in Table 1.

Table A.6: Effect of skill path length (Qwen3.5-35B-A3B; avg@8 on PinchBench, avg@4 elsewhere). Each length trains on a matched budget of 2,000 tasks drawn only from that length, so differences reflect path composition rather than scale; every point uses far fewer tasks than the full model and none coincides with it. Corresponds to the blue curve of Figure 4.

Length	PinchBench	SkillsBench	SWE-bench Multi.	TB2.0 Verified
1	78.72	21.02	55.92	29.21
2	78.95	23.14	56.00	29.78
3	79.08	23.62	55.92	29.49
4	79.86	24.28	56.67	31.18
5	79.71	24.95	57.08	31.74
6	80.34	24.71	57.42	32.30
7	80.68	26.33	57.83	33.15

A.12 Task-Count Ablation

Table A.7 reports benchmark scores as the number of validated constructed tasks used for SFT grows from 10 through 5,000 to the full pool of 10,916 on a $\log_{10}$ axis, with the path-length mixture and all other components held fixed (setup in Appendix A.10). The largest point, 10,916 tasks, is the full released pool and coincides with the Skill2Task-35B-A3B model in Table 1; unlike the path-length curve, this axis is the only one of the two scaling curves in Figure 4 that reaches the released model.

Table A.7: Effect of constructed task count (Qwen3.5-35B-A3B; avg@8 on PinchBench, avg@4 elsewhere). Each row trains on a nested subset of the validated pool with the full path-length mixture fixed; the largest, 10,916 tasks, is the released Skill2Task-35B-A3B model in Table 1. Corresponds to the red curve of Figure 4.

Tasks	PinchBench	SkillsBench	SWE-bench Multi.	TB2.0 Verified
10	77.62	20.38	54.50	26.97
100	78.05	21.44	54.08	26.40
1,000	79.28	23.66	55.75	29.78
5,000	80.94	26.42	57.50	32.30
10,916	82.06	28.39	59.00	35.96

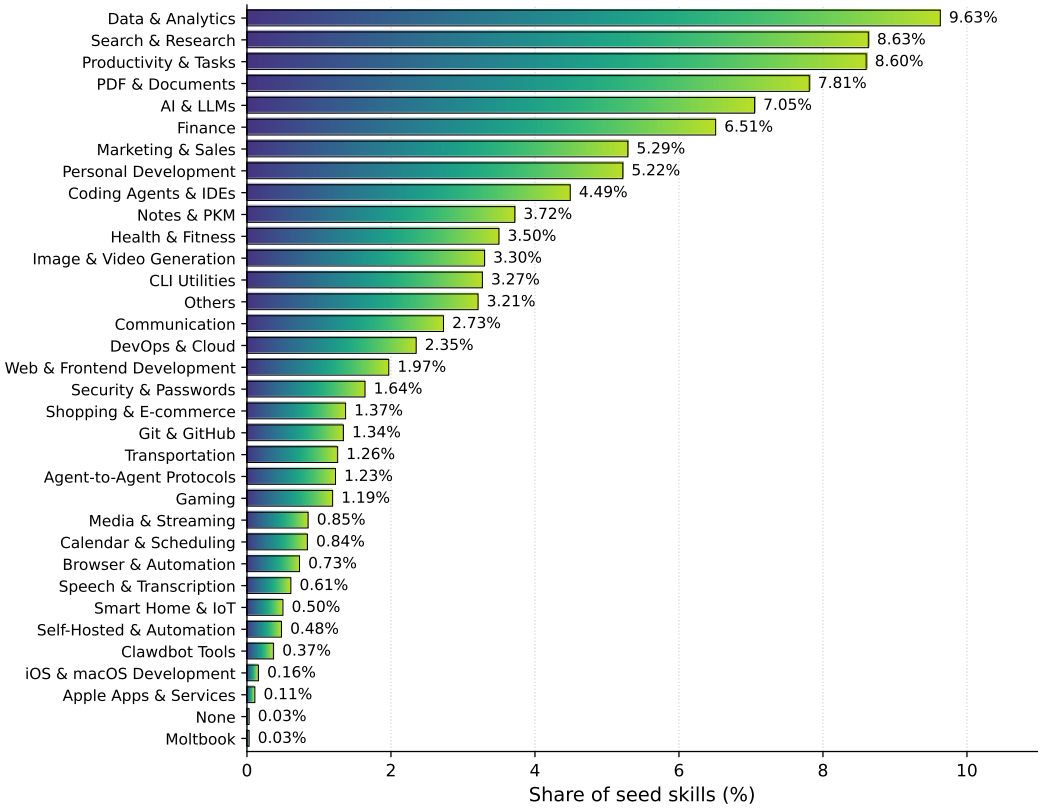


Figure A.1: Scene distribution of the initial skills underlying the final constructed task pool. Each of the 10,916 validated tasks is traced back to its seed skills, and skills are grouped by their community-assigned scene category; each bar is the share of that scene among all seed skills (summing to 100%).

A.13 Skill Scene Distribution

Figure A.1 reports the scene distribution of the initial community skills that seed the final constructed task pool. We trace each of the 10,916 validated tasks back to its seed skills and group those skills by their community-assigned scene category. The distribution is long-tailed but broad: no single scene exceeds 10% and the coverage spans 34 categories, from data analytics, research, and productivity workflows down to niche domains such as smart-home automation and speech transcription. This breadth indicates that the skill-state graph composes tasks from a diverse capability pool rather than concentrating on a few dominant domains.

A.14 Independent Verifier Reliability Audit

To evaluate verifier reliability independently of the automatic verifier-validation loop, we randomly sample 100 synthesized verifiers from the final task pool, stratified by task domain. The audit is carried out by two reviewers who are computer-science Ph.D. students and did not participate in authoring any of the tasks or verifiers. To keep the audit unbiased, both reviewers are kept blind to the task-generation method and to the automatic verification results: they see only the executable checks and their test cases, not which pipeline produced a verifier or whether the automatic validation loop had accepted it. For each verifier, they independently assess whether it rejects the unchanged initial state, accepts a reference correct final state, rejects deliberately incorrect final states, and resists common shortcut attempts. A verifier is counted as reliable only when it passes all four checks. The two reviewers label every verifier independently, and we report each criterion as the average of their two judgments before adjudication, so a verifier on which they disagree contributes 0.5 rather than 0 or 1; inter-reviewer agreement is measured with Cohen's κ on these pre-adjudication labels.

Table A.8: Independent manual audit of 100 synthesized verifiers by two computer-science Ph.D. students who did not author the tasks and are blind to the task-generation method and the automatic verification results. Each criterion is the average of the two reviewers' independent judgments before adjudication, so a verifier they disagree on contributes 0.5; percentages are over the audited sample.

Audit criterion	Passing verifiers	Rate (%)
Rejects unchanged initial state	100.0/100	100.0
Accepts reference correct state	99.5/100	99.5
Rejects deliberately incorrect states	99.0/100	99.0
Resists common shortcut attempts	98.5/100	98.5
All four checks (overall reliable)	97.0/100	97.0
Inter-reviewer agreement (Cohen's κ)	0.87	

The audit indicates that most verifiers satisfy the intended executable checks, while shortcut resistance is the most frequent weakness. These failures are fed back to the verifier-synthesis and verifier-validation prompts in subsequent evolution rounds.

A.15 Case Studies of Progressive Repair

The progressive pipeline of Section 3.3 is a per-stage generate–validate–repair loop: whenever a stage validator rejects its output, it returns a concrete critique and the constructor regenerates only that stage. Here we trace three real repair loops from our construction logs, one per stage, each showing the first-round output, the validator's critique, the regenerated output, and the final verdict. All three converge in two rounds, and together they illustrate that failures are attributed to—and fixed at—the stage that produced them, rather than caught by a single terminal filter.

Figure A.2 shows an *environment*-loop repair: the first `init.sh` installs an `npm` package that conflicts with the `nodejs` package already providing it, so the image fails to build with a held-broken-packages error; the second attempt removes the redundant line and the environment builds and starts. Figure A.3 shows a *task*-loop repair in which the first instruction chains skills whose data formats do not connect (an OrcaFlex `.sim` output feeding an HDF5-only tool), producing an all-zero output that a checker cannot distinguish from a correct result; the validator attributes the failure to the composition itself, and the rewrite grounds the task in a compatible artifact and adds recomputable statistics, making it verifiable. Figure A.4 shows a *verifier*-loop repair where the first rubric's check weights sum to 1.21 rather than 1.0; the scoring engine silently renormalizes, hiding a distorted weighting that over-rewards shallow structure checks, and the repair rebalances the weights to sum to one and shifts mass toward discriminative semantic checks.

These loops are not rare corner cases. Table A.9 reports, per stage, the fraction of constructed tasks whose first-round output is rejected and then repaired in a later round. Environments require at least one repair about half the time, and verifiers are the hardest component to synthesize correctly on the first attempt—the large majority need repair—whereas task instructions usually pass immediately. This ordering mirrors the independent audit in Appendix A.14, where verifier shortcut resistance is likewise the most frequent weakness, and it is exactly this concentration of failures in the environment and verifier stages that the harness self-evolution (Section 4.3) targets.

Table A.9: Share of constructed tasks whose first-round stage output is rejected by the validator and then fixed in a subsequent repair round, aggregated over the construction logs. Verifiers are hardest to synthesize correctly on the first attempt; task instructions usually pass immediately.

Construction stage	Tasks needing ≥ 1 repair round (%)
Environment	54.2
Task instruction	8.1
Verifier	88.0

Environment loop — broken dependency repair

Round 1 — generated `init.sh` (excerpt)

```
apt-get install -y -qq \
    nodejs \
    npm \   # also pulls its own nodejs
    git \ curl \ jq \ sqlite3
```

Validator critique (environment fails to build)

Image build aborted during package installation:
`env/init.sh failed: E: Unable to correct problems, you have held broken packages.`
Diagnosis: `npm` conflicts with the `nodejs` package, which already bundles `npm`; `apt` cannot satisfy both.

Round 2 — regenerated `init.sh` (repaired)

```
apt-get install -y -qq \
    nodejs \   # npm provided by nodejs; conflicting line removed
    git \ curl \ jq \ sqlite3
```

Validator verdict

PASS — image builds, all services start, sandbox check succeeds (`env verified: 3 files`).

Figure A.2: Environment-loop repair. The first `init.sh` installs a conflicting `npm` package on top of `nodejs`, so the image fails to build; the validator reports the concrete `apt` error, and the second attempt drops the redundant package and passes.

Task loop — unverifiable-composition repair

Round 1 — generated instruction (excerpt)

Run the SCR analysis: build the OrcaFlex model, run the batch simulation, then extract a cross-section of **water_surface_elevation** from the result, start `(0,-200)` to `(0,800)`, 50 points; write `cross_section_result.json`.

Validator critique (semantically broken, unverifiable)

FAIL — the three chained skills do not connect: OrcaFlex emits a `.sim` file but `hmt-cli` consumes HDF5, with **no bridge between them**, and `get_cross_section_profile` returns **all-zero values** instead of interpolating.
Root cause: unrelated skills forced into one chain; the output looks complete but **a checker cannot tell a correct result from an all-zero placeholder**.

Round 2 — regenerated instruction (repaired)

Read the companion `hydro_results.h5` directly; extract **water_surface_elevation** on the in-domain section `(0,0)`→`(200,0)`, 10 points, **and report domain statistics** (`min/max/mean/std/num_cells`, 4 dp) in the required JSON.

Validator verdict

PASS — data flow is now consistent and the statistics are recomputable from the HDF5 file (`mean`=2.5068, `std`=0.4327, `num_cells`=40401), so the task is verifiable.

Figure A.3: Task-loop repair. The first instruction chains skills whose data formats do not connect, yielding an all-zero, unverifiable output; the validator attributes the failure to the composition, and the rewrite grounds the task in a compatible artifact and adds recomputable statistics.

A.16 Case Study of Harness Self-Evolution

The repair loops above fix a *single* task, whereas self-evolution (Section 4.3) turns recurring failure modes into edits of the *synthesis harness* so that later tasks avoid the same mistake by construction. After each round of rollouts we aggregate the construction and rollout diagnostics into the report R_t separately for the three stages and edit the synthesis skill each stage owns—`env-synthesis`, `task-synthesis`, and `verifier-synthesis`—so a stage-specific failure pattern is fixed once, at its source, rather than re-repaired per task. Every edit is proposed by the evolution LLM with no human editing and is accepted only if it does not lower the held-out valid-bundle rate while preserving the guardrails (Section 4.3). Following the presentation of Zhang et al. (2026), we show

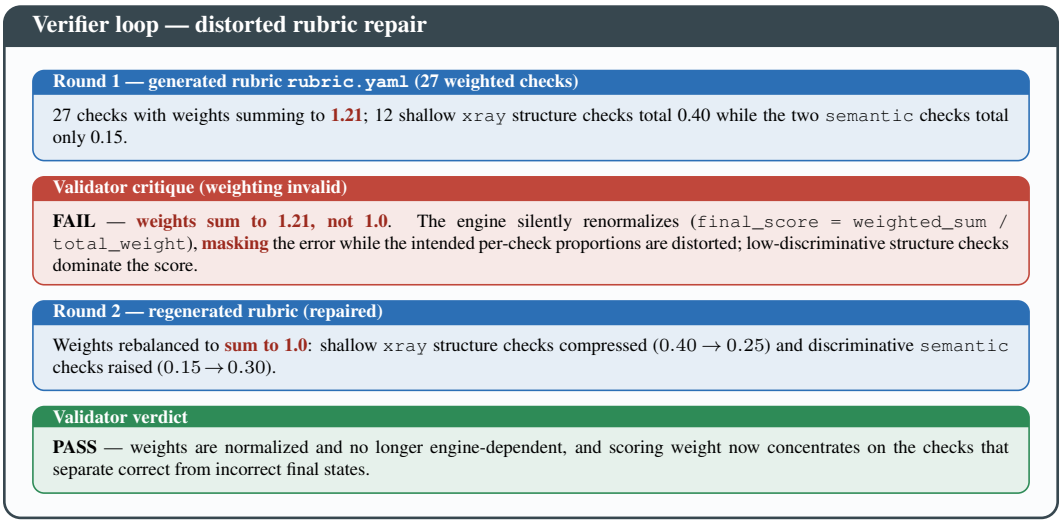


Figure A.4: Verifier-loop repair. The first rubric's weights sum to 1.21, so the scoring engine's internal renormalization hides a distorted weighting that over-rewards shallow structure checks; the repair rebalances the weights to sum to one and shifts mass toward discriminative semantic checks.

each update as a git-style diff of the skill file, with removed lines in red and added lines in green; the environment round-1 diff is taken verbatim from our logs, while the remaining hunks are lightly edited excerpts of the accepted changes. We trace two consecutive accepted updates for each stage; Figures A.5–A.7 show the aggregated diagnostic and the before/after skill edit for each round, and Table A.10 summarizes the six edits.

A useful pattern spans the task and verifier stages. In Round 1 the two stages fix the *same* underlying problem—verifiers checking details the instruction never fixed—from both sides: the task-authoring skill is tightened to state every checkable detail explicitly (Figure A.6), and the verifier-validation prompt is tightened to forbid assertions that are not grounded in the instruction (Figure A.7). Round 2 then addresses the new failure mode this introduces: once instructions must be fully checkable, authors begin to over-specify and leak the solution procedure, so the Round 2 task edit strips method and tool-sequence hints and keeps only goals and acceptance criteria. The rounds are therefore not independent tweaks but a causal chain in which each fix surfaces the next bottleneck.

Table A.10: Summary of the six accepted harness edits (three stages × two rounds) traced in Figures A.5–A.7. Each edit targets the artifact owned by its stage and removes a recurring failure class surfaced by that round's aggregated diagnostics R_t.

Stage	Round 1 edit	Round 2 edit
Environment (`env-synthesis`)	Install runtimes with bundled package managers as one unit; do not reinstall interpreters the image already ships (removes dependency conflicts).	Validate with network access disabled; fix locale, timezone, seeds, and deterministic seed data (removes non-reproducibility).
Task (`task-synthesis`)	Require an explicit machine-checkable deliverable: output path, typed schema, recomputable evidence fields.	State only goals and acceptance criteria; forbid leaked tool-by-tool solution steps.
Verifier (`verifier-synthesis`)	Reject rubrics whose weights do not sum to one; concentrate weight on discriminative checks.	Require every assertion to be grounded in an explicit instruction requirement; probe with a legitimate alternative solution.

Environment-synthesis skill update (environment stage)

Diagnostic report R_t (round 1): a recurring cluster of environment build failures comes from the synthesizer re-installing an interpreter and package manager the base image already ships (e.g. `python3`, `python3-pip`, or `npm` on top of `nodejs`), producing held-broken-package conflicts (cf. Figure A.2).

```
@@ package-installation guidance @@
  ### Self-check:  preinstalled runtimes
- The image is minimal; apt-get install python3, python3-pip,
- python-is-python3 and the matching npm before using them.
+ The image ships Python 3 and pip already; do not apt-get
+ install python3 / python3-pip / python-is-python3, and install a
+ language runtime together with its bundled package manager as one
+ unit (npm ships with nodejs).  Extra deps go through pip directly.
+ A pip mirror is preconfigured, so do not pass -i or edit sources.
```

Diagnostic report R_t (round 2): some environments that build still validate non-deterministically—setup steps reach the network or leave locale, timezone, and random seeds unpinned—so the same bundle is accepted on one run and rejected on another.

```
@@ reproducibility @@
  ### Validation environment
- Run the setup and validation steps as generated.
+ Validate with network access disabled; pin locale, timezone,
 + and all random seeds
, and ship deterministic seed data so the
+ bundle validates identically on every run.
```

Effect: round 1 removes the dependency-conflict failure class at its source and round 2 removes non-reproducible acceptances; later tasks avoid both by construction, with no per-task repair.

Figure A.5: Environment-stage harness update over two accepted rounds (git-style diff: red = removed, green = added). The round-1 edit is taken verbatim from our synthesis logs: the evolution operator rewrites the package-installation guidance so later tasks stop reinstalling runtimes the image already provides; round 2 makes validation reproducible. Both eliminate a recurring failure class rather than repairing each task.

Across these rounds the held-out valid-bundle rate rises from 42.3% to 91.5% (Section 4.3, Figure 3), and because acceptance is gated only on construction quality, none of these edits is selected against the four evaluation benchmarks. In contrast to prior harness-optimization work that edits an agent's *runtime* harness to raise its pass rate on a fixed test set (Lee et al., 2026; Zhang et al., 2026), here the object of evolution is the *synthesis* harness that produces training data, and the observed benchmark gains are a downstream consequence rather than the selection target.

A.17 Limitations and Future Work

We note several practical scope limitations that also point to natural extensions. First, our released pool is collected from two community registries (SkillsMP and ClawHub); as these registries grow, periodically re-collecting skills and re-running the pipeline would keep the skill graph current, and we leave a rolling-update schedule to future work. Second, to keep every task reproducible inside a CPU sandbox we deliberately exclude skills that require UI interaction, interactive login, external API keys, real transactions, or GPU acceleration; extending the sandbox to cover these categories—for example with recorded browser sessions or mocked paid APIs—would broaden coverage. Third, all construction, rollout, and step-level annotation use fixed model choices (GLM-5.2, `gpt-oss-120b`, and DeepSeek-V4-Pro); we did not sweep alternative generators or labelers, and a systematic study of how these choices affect the synthesized distribution is left for future work. Finally, this paper studies the synthesized data purely under supervised fine-tuning; because every task ships with an executable verifier, the same pool is directly usable as a verifiable-reward source for reinforcement learning, which we view as a promising next step rather than a claim of the present work.

Task-synthesis skill update (task stage)

Diagnostic report R_t (round 1): many downstream verifier failures trace to instructions with no concrete, checkable deliverable, so a trivial placeholder cannot be told from a correct output (cf. Figure A.3).

```
@@ output specification @@
  ### Required output
- Describe in prose what the task should ultimately produce.
+ Specify a concrete output path, a typed JSON schema, and
+ recomputable evidence fields (e.g. summary statistics) so the
+ result is machine-checkable and cannot be a placeholder.
```

Diagnostic report R_t (round 2): once outputs are pinned, authors over-specify and leak the solution procedure into the instruction, which trivializes the task.

```
@@ instruction style @@
  ### Writing the instruction
- Walk the user through the steps:  which tool to call first, what
- code to write next, and the order of operations to follow.
+ State only the goal and acceptance criteria.  Do not reveal
+ tools, commands, or a step-by-step procedure; the model must
+ discover the method while the task stays verifiable.
```

Effect: instructions become verifiable by design (round 1) without leaking solutions (round 2), keeping the training signal intact.

Figure A.6: Task-stage harness update over two accepted rounds (git-style diff). Round 1 forces every instruction to declare a concrete, checkable deliverable; round 2 removes the over-specification this induced, stripping leaked tool-by-tool solution steps so instructions state goals and acceptance criteria only.

Verifier-synthesis skill update (verifier stage)

Diagnostic report R_t (round 1): a cluster of verifiers use rubric weights that do not sum to one; the engine silently renormalizes, hiding a distorted weighting dominated by shallow structure checks (cf. Figure A.4).

```
@@ rubric weighting @@
  ### Rubric weights
- Assign a weight to each check; the engine normalizes them.
+ Weights must sum to 1.0 (reject the rubric otherwise), and
+ most weight must go to discriminative semantic checks, not to
+ shallow file-exists / substring checks.
```

Diagnostic report R_t (round 2): verifiers assert details the instruction never fixed (a specific CSV column name, exact float formatting, output order) and false-reject legitimate alternative solutions.

```
@@ what a check may assert @@
  ### Grounding checks
- Check the output against the expected reference values.
+ Every assertion must be grounded in an explicit instruction
+ requirement; do not hard-code unspecified names, formats, or
+ ordering.  Probe each verifier with a legitimate alternative
+ solution that it must still accept.
```

Effect: rubrics become engine-independent and weighted toward what matters (round 1), and stop false-rejecting valid alternatives (round 2), directly reducing the shortcut / over-verification weakness in Appendix A.14.

Figure A.7: Verifier-stage harness update over two accepted rounds (git-style diff). Round 1 makes the verifier-synthesis skill reject mis-weighted rubrics and push weight onto discriminative checks; round 2 forbids assertions that are not grounded in an explicit instruction requirement and adds an alternative-solution probe.

B USE OF AI ASSISTANTS

In accordance with the ICLR 2027 policy on AI use by authors, we disclose that AI assistants were used *only to aid or polish writing*. Specifically, we used a general-purpose LLM-based assistant to check grammar, improve clarity and readability, and suggest minor rephrasings of author-written text. All technical content—the research idea, method design, experiments, analyses, and claims—was conceived, implemented, and verified by the authors, who take full responsibility for the entire content of the paper, including any text refined with such assistance. We did not use AI assistants for research ideation or execution, for retrieval and discovery of related work, for drafting sections of the paper, for generating synthetic datasets, or for proving mathematical claims. We note that this disclosure concerns tools used to *write the paper*; it is separate from the language models that are studied as objects of, or components in, our synthesis pipeline (e.g., GLM-5.2, `gpt-oss-120b`, and DeepSeek-V4-Pro), whose roles are fully described in the main text and appendices.