SkillComposer: Learning to Evolve Agent Skills for Specification and Generalization

Qi Zhang[1,2*†], Zhaopeng Feng[2,3*], Xiaonan Shi[2], Xiaomeng Hu[1], Chu Liu[2], Pengjun Xie[2], Xiaobin Wang[2‡], Jieping Ye[2], Bryan Hooi[3], Haobo Wang[1], Junbo Zhao[1‡]

[1]Zhejiang University, [2]Alibaba Tongyi Lab, [3]National University of Singapore

Abstract

Agent skills, which consist of reusable strategies that guide agent reasoning and action, have shown strong potential for improving model capability at inference time. However, current skill construction methods treat the problem as one-shot extraction, overlooking a fundamental tension: a skill tailored to the specific task fails to transfer, while the abstracted skill often provides insufficient guidance. We attribute this fragility to the absence of explicit mechanisms for skill *specification* and *generalization*. To address this gap, we introduce SkillComposer, a framework that decomposes skill construction into three learnable operations: *create*, *improve*, and *merge*. Trained via systematic rejection sampling recipe, SkillComposer enables language models to self-evolve skills at inference time and supports three deployment modes: offline for building generalized libraries, online for task-specific refinement, and hybrid for combining both. Comprehensive experiments on τ^2-Bench, LiveCodeBench v6, and AppWorld show that SkillComposer consistently outperforms baselines. Our SkillComposer-4B improves a 27B executor by up to +4.5 on agent tasks and +3.4 on code tasks, while generalizing across domains and task types unseen during training. Analysis reveals that *merge* and *improve* address orthogonal quality dimensions and that skill composition is a transferable meta-ability, providing a practical recipe for skill-augmented inference.

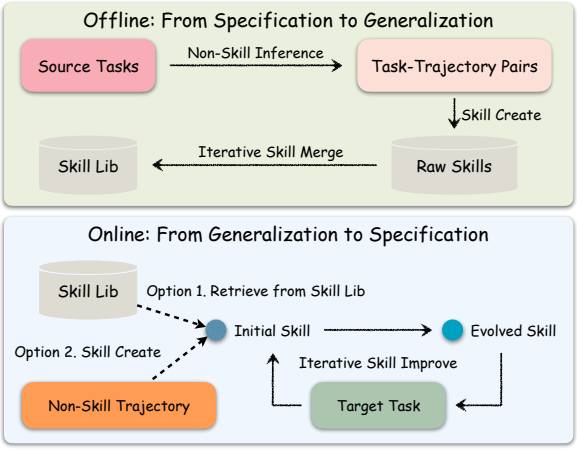

Figure 1: Three deployment modes of SkillComposer. *Offline* builds a generalized skill library via creation and merging; *Online* creates and iteratively refines skills per task; *Hybrid* initializes from an offline library and specializes via online evolution.

1 Introduction

Large language models (LLMs) have demonstrated remarkable progress across a wide range of tasks, yet they still struggle with novel scenarios that demand structured problem-solving strategies (Xu and Yan, 2026). To bridge this gap, recent work has introduced *agent skills*: reusable natural-language instructions, each comprising a name, a trigger condition, and a procedural body, that encode domain-specific procedures in a modular format (Anthropic, 2025; Wang et al., 2025). By loading relevant skills into the context, models can leverage accumulated experience without parameter updates, offering a lightweight and interpretable mechanism for capability enhancement (Liang et al., 2026).

Despite their promise, current skill-based approaches face several fundamental limitations: they rely on high-quality human-written skills (Li et al., 2026) or on extracting procedural knowledge from successful trajectories (Fang et al., 2026), both of which are costly and difficult to scale. When models generate skills autonomously, the results often yield limited or even negative performance gains, as the models lack mechanisms for quality control, refinement, and abstraction (Ni et al., 2026; Ling et al., 2026). More critically, effective skills require both *generalization* (abstracting across sim-

*Equal contribution.
†Correspondence to: cheung_se@zju.edu.cn, xuanjie.wxb@alibaba.com
‡Corresponding authors.

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 15997–16011
October 24-29, 2026 ©2026 Association for Computational Linguistics

ilar tasks to produce transferable strategies) and *specification* (tailoring skills to particular task patterns), yet existing methods provide no systematic way to achieve either. A skill that is too specific fails to transfer to other relevant tasks, while one that is too general provides insufficient guidance for other particular task (Ling et al., 2026; agentskills, 2025).

In this work, we observe that skill construction can be decomposed into three learnable abilities: (1) **Skill Create** extracts reusable procedural knowledge from raw execution trajectories, (2) **Skill Merge** drives generalization by consolidating semantically similar skill pairs into broader, more transferable ones, and (3) **Skill Improve** drives specification by refining a skill to better capture task-specific patterns based on new execution experience. Together, these operations define a complete lifecycle for skill evolution: from initial extraction, through cross-task abstraction to iterative refinement. We propose **SkillComposer**, which enables language models to self-evolve its skills at inference time **without ground-truth supervision**. Moreover, we use delta pass rate guided rejection sampling to synthesize supervised fine-tuning data to further enhance these three abilities.

Building on these abilities, SkillComposer supports three modes of operation that address different practical scenarios (Figure 1). In *Offline* mode, the model builds a general-purpose skill library from a training set via iterative creation and merging, and retrieves relevant skills for new tasks at inference time. In *Online* mode, the model starts with an empty library and iteratively creates and refines skills on the fly, driving specification in real time. This mode requires no prior data and is applicable to open-ended or streaming scenarios. *Hybrid* mode combines both by initializing with a generalized offline library and then specializing it through online evolution, balancing broad coverage with task-specific adaptation. We evaluate SkillComposer on code generation (LiveCodeBench v6) and agent tasks (τ^2-Bench, AppWorld), demonstrating consistent improvements across models and domains. Our contributions are as follows:

- We identify specification and generalization as two orthogonal dimensions of skill quality, and decomposing skill construction into creating, improving, and merging provides an explicit mechanism for this trade-off.
- We propose SkillComposer, a rejection-sampling-based framework that teaches language models to compose, refine, and abstract skills, supporting offline, online, and hybrid deployment modes.
- Comprehensive experiments across multiple benchmarks, model scales, and both in-domain and out-of-distribution settings demonstrate the effectiveness and generalization of SkillComposer. Analysis further reveals that Skill Merge and Improve address complementary quality dimensions and that skill composition transfers across task types, providing a practical recipe for skill-augmented inference.

2 Related Work

Agent skills and skill libraries. Agent skills have recently emerged as a practical abstraction for extending LLM agents at inference time. Unlike a single tool call, a skill packages reusable procedural knowledge, trigger conditions, and sometimes executable resources that can be loaded on demand (Anthropic, 2025; Xu and Yan, 2026). Recent work has therefore studied how to build, organize, and reuse skill libraries. SkillX constructs a plug-and-play skill knowledge base by distilling trajectories into hierarchical skills and refining them with execution feedback (Wang et al., 2026). SkillRL distills past experience into a hierarchical SkillBank and lets the skill library co-evolve with an agent policy during reinforcement learning (Xia et al., 2026). Complementary work upgrades textual skills into executable program functions that intervene in the agent loop (Liu et al., 2026a). These works show that skills can serve as compact experience carriers for LLM agents. However, they mainly treat skills as external artifacts to be collected, retrieved, or optimized, while the ability to construct high-quality skills remains dependent on human authoring, successful trajectories, or external optimization loops.

Skill evolution and quality control. Closest to our work are recent methods that move from static skill reuse to automatic skill evolution. EvoSkill discovers and edits skills through iterative failure analysis and validation (Alzubi et al., 2026); CoEvoSkills couples a skill generator with a surrogate verifier to build multi-file skill packages without ground-truth test content (Zhang et al., 2026); and SkillClaw aggregates cross-user interaction traces to collectively refine a shared skill repository (Ma et al., 2026).

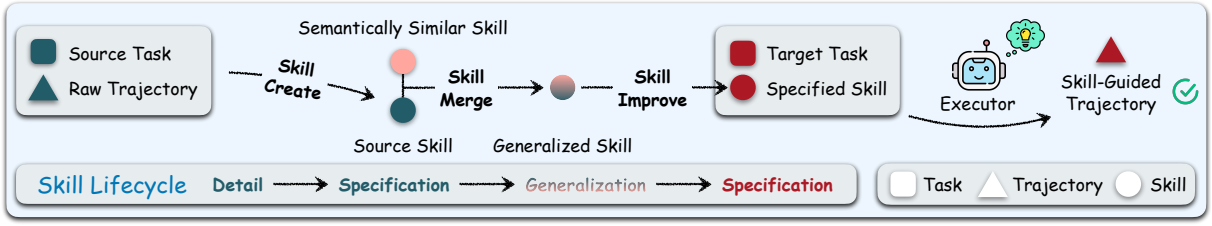

Figure 2: The lifecycle of a skill under SkillComposer. A skill is first created from a raw execution trajectory (*creation*). Multiple semantically similar while task-specific skills are then consolidated via skill merge to form broader, more transferable skills (*specification* → *generalization*). When applied to a new task, a generalized skill is iteratively refined via Skill Improve to better fit the target task (*generalization* → *specification*). The resulting specialized skill is finally consumed by the skill executor to guide inference.

Meanwhile, benchmark studies reveal that skill benefits are fragile: curated skills can substantially improve agents, but self-generated skills may provide no average gain (Li et al., 2026), and skill utility degrades under realistic retrieval and adaptation settings (Liu et al., 2026b). These findings suggest that the core bottleneck is not merely having more skills, but constructing skills with explicit quality control, task-level specification, and cross-task generalization. Different from prior work, SkillComposer decomposes skill construction into three trainable abilities–*create*, *improve*, and *merge*–and learns them through rejection-sampled supervision filtered by delta pass rate. In this way, skill evolution becomes an intrinsic capability of the language model itself: improvement drives specification, merging drives generalization, and the resulting model can self-evolve skill libraries in offline, online, and hybrid settings.

3 Methodology

In this section, we dive into the framework of SkillComposer. We first formalize three core skill operations (§3.2), then describe how they compose into three inference-time application scenarios (§3.3), and finally introduce a rejection-sampling-based training procedure that strengthens these operations (§3.4). Figure 2 provides an overview of the lifecycle of a skill under our proposed SkillComposer.

3.1 Preliminary

We begin by formalizing the concept of agent skills. In the most general form, agent skills comprise a markdown-style document `SKILL.md` along with optional auxiliary files such as references, scripts, and assets (Anthropic, 2025). In this work, we focus on the main body of agent skills, where each skill s is defined as a tuple of three components: a name n, a description d, and a body b. The name serves as an identifier, the description specifies the trigger condition under which the skill should be activated, and the body encodes the procedural strategy to be followed.

3.2 Skill Operations Formulation

Maintaining a high-quality skill library requires addressing three fundamental challenges: *i*) how to extract a reusable skill from a raw trajectory produced without any skill guidance, *ii*) how to prevent the unbounded growth of a skill library while promoting generalization, and *iii*) how to refine an existing skill based on new execution experience. We address these challenges by introducing three composable operations: *create*, *merge*, and *improve*.

Skill Create. When the agent solves a task without leveraging any prior skill, the resulting trajectory contains implicit procedural knowledge that can benefit future tasks with similar structure. Skill Create distills such a trajectory into a reusable skill:

$$s = \text{SkillComposer}_{\text{create}}(x, \text{LLM}(x))$$

Skill Merge. As skills accumulate, the library risks unbounded growth and over-specification, where many narrowly-scoped skills overlap in functionality. To mitigate this, skill merge consolidates two semantically similar skills, s_1 and s_2, into a single, more general one:

$$s = \text{SkillComposer}_{\text{merge}}(s_1, s_2)$$

Specifically, to measure the semantical similarity of two skills, we introduce a multi-view similarity mechanism where we compute similarity cross every dimension for each skill pair:

$$\text{sim}(s_1, s_2) = \frac{1}{3} \sum_{I \in \{n,d,b\}} \text{sim}(I_1, I_2) \quad (1)$$

Only pairs with similarity larger than a threshold δ will be selected to conduct skill merge.

Skill Improve. When the agent solves a task under the guidance of an existing skill s_{o}, the resulting trajectory may reveal new patterns or refinements that the original skill does not capture. Skill improve incorporates these insights to produce a more effective skill:

$$s = \mathrm{SkillComposer}_{\mathrm{improve}}(x, s_{\mathrm{o}}, \mathrm{LLM}(x, s_{\mathrm{o}}))$$

3.3 Inference-Time Skill Evolution

The three operations can be chained in different orders depending on whether a pre-existing task set is available and whether task-specific refinement is desired at inference time. We organize these compositions into three deployment modes. Algorithm 1 also unifies the three modes into a single procedure:

Offline. The most straightforward way to leverage skills at inference time is to pre-build a skill library from a given task set. In offline mode, we first generate raw trajectories for all tasks and apply Skill Create to extract one skill per trajectory. We then compute pairwise similarities using Equation 1 and iteratively merge the most similar pairs until no pair exceeds δ. The resulting library captures generalized procedural knowledge. At inference time, we retrieve the top-k skills for each target task and employ a self-select mechanism that allows the executor to choose the most appropriate skill, or abstain if none is relevant.

Online. In scenarios where no pre-existing task set is available, we operate in online mode. For each incoming task, the agent generates a raw trajectory and applies Skill Create to initialize a skill. The skill is then iteratively refined: at each iteration, the agent re-executes the task under the current skill's guidance, and Skill Improve updates the skill based on the new trajectory. This process repeats for a fixed number of iterations T, progressively driving specification.

Hybrid. The hybrid mode combines the strengths of both approaches. It initializes with an offline-constructed skill library, then applies online evolution on top. For each new task, we retrieve and self-select a relevant skill from the library, then refine it through iterative improvement. This enables the model to start from a generalized foundation and progressively specialize toward the target task.

3.4 Training via Rejection Sampling

While any language model can perform the three operations zero-shot, output quality varies considerably. We strengthen them via supervised fine-tuning on rejection-sampled data (Yuan et al., 2023), using a unified quality signal: *delta pass rate*, the performance change caused by a candidate skill. A training example is accepted only when the candidate improves the executor's pass@k relative to the appropriate baseline by at least a threshold ϵ.

Concretely, we generate n raw trajectories per task and apply the base model to produce candidate skills. For *create*, we compare the executor's pass@k with versus without the candidate skill on the source task and retain the example if the improvement exceeds ϵ_{c}:

$$\begin{aligned}\Delta_{\text{create}} = \text{pass@}k(\mathrm{LLM}(x, s)) \\ - \text{pass@}k(\mathrm{LLM}(x)) \geq \epsilon_{\mathrm{c}}\end{aligned} \quad (2)$$

For *merge* and *improve*, the criterion generalizes to multiple evaluation tasks. Merge assesses the consolidated skill s against the two original skills s_1, s_2 on their respective source tasks x_1, x_2:

$$\begin{aligned}\Delta_{\text{merge}} = \frac{1}{2}\sum_{i=1}^{2}\big[\text{pass@}k(\mathrm{LLM}(x_i, s)) \\ - \text{pass@}k(\mathrm{LLM}(x_i, s_i))\big] \geq \epsilon_{\mathrm{m}}\end{aligned} \quad (3)$$

Improve evaluates the refined skill against the original skill s_{o} on both the source task x (from which s_{o} was derived) and a target task x':

$$\begin{aligned}\Delta_{\text{improve}} = \frac{1}{2}\sum_{x_i \in \{x, x'\}}\big[\text{pass@}k(\mathrm{LLM}(x_i, s)) \\ - \text{pass@}k(\mathrm{LLM}(x_i, s_{\mathrm{o}}))\big] \geq \epsilon_{\mathrm{i}}\end{aligned} \quad (4)$$

For improve, we include both cross-task ($x' \neq x$) and same-task ($x' = x$) pairs, so the training signal captures both specification and generalization. Detailed algorithms for all three procedures are provided in Appendix B.

4 Experiments

4.1 Experimental Settings

Training and Evaluation For coding task, we use OpenCodeReasoning (Ahmad et al., 2025) for both rejection sampling and building the initial skill library. For rejection sampling of the agent task, we follow AutoForge (Cai et al., 2025) to synthesize datasets similar to the Retail domain in τ^2-Bench. In summary, we synthesize around 7,000

Algorithm 1 Inference-Time Skill Evolution

Require: Task set $\mathcal{X}$, skill executor LLM, mode $\in$ {offline, online, hybrid}, max iterations T, skill library $\mathcal{L} \leftarrow \emptyset$
Ensure: Evolved skill library $\mathcal{L}$

Stage 1: Skill Library Initialization
if mode $\in$ {offline, hybrid} **then**
  for each task $x \in \mathcal{X}$ **do**
    $\tau \leftarrow \text{LLM}(x)$ ▷ Raw trajectory
    $s \leftarrow \text{SkillComposer}_{\text{create}}(x, \tau)$
    $\mathcal{L} \leftarrow \mathcal{L} \cup \{s\}$
  end for
  repeat
    $(s_i, s_j) \leftarrow \text{RetrieveSimilarPair}(\mathcal{L}, \delta)$
    $s \leftarrow \text{SkillComposer}_{\text{merge}}(s_i, s_j)$
    $\mathcal{L} \leftarrow (\mathcal{L} \setminus \{s_i, s_j\}) \cup \{s\}$
  until no similar pairs remain
end if
Stage 2: Skill Evolution
for each target task x' **do**
  if mode = offline **then**
    $s \leftarrow \text{SelfSelect}(\text{Retrieve}(\mathcal{L}, x', k))$
    $\tau \leftarrow \text{LLM}(x', s)$
  else
    if mode = online **then**
      $\tau \leftarrow \text{LLM}(x')$
      $s \leftarrow \text{SkillComposer}_{\text{create}}(x', \tau)$
    else if mode = hybrid **then**
      $s \leftarrow \text{SelfSelect}(\text{Retrieve}(\mathcal{L}, x', k))$
    end if
    for $t = 1$ to T **do**
      $\tau \leftarrow \text{LLM}(x', s)$
      $s \leftarrow \text{SkillComposer}_{\text{improve}}(x', s, \tau)$
    end for
  end if
  $\mathcal{L} \leftarrow \mathcal{L} \cup \{s\}$
end for

supervised fine-tuning data points. We further train Qwen3.5-4B (Team, 2026) with LlamaFactory, and run skill executor and SkillComposer using vllm. We extend more implementation details in Appendix A.

Benchmarks We evaluate SkillComposer on three benchmarks spanning agent and code tasks. (1) **τ^2-Bench** (Barrès et al., 2025) is a multi-turn agent benchmark that simulates realistic customer service interactions across three domains: Retail, Airline, and Telecom. Each domain requires the agent to follow domain-specific policies and execute tool calls to resolve user requests. Since the training data only consists of Retail domain, evaluation on Airline and Telecom can be regarded as a cross-domain scenario. (2) **LiveCodeBench v6** (Jain et al., 2025) is a contamination-free code generation benchmark comprising problems of varying difficulty (Easy, Medium, Hard) sourced from recent programming contests. (3) **AppWorld** (Trivedi et al., 2024) is an interactive agent benchmark that requires autonomous app usage through API calls across diverse everyday applications. We use AppWorld as an out-of-distribution evaluation to assess cross-task generalization, as it is entirely unseen during training.

4.2 Main Results

Table 1 presents results on τ^2-Bench and LiveCodeBench v6. We compare against *No Skill* (vanilla LLM) and *MemP* (Fang et al., 2026) (procedural memory extracted from valid trajectories), and additionally report results for the untrained Qwen3.5-4B to isolate the contribution of rejection sampling SFT. All training data is collected using the 4B model. For offline skill library construction, we use the training set of τ^2-Bench for agent tasks and 500 randomly sampled instances from OpenCoderReasoning for code tasks.

Effectiveness of SkillComposer. On the 4B executor, SkillComposer consistently outperforms both baselines and its untrained counterparts across all modes. The hybrid mode achieves the best τ^2-Bench score (85.7, +6.2 over No Skill), while the online mode leads on LiveCodeBench (59.1, +2.5). MemP degrades LiveCodeBench performance to 47.4, well below the No Skill baseline of 56.6, underscoring the risk of uncontrolled skill extraction. In contrast, SkillComposer online improves over its untrained counterpart by 2.0 points on τ^2-Bench and 2.4 points on LiveCodeBench, confirming that rejection-sampling-based training yields meaningful quality gains. We provide detailed performance curves across iterations in Appendix C (Figure 5).

Moreover, we further provide inference efficiency comparison across the base model and our trained SkillComposer-4B. As shown in table 6, as skills evolve, the average token consumption of both methods decreases while the average performance keep increasing, which indicates the steady performance and efficiency of our methods. Notably, except for the performance superiority, our trained SkillComposer-4B also consistently show

Method		Tau2Bench				LiveCodeBench v6			
		Retail	Airline	Telecom	Overall	Easy	Medium	Hard	Overall
Skill Executor: Qwen3.5-4B									
No Skill		71.3	68.8	93.1	79.5	*99.1*	67.3	26.8	56.6
MemP		72.5	77.5	93.8	82.0	**100.0**	42.3	22.5	47.4
Qwen3.5-4B	offline	72.5	76.3	92.5	81.3	97.7	61.5	25.0	53.7
	online	70.6	*80.0*	*96.3*	82.8	98.1	68.9	26.5	56.7
	hybrid	73.8	77.5	93.8	82.5	95.4	64.1	29.6	56.0
SkillComposer-4B	offline	72.5	75.0	92.5	81.0	95.4	65.4	25.8	54.7
	online	*74.4*	78.8	**98.1**	84.8	98.1	**72.7**	*29.3*	**59.1**
	hybrid	**77.5**	**82.5**	95.6	**85.7**	98.5	*69.2*	**31.3**	*59.0*
Skill Executor: Qwen3.5-27B									
No Skill		75.6	77.5	94.4	83.5	99.5	90.4	62.0	79.7
MemP		71.3	78.8	97.5	83.3	**100.0**	67.3	47.5	66.3
Qwen3.5-4B	offline	72.5	73.8	96.3	82.3	**100.0**	84.6	55.0	74.9
	online	77.5	78.8	**98.8**	86.3	**100.0**	91.9	64.5	81.4
	hybrid	77.5	80.0	97.5	86.0	99.2	90.4	64.2	80.6
SkillComposer-4B	offline	73.8	73.8	97.5	83.3	98.3	89.4	59.7	78.0
	online	**79.4**	**83.5**	**98.8**	**88.0**	**100.0**	**94.2**	*66.8*	**83.1**
	hybrid	*78.1*	*81.3*	**98.8**	*87.0*	99.4	*93.3*	**67.2**	*82.9*

Table 1: Main results on τ^2-Bench (agent) and LiveCodeBench v6 (code). Rejection sampling and offline library construction use Qwen3.5-4B as the skill executor, with the τ^2-Bench training set and randomly sampled 500 OpenCoderReasoning samples as data sources. **Bold** denotes the best and *italic* the second best within each executor.

better efficiency.

Comparison across modes. Online and hybrid consistently outperform offline, indicating that iterative skill refinement at inference time provides substantial gains beyond static retrieval. Hybrid achieves the strongest τ^2-Bench results by combining a generalized library with online specialization. Notably, the performance gains of offline and hybrid modes are more pronounced on τ^2-Bench than on LiveCodeBench. We attribute this to the offline library construction: the τ^2-Bench library is built from its in-distribution training set, while the code library draws from only 500 randomly sampled OpenCoderReasoning instances, which offers limited coverage of LiveCodeBench's task distribution.

In contrast, the online mode, which requires no pre-built library and instead refines skills per task, excels on LiveCodeBench where task-specific specification is most beneficial. Furthermore, on Hard-level code problems, hybrid consistently outperforms online (31.3 vs. 29.3 on 4B; 67.2 vs. 66.8 on 27B), suggesting that an initial skill library serves as valuable prior experience that particularly benefits the executor on more challenging tasks where structured guidance is most needed. However, on the Simple- and Medium-Level problems, hybrid lags behind the online setting, which indicates that unnecessary skills might mislead the language models when facing simpler tasks.

4.3 Cross-Model Generalization

Since all training data is collected with Qwen3.5-4B as the skill executor, the 27B results directly evaluate cross-model transfer. As shown in Table 1, SkillComposer generalizes effectively across model scales: SkillComposer-4B-online achieves 88.0 on τ^2-Bench (+4.5 over No Skill) and 83.1 on LiveCodeBench (+3.4), while the hybrid mode reaches 67.2 on Hard-level code problems. These results confirm that skills composed by a smaller model remain effective when consumed by a substantially larger executor, suggesting that skill quality is largely decoupled from composer's capacity.

4.4 Cross-Domain Generalization

Our τ^2-Bench training data is sourced exclusively from the Retail domain, making Airline and Telecom results a direct test of cross-domain generalization. On the 4B executor, SkillComposer-4B-hybrid achieves 82.5 on Airline (+13.7 over No Skill) and 95.6 on Telecom (+2.5). On the 27B executor, SkillComposer-4B-online reaches 83.5 on Airline (+6.0) and 98.8 on Telecom (+4.4). The Airline gains suggest that skills learned from Retail, which shares structural similarities as a customer-service domain, transfer effectively to related but unseen domains. This indicates that SkillComposer acquires generalizable composition abilities rather than domain-specific patterns.

5 Ablation and Analysis

We now investigate the mechanisms behind SkillComposer's gains. We first ablate the individual skill operations to verify that merge and improve serve complementary roles (§5.1). We then ask whether the learned composition ability is tied to the training domain or transfers across task types (§5.2), whether it generalizes to entirely unseen benchmarks (§5.3), and whether iterative skill evolution offers genuine advantages over brute-force repeated sampling (§5.4).

5.1 Ablation on Skill Operations

We ablate the skill operations used during offline library construction to isolate each operation's contribution. Table 2 reports offline-mode results with Qwen3.5-4B as the skill executor.

Skill Operations	τ^2-Bench	LiveCodeBench v6
Create	73.8	*58.5*
Create/Improve	75.6	57.3
Create/Merge	**77.5**	**59.0**
Create/Merge/Improve	*76.9*	57.9

Table 2: Ablation on skill operations used during offline library construction (Qwen3.5-4B executor). Merge provides the largest gain by promoting generalization, while Improve is less beneficial in the offline setting where transferability is prioritized.

Merge is the most impactful operation: adding it to Create yields +3.7 on τ^2-Bench and +0.5 on LiveCodeBench. This aligns with our framework's design, as offline libraries prioritize generalized skills, and Merge is precisely the operation that drives generalization. To visualize this effect, Figure 3 shows t-SNE embeddings of the Retail skill library before and after merging. After merge, the skill distribution becomes notably sparser, confirming that merge effectively consolidates redundant skills into fewer, more general ones. Adding Improve alone to Create provides moderate gains on τ^2-Bench (+1.8) but slightly hurts LiveCodeBench (−1.2), suggesting that specification can reduce transferability for unseen tasks. Combining all three operations performs slightly below Create/Merge, indicating that indiscriminate specification during offline construction partially counteracts the generalization benefit of Merge. These results reinforce that Improve is better suited for online and hybrid settings, where skills are refined toward specific target tasks.

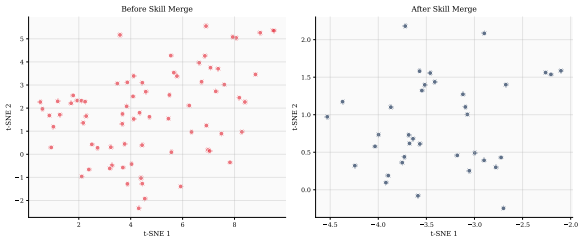

Figure 3: t-SNE visualization of the Retail skill library before (left) and after (right) skill merge. Merging consolidates semantically similar skills, producing a **sparser** and **more general** library with reduced redundancy.

5.2 Transferability of Skill Composition

We investigate whether the skill composition ability generalizes across task types or remains specific to the training scenario. We train SkillComposer with agent-only, code-only, and mixed (agent & code) rejection sampling data, evaluating all variants on LiveCodeBench v6 in online mode with Qwen3.5-27B as the skill executor (Table 3).

SkillComposer	Easy	Medium	Hard	Overall
Qwen3.5-4B	**100.0**	91.9	64.5	81.4
+agent	**100.0**	93.5	*66.8*	82.9
+code	99.5	**95.8**	**68.0**	**84.0**
+agent & code	**100.0**	*94.2*	*66.8*	*83.1*

Table 3: Transferability of skill composition across task types, evaluated on LiveCodeBench v6 (online mode, Qwen3.5-27B executor). Agent-only training still improves code performance, confirming that skill composition is a general-purpose ability.

Training with agent-only data improves overall performance from 81.4 to 82.9 (+1.5), despite

never encountering code tasks during training. This confirms that skill composition is a general-purpose ability: structuring procedural knowledge and iterative refinement transfer across task types. Code-only training yields the strongest results (84.0, +2.6) due to direct domain alignment, while the mixed setting used in practice achieves 83.1, offering a balanced trade-off.

5.3 Cross-Task Generalization

To further evaluate whether SkillComposer generalizes to entirely unseen agent tasks, we test the online mode on AppWorld (Trivedi et al., 2024), a benchmark not used during training or rejection sampling. We compare the vanilla model (No Skill), the untrained Qwen3.5-4B composer, and SkillComposer-4B, all using Qwen3.5-4B as the skill executor. Results are shown in Table 4.

SkillComposer-4B achieves the best performance on both metrics, improving accuracy to 57.7 (+2.5 over Vanilla) while maintaining a high pass rate of 88.3. Notably, the untrained composer improves accuracy (57.1) but degrades pass rate from 88.1 to 86.0, indicating that naively generated skills can introduce errors that reduce task completion reliability. In contrast, SkillComposer improves both metrics simultaneously, confirming that rejection-sampling-based training produces skills that enhance correctness without sacrificing robustness. This result demonstrates that the skill composition ability learned from τ^2-Bench transfers effectively to a structurally different agent benchmark.

Method	Vanilla	Qwen3.5-4B	SkillComposer-4B
Accuracy	55.2	57.1	**57.7**
Pass Rate	88.1	86.0	**88.3**

Table 4: Cross-task generalization on AppWorld (online mode, Qwen3.5-4B executor). SkillComposer improves both accuracy and pass rate, while the untrained composer degrades pass rate.

5.4 Iterative Evolution vs. Repeated Sampling

A natural question is whether iterative skill evolution offers advantages beyond simply sampling more attempts. To answer this, we compare two strategies under the same inference budget of k rounds: *i*) vanilla pass@k, which independently samples k solutions without any skill and selects the best one, and *ii*) SkillComposer online evolution, which creates a skill from the first trajectory and iteratively improves it across k rounds.

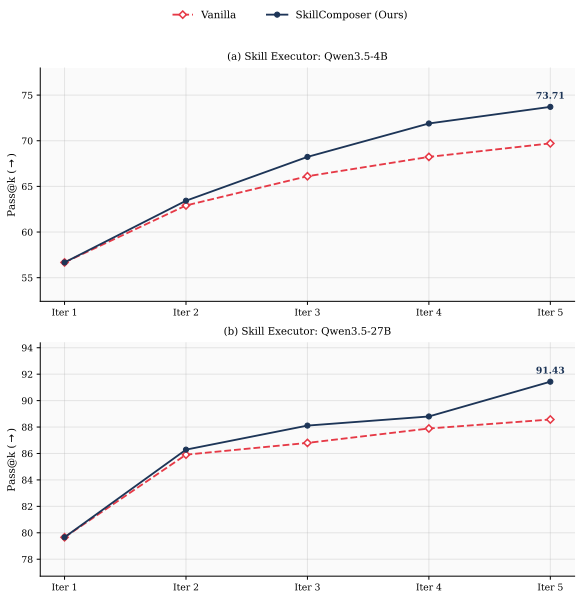

Figure 4: Pass@k vs. SkillComposer online evolution over k rounds on LiveCodeBench v6. Both strategies share the same starting point at k=1. Iterative skill refinement consistently outperforms independent repeated sampling, with the gap widening as k increases.

As shown in Figure 4, both strategies start from the same baseline at k=1, but SkillComposer consistently outperforms vanilla pass@k as the number of rounds increases. On the 4B executor, SkillComposer reaches 73.71 at k=5 compared to 69.71 for vanilla pass@k, a gain of 4.0 points. On the 27B executor, the gap persists at 91.43 vs. 88.57 (+2.86). Notably, the performance gap widens with each additional round, indicating that iterative skill refinement compounds its advantage over independent sampling. This demonstrates that structured skill evolution is a more effective use of additional inference budget than brute-force repeated sampling.

6 Conclusion

We presented SkillComposer, a framework that decomposes skill construction into three learnable operations: create, improve, and merge. By training these operations via rejection sampling filtered by delta pass rate, SkillComposer enables language models to self-evolve skill libraries at inference time without ground-truth supervision. The framework supports three complementary modes: offline for building generalized libraries, online for task-specific refinement, and hybrid for combining both. Experiments on τ^2-Bench, LiveCodeBench v6, and AppWorld demonstrate that SkillComposer consistently outperforms baselines and generalizes across

model scales, domains, and task types. Our analysis further reveals that merge drives generalization while improve drives specification, and that skill composition is a transferable ability that benefits from but does not require in-domain training data.

Limitations

The rejection sampling procedure requires multiple inference passes per task to compute delta pass@k, making training data collection computationally expensive. Additionally, our experiments are conducted with a limited number of model families (Qwen3.5-4B and Qwen3.5-27B); evaluating SkillComposer across a broader range of architectures and scales would strengthen the generality of our findings.

Ethical Considerations

We used large language models only for language polishing and grammar refinement during manuscript preparation. All research ideas, method design, experiments, analyses, and conclusions were developed and verified by the authors. No large language models were used to generate experimental results, fabricate data, or make scientific claims.

Acknowledgments

This work was supported by the Noncommunicable Chronic Diseases-National Science and Technology Major Project (Grant No. 2026ZD0558202). Haobo Wang is also supported by the Ningbo Yongjiang Talent Introduction Programme (No. 2023A-399-G).

References

agentskills. 2025. agentskills. https://github.com/agentskills/agentskills. GitHub repository.

Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. 2025. Opencodereasoning: Advancing data distillation for competitive coding. *CoRR*, abs/2504.01943.

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

Anthropic. 2025. skills. https://github.com/anthropics/skills. GitHub repository.

Victor Barrès, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-bench: Evaluating conversational agents in a dual-control environment. *CoRR*, abs/2506.07982.

Shihao Cai, Runnan Fang, Jialong Wu, Baixuan Li, Xinyu Wang, Yong Jiang, Liangcai Su, Liwen Zhang, Wenbiao Yin, Zhen Zhang, Fuli Feng, Pengjun Xie, and Xiaobin Wang. 2025. Autoforge: Automated environment synthesis for agentic reinforcement learning. *CoRR*, abs/2512.22857.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. 2026. Memp: Exploring agent procedural memory. In *Findings of the Association for Computational Linguistics, ACL 2026, San Diego, California, United States, July 2-7, 2026*, pages 17490–17502. Association for Computational Linguistics.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, Shuyi Wang, Qunhong Zeng, Di Wang, Xuandong Zhao, Yuanli Wang, Roey Ben Chaim, Zonglin Di, Yipeng Gao, Junwei He, and 21 others. 2026. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *CoRR*, abs/2602.12670.

Yuan Liang, Ruobin Zhong, Haoming Xu, Chen Jiang, Yi Zhong, Runnan Fang, Jia-Chen Gu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, Xin Xu, Tongtong Wu, Kun Wang, Yang Liu, Zhen Bi, Jungang Lou, Yuchen Eleanor Jiang, Hangcheng Zhu, and 30 others. 2026. Skillnet: Create, evaluate, and connect AI skills. *CoRR*, abs/2603.04448.

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

Hongjun Liu, Yifei Ming, Shafiq Joty, and Chen Zhao. 2026a. Harnessing LLM agents with skill programs. *CoRR*, abs/2605.17734.

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

Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. 2026. Skillclaw: Let skills evolve collectively with agentic evolver. *CoRR*, abs/2604.08377.

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

Qwen Team. 2026. Qwen3.5: Accelerating productivity with native multimodal agents.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 16022–16076. Association for Computational Linguistics.

Chenxi Wang, Zhuoyun Yu, Xin Xie, Wuguannan Yao, Runnan Fang, Shuofei Qiao, Kexin Cao, Guozhou Zheng, Xiang Qi, Peng Zhang, and Shumin Deng. 2026. Skillx: Automatically constructing skill knowledge bases for agents. *CoRR*, abs/2604.04804.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. 2025. Inducing programmatic skills for agentic tasks. *CoRR*, abs/2504.06821.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. 2026. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *CoRR*, abs/2602.08234.

Renjun Xu and Yang Yan. 2026. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *CoRR*, abs/2602.12430.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Chuanqi Tan, and Chang Zhou. 2023. Scaling relationship on learning mathematical reasoning with large language models. *CoRR*, abs/2308.01825.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, Xue Liu, Xiaoxiao Li, and Philip S. Yu. 2026. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *CoRR*, abs/2604.01687.

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. 2025. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *CoRR*, abs/2506.05176.

A Implementation Details

In this paper, we use Qwen3-Embedding-8B (Zhang et al., 2025) as the embedding model to caculate the similarity. We here list the global hyperparameters settings across all tasks as below:

Parameter	Value
similarity threshold δ	0.8
delta pass rate for skill create ϵ_{c}	0.4
delta pass rate for skill merge ϵ_{m}	0.4
delta pass rate for skill improve ϵ_{i}	0.4

Table 5: Hyperparameters details for rejection sampling.

A.1 Code

Rejection Sampling For rejection sampling, we use 8,000 data points sampled from OpenCodeReasoning, and run 5 separate runs for each data point.

Skill Library Building For the initial skill library building of offline and hybrid modes, we incorporate 500 randomly sampled data points from OpenCodeReasoning dataset.

A.2 Agent

Skill Library Building For the initial skill library building of offline and hybrid modes, we use the training split of τ^2-Bench.

A.3 Evaluation Details

For agent tasks, we conduct 4 separate runs across all methods and settings. For code tasks, we conduct 5 separate runs across all methods and settings. Finally, we report the average score of the best iteration across all runs for each setting. Moreover, during the skill library building of evaluation process, we use the same similarity threshold δ as rejection sampling.

A.4 Training Details

For supervised fine-tuning, we trained Qwen3.5-4B with LlamaFactory. We list the detailed hyperparameters below:

B Rejection Sampling Procedures

We provide detailed pseudo-code for the three rejection sampling procedures introduced in §3.4.

Parameter	Value
Learning Rate	1e-6
Train Batch Size	64
Number of Training Epochs	3
Max Length	65536
LR Scheduler	cosine
Warmup Ratio	0.1

Table 6: Supervised fine-tuning details for SkillComposer.

Algorithm 2 Rejection Sampling for Skill Create

Require: Task set $\mathcal{X}$, skill executor LLM, base composer $\mathcal{C}$, samples n, threshold ϵ_{c}
Ensure: Training set $\mathcal{D}_{\text{create}}$

$\mathcal{D}_{\text{create}} \leftarrow \emptyset$
for each task $x \in \mathcal{X}$ **do**
  $p_{\text{base}} \leftarrow \text{pass@}k(\text{LLM}(x), n)$
  $\tau \leftarrow$ sample one trajectory from $\text{LLM}(x)$
  $s \leftarrow \mathcal{C}_{\text{create}}(x, \tau)$
  $p_{\text{skill}} \leftarrow \text{pass@}k(\text{LLM}(x, s), n)$
  if $p_{\text{skill}} - p_{\text{base}} \geq \epsilon_{\text{c}}$ **then**
    $\mathcal{D}_{\text{create}} \leftarrow \mathcal{D}_{\text{create}} \cup \{(x, \tau) \rightarrow s\}$
  end if
end for

Algorithm 3 Rejection Sampling for Skill Merge

Require: Skill-task pairs $\{(s_i, x_i)\}$, skill executor LLM, base composer $\mathcal{C}$, samples n, threshold ϵ_{m}
Ensure: Training set $\mathcal{D}_{\text{merge}}$

$\mathcal{D}_{\text{merge}} \leftarrow \emptyset$
Compute pairwise similarity among all skills
for each similar pair (s_1, x_1), (s_2, x_2) **do**
  $s \leftarrow \mathcal{C}_{\text{merge}}(s_1, s_2)$
  $\Delta \leftarrow \frac{1}{2}\sum_{i=1}^{2}[\text{pass@}k(\text{LLM}(x_i, s), n) - \text{pass@}k(\text{LLM}(x_i, s_i), n)]$
  if $\Delta \geq \epsilon_{\text{m}}$ **then**
    $\mathcal{D}_{\text{merge}} \leftarrow \mathcal{D}_{\text{merge}} \cup \{(s_1, s_2) \rightarrow s\}$
  end if
end for

Algorithm 4 Rejection Sampling for Skill Improve

Require: Skill-task pairs $\{(s_i, x_i)\}$, skill executor LLM, base composer $\mathcal{C}$, samples n, threshold ϵ_{i}
Ensure: Training set $\mathcal{D}_{\text{improve}}$

$\mathcal{D}_{\text{improve}} \leftarrow \emptyset$
Compute skill-task similarity matrix
for each selected pair (s_{o}, x') with original task x **do**
  $\tau' \leftarrow$ sample one trajectory from $\text{LLM}(x', s_{\text{o}})$
  $s \leftarrow \mathcal{C}_{\text{improve}}(x', s_{\text{o}}, \tau')$
  $\Delta \leftarrow \frac{1}{2}\sum_{x_i \in \{x, x'\}}[\text{pass@}k(\text{LLM}(x_i, s), n) - \text{pass@}k(\text{LLM}(x_i, s_{\text{o}}), n)]$
  if $\Delta \geq \epsilon_{\text{i}}$ **then**
    $\mathcal{D}_{\text{improve}} \leftarrow \mathcal{D}_{\text{improve}} \cup \{(x', s_{\text{o}}, \tau') \rightarrow s\}$
  end if
end for

C Performance by Iteration

Figure 5 presents the performance trajectory across iterations for both online and hybrid modes. For τ^2-Bench, we report the overall score across all three domains (Retail, Airline, and Telecom). The curves illustrate how skill quality evolves over successive iterations, with SkillComposer consistently achieving steeper improvement curves compared to the untrained model in both settings.

D Inference Efficiency across Iterations

E Prompt Template

We present the prompt templates used for the four core operations in SkillComposer. Each prompt is instantiated with task-specific variables (denoted by `{{variable}}`) at inference time.

E.1 Skill Create Prompt

The following prompt instructs the skill composer to extract a reusable skill from a completed task trajectory.

Skill Create Prompt

```
You are a professional skill creator.
A skill is an executable procedure
that provides specialized instructions for
specific tasks.

[Task Description]
1. Given a task and a corresponding
trajectory generated by a language
model, your task is to generate a
reusable/generalizable skill based on the
```

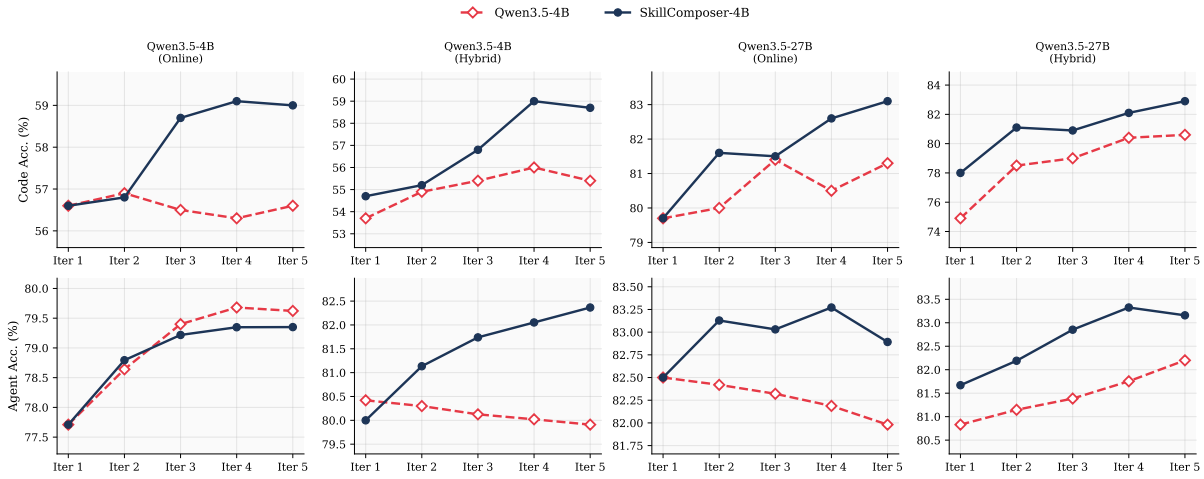

Figure 5: Average performance by iteration for online and hybrid modes on τ^2-Bench (average scores across Retail, Airline, Telecom) and LiveCodeBench v6. SkillComposer exhibits steeper and more sustained improvement across iterations compared to the untrained baselines in both settings.

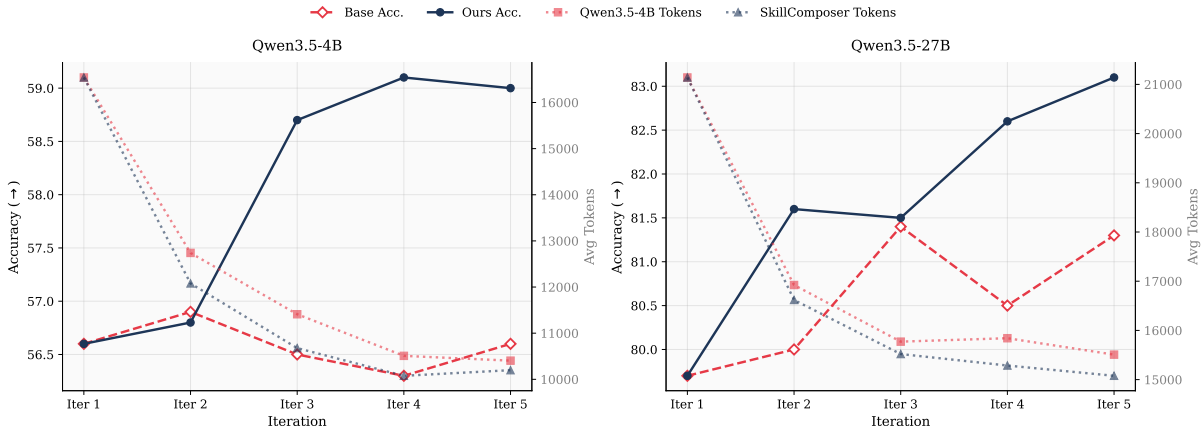

Figure 6: The token consumption across iterations of Qwen3.5-4B and SkillComposer-4B.

```
given contents.
2. The reusability/generalizability should
be both model-level and task-level:
- The skill should help current language
model not only solve current task but this
type of tasks
- The skill should help other language
models not only solve current task but this
type of tasks

[Skill Formulation]
The generated skill should consist of three
components including name, description and
body:
1. **name**: The identifier of the skill.
Once the name is generated, it will never
be further modified. Therefore, make its
name as general and abstract as possible,
rather than limiting it to the current input
problem and trajectory.
2. **description**: When to trigger, what
it does. This is the primary triggering
mechanism - include both what the skill
does AND specific contexts for when to use
```

```
it. All "when to use" info goes here, not
in the body.
3. **body**: A document in markdown-style.
It should contain the name of the skill and
several sections with detailed instructions
(i.e., what you want the skill to do and
roughly how it should do it) to solve this
type of tasks.

[Skill Create Instructions]
1. When writing a skill, try to explain to
the model why things are important in lieu
of heavy-handed musty MUSTs.
2. Use theory of mind and try to make
the skill general and not super-narrow to
specific examples.
3. Start by writing a draft and then look
at it with fresh eyes and improve it.
4. Make sure not to introduce any specific
concepts, examples or details from the
input.

[Input Details]
Here are the task and the trajectory:
```

Task
{{task}}
Trajectory
{{trajectory}}

[Output Format]
Put your final output in a JSON object enclosed by a json code block like:
“‘json
{{
"name": <skill-name>,
"description": <skill-description>,
"body": <skill-body>
}}
“‘

E.2 Skill Merge Prompt

The following prompt instructs the skill composer to merge two related skills into a more general one.

Skill Merge Prompt

You are a professional skill merger. A skill is an executable procedure that provides specialized instructions for specific tasks.

[Task Description]
1. Given two seperate skills, your task is to merge them into a more generalizable and reusable skill.
2. The merged skill should cover both the common and diverse instructions within the given two skills. The reusability/generalizability should be both model-level and task-level:
- The skill should help current language model not only solve current question but this type of questions
- The skill should help other language models not only solve current question but this type of questions

[Skill Formulation]
The generated skill should consist of three components including name, description and body:
1. **name**: The identifier of the skill. Once the name is generated, it will never be further modified. Therefore, make its name as general and abstract as possible, rather than limiting it to the current input problem and trajectory.
2. **description**: When to trigger, what it does. This is the primary triggering mechanism - include both what the skill does AND specific contexts for when to use it. All "when to use" info goes here, not in the body.
3. **body**: A document in markdown-style. It should contain the name of the skill and several sections with detailed instructions (i.e., what you want the skill to do and roughly how it should do it) to solve this type of tasks.

[Skill Merge Instructions]
1. When merging two skills, try to explain to the model why things are important in lieu of heavy-handed musty MUSTs.
2. Use theory of mind and try to make the skill general and not super-narrow to specific examples.
3. The merged skill should cover both the common and diverse instructions within the given two skills.
4. Merge the two skills into one standalone, executable skill, not a simple combination.
5. Remove overlap and unify similar instructions into the clearest version.
6. Make the final result clearer, cleaner, more reusable and more generalizable than either original skill alone.

[Input Details]
Skill A
—
name: {name_a}
description: {description_a}
—
{body_a}

Skill B
—
name: {name_b}
description: {description_b}
—
{body_b}

[Output Format]
Put your final output in a JSON object enclosed by a json code block like:
“‘json
{{
"name": <skill-name>,
"description": <skill-description>,
"body": <skill-body>
}}
“‘

E.3 Skill Improve Prompt

The following prompt instructs the skill composer to refine an existing skill based on a new task trajectory.

Skill Improve Prompt

You are a professional skill improver. A skill is an executable procedure that provides specialized instructions for specific tasks.

[Task Description]
1. Given a question, an original skill, and a corresponding trajectory generated by a language model when injecting this original skill into its context, your task is to generate an improved version of the skill.
2. The improved skill should maintain its reusability/generalizability at both model-level and task-level:
- The skill should help current language model not only solve current question but this type of questions

- The skill should help other language models not only solve current question but this type of questions
[Expert Analysis Before Improvement]
Before designing the improved skill, you must act as an expert evaluator:
- Carefully examine the trajectory and judge whether it fully completed the user's request.
- Decompose the trajectory's completion degree: identify specific parts that were executed well and parts that were insufficient or incorrect.
- If the request was completed successfully, extract reliable, generalizable experiences (best practices) that likely contributed to the success, and check whether the current skill already captures them.
- If certain aspects were not handled well, summarize the lessons learned and points of attention that the improved skill should explicitly warn about or guide the model to avoid.
- Use this analysis to inform the improved skill, ensuring that the final skill encodes both proven strategies and preventive guidance.

[Skill Formulation]
The generated skill should consist of three components including name, description and body:
1. **name**: The identifier of the skill. Once the name is generated, it will never be further modified. Therefore, make its name as general and abstract as possible, rather than limiting it to the current input problem and trajectory.
2. **description**: When to trigger, what it does. This is the primary triggering mechanism - include both what the skill does AND specific contexts for when to use it. All "when to use" info goes here, not in the body.
3. **body**: A document in markdown-style. It should contain the name of the skill and several sections with detailed instructions (i.e., what you want the skill to do and roughly how it should do it) to solve this type of tasks.

[Skill Improve Instructions]
1. Use the original skill as a foundation and produce an improved version that is clearer, more complete, and more reusable.
2. Preserve what already works, strengthen weak or missing guidance, and reflect the lessons from the trajectory.
3. Keep the skill generalizable across similar tasks rather than overfitting to the current example.
4. If the trajectory does not provide useful improvement signal, return the original skill unchanged.

[Input Details]
Here are the question, the trajectory, and the original skill:
Question
{{question}}
Trajectory
{{trajectory}}
Skill
—
name: {name}
description: {description}
—
{body}

[Output Format]
Put your final output in a JSON object enclosed by a json code block like:
“‘json
{{
"name": <skill-name>,
"description": <skill-description>,
"body": <skill-body>
}}
“‘

E.4 Skill Use System Prompt

The following system prompt equips the skill executor with a skill to guide task solving.

Skill Use Prompt

You are a professional skill user. The following is a skill that provides specialized instructions for specific tasks.
name: {name}
—
{body}
Read the skill above carefully, and use its instructions to solve problems. If you think the skill is unuseful or even wrong, ignore it and use your own knowledge instead.

F Case Studies

Figure 7 gives a case study on how skill evolves before and after skill improve. With the improved skill, the skill executor get higher pass@k on the same task.

Before Skill Improve: Pass@k=0.0

```
# solve_quadratic_budget_allocation
# Problem Type: Optimization / Greedy Selection with Convex Costs
#
## Overview
When faced with a constraint like "maximize total units" subject to "total cost <= M",
and where the cost to acquire k items from a source i is k^2 * P_i (or generally
convex), the most efficient units to buy are those with the lowest marginal cost.

For product i, the marginal cost of the 1st unit is P_i, 2nd is 3*P_i, 3rd is 5*P_i, etc. In
general, the j-th unit costs (2*j - 1)*P_i.

Instead of enumerating all units (which could reach billions), we determine a global
**cost threshold** (the price of the last unit we afford) and calculate the quantities
derived from it.

## Steps [Over-long Steps Illustration]

[Omit ~1300 tokens]

**Summary Algorithm:** [Redundant Skill Summary]
1. Sort `P` (optional optimization).
2. Binary search for `V` in `[0, M]`. Maintain `min_cost` function that returns
`sum(P[i]*x_i^2)` and `sum(x_i)` for given `x_i` derived from `V`.
3. Find max `V` s.t. `min_cost(V).cost <= M`.
4. Take base answer `base_ans = count_from_V(V)`.
5. Compute `rem_budget = M - cost_from_V(V)`.
6. Try incrementing `V` slightly or re-evaluating `count` for `V+1`. If `cost_for(V+1)
> M`, we can potentially add fractional part of `(V+1)` group. Add `floor(rem_budget /
(V+1))` (bounded by actual availability of `(V+1)` items).
7. Output total.
```

After Skill Improve: Pass@k=1.0

```
# Quadratic Budget Allocation Optimization

## Overview
This skill addresses resource allocation problems where multiple categories of items e
xist, each with a quadratic cost function. Unlike linear costs (where you simply pick ch
eapest per-
unit), here the marginal cost of subsequent units increases linearly (e.g., first unit costs
P, second costs 3P, third 5P...). The objective is to maximize total items purchased give
n a fixed budget M.

## Prerequisites & Constraints [Detailed Requirements]
-
 **Time Complexity Requirement:** Solution must scale to N <= 2*10^5 items and
M <= 10^18 budget within typical CP limits (~2 seconds).
- **Integer Precision:** All calculations must support 64-
bit integers; Python handles arbitrarily large integers automatically, ensuring safety ag
ainst overflow during intermediate summations.

## Implementation Steps [Wrought Step Illustration]

[Omit ~600 tokens…]

## Edge Cases & Best Practices [Valuable Experience]
- **Empty List:** If `N=0` or `M=0`, answer is 0.
- **Large Values:** Ensure no division by zero (check `P[i] !
= 0`) and handle big int operations if using languages without native BigInt support.
-
 **Performance:** Avoid recomputing `P[i]` divisions unnecessarily. Sorting `P` initia
lly can optimize finding terms quickly but linear scan is generally sufficient for `O(N l
og M)` given constraints.

## Example Walkthrough [Detailed Example]
**Input:** N=3, M=9, P=[4, 1, 9]
1. Filter: All valid.
2. BS Range [0, 9]. Try mid=4.
   - For P=4: k=floor((4//4+1)/2)=1. Cost=4.*1=4.
   - For P=1: k=floor((4//1+1)/2)=2. Cost=1.*4=4.
   - Total Cost=8 <= 9. Feasible. `best_V`=4. Low=5.
3. Continue BS until converging around 4.
4. Base Units = 3 (1 from P=4, 2 from P=1). Cost=8.
5. Remainder = 1.
6. Next Cost candidate (V+1=5). Available P=1 units at cost 5? Yes. Diff Count > 0.
7. Can afford 5? Rem(1)//5 = 0.
8. Final Answer = 3.
```

Figure 7: A case study on under-improved skill (left) and improved skill (right).