SKILLGYM: TRAINING SKILL-USE AGENTS WITH AUTOMATIC VERIFIABLE ENVIRONMENT GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Skills equip LLM agents with professional knowledge and guidance to complete long-horizon and complex tasks. Although skills have been widely adopted in recent agent paradigms and harnesses, how to synthesize reliable training data and how to train agents for skill use remain underexplored. In this work, we propose SkillGym, an automatic pipeline to build verifiable environments, collect trajectories, and train skill-use agents. SkillGym first crawls a large volume of skills from the internet, then keeps those whose workflows can run reproducibly offline. A builder-reviewer pipeline is used to construct difficulty-controlled tasks, spanning four task types, each with a reference solution and an executable verifier. With this pipeline, we build 6.8k environments and collect 19k verified successful trajectories for supervised finetuning. Finetuning on these trajectories improves LLMs of different families and sizes, from 2B to 27B parameters across four skill-use benchmarks; Our Qwen3.5-9B SFT model outperforms the 397B untrained model on two of them. Further analysis shows that training teaches agents to invoke skills, raising the rate of reading the relevant skill from 28% to 96%, and that the gains hold across reasoning structures, extending to task types that form a minority of the training data and to skills held out from training.

1 INTRODUCTION

Agent skills are reusable packages that provide task guidance, factual knowledge, or runnable scripts to large language model (LLM) agents to help them complete tasks (Zhang et al., 2025) (e.g. Anthropic's pptx and mcp skills). LLM agents integrate and discover skills at inference time. Augmenting agents with skills has been shown to improve their performance on tasks requiring domain knowledge or expertise (Li et al., 2026b). Because they are flexible and require no weight updates, skills are now standard in modern agent harnesses, such as Claude Code (Anthropic), Codex (OpenAI), and OpenClaw (Steinberger & OpenClaw). However, effectively using skills requires the agent to interpret skill content, determine how to apply it to current task, and translate it into appropriate actions (Han et al., 2026a; Tan et al., 2026; Han et al., 2026b). For example, a documented workflow may require adapting its steps to the available inputs, resolving dependencies, and responding to unexpected execution outcomes. This motivates training agents to apply externally provided skills more effectively, with the aim of transferring the learned behavior to new skills and tasks.

Existing skill-use training approaches mostly derive skills from an agent's own experience in fixed environments (Xia et al., 2026; Shi et al., 2026b; Yang et al., 2026), which confine skill-use learning to a few task domains. While we reverse the direction to start from skills and build environments. Public community-written skills cover many domains, such as software engineering, science, finance, document processing, and marketing (skills.sh, 2026; majiayu000, 2026; Li et al., 2026a). However, these skills are written as reusable resources rather than training materials: none comes with a task, an environment, or a way to check success. Turning them into training data raises challenges: First, tasks must be skill-critical. Applying the skill should decide the outcome while the task stays solvable without it, so that agents learn to use the skill rather than bypass it. Second, outcomes must be verified reliably. Such verified outcomes should reflect the task requirements, recognize valid alternative solutions, and distinguish successful completions from superficially plausible outputs.

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.

In this work, we introduce SkillGym, an automatic agentic pipeline that transforms community-written skills into skill-critical tasks. From a curated collection of skills, SkillGym builds task environments and constructs problems around the applications of the knowledge, procedures, and scripts these skills are intended for. These tasks are based on four reasoning structures: procedural execution (Shridhar et al., 2020), abductive diagnosis (Jimenez et al., 2024; Zhao et al., 2023), constraint satisfaction (Xie et al., 2024), and partial-order planning (Lin et al., 2024; Qiao et al., 2025). A two-stage builder-reviewer agent system first prepares the environments and then develops task instructions, initial workspaces, reference solutions, and executable verifiers. The final task is formed by combining the builder's exploration. Execution checks establish that the reference solution completes an initially unsolved task (Jimenez et al., 2024), while reviewer feedback helps identify and repair ambiguous requirements, information leakage, and overly restrictive or insufficient verification.

In total, we construct 6.8k tasks and collect 19k verified successful trajectories from three teacher models (Team et al., 2026; Zeng et al., 2026; Xu et al., 2026) across four agent harnesses. Supervised finetuning (SFT) on these trajectories improves five LLMs from three families, ranging from 2B to 27B parameters on four skill-use benchmarks. The finetuned models are even competitive with much larger ones. Our 9B model outperforms Qwen3.5-397B-A17B (Qwen Team, 2026) on our test set and on SkillEval (Tan et al., 2026). To summarize, our contributions are as follows:

- We introduce SkillGym, an automated pipeline that transforms community-written skills into executable training environments. It constructs tasks across four reasoning structures and uses a builder-reviewer agent system to refine environments, reference solutions, and outcome verifiers.
- We construct 6.8k tasks and collect 19k interaction trajectories across multiple agent harnesses. Supervised finetuning on the resulting data improves agents' skill-use task performance, including on tasks involving skills held out from finetuning.
- We find training teaches agents to consult the provided skills, raising the rate from 28% to 96%. Annotating tasks from three benchmarks with a shared rubric, we find that the gains hold across reasoning structures and transfer to structures that form a minority of the training data.

2 Related Work

Learning to Use Agent Skills Several studies have been released to improve LLM agents' skill-use capabilities, which broadly fall into training-free and training-based methods. Training-free methods usually gather experiences from interaction trajectories, and use them to guide future exploration. Voyager (Wang et al., 2023) builds, retrieves, and composes an expanding library of executable skills through environment feedback. SkillWeaver (Zheng et al., 2025) explores websites and develops reusable APIs that improve web-agent interaction. AgentSkillOS (Li et al., 2026a) organizes existing skills into a capability hierarchy and retrieves relevant skills on demand. In this work, we focus on training-based methods, which update model weights with skill-related trajectories. Skill-to-LoRA (Zhang & Qi, 2026) uses synthetic trajectories to train skill-specific adapters. SAGE (Wang et al., 2026a) starts from SFT and uses skill-augmented RL to improve skill creation and utilization. SkillRL (Xia et al., 2026) extracts reusable knowledge into a hierarchical SkillBank and jointly evolves the skill library and agent through RL. These methods derive skills from an agent's own experience in a small set of environments or compile specific skills into model weights. SkillGym instead keeps skills external and trains the general capability to use public skills, spanning 3.5k skills across 18 domains, and the learned behavior transfers to skills held out from training.

Task and Environment Synthesis Automatically synthesizing tasks and environments provides a scalable source for training LLM agents. We group these studies based on what drives the synthesis pipeline. Task-driven synthesis starts from a task and then constructs the related environment. Endless Terminals (Gandhi et al., 2026) generates terminal-related task descriptions, then builds their environment container and refines it iteratively. CLI-Universe (Hua et al., 2026) starts with three task dimensions, creates task candidates and refines them iteratively. Environment-driven synthesis starts from the environments, tools, or states (Song et al., 2026; Wang et al., 2026b; Dong et al., 2026). EnvScaler (Song et al., 2026) collects diverse environment themes, then uses LLMs to enrich environment descriptions and construct environment states. Agent-World (Dong et al., 2026) collects

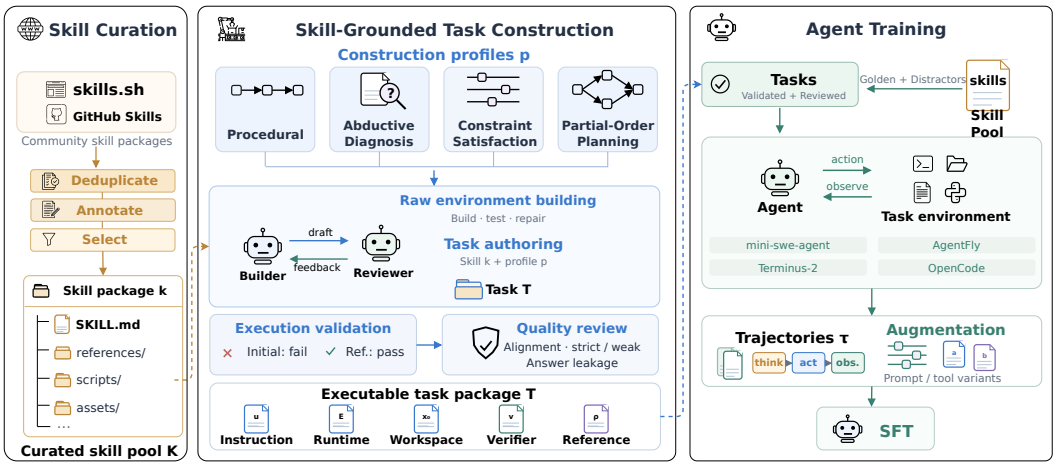


Figure 1: Overview of SkillGym. Public skill packages are curated and transformed into executable tasks through two stages of builder-reviewer interaction: raw environment building and task authoring. Execution validation and quality review support task refinement, followed by multi-harness trajectory collection and supervised finetuning.

thousands of real-world environment themes, then uses a deep-search pipeline to mine databases and executable tool interfaces. Tasks are synthesized on top of these environments. Skill-driven synthesis creates tasks from skills. SKT (Tan et al., 2026) synthesizes template-driven task packages from skills.sh skills with difficulty control and verified trajectories, and trains agents on 4k such tasks across two harnesses. SkillGym is also skill-driven, but builds tasks around explicit reasoning structures and pairs execution checks with a reviewer that audits verifiers for overly strict or insufficient checks and information leakage. The resulting tasks cover all five reasoning structures we annotate, whereas SKT's evaluation set concentrates on applying skill-provided rules (Section 5.2).

Benchmarking Agent Skill Use Existing benchmarks mainly evaluate LLM agent skill-use capabilities through task completion. SkillsBench (Li et al., 2026b) collects expert-written tasks that require skills, span multiple domains, and are all verifiable by deterministic checks. SWE-Skill-Bench (Han et al., 2026b) curates skills for SWE-style tasks, studying LLM agents on real software repositories with execution-based tests. AgentSkillOS (Li et al., 2026a) evaluates skill retrieval and orchestration through pairwise assessment of generated artifacts. Skill-Use-Bench (Han et al., 2026a) decomposes skill use into triggering, procedural compliance, and boundary adherence, scoring trajectories under progressive disclosure. These benchmarks contain at most a few hundred tasks and are designed for evaluation. SkillGym instead provides thousands of verifiable tasks for training, and we use these benchmarks to measure transfer (Section 4); our structure annotation further shows that they place different demands on agents (Section 5.2).

3 SkillGym

SkillGym converts public skills into executable task environments with a three-stage pipeline: (I) collecting and curating skill packages; (II) constructing skill-critical tasks; and (III) collecting trajectories for agent training.

3.1 Skill Collection and Curation

A skill can serve as the basis of a task only if it is (i) complete and substantive, with non-empty documentation and all referenced files available, (ii) executable in an isolated container without internet access, GPU requirements, or interactive input, and (iii) sufficiently clear to support the construction of meaningful workflow tasks.

Skill Crawling Each skill is downloaded and stored as a single folder, where a `SKILL.md` file must exist to specify basic skill information, such as its name, description, and body. We collect skills from two main sources. The first is `skills.sh`[1], from which we collect around 9.7k top-ranked skills that represent the most widely used skills in the community. The second is claude-skill-registry[2], a public aggregation of agent skills from GitHub. We download all files from skills' original sources to ensure every skill is complete. We crawl 184k skill entries from the two sources, of which 51k unique skills can be fetched after deduplication; Table 6 in Appendix F lists the counts per source and stage.

Skill Annotation Each skill is annotated on three dimensions using a combination of rule-based checks and LLM-based annotation: (i) basic properties, covering package completeness, language, file count, and folder size; (ii) runtime requirements, covering network access, GPU requirements, and dependencies; and (iii) quality, covering the coherence of the skill description and clarity of its requirements. A detailed description of the annotation properties is provided in Appendix A. To ensure annotation quality, a human expert independently annotates 100 sampled skills, resulting in an agreement of 94% with the automatic annotations.

Skill Filtering & Selection Skills are first filtered by basic requirements, retaining only valid, non-empty, and coherently written skills in English. The remaining skills are selected for task creation based on whether they can support stable execution with modest resources. Specifically, selected skills must operate without runtime network access or GPUs, avoid destructive actions such as writing outside the working directory, and remain within a file-count limit of at most 300 files. The complete selection criteria are provided in Table 4. This process yields a curated pool $\mathcal{K}$ of $11,897$ skills, whose domain distribution is shown in Figure 2b and Table 7.

3.2 Task Creation

3.2.1 Design Principle

Task construction follows two main principles. First, each task must be skill-critical. Completing it requires applying a procedure or utility provided by the skill, with at least one consequential step relying on skill-specific knowledge not stated in the task instruction. The task should remain solvable through inspection and experimentation, but access to the skill should provide a clear advantage. Second, task outcomes must be reliably verifiable. Each task therefore includes a reference solution and an executable verifier that checks the required outcome while allowing alternative valid solutions. Each finalized task is represented as $T = (u, \mathcal{E}, x_0, v, \rho)$, consisting of a task instruction u, an executable environment $\mathcal{E}$, an initial workspace state x_0, an executable verifier v, and a reference solution ρ.

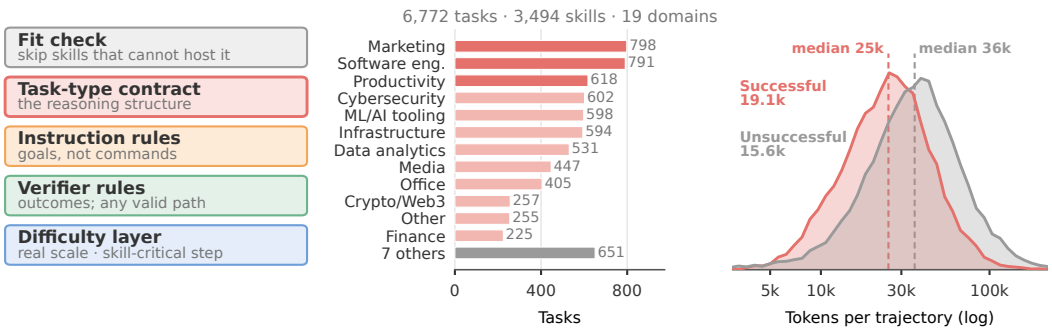
3.2.2 Task Profiles

Prior work has explored a range of reasoning structures, including multi-hop tasks solved step by step (Shi et al., 2026a; Fan et al., 2026; Tao et al., 2026), diagnosis of faulty systems (Jimenez et al., 2024; Zhao et al., 2023), planning under competing constraints (Xie et al., 2024), and plans whose steps are only partially ordered (Lin et al., 2024; Qiao et al., 2025). Each of these studies centers on a single structure, and to our knowledge none combines several structures to synthesize verifiable, skill-grounded tasks. We therefore define four task profiles, one for each reasoning structure, to diversify how skills are applied.

- **Procedural.** The task consists of a sequence of dependent steps, where each step uses the artifact produced by the previous step. The verifier checks the result of each step.
- **Abductive.** The task starts from a system with incorrect observed behavior. The agent must identify the unstated cause, repair the system, and demonstrate the corrected behavior. The instruction provides the symptoms but does not reveal the cause.

[1] `https://www.skills.sh`

[2] `https://github.com/majiayu000/claude-skill-registry`



(a) Task profile (b) Tasks by domain (c) Trajectory length

Figure 2: **(a)** The parts shared by every task profile; the task-type contract differs across the four reasoning structures. **(b)** Tasks per domain (6,772 tasks from 3,494 skills). **(c)** Token length of successful (19.1k) and unsuccessful (15.6k) teacher trajectories; dashed lines mark the medians.

- **Constraint satisfaction.** The task requires a deliverable that satisfies multiple measurable constraints. Since satisfying one constraint may violate another, the agent must measure the results and iteratively refine the solution.
- **Partial order.** The task defines dependencies as a directed acyclic graph sampled for each task. This includes joins where outputs from different branches must agree and inputs that are reused later and therefore cannot be modified in place. The agent must determine a valid execution order and produce all required deliverables.

Each task profile is a prompt block given to the builder agent, and Figure 2a shows its parts. Besides the task-type contract above, it contains a *difficulty layer* that makes skill-specific knowledge important for solving the task. It requires realistic input scales that prevent solutions from being easily computed or guessed, value-based verification that recomputes expected values from the inputs rather than hard-coding them, and at least one consequential step that depends on non-obvious skill-specific knowledge left unstated in the instruction. Depending on the profile, this step involves a documented edge case, the hidden cause of a failure, an adversarial constraint, or a join that produces an incorrect result by default. Each profile also carries a fit check and rules for the instruction and verifier; Section 3.2.3 describes how the builder-reviewer loop enforces them. Appendix B gives the full prompts of the four profiles, and Appendix D shows an example task for each.

3.2.3 Two-Stage Agentic Construction

Environment Building Basic environments are constructed with all required dependencies installed using a builder-reviewer agentic system. The builder creates a Dockerfile and any required dependency files based on the skill requirements (if specified). The reviewer builds the image, launches the container, and independently validates the environment by running relevant tests and commands. If all checks pass, the Dockerfile and dependency files are accepted and retained as the final deliverables. Otherwise, the reviewer provides feedback to the builder, which revises the environment accordingly. This process repeats until the environment passes validation or the maximum number of iterations is reached.

Task Construction Task construction reuses the builder-reviewer system with three gates. First, the builder applies the profile's *fit-check gate* and skips the skill if it cannot support the target task type; otherwise, it builds a task package containing a task instruction, initial workspace files, a reference solution script, and a verifier script. Second, a *validity gate* executes the package and accepts it only if the verifier fails on the initial workspace x_0 and passes on the workspace produced by the reference solution ρ in environment $\mathcal{E}$:

$$v(x_0) = 0, \qquad v\left(\mathrm{Exec}(\rho; \mathcal{E}, x_0)\right) = 1.$$

Table 1: Main evaluation results, higher is better. SkillEval and SkillsBench show mean ± standard deviation over three runs; the other benchmarks use one run. Bold marks the higher score within each pair where both results are available. ◇Teacher models whose trajectories form our training data. More reference models are in Appendix E. §Reported by Tan et al. (2026) from their paper.

Model	Size	SkillGym	SkillEval	SkillsBench	Skill-Use-Bench
Qwen3.5	397B/17B	55.8	73.8 ± 0.8	33.2 ± 3.0	23.0
GLM-5.2◇	753B	66.5	79.6 ± 0.8	57.8 ± 4.8	76.3
DeepSeek-V4-Flash◇	284B/13B	61.5	78.6 ± 0.2	58.7 ± 1.0	56.1
Kimi-K3◇	2.8T/104B	68.8	81.0 ± 0.9	53.5 ± 4.8	76.3
Base and SkillGym SFT comparisons					
MiniCPM5	2B	33.2	63.0 ± 0.6	$\mathbf{10.8 \pm 2.7}$	24.1
+ SkillGym SFT	2B	33.2	$\mathbf{68.1 \pm 2.0}$	6.1 ± 0.8	**44.7**
Ministral-3	8B	17.5	55.7 ± 3.1	3.9 ± 2.7	4.3
+ SkillGym SFT	8B	**48.2**	$\mathbf{77.5 \pm 0.4}$	$\mathbf{16.9 \pm 1.3}$	**55.1**
Qwen3.5	4B	33.8	62.8 ± 1.5	10.1 ± 1.3	8.8
+ SkillGym SFT	4B	**47.0**	$\mathbf{70.8 \pm 1.6}$	$\mathbf{14.3 \pm 1.0}^{\dagger}$	**48.7**
Qwen3.5	9B	41.3	65.7 ± 1.1	14.8 ± 3.5	14.8
+ SKT SFT§	9B	—	74.1 ± 1.4	14.2 ± 0.6	—
+ SkillGym SFT	9B	**59.5**	$\mathbf{74.7 \pm 1.2}$	$\mathbf{22.4 \pm 4.2}$	**49.6**
Qwen3.5	27B	55.3	76.2 ± 1.1	32.6 ± 2.3	28.3
+ SkillGym SFT	27B	**62.8**	$\mathbf{80.1 \pm 1.5}$	$\mathbf{47.4 \pm 5.1}^{\dagger}$	**74.2**

Third, at the *quality gate*, the reviewer checks the package against the profile's instruction and verifier rules: it looks for information leakage among the skill, instruction, and verifier, and for checks that are too strict to accept valid alternative solutions or too weak to reject incorrect ones. Tasks that pass all gates are accepted; otherwise, the reviewer returns feedback to the builder, and the loop repeats until the task passes or the maximum number of iterations is reached. Appendix C gives the prompts of the builder and reviewer agents in both stages.

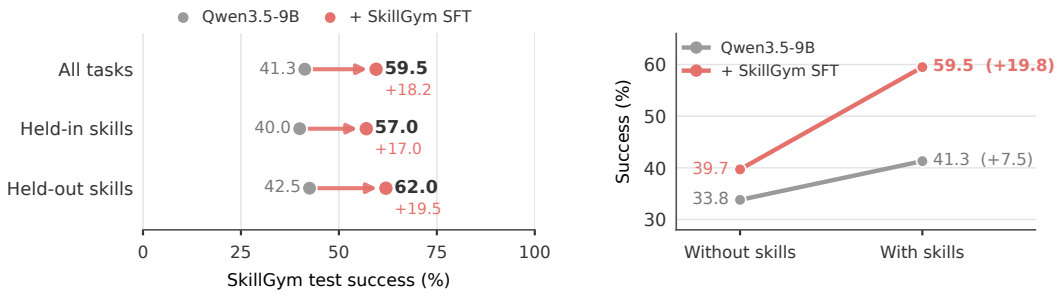
3.3 Trajectory Collection

Training trajectories are generated by three open-weight LLMs, Kimi-K3, DeepSeek-V4-Flash, and GLM-5.2, using four agent harnesses, MiniSwe-Agent, AgentFly, Terminus-2, and OpenCode. Using multiple LLMs captures variations in reasoning, action sequences, and tool-use behaviors, while reducing dependence on a single model. Using multiple harnesses further exposes the models to different interaction interfaces and tool-use formats: MiniSwe-Agent uses Bash commands for all agent actions, AgentFly and OpenCode provide dedicated tools for file operations and command execution, while Terminus-2 uses a JSON-based protocol for command execution. Further diversity is introduced by varying the system prompts, tool names, and tool schemas. Executing an agent within a task environment produces a trajectory $\tau = (o_0, a_0, \ldots, o_L)$, consisting of observations o_t and actions a_t.

4 Experiments

4.1 Setup

Training For SFT, we use our collected trajectories to train LLMs for 2 epochs. Learning rate is set to 10^{-5}, with a linear scheduler decaying to zero and AdamW optimize to update weights. We use 128 as the batch size, and train the model for 64 GPU hours. For LLMs, we select MiniCPM5-2B (MiniCPM, 2025), Ministral-3-8B (Liu et al., 2026), and Qwen3.5 series (Qwen Team, 2026), including 4B, 9B, and 27B.



(a) Held-in vs. held-out skills

(b) Benefit from skill access

Figure 3: (a) Qwen3.5-9B success rate before and after SkillGym SFT, overall (400 tasks) and per skill split (200 each), with skills available. (b) Overall success rate without and with skills.

Evaluation For main experiments, we evaluate models on (I) SkillGym splited test set, which contain a held-in and held-out subset, held-in consists of unseen tasks with seen skills during training, while held-out consists of unseen tasks with unseen skills. (II) SkillEval (Tan et al., 2026) is a synthesitic dataset constructed by SKT, aother automatic agent pipeline, consisting of single and multiple-skill tasks; (III) SkillsBench (Li et al., 2026b), a general skill-use benchmarks with all samples crurated by human experts spanning 8 domains; (IV) Skills-Use-Bench (Han et al., 2026a), which measures the agent in three dimensions: whether invokes the relevant skill, whether faithfully follows prescribed procedure and whether it avoids forbidden operations. We report overall task success on SkillGym, mean normalized reward on SkillEval and SkillsBench, and the SU score on Skill-Use-Bench, which combines the three skill-use dimensions.

We use MiniSwe-Agent (Yang et al., 2024) as the evaluation harness, which allows only bash tool for agent to use. Skills' names and descriptions are put in the system prompt, while detailed contents need to be disclosed by the agent itself.

4.2 Results

Table 1 compares five backbones from three families, from 2B to 27B parameters, before and after SkillGym SFT. Training improves the model in 18 of the 20 comparisons, with average gains of 13.9 points on the SkillGym test set, 9.6 on SkillEval, 7.0 on SkillsBench, and 38.4 in Skill-Use-Bench SU. The largest and most uniform gain is in skill use itself: SU rises by 21 to 51 points for every backbone (Section 5.1). Gains on the SkillGym test set and SkillEval are largest for Ministral-3, the weakest base model, whereas gains on SkillsBench, whose human-written tasks are the hardest, grow with model size within the Qwen3.5 family, from +4.2 at 4B to +14.8 at 27B; the 2B model is the only one whose SkillsBench score drops. The trained models also compare well with much larger ones: the 9B model outperforms Qwen3.5-397B-A17B on the SkillGym test set and SkillEval, the 27B model exceeds two of its three teachers on SkillEval, and our 9B model matches the SkillEval score reported for SKT without using its data, although the two use different harnesses.

Generalization to held-out skills. Figure 3a separates the SkillGym results by skill split. SFT increases success from 40.0% to 57.0% on held-in skills and from 42.5% to 62.0% on held-out skills, 17.0 and 19.5 percentage points respectively. The gains on both splits show that the improvement extends to skills excluded from training. On these evaluation sets, held-out success is higher than held-in success for both models, with a difference of 2.5 percentage points for the base model and 5.0 points for SFT.

Benefit from skill access. Figure 3b compares overall success on the SkillGym test set with and without skills. Providing skills increases base-model success from 33.8% to 41.3%, a gain of 7.5 percentage points. For the SFT model, success increases from 39.7% to 59.5%, a gain of 19.8 percentage points. SFT thus improves success even without skills, by 5.9 points, and more than doubles the benefit that the model draws from skill access. The training therefore does not only strengthen general task-solving ability, it teaches the agent to make better use of the skills it is given.

5 Analysis

5.1 Training improves model's behavior

Figure 4 breaks the Skill-Use-Bench score of Qwen3.5-9B into its three components, refer to Section 4.1 for definitions. All three improve after SFT. The largest change is in Trigger, which rises from 28.0 to 96.0, showing that the base model opens the relevant skill in fewer than a third of tasks, the main reason its skill-use score is low, whereas the trained model consults it almost always. Consulting the skill is a prerequisite for applying it, and the trained model also follows what it reads more faithfully. The Compliance score rises from 32.5 to 49.0 and Boundary from 57.7 to 64.0. The consulted skills are also put to use. With skills available, the trained model gains more than twice as much success on the SkillGym test set as the base model as shown in Figure 3b.

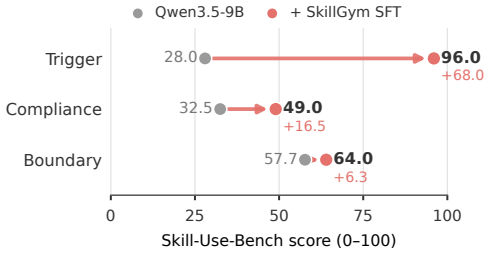


Figure 4: Mean component scores on Skill-Use-Bench for Qwen3.5-9B.

5.2 Task Structures

To test whether the gains merely reflect agreement between training and test distributions, we annotate tasks from all three benchmarks with a shared rubric of five reasoning structures, annotation details are in Appendix G. The benchmarks differ markedly, as shown in Figure 6. Nearly all SkillEval tasks apply skill-provided rules to a list of items, while SkillsBench relies more on skills that supply domain methods or reference documentation. Yet SFT improves every structure by similar amounts, by 16.9 to 22.1 points on SkillGym and 15.4 to 16.4 on SkillEval, and a regression controlling for task difficulty finds no structure-specific effect as shown in Figure 7. Gains also reach structures that are rare in training: rule application is the primary structure of only 12% of training tasks, yet SkillEval's rule-application tasks improve by 13.9 points. Transfer is weakest where a skill must be *applied*. As Figure 5 shows, on SkillsBench, tasks whose skills provide reference documentation gain 28.0 points, but those requiring a domain method or a shipped tool, or modifying an existing system, do not improve. Together with Section 5.1, this suggests that training teaches agents to consult skills more than to apply their methods, pointing to method and tool-centric tasks as a target for data construction.

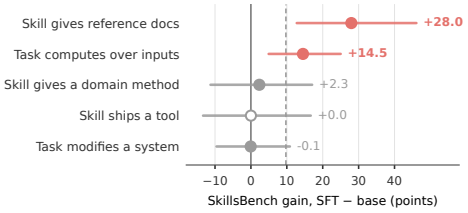


Figure 5: SkillsBench gains by what the skill provides and what the task asks. Lines show 95% task-bootstrap intervals; the dashed line is the overall gain; the hollow marker has fewer than 15 tasks.

5.3 Data Ablations

We ablate two design choices on token-matched subsets of the training data (Table 2). **Quality review.** We compare training on trajectories from *review-approved* tasks, which passed both the validity and quality gates (Section 3.2.3), with trajectories from tasks that were *validated only*: they pass the validity gate, but the reviewer did not approve them within its round budget. At the same token budget, training on review-approved tasks outperforms training on validated-only tasks on every metric, by 4.7 points on our test set, 6.7 on SkillEval, and 9.3 on Skill-Use-Bench completion, indicating that quality review improves the training signal beyond execution validation. **Task structure.** Training on a mix of all four task profiles performs on par with training on procedural tasks alone at the same token budget: the two are within run-to-run variation on SkillEval and Skill-Use-Bench, and the mix is slightly lower on SkillsBench. Together with the uniform gains across reasoning structures (Section 5.2), this suggests that the skill-use behavior acquired in training does not depend on matching the structure of training and test tasks; at this scale, the additional profiles broaden coverage rather than raise average scores. Both subsets are much smaller than the full data, and all ablation models fall below the full SkillGym model; on SkillEval they fall below the base model because many

Table 2: Data ablations on Qwen3.5-9B (64k context, 2 epochs). Each pair is trained on a token-matched subset of SkillGym trajectories (110M tokens for review, 132M for structure).

	Ours	SkillsBench	SkillEval		Skill-Use-Bench	
Training data			Pass	Strict	SU	Compl.
Qwen3.5-9B (no SFT)	41.3	14.8	65.7	17.3	14.8	51.4
SkillGym (full)	59.5	22.4	74.7	33.3	49.6	46.9
Validated only	40.5	10.8	41.2	18.3	45.1	43.1
Review-approved	**45.2**	**12.0**	**47.9**	**24.7**	**47.4**	**52.4**
Procedural only	**42.5**	**17.4**	**56.0**	24.7	**52.2**	**56.2**
Four-profile mix	39.4	13.2	55.0	**26.3**	50.1	55.4

rollouts end in repeated reasoning that exhausts the output budget. The ablations should therefore be read as comparisons within each pair.

5.4 Are the Tasks Skill-Critical?

The design principle in Section 3.2.1 requires tasks that become easier with the skill but remain solvable without it. We test this with Kimi-K3, one of the strongest available models and one of our teachers, which we evaluate on the SkillGym test set with and without skills, as reported in Table 3. Even this model benefits from skills: success rises from 60.5% to 68.8%. The effect is consistent across tasks: 52 tasks are solved only with the skill and 19 only without it (McNemar exact test, $p \approx 10^{-4}$). At the same time, 60.5% of tasks are solved without skills, so the skills make tasks easier rather than gating them. The benefit is largest for procedural tasks (+16.0), whose solution follows the skill's workflow, and smallest for abductive and partial-order tasks (+4.0 and +3.0), where a strong model can often recover the needed knowledge by inspection and experimentation. Because Kimi-K3 also served as a teacher, these tasks are not adversarial to it; that it still gains from skills indicates that the tasks encode skill-specific knowledge that even a strong model does not reliably possess.

Table 3: Kimi-K3 on the SkillGym test set without and with skills.

Task type	w/o	w/	Gain
Procedural	31.0	47.0	+16.0
Constraint sat.	79.8	89.9	+10.1
Abductive	65.7	69.7	+4.0
Partial order	65.7	68.7	+3.0
All	60.5	68.8	+8.3

6 Conclusion

We presented SkillGym, an automatic pipeline that turns community-written skills into executable, verifiable training tasks. Starting from 184k crawled skills, a builder-reviewer agent system constructs 6.8k skill-critical tasks across four reasoning structures, each with a reference solution and an outcome-based verifier, from which we collect 19k successful trajectories with four agent harnesses. Supervised finetuning on these trajectories improves LLMs of different families and sizes on four skill-use benchmarks, and the gains extend to skills held out from training. The largest behavioral change is that trained agents consult the skills they are given, and training on review-approved tasks provides a stronger signal than training on validated-only ones. Two directions remain open. Gains on SkillsBench concentrate on skills that supply reference documentation, while skills built around a domain method or a bundled tool improve little. We also train only with supervised finetuning; reinforcement learning on SkillGym environments, whose verifiers already provide outcome rewards, is a natural next step.

AI use statement

Generative AI is part of the method used in this paper. LLM agents build and review the tasks, generate the training trajectories, and annotate skills and task structures. Sections 3 and 4 name the models used. Beyond the method, the research ideas and directions originated with the authors. We used AI to assist drafting code and initial testing and drafting figure. The authors reviewed all code and verified the reported results. We also used AI assistants to smooth the writing, which the

authors further reviewed and edited. The authors take full responsibility for the content of this paper, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Anthropic. Claude Code: Overview. Claude Code Docs. URL https://code.claude.com/docs/en/overview. Accessed: 2026-09-16.

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.

Jinyi Han, Yuanjian Xu, Ying Liao, Xinyi Wang, Zishang Jiang, Zixiang Di, Fanyang Lu, Zhichao Hu, and Yanghua Xiao. Skill-use: Can llms actually use skills in agentic harnesses? *arXiv preprint arXiv:2608.04828*, 2026a.

Tingxu Han, Yi Zhang, Wei Song, Chunrong Fang, Zhenyu Chen, and Youcheng Sun. Do agent skills actually help in real-world software engineering. *arXiv preprint arXiv:2603.15401*, 2026b.

Zhanbo Hua, Yifan Yao, Weihao Xie, Yongchi Zhao, Minghao Liu, Ruizhi Qiu, Zhewei Huang, Zun Wang, Yiyan Ji, Yunhai Ye, et al. Cli-universe: Towards verifiable task synthesis engine for terminal agents. *arXiv preprint arXiv:2606.22883*, 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.

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.

Fangru Lin, EL Malfa, Valentin Hofmann, Elle Michelle Yang, Anthony Cohn, and Janet B Pierrehumbert. Graph-enhanced large language models in asynchronous plan reasoning (2024). *URL https://arxiv. org/abs/2402.02805*, 2024.

Alexander H Liu, Kartik Khandelwal, Sandeep Subramanian, Victor Jouault, Abhinav Rastogi, Adrien Sadé, Alan Jeffares, Albert Jiang, Alexandre Cahill, Alexandre Gavaudan, et al. Ministral 3. *arXiv preprint arXiv:2601.08584*, 2026.

majiayu000. Claude skills registry. https://github.com/majiayu000/claude-skill-registry, 2026.

Team MiniCPM. Minicpm4: Ultra-efficient llms on end devices. *arXiv preprint arXiv:2506.07900*, 2025.

OpenAI. Codex CLI. ChatGPT Learn. URL https://learn.chatgpt.com/docs/codex/cli. Accessed: 2026-09-16.

Shuofei Qiao, Runnan Fang, Zhisong Qiu, Xiaobin Wang, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. Benchmarking agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 69679–69703, 2025.

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

Dingfeng Shi, Jingyi Cao, Qianben Chen, Weichen Sun, Weizhen Li, Hongxuan Lu, Fangchen Dong, Tianrui Qin, Minghao Liu, Yuchen Jiang, et al. Taskcraft: Automated generation of agentic tasks. In *International Conference on Learning Representations*, volume 2026, pp. 43714–43734, 2026a.

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

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

skills.sh. skills.sh. https://www.skills.sh/, 2026. Accessed: 2026-09-24.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 8326–8357, 2026.

Peter Steinberger and OpenClaw. OpenClaw. GitHub repository. URL https://github.com/openclaw/openclaw. Accessed: 2026-09-16.

Zelin Tan, Yiqun Zhang, Hao Li, Zhiyao Cui, Hejia Geng, Shao Zhang, Hangfan Zhang, Yang Chen, Xiaosong Wang, Lilong Wang, Zhenfei Yin, Shuyue Hu, Chen Zhang, and Lei Bai. Skt: Skill-use training at scale via verified synthetic data generation, 2026. URL https://arxiv.org/abs/2608.02287.

Zhengwei Tao, Jialong Wu, Wenbiao Yin, Pu Wu, Junkai Zhang, Baixuan Li, Haiyang Shen, Kuan Li, Liwen Zhang, Xinyu Wang, et al. Webshaper: Agentically data synthesizing via information-seeking formalization. In *International Conference on Learning Representations*, volume 2026, pp. 101872–101889, 2026.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

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.

Jiongxiao Wang, Qiaojing Yan, Yawei Wang, Yijun Tian, Soumya Smruti Mishra, Zhichao Xu, Megha Gandhi, Panpan Xu, and Lin Lee Cheong. Reinforcement learning for self-improving agent with skill library. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1529–1550, 2026a.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. *arXiv preprint arXiv:2602.10090*, 2026b.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. Travelplanner: A benchmark for real-world planning with language agents. *arXiv preprint arXiv:2402.01622*, 2024.

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 E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Shidong Yang, Ziyu Ma, Tongwen Huang, Xucong Wang, Renda Li, Yiming Hu, Yong Wang, and Xiangxiang Chu. Skillforge: Evolving verifiable skills for reinforcement learning agents. *arXiv preprint arXiv:2608.24747*, 2026.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

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

Tianyi Zhang and Zhonghao Qi. Skill-to-lora: From using skills to learning behaviors for token-efficient llm agents. *arXiv preprint arXiv:2606.16769*, 2026.

Wenting Zhao, Justin Chiu, Claire Cardie, and Alexander M Rush. Abductive commonsense reasoning exploiting mutually exclusive explanations. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14883–14896, 2023.

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, et al. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

A Skill Annotation

Table 4 lists the annotated properties. An LLM annotator reads each skill's `SKILL.md` and returns labels from closed vocabularies; replies outside the vocabulary are rejected and re-requested. Fields are grouped into four levels, from surface properties to semantic judgments. The LLM screening labels cover all 51,131 deduplicated skills; skills from archived repositories or bulk publishers are then removed by rule, and the identity, runtime, and quality levels are annotated on the remaining 27,569 skills. Applying the selection rules in the last column yields the $11,897$ skills used for task creation. Screening, runtime, and quality labels use GPT-5.4; identity labels use GPT-5.5.

B Task Profile Prompts

Each task profile is a prompt block inserted into the builder agent's system prompt; a matching addendum is appended to the reviewer's prompt so that the reviewer judges the task by the same contract. The boxes below condense the prompts for the four profiles used in SkillGym; quoted phrases follow the prompt wording, and implementation details (tool names, file layout, optional difficulty dials) are omitted. Box 1 lists the rules shared by all profiles; Boxes 2–5 give each profile's fit check, task-type contract, and difficulty layer.

Table 4: Skill annotation fields, their values (share of annotated skills, %), and the rule used to select skills for task creation. Fields marked † allow several values per skill, so shares can exceed 100%. Fields marked ‡ are rule-based, computed from repository metadata and the file listing; all others are LLM labels. "–" means the field is recorded but not used for selection.

Level	Field	Values (%)	Selection
(i) Screening (n=51,131)	Label	real skill 88.9, unclear 5.5, meta/doc 3.3, template 1.0, test/demo 0.7, placeholder 0.5	real skill
	Language	English 92.4, Chinese 3.0, Japanese 1.8, Korean 0.9, mixed 0.9, other 1.0	English
	Repository‡	archived flag, bulk-publisher flag, license, stars	not archived or bulk
	Size‡	number of files in the skill folder	$\leq$300
(ii) Identity (n=27,569)	Domain	21 domains; software engineering 53.0, agents/meta 8.8, productivity 3.9, infrastructure 3.9, writing 3.8, media 3.7, …	–
	Task type†	generate 51.7, validate 48.4, analyze 46.8, integrate 30.7, plan 26.7, orchestrate 25.2, transform 18.4, …	–
	Resources‡	presence of scripts, references, and assets	–
(iii) Runtime (n=27,569)	Network access	none 63.8, runtime 31.4, setup only 4.9	none, setup only
	GPU required	no 99.9, yes 0.1	no
	Interactive	no 74.3, yes 25.7	no
	Destructive	no 85.8, yes 14.2	no
	Credentials	none 75.9, required 15.5, optional 8.6	–
	Verification	programmatic 39.2, none 32.2, LLM/human judge 28.6	–
	Output kind†	stdout 48.4, files 40.3, none 31.6, external state 13.1	–
	Runtimes†	none 55.6, bash 31.7, python 15.6, node 8.9, other $<$1 each	–
	Services, packages	free-form lists of external services and system packages	–
(iv) Quality (n=27,554)	Procedurality	procedural 58.4, mixed 38.0, declarative 3.3, persona 0.2	not persona
	Specificity	moderate 60.2, specialized 33.6, generic 6.2	–
	Coherence	coherent 66.7, minor issues 31.9, broken 1.3	not broken
	Task scope	class of tasks 53.5, narrow 45.7, single instance 0.8	not single instance
	Abstraction	balanced 80.8, adaptable 16.7, brittle templates 2.5	–

Box 1: Rules shared by all profiles

Instruction. *"instruction.md must read like a real task a user would give"*: natural prose stating the goal and the required deliverables, named by relative path. Banned: a hints/notes/tips section; exact commands, subcommands, flags, or flag values; parameter values the task does not require; meta-commentary about the task's structure; pointers to the skill; absolute or harness paths.

Skill requirement. *"If a competent agent could complete the task from instruction.md alone without using the skill's specific knowledge … the task is too easy."* The exact commands live only in the reference solution.

Verifier. Check the outcome the instruction requires, never the incidental form of the reference solution: prefer running or importing the produced artifact and asserting on what it does or outputs, and never grep the solver's source for identifiers. *"test.sh must accept any correct solution."* Do not assert names, import styles, formatting, or values the instruction leaves to the solver, and never let a check contradict a choice the instruction grants.

Validity gate. The untouched workspace must fail the verifier and the reference solution must pass it; all checked outputs must be deterministic (sorted collections, fixed ordering, seeded randomness, pinned number formats).

Difficulty layer (common core). (1) *Realistic scale*: inputs large enough that the answer cannot be computed by hand or guessed. (2) *Recompute-verify*: the verifier derives every expected value from the inputs with an independent reference implementation; *"If the inputs were regenerated with different values, would test.sh still compute the correct expected answer with no edits?"* (3) *Skill-sourced edge case*: plant inputs that trigger a caveat the skill documents. (4) *Skill-critical, underspecified step*: the correct result hinges on a non-obvious, skill-specific fact that the instruction leaves unstated; *"with the skill the fix is direct; without it the solver must research or experiment to recover the fact—harder, but still possible."*

Box 2: Procedural

Fit check. If the skill cannot honestly support the required number of distinct dependent steps, skip the skill rather than pad.

Task-type contract. *"Design the task as a linear sequence of ordered steps that exercise the skill's workflow."* The steps form a true dependency chain: *"Step k must consume the artifact produced by step $k-1$"* (raw $\rightarrow$ A $\rightarrow$ B $\rightarrow$ C). No hub-and-spoke designs in which every step reads the original input, and no orphan artifacts that no later step reads. The workspace ships the inputs but none of the step outputs, and step 1 must be a real transformation of them. The verifier checks the result of every step and prints one PASS/FAIL line per step.

Difficulty layer. Common core, plus: at least two edge cases drawn from the skill's own caveats; *inclusion and exclusion*—at least one step must drop items that superficially look processable, and the verifier checks both that the right items are present and that the wrong ones are absent; the difficulty items spread over different steps, with at least one in the second half of the chain.

Self-check. *"Could a strong agent solve this without the skill's specific knowledge? If yes $\rightarrow$ not hard enough; add a skill-critical step."* Could the answer be eyeballed or hardcoded from the inputs? Does the verifier recompute expected values?

Box 3: Abductive

Fit check. *"Can this skill host a task with a hidden explanation the solver must infer, and a verifiable check on whether they got it right?"* Otherwise skip (*"not abductive-amenable"*), or skip if the skill offers no uplift anchor.

Task-type contract. The solver observes evidence and must infer a hidden cause, rule, or explanation, then act on it. Forms include diagnosing a running but wrong system, inducing a rule from input–output examples, reconstructing a cause from logs or corrupted artifacts, and selecting among competing hypotheses. The instruction states the observations and the goal, never the cause or a path to it. The authoritative check stays hidden from the solver and must fail symptom-hiding fixes (swallowing the error, hardcoding the expected output, deleting a failing assertion).

Difficulty layer. Common core, applied to diagnosis: a realistic multi-module system whose wrong behavior cannot be spotted by reading one file, so the solver must run it and trace the symptom back; verification by recomputation or by invariants on inputs the instruction never enumerates; the planted cause ideally *is* naive code getting a skill-documented caveat wrong. Calibrate the anchoring fact between two failure modes: not general programming knowledge or already present in the workspace (too easy), and not an arbitrary token that exists only in the skill (too hard).

Self-check. *"Could a generalist spot and fix the defect by reading a few files without the skill? If yes → the cause is too shallow."*

Box 4: Constraint satisfaction

Fit check. The skill must offer several competing, deterministically checkable constraints; otherwise skip (*"not CSP-amenable"*) rather than force a checklist into this shape.

Task-type contract. *"Several simultaneous, individually measurable constraints where the starting state violates at least one and naively fixing one tends to break another."* The instruction lists the constraints as required outcomes, never how to reconcile them. The verifier checks each constraint as a separate, deterministic, outcome-based assertion and requires all of them.

Difficulty layer. Common core, plus: *real tension*—at least three constraints with at least one pair where *"the naive fix for constraint A measurably breaks constraint B"*; a setup in which every constraint is independently satisfiable in any order is a checklist and is rejected. No golden configuration: each constraint accepts any configuration that satisfies it. At least one constraint or tension comes from a skill caveat, and at least one satisfying value hinges on a skill-specific threshold, convention, or compatibility rule that the instruction does not restate.

Self-check. *"Can every constraint be satisfied independently in any order? If yes → it is a checklist, not CSP."* Could a generalist reconcile the constraints without the skill?

Box 5: Partial order

Fit check. If the skill's workflow cannot support an edge of the sampled graph, or provides no combining knowledge, skip rather than fake it.

Task-type contract. The solution is a set of artifact-producing steps *"ordered only where a real dependency demands it—a DAG, not a line."* For each task a random DAG of 4–6 nodes is sampled and given to the builder as an edge list; every node becomes a concrete artifact and every edge must be real (the consuming step opens the consumed artifact). The graph's motifs force distinct behavior: a *join* combines independently produced artifacts that must agree on a convention (field names, units, ordering, precision); a *skip edge* forces an early artifact to be kept intact until a late consumer; a *fan-out* source must survive all its consumers. At least two intermediate artifacts are *checked*: the verifier confirms that each exists, is correct, and survived (immutable, append-only, or valid), and prints a PASS/FAIL line per node. The instruction may name deliverables but never the combining knowledge or structural vocabulary ("DAG", "branch", "parallel").

Difficulty layer. Common core, plus: *the natural strategy must fail*—a solver that works serially and transforms artifacts in place must fail at least one checked node; a *wrong-by-default primary join*, where the obvious combination yields plausible-looking output that fails on values and the correct combination hinges on skill-provided knowledge that is also recoverable from workspace evidence; and *structure is not the only wall*—*"assume the solver guesses the complete graph"*; the task must still be harder without the skill.

Self-check. Would an in-place, keep-only-the-latest solver fail a checked node? *"If the solver were handed the full graph, would the task still be much harder without the skill?"*

C Builder and Reviewer Prompts

Both construction stages (Section 3.2.3) pair a builder agent with a reviewer agent. The boxes below condense their system prompts; quoted phrases follow the prompt wording. The task builder's prompt is completed by the task profile of Appendix B, and the task reviewer's prompt by the profile's reviewer addendum.

Box 6: Environment builder

Role. *"Produce a working Docker image for a single skill so it can later be exercised by a downstream evaluator."* Tools: read the skill (`SKILL.md` and its file manifest), read a skill file, build an image from a Dockerfile plus dependency files, and run a script in the built image with the skill mounted read-only.

Procedure. Read `SKILL.md` for the language, runtime, package manager, declared dependencies, and system packages. Trust dependency files shipped with the skill (`requirements.txt`, `package.json`, `pyproject.toml`, …); only if a language the skill ships scripts in has none, read a few entry-point scripts and collect their third-party imports. After at most a few reads, build: *"a failed build teaches you more than reading another script."* On failure, read the build log, find the root cause, and make a targeted edit.

Rules. Default to `ubuntu:24.04`, with `bash` and `coreutils` available. Declare dependencies in files the builder writes itself and install from those, never from the skill's own copy. Never copy the skill into the image: the skill is mounted at run time, so the image provides dependencies only.

Box 7: Environment reviewer

Role. *"Verify that a previously built Docker image actually has the dependencies the skill needs to run."* The reviewer reads `SKILL.md`, then runs one script in the image that checks each declared dependency, and reports success or a precise diagnosis for the rebuild. It verifies but never installs or patches.

Rules. *"Tests must exercise the dependency, not just inspect it"*: import a package and call something on it, or run a real subcommand of a tool; metadata checks (`pip show`, `npm list`, `which`) are insufficient. If the skill ships scripts, their imports must resolve, which catches dependencies declared nowhere but in the code. Skill files baked into the image are a defect. At most two or three test runs.

Box 8: Task builder

Role. *"From a single skill and its working environment, author one concrete, automatically-verifiable task that a downstream agent will solve."* The builder works inside a live container of the skill's environment and can read and run the skill, run shell commands, and create files.

Package. `instruction.md` (what the solver reads); `workspace_state/` (the solver's starting files); `solution/solve.sh` (the reference solution); `tests/test.sh` and helpers (the verifier); and `grading.json`, which declares the files that carry facts and a checklist of the outputs the verifier grades.

Design principles. *"Frame the task around a checkable output"*: the deliverable is data the solver produces, checked by value or behavior; a skill whose only possible check is grepping the solver's source is skipped. The workspace is a realistic starting point whose gap to the solution is the work. *"No answers in the workspace"*: no author vocabulary, no comments that explain the hidden cause or rule, no expected values in examples or configs, and no leftovers from the builder's own runs.

Loop. (1) Read the skill and try its scripts, within a small exploration budget. (2) *Verifiability gate*: *"Can the solver's deliverable be checked by value?"* If not, call `skip_task`. (3) Author the package, then call `validate_task`, which runs the validity gate and grading checks: the verifier must still pass when fact-carrying files are altered and the reference outputs kept, and must fail when each checklist output is corrupted. Iterate until valid. (4) Call `review_task`; on *revise*, fix the verifier (or remove a leak at its source), re-validate, and review again.

Rules. *"Verify the outcome, not the implementation"*: run the produced artifact or parse its output, and never grep the solver's source for identifiers. *"If a check enforces something the instruction does not already require, drop the check"*; never edit the instruction to justify a check. The instruction uses relative paths only.

Box 9: Task reviewer

Role. *"Judge whether test.sh is a sound reward signal."* The task has already passed the validity gate, so solvability is not re-examined. The reviewer reads the instruction, the reference solution, and the verifier, and can open any file in the package with read-only tools. *"A sound test is the tightest test that still accepts every correct solution."*

Defect A: too strict. *"If the task had been solved a different but valid way, would this check still pass?"* Flags checks on identifiers, imports, formatting, key names, or wording the instruction does not dictate, on one of several allowed options, or on artifacts the instruction never asked for.

Defect B: too weak. *"Would a lazy or partially-wrong solution still pass?"* Flags existence-only checks, keywords a stub or the starting state already satisfies, structure without values, count thresholds, and multi-step tasks whose intermediate steps are never checked.

Further checks. Answer leaks in files the solver can see (expected values, thresholds, or the fix in examples, docstrings, comments, or leftover outputs); a declared structure padded to reach its count; an instruction that names the technique or the location of a defect; toy material or a skill used only as a theme.

Verdict. Identify the task's core outcome and whether the verifier checks it soundly; if so, remove over-strict checks and pass. Otherwise, or on any concrete defect, return *revise* with the file, the line, and the exact fix. The builder has at most five review rounds.

D EXAMPLE TASKS

We show one task per profile from the SkillGym test set, excerpted and lightly formatted. All four are tasks that Kimi-K3 solves with the skill but not without it (Section 5.4), and that Qwen3.5-9B fails before SkillGym SFT and solves after it. Box 10 shows a procedural task in more detail; Boxes 11–13 summarize one task for each of the other profiles.

Box 10: Procedural task, `co2lpm` (natural science, held-in skill)

Skill. Physics knowledge for a 0-D lumped-parameter model of CO_2 in geothermal reservoirs: symbols, governing equations, derivations, and sanity checks in four reference files (`SYMBOLS.md`, `EQUATIONS.md`, `DERIVATIONS.md`, `SANITY_CHECKS.md`).

Workspace. `params.json` (five reservoir scenarios, e.g., `baseline`, `reversal`, `low_gas_nodagas`) and `times.csv` (60 evaluation times from 0 to 100 years).

Instruction (excerpt). *"The model equations, symbols, derivations, and sanity checks are defined by the skill's reference material … Apply those equations exactly as documented. Produce the three deliverables below, in order. Each one feeds the next."* (1) `derived.json`: per scenario, the time-independent quantities (conductivity K, initial and long-term pressure P_0, P_∞, critical and effective extraction rates, solubility slope dC_s/dP, solubility $C_{s,\infty}$, baseline emissions), *"follow[ing] the solubility convention exactly"*. (2) `trajectory.csv`: read `derived.json` and compute pressure, upflow, outflow, and solubility at every time, *"using the sign convention stated in the skill."* (3) `report.json`: read both files, decide whether pressure reversal occurs and when, and integrate the outflow over the 100-year window.

Verifier (excerpt). Recomputes every expected value from the inputs with an independent implementation and prints one PASS/FAIL line per stage:

```
def slope(T): return (A0 + A1*T + A2*T*T) / 1e6              # A0, A1, A2 from the skill's solubility law
def Cs_at(dCsdP, Pref, P, degas, C0):  return dCsdP*((Pref+P) - P_SOL_REF) + C_00 if degas else C0
P = P0 - (q_eff/K)*(1 - exp(-t/tp));  q_up = -Kup*(Pup - P);  q_out = Kout*P
rev = q_eff > q0c;  t_r = log(q_eff/(q_eff - q0c))*tp if rev else None
check("stage1-derived", approx(got[k], exp[k]), ...)                              # relative tolerance 1e-6
```

Why the skill matters. The instruction names each quantity but never its formula. The solubility-slope coefficients, the degassing rule for C_s, the sign convention for upflow, and the reversal-time formula appear only in the skill's reference files, so a solver without the skill must guess them, and any guess fails the recomputed values.

Box 11: Abductive task, `oasis-score` (healthcare, held-in skill)

Instruction (excerpt). *"The scores in `patient_scores.csv` are wrong for some patients. The scoring pipeline completes without errors … diagnose why the scores are wrong and fix the root cause so the output is correct for all patients."* The workspace holds a multi-module pipeline for the OASIS ICU severity score and 80 patient records.

Hidden cause. Three rule modules (heart rate, mean arterial pressure, temperature) evaluate their bins in the wrong order; because the first matching bin wins, patients with both low and high extremes receive the wrong component score. The instruction only says that bins have a *"defined evaluation order"*; the order itself is given in the skill's MIMIC OASIS tables.

Verifier. Recomputes the total and component scores of all 80 patients from the specification and compares them with the output of the repaired pipeline.

Box 12: Constraint-satisfaction task, `applying-brand-guidelines` (design, held-out skill)

Instruction (excerpt). *"Edit `report_config.json` so that the report configuration satisfies all of the following constraints simultaneously"*: approved color palette, approved font stack, WCAG AA contrast for every text/background pair, standard number and date formats, no prohibited terms, logo size and placement, and table styling. The instruction names the constraints but not the palette, fonts, formats, or logo values.

Tension. Palette and contrast conflict: several text/background pairs fall below the 4.5:1 ratio (white text on amber reaches only 1.6:1), and each must be repaired using palette colors only, while replacing an off-palette color can in turn break a contrast pair. Four such pairs must be resolved together.

Verifier. Checks each of the eight constraints separately, recomputing contrast ratios from the final colors, and requires all of them.

Box 13: Partial-order task, `edge-strategy-designer` (finance, held-out skill)

Instruction (excerpt). Turn a batch of trading *edge concepts* into four deliverables: an exit-calibration report, an entry-settings report, one strategy draft per concept variant, and export tickets for the downstream exporter, with *"every calibrated value … the one the strategy-design stage's standard rules produce."*

Graph. Five nodes: the drafts join the concepts, the exit calibration, and the entry settings, and the tickets reuse the entry settings directly (a skip edge), so the entry report must survive until the last step.

Hidden knowledge. At the primary join, the obvious choice copies the risk profile's base values (stop loss 0.07, reward-to-risk 3.0) into every draft. This looks valid but is wrong: the skill's script adjusts these values by hypothesis type (e.g., a breakout stop of $0.07 \times 0.85 = 0.0595$). A solver without the skill can recover the adjustments only by reverse-engineering example drafts from a previous run in the workspace.

Verifier. Recomputes the calibrated values with the skill's rules and checks the two reports, every draft, and every ticket, printing one PASS/FAIL line per node.

E Additional Reference Models

Table 5 reports all reference models evaluated under the same protocol as Table 1.

Table 5: All reference models, with skills available, evaluated under the same protocol as Table 1. *SkillGym score over the tasks graded so far. ◇Teacher models.

Model	Size	SkillGym	SkillEval	SkillsBench	Skill-Use-Bench
Qwen3.5	397B/17B	55.8	73.8 ± 0.8	33.2 ± 3.0	23.0
Qwen3.8	27B	69.0	81.6 ± 0.8	65.0 ± 1.5	68.0
GLM-5.2◇	753B	66.5	79.6 ± 0.8	57.8 ± 4.8	76.3
GLM-5.3	753B	68.8	80.8 ± 0.7	56.9 ± 2.0	78.2
DeepSeek-V4-Pro	1.6T/49B	61.5	78.8 ± 0.1	57.0 ± 2.6	57.7
DeepSeek-V4-Flash◇	284B/13B	61.5	78.6 ± 0.2	58.7 ± 1.0	56.1
MiniMax-M3	428B/23B	64.5	79.1 ± 1.4	53.4 ± 2.0	72.3

F Dataset Statistics

Table 6 traces skills from crawling to selection, Table 7 gives the domain distribution of selected skills and constructed tasks, Table 8 summarizes the tasks, and Table 9 summarizes the trajectories used for training.

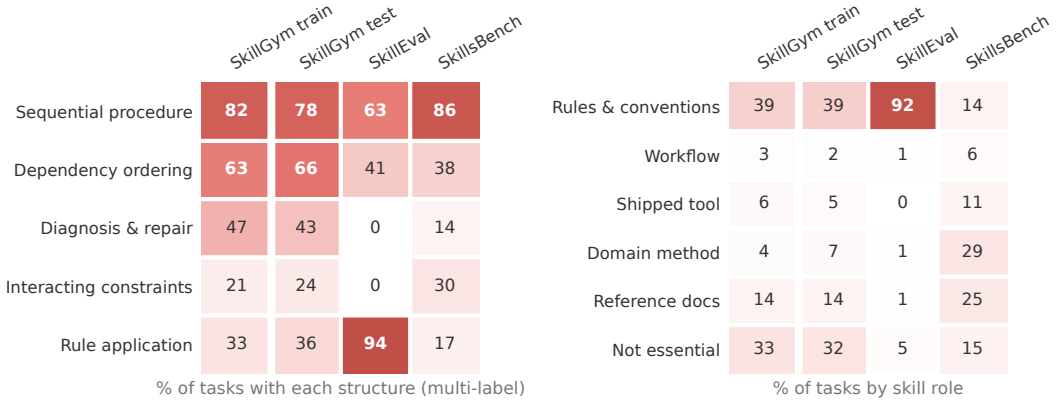


Figure 6: Share of tasks annotated with each reasoning structure (left; a task can have several) and each skill role (right). Columns: a stratified SkillGym training sample (n=300), the SkillGym test set (n=400), SkillEval (n=100), and SkillsBench (n=87).

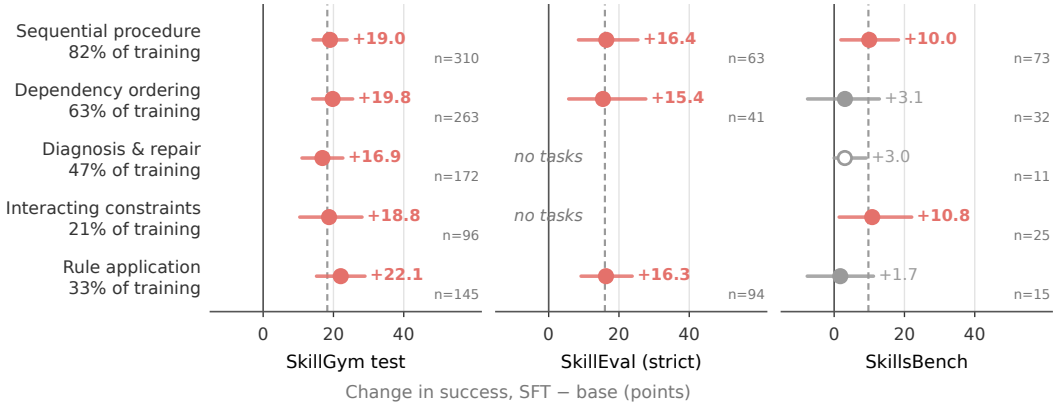


Figure 7: Change in success from Qwen3.5-9B base to SkillGym SFT for tasks with each structure, with skills available. Dots show mean paired differences and lines 95% task-bootstrap intervals; coral marks intervals that exclude zero, and hollow markers denote fewer than 15 tasks. Dashed lines show each set's overall gain. Row labels give each structure's share of the training sample. SkillEval uses strict success averaged over three runs; SkillsBench uses reward averaged over three runs, excluding trials lost to an environment permission failure.

Table 6: Skill curation funnel by source. Registry skills are fetched from their GitHub repositories; skills that could not be fetched and exact duplicates are removed before screening. The last row removes skills that appear in both sources.

Stage	skills.sh	Registry	Total
Crawled (manifest)	9,724	174,706	184,430
Fetched and deduplicated	8,593	42,538	51,131
Screened as a real skill	8,104	37,333	45,437
Annotated (not archived or bulk-published)	6,776	20,793	27,569
Selected (rules in Table 4)	2,838	9,268	12,106
After cross-source deduplication	2,838	9,059	11,897

Table 7: Domain distribution of the selected skill pool and of the constructed tasks. Task construction samples skills with per-domain quotas, which cap software engineering and exclude agents/meta and writing skills, so the task distribution is more balanced than the pool.

Domain	Selected skills	Skills with tasks	Tasks
Marketing	406	374	798
Software engineering	7,726	707	791
Productivity	321	285	618
Cybersecurity	341	289	602
ML / AI tooling	286	263	598
Infrastructure	317	283	594
Data analytics	286	249	531
Media	242	221	447
Office	234	201	405
Crypto / Web3	121	116	257
Finance	105	97	225
Healthcare	75	71	167
Communication	84	79	151
Natural science	68	59	137
Mathematics	32	29	74
Robotics	38	34	66
Manufacturing	24	21	46
Energy	6	5	10
Agents / meta	651	0	0
Writing	414	0	0
Other	120	111	255
Total	11,897	3,494	6,772

G Task Structure Annotation

Rubric. Each task is labeled on three facets. *Reasoning structure* (multi-label): sequential procedure ($\geq$3 dependent steps in which a checked output depends on an intermediate result); dependency ordering ($\geq$2 checked deliverables whose dependencies form a graph rather than a chain); diagnosis and repair (an unstated defect must be located and fixed, and the verifier checks corrected behavior); interacting constraints ($\geq$2 checked constraints where satisfying one can violate another); and rule application (rules, thresholds, or conventions given in the skill applied to each item of a collection). An additional *other* label requires a free-text description; it was used once among 887 tasks. *Skill role* (single label): rules and conventions, workflow, shipped tool, domain method, reference documentation, not essential, or other. *Action* (single label): read and answer, compute over inputs, modify an existing system, or generate many artifacts. Two flags record leakage of the intended reasoning in solver-visible material and verifier checks of conventions that neither the instruction nor the skill determines. For every positive label, the annotator must quote supporting evidence from the task, and it reports its confidence for each facet.

Inputs and model. The annotator sees the instruction, a listing and previews of the initial workspace, the skill documents, and the verifier code. Reference solutions, design notes, and task metadata are

Table 8: Constructed tasks. Top: counts by reasoning profile and review outcome (*passed*: validity gate and reviewer approval; *unresolved*: valid, but the reviewer's strictness concerns were not fully resolved within the round limit). Partial-order tasks were built in separate construction runs and are not included in these counts. Bottom: task size over the 6,692 locally available packages (median, mean, and 10th–90th percentile).

Profile	**Passed**	**Unresolved**	**Total**
Procedural execution	1,293	869	2,162
Abductive diagnosis	2,400	430	2,830
Constraint satisfaction	1,498	282	1,780
Total	5,191	1,581	6,772
Size	**Median**	**Mean**	**P10–P90**
Instruction length (words)	336	365.4	215–556
Workspace files	9	25.1	1–30
Workspace size (KiB)	19	1,261.2	2–99
Verifier code (lines)	281	305.3	148–494

Table 9: Collected trajectories. Successful trajectories pass the task verifier; unsuccessful ones are graded and fail. Length statistics give the median (mean) per trajectory; Terminus-2 issues commands inside its JSON reply rather than as tool calls.

		Successful	**Unsuccessful**
Trajectories		19,070	15,649
Harness	MiniSwe-Agent	6,433	4,360
	AgentFly	5,936	6,909
	Terminus-2	6,701	4,380
Teacher	Kimi-K3	10,603	7,649
	DeepSeek-V4-Flash	6,521	5,507
	GLM-5.2	1,946	2,493
Length	Tokens	25,146 (29,254.2)	35,977 (43,157.5)
	Reasoning tokens	7,008 (9,952.2)	12,460 (17,854.6)
	Assistant turns	12 (13.9)	15 (17.9)
	Tool calls	14 (14.6)	18 (18.9)

withheld, and construction-profile identifiers are redacted from every set. We use GPT-5.4, which did not construct or review SkillGym tasks. Replies that violate the schema are returned to the model with the validation error. As a check of the verifier reading, the annotated number of checks matches the exact count in SkillEval's evaluator specification for 99 of 100 tasks.

Gain analysis. Outcomes are paired per task: SkillGym success from one run; SkillEval strict success averaged over three runs; SkillsBench reward averaged over the runs in which the task was graded, excluding trials lost to an environment permission failure. These per-task outcomes reproduce the aggregates in Table 1. Intervals are 95% bootstrap intervals over tasks (2,000 resamples). Table 10 regresses the paired difference on the structure labels, the evaluation set, and difficulty, defined as the mean outcome of nine reference models on the task (excluding the base and SFT models).

H Cost Estimation

We estimate the API cost of building SkillGym at DeepSeek's off-peak prices, pricing task construction at DeepSeek-V4-Pro and trajectory collection at DeepSeek-V4-Flash. Since agent loops re-send the conversation history, we split input tokens into cached tokens T_{hit} and new tokens T_{miss}, and the cost of a stage with output tokens T_{out} and prices p per million tokens is

$$C = \left(T_{\text{hit}}\, p_{\text{hit}} + T_{\text{miss}}\, p_{\text{miss}} + T_{\text{out}}\, p_{\text{out}}\right) / 10^{6}.$$

Table 10: Regression of the paired SFT−base difference (points) on task properties (n=585 tasks). Intervals are 95% bootstrap intervals over tasks (1,000 resamples). Set effects are relative to SkillGym; difficulty ranges from 0 to 1.

Term	Coefficient	95% CI
Sequential procedure	$+4.6$	$[-4.2, +12.5]$
Dependency ordering	$+0.4$	$[-6.5, +7.2]$
Diagnosis and repair	-2.3	$[-9.8, +5.6]$
Interacting constraints	$+1.0$	$[-8.7, +11.2]$
Rule application	$+6.3$	$[-2.0, +15.0]$
SkillEval	-1.2	$[-11.7, +9.2]$
SkillsBench	-6.2	$[-14.7, +3.2]$
Difficulty (reference success)	$+18.0$	$[+11.0, +25.6]$
Intercept	$+1.7$	$[-8.3, +11.8]$

Token counts come from the saved builder transcripts and a sample of 3,000 trajectories. Table 11 shows that task construction dominates the cost, including failed and skipped attempts. Reviewer transcripts were not stored, so the review cost assumes 12k input and 15k output tokens per call. In total, SkillGym costs \$5.7k–8.4k, or about \$1 per released task, excluding environment building, annotation, and evaluation.

Table 11: Estimated API cost of building SkillGym at DeepSeek off-peak prices per million tokens, with \$0.022, \$0.66, and \$1.98 for cached input, new input, and output on DeepSeek-V4-Pro, and \$0.003, \$0.15, and \$0.60 on DeepSeek-V4-Flash. Token counts are in billions.

Stage	Priced as	Cached in	New in	Output	Cost (USD)
Task construction, measured	V4-Pro	58.4	0.54	1.00	3.6k
Task construction, all attempts	V4-Pro	extrapolated			up to 6.3k
Quality review, assumed	V4-Pro	0.28	0.39	0.84	1.9k
Trajectory collection	V4-Flash	6.44	0.48	0.08	0.14k
Total					5.7k–8.4k