You Live More Than Once: Towards Hierarchical Skill Meta-Evolving

Anonymous ACL submission

Abstract

Test-time skill evolving is regarded as a new paradigm for enhancing deployed agentic systems. Existing works mainly focus on hard-coded skill evolving strategies or parametric learning that rely on expensive parameter updates in the underlying LLMs. In this paper, we demonstrate that test-time refinement of the skill evolving framework itself is necessary for continuous improvement of the agent systems in different downstream scenarios, and lightweight algorithmic adaptation is feasible. Specifically, we propose HiSME, a lightweight hierarchical skill meta-evolving solution that jointly optimizes skills and the skill evolving strategy by learning meta-skills from agents' task execution traces. Experiments on diverse agentic benchmarks show that meta-evolving can produce a higher-quality skill library than pure skill evolving and can derive diverse meta-skills for different scenarios, thereby facilitating future continual experience learning. Our code is temporarily public at https://anonymous.4open.science/r/HiSME-BD45.

1 Introduction

Skills have recently been introduced as plug-and-play modules for large language model-based agents. As a desired way to extend the manually curated skills, **skill evolving** refers to a test-time learning paradigm in which, after deployment, an agent improves its behavior by distilling execution traces into reusable skills and maintaining them over time (Ni et al., 2026; Wang et al., 2026a; Liu et al., 2026b). In this sense, skill evolving plays a role analogous to model parameter updates, while remaining lightweight, modular, and explainable in text space.

Recent work has begun to develop skill evolving strategies, largely focusing on maintaining a skill library that improves agent performance (Wang et al., 2026a; Ma et al., 2026b; Miao et al., 2026; Gong et al., 2026; Shi et al., 2026a; Liu et al., 2026a). However, in most cases, the evolving strategy itself remains "static": it does not accumulate evidence to improve its own extraction, maintenance, or refinement process over the course of deployment. This creates two practical limitations. First, when the skill generation algorithm has a systematic flaw, the agent system must repeatedly spend maintenance effort repairing the same defect pattern. Second, a fixed evolving strategy is unlikely to fit all target domains, where requirements on skill specification, scope, and maintenance may vary substantially or even conflict. For example, data-processing tasks may benefit from general callable functions, whereas safety-critical settings may favor narrow checklist-type skills. Some works construct skill evolving frameworks through parametric learning (Xia et al., 2026; Shi et al., 2026b; Tu et al., 2026; Jiang et al., 2026; Pan et al., 2026; Wang et al., 2026b), which however impose substantial computational overhead in practice.

Therefore, the core question we aim to study is:

Can the skill evolving framework itself also be optimized automatically at test time?

We answer this question with **Hierarchical Skill Meta-Evolving** (HiSME), a framework that optimizes both task skills and the skill maintenance process itself by producing and maintaining meta-level experiences. HiSME aims not only to build a reusable skill library, but also to adapt the skill evolving framework itself so that it can keep improving skills for future tasks and domains. In HiSME, all updates are performed at test time in the text space, without modifying the parameters of the underlying LLMs, therefore light-weighted.

We evaluate HiSME on several agentic benchmarks. The results show that HiSME produces stronger skill libraries than static evolving baselines, improves end-to-end agent performance, and learns evolving strategies that surpass the framework's initial capability. These findings suggest

that skill evolving can benefit from optimizing not only the skills being created, but also the process that creates and maintains them.

Our contributions are as follows:

- We identify the limitations of static skill evolving solutions and formulate skill meta-evolving as a multi-level residual optimization problem.
- We propose HiSME, a lightweight text-based framework that uses unified execution feedback to improve both task-level skills and the evolving procedure that maintains them.
- We validate HiSME across diverse skill evolving scenarios through experiments and analyses, demonstrating the effectiveness of test-time meta-evolving.

2 Preliminaries

2.1 Agentic Systems

Considering a ReAct-style agent system (Yao et al., 2022) based on LLMs with fixed parameters, we denote the **executor** as EXE. Given a task x from dataset $\mathcal{D}$, the executor generates a corresponding **trace** by taking actions and receiving observations until the task is solved in N steps. This is regarded as a Markovian Decision Process (MDP) with initial state $x_0 = x$:

$$\begin{aligned} a_i &\sim \textsc{Exe}(\cdot|x_i), \\ x_{i+1} &= \mathcal{T}(x_i, a_i, o_i) \coloneqq x_i \oplus \{(a_i, o_i)\}, \\ y &\coloneqq x_N = \{x\} \cup \{(a_i, o_i)\}_{i=1}^{N}, \end{aligned} \tag{1}$$

denoted briefly $y \sim \textsc{Exe}(\cdot|x)$. Its utility $U(y)$ is measured by task success, efficiency, or benchmark-specific scoring.

2.2 Skill Evolving as Approximate Parametric Update

We aim to maximize

$$J_0(\textsc{Exe}) \coloneqq \mathbb{E}_{x\sim\mathcal{D}, y\sim\textsc{Exe}(\cdot|x)}\left[U(y)\right]. \tag{2}$$

Since the executor is fixed, parametric update like $\textsc{Exe}_{t+1} = \textsc{Exe}_t + \alpha_0 \nabla J_0(\textsc{Exe}_t)$ is infeasible. We instead propose **skills** $S \coloneqq \{s_i\}_{i=1}^{|S|}$, reusable artifacts distilled from past traces, conditioned on which the executor can solve future problems,

$$\textsc{Exe}_S(\cdot|x) = \textsc{Exe}(\cdot|x, S). \tag{3}$$

In practice, we use a lightweight retriever to select relevant skills from the library and expose them to the executor. The design of skill retrieval is not the main focus of this paper. We elaborate our implementation details in appendix B.

Let $\widehat{S}_0$ denote the initial skill library, and in our setting $\widehat{S}_0 = \emptyset$. To optimize the expected utility J_0, a **skill evolving** algorithm EVO consistently updates the skill library by learning from datasets $\mathcal{D}_t$ and corresponding traces generated by EXE at each timestep t,

$$\widehat{S}_{t+1} \sim \textsc{Evo}(\widehat{S}_t, \mathcal{D}_t, \textsc{Exe}), 1 \le t \le T, \tag{4}$$

$$\widehat{S}_T \approx \arg\max_S J_0(\textsc{Exe}_S). \tag{5}$$

From this view, skill evolving approximates the executor's ideal parametric update step.

3 Methodology

We first illustrate the high-level framework of our method in §3.1, then elaborate the design of each algorithmic role in §3.2 to 3.4.

3.1 Hierarchical Residual Optimization

If we dive deeper into the perspective proposed in §2: considering the agent system (executor) as a flawed prior for solving a problem:

$$\epsilon_0 = \max_{\textsc{Exe}} J_0(\textsc{Exe}) - J_0(\textsc{Exe}_0) > 0, \tag{6}$$

A skill-evolving algorithm serves as a posterior optimizer on the **residual of its utility** at test time. The improved utility from a skill evolving algorithm $\textsc{Evo}_0$ on the executor is

$$\begin{aligned} J_1(\textsc{Evo}_0) = &\mathbb{E}_{\{S_t\}_{t=1}^{T}\sim\textsc{Evo}_0(S_0,\mathcal{D},\textsc{Exe}_0)} \\ &\left[J_0(\textsc{Exe}_{S_T}) - J_0(\textsc{Exe}_0)\right]. \end{aligned} \tag{7}$$

Hopefully $J_1 > 0$ if the algorithm is valid.

As a natural extension of this perspective, we introduce the core claim of HiSME: **the skill evolving algorithm EVO itself, designed with human prior, can also be flawed, and thus further optimized at test time**:

$$\epsilon_1 = \max_{\textsc{Evo}} J_1(\textsc{Evo}) - J_1(\textsc{Evo}_0) > 0. \tag{8}$$

The remedy is the same as how skill evolving works on the executor: we can learn run-time experiences, namely **meta-skills**, to improve the evolving algorithm as a posterior:

$$\widehat{\textsc{Evo}}_{t+1} = \widehat{\textsc{Evo}}_t(\cdot|M_t), \tag{9}$$

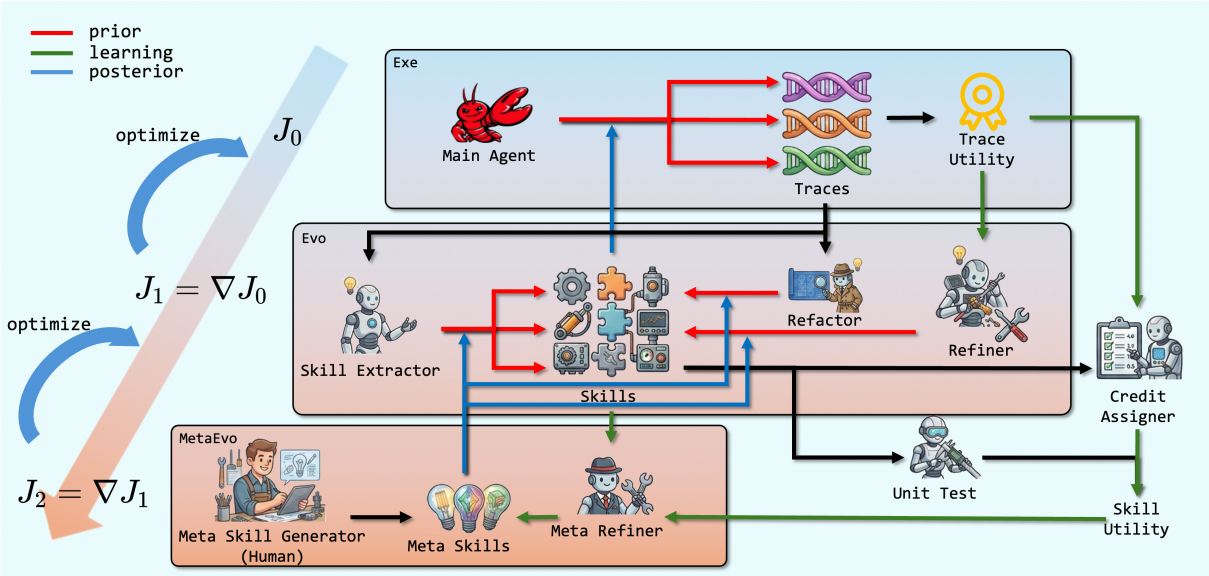


Figure 1: Overview of HiSME. Red arrows denote pre-hoc guidance to the executor, blue arrows denote posterior evidence from traces to skills, and green arrows denote learned feedback to the evolving procedure.

where M_t denotes the meta-skill set at timestep t. HiSME implements a meta-evolving algorithm that periodically updates the meta-skills:

$$M_{t+1} = \textsc{MetaEvo}(M_t, \mathcal{D}_t). \tag{10}$$

As summarized in Table 1 from the perspective of optimization, skills represent the first-order information of trace utility, and meta-skills represent the second-order information of trace utility. We elaborate in the following subsections on the detailed design of HiSME on skill and meta-skill evolving. The overall hierarchy is illustrated in Figure 1.

3.2 Skill Generation

HiSME generates skills from the execution traces and corresponding utility feedback with two strategies. The **extractor** receives each single trace and identifies potential reusable fragments or experiences according to human engineer's prior knowledge embodied in its general ability. The **refactorer** receives groups of similar or related traces or skills and abstracts potential shared structures or knowledge. For refactoring, we design a two-level ranking pipeline: first build a graph with nodes being all the segments from historical traces, and edge weights being semantic similarity measured by embeddings and N-gram statistics. Then we heuristically extract clique from this graph to find potentially similar segments. The final top-k cliques will be passed to the refactorer for concise reusable skills. The details of overlap graph construction and maintenance are provided in Appendix C.

The resulting new skills at step t is

$$\mathcal{G}_t = \textsc{Ext}(\tau_t) \cup \textsc{Rfc}(\mathcal{C}_t), \tag{11}$$

where τ_t is the current trace, $\mathcal{C}_t$ is the set of overlap-supported trace/skill groups.

Following (Wang et al., 2026a), we consider three types of skill semantics in this work: callable functions, workflow frameworks, and domain-specific knowledge. The full skill bodies are injected into context of the executor directly.

3.3 Skill Evaluation and Maintenance

Since the skills are generated basically from LLMs' prior knowledge, they require further evaluation according to their future usage and maintenance based on the evaluation signals.

For evaluation, HiSME checks both short-term and long-term utility of a skill. In respect of short-term utility, the **bundle tester** builds and maintains with LLMs a test bundle for each skill, including both unit test cases and integration test cases. It mainly checks whether a skill implements the contract it declares faithfully. In respect of long-term utility, the **credit assigner** uses real executor traces to estimate with LLMs whether each exposed skill was helpful, harmful or neutral to the given task. Besides, **usage statistics** mechanism will record the lifetime events for each skill, among which here we concern retrieval and execution times.

Putting all these together, HiSME keeps an evidence state for each skill:

$$e_t(s) = \{B(s),\ c_t(s),\ u_t(s)\}. \tag{12}$$

Object	Order	Role in HiSME
Trace	0	Ultimate utility target produced by the frozen executor.
Skill evolving	1	Create, revise or filter skills as experiences to optimize trace utility.
Skill meta-evolving	2	Create, revise or filter meta-skills as experiences to optimize skill utility.

Table 1: Optimization hierarchy in HiSME.

Here $B(s)$ is the latest bundle for the current version of the skill. $c_t(s)$ are credit events collected after exposure or use. $u_t(s)$ are lifetime events.

With these evaluation signals, maintenance on the skill library becomes possible. HiSME adopts mainly two types of maintenance mechanism. First, the **refiner** revises with LLMs an existing skill when its evidence state indicates that its scope, interface, implementation, or trigger condition can be improved. The corresponding test bundle may alter as well due to the skill being revised:

$$\begin{aligned}&(s_{t+1}, B(s_{t+1})) \sim \textsc{Rfn}(s_t, e_t(s_t)),\\&\textsc{release}(s_{t+1}) = \mathbb{1}[\textsc{Gate}(s_{t+1}, B(s_{t+1}))].\end{aligned} \tag{13}$$

The refined version is released only after passing the maintained bundle.

Second, **filter** mechanism simply changes the exposure level of a skill according to its performance. Skills with explicit harmful evidence and no helpful evidence can be marked as disabled. Let $\mathcal{E}_s$ denote the set of credit events assigned to skill s in a maintenance window. The filter first aggregates two counts:

$$h_s = \sum_{e \in \mathcal{E}_s} \mathbb{1}[j_e = \text{harmful}], \tag{14}$$

$$p_s = \sum_{e \in \mathcal{E}_s} \mathbb{1}[j_e = \text{helpful}], \tag{15}$$

With the credit-filter threshold τ and the positive-protection threshold τ_p, the filter gate is

$$\textsc{Filter}(s) = h_s \geq \tau \ \wedge \ p_s < \tau_p. \tag{16}$$

Thus, a skill is disabled only when harmful evidence exceeds the filter threshold and the skill lacks enough positive evidence.

3.4 Meta-skill Generation

Analogous to optimizing skills with evolving framework in §3.3, the maintenance algorithm itself is not perfect and requires evaluation and optimization.

For meta-evaluation, HiSME stores experiences as meta-skills for each LLM-based algorithm $r \in \{\textsc{Ext}, \textsc{Rfc}, \textsc{Rfn}\}$ in a role-specific replay buffer $\mathcal{B}_t^r$. An experience row for role r links each produced artifact to its later utility:

$$\mathcal{B}_t^r = \{\rho^r(s) = (s, e_t(s)) : s \text{ is produced by } r\}, \tag{17}$$

where s is the skill or skill group produced from that input, and e is the utility observed after publication or refinement, as defined in (12).

For meta-evolving, HiSME periodically samples experience rows from the corresponding replay buffer of each role, and call an LLM agent METAEVO to summarize one or more rules-of-thumb as meta-skills. In practice, we maintain no more than 5 pieces of meta-skills. Thus, we prompt METAEVO to carefully decide which rules are most important to keep.

$$M_{t+1}^r = \textsc{MetaEvo}^r(M_t^r, \textsc{Sample}(\mathcal{B}_t^r)). \tag{18}$$

The resulting meta-skills are supplied to the role's context as learned posterior:

$$r_t(\cdot) = r(\cdot; M_t^r), \quad r \in \{\textsc{Ext}, \textsc{Rfc}, \textsc{Rfn}\}. \tag{19}$$

In summary, HiSME is a simple but effective meta-evolving framework with the following advantages:

- **Lightweight**. The only additional developing cost comes from adding a meta-skill extractor.
- **General**. The meta-evolving system of HiSME can be deployed to any LLM-based framework with context.

4 Experiments and Analyses

4.1 Experimental Setup

Benchmark and dataset. We evaluate HiSME on two agentic settings. BFCL-v3 multi-turn base is our primary completed benchmark (Patil et al., 2025). It evaluates function selection, argument binding, state tracking, and official multi-turn tool-call validity. MineDojo evaluates long-horizon embodied interaction in an open-world environment (Fan et al., 2022).

Metrics and baselines. For BFCL-v3, we report `official_valid`, the official BFCL multi-turn validation rate (Patil et al., 2025), as the primary metric, and `avg_score`, the average partial-credit score assigned by the BFCL evaluator. We further report call-level recall and precision to diagnose tool-call coverage and correctness, respectively. For MineDojo, we report `success_rate` as the primary metric and use average environment steps and executor–environment interaction rounds as efficiency diagnostics.

For both benchmarks, token cost is measured on held-out evaluation tasks and averaged per task. `in tok.` counts the prompt-side tokens consumed by the executor across all turns. `out tok.` counts the averaged generation tokens in the form of assistant/tool-call messages. `total tok.` is the sum of both.

We compare our method with `HiSME-static` ablation intended for showing performance advantage against a mostly aligned static evolving algorithm. Since crucial for our main conclusion, we set this comparison in our main results. We also compare our method with SkillX (Wang et al., 2026a), a recent related static evolving algorithm for skill lifetime maintenance. We re-implement their method based on their open-source implementation.[1]

Training configuration. All experiments use `claude-sonnet-4-6` as both the base ReAct executor and the LLM behind maintenance roles. We use deterministic decoding for executor/tool-use evaluation. For BFCL-v3, the executor outputs thoughts and tool callings and observes tool call feedbacks. For MineDojo, the executor outputs Python programs representing primitive compositions to act in the environment, and observes the resulting environment state. The max rounds for these benchmarks are 20 and 100, respectively.

For each benchmark, we use a fixed train/test split: training tasks are used for online skill evolving, while held-out test tasks are evaluated with a frozen skill repository. The sizes of training set / test set for these benchmarks are 50 / 50 and 30 / 30, respectively. For our methods, we invoke refactoring, filtering and meta-skill updating at every 5 tasks. Our method design is based on single-thread assumption. For efficiency concern, however, in practice we run all tasks between two meta updates in parallel and update skills following a thread-safe mechanism.

We sample 3 extractor candidates for each input because single-trace extraction has higher uncertainty and is more sensitive to prompt priors. Both refactorer and refiner generates one candidate at each time. Implementation details of prompt formats, truncation limits, skill fields, bundle fields, retry budgets, graph parameters, and retrieval score constraints are elaborated in Appendix B.

4.2 Main Results

Table 2 summarizes the held-out results on BFCL-v3 and MineDojo. We highlight three main observations from these results.

Skills act as effective test-time residuals. Across all benchmarks, HiSME improves the frozen executor without changing model parameters. On BFCL-v3, the gain appears not only in official validity but also in call recall and precision, indicating that the repository changes the structure of tool-call trajectories rather than merely adding more prompt context. The increased token cost is due to producing more otherwise missing calls. On MineDojo, HiSME obtains the highest success rate while reducing execution rounds and total token use, suggesting that useful skills can shorten interaction even in long-horizon embodied tasks.

Our framework has architectural advantage over the representative baseline work. Our results show that HiSME-static already improves over SkillX, isolating the value of HiSME's static lifecycle.

Meta-evolving succeeds in optimizing further residual utility of the agent system. HiSME-static already provides a strong static evolving framework through bundle-gated release, posterior refactoring, credit-based repair, and conservative filtering. Full HiSME uses the same framework as the static variant except for adding a meta-evolving mechanism and substantially improves over it. The performance gain is therefore attributable to learned role guidance that affects how candidate skills should be built and maintained.

In the following sections, we perform further analyses on BFCL-v3 benchmark in consideration of cost.

4.3 Ablation Studies

We ablate the key algorithmic roles of HiSME to explore the effect of each role on the final performance. The results are in Table 3.

[1] Appendix D reports an executor-alignment check on BFCL-v3. Injecting the same SkillX library through our executor performed worse than the SkillX official-style runner, so the main table reports the stronger SkillX result.

(a) BFCL-v3

Method	official_valid ↑	avg_score ↑	recall ↑	precision ↑	in tok. ↓	out tok. ↓	total tok. ↓
No Skill	0.5200	0.7691	0.8434	0.7254	61142.3	1016.8	62159.1
SkillX	0.6600	0.7689	0.8837	0.7026	73514.6	1150.3	74664.9
HiSME-static (ours)	0.6939	0.8143	0.8670	0.7775	68813.4	1052.3	69865.7
HiSME (ours)	**0.7800**	**0.8511**	**0.9040**	**0.8136**	**68012.2**	**1038.0**	**69050.2**

(b) MineDojo

Method	success_rate ↑	env_steps ↓	exec_rounds ↓	in tok. ↓	out tok. ↓	total tok. ↓
No Skill	0.5667	10072.0	54.0	432576.6	71235.2	503811.8
SkillX	0.7333	11209.1	63.1	**279876.9**	212131.6	492008.6
HiSME-static (ours)	0.7333	**4643.8**	63.8	442915.7	87021.3	529937.0
HiSME (ours)	**0.8333**	5570.9	**38.5**	337374.5	**74910.8**	**412285.3**

Table 2: Held-out test results on BFCL-v3 and MineDojo. Arrows indicate the preferred direction. Bold highlights the best value among skill-based methods in each column; No Skill is included as a reference baseline.

Method	official_valid ↑	Δ	avg_score ↑	Δ	call recall ↑	call precision ↑
HiSME	**0.7800**	-	**0.8511**	-	**0.9040**	**0.8136**
static	0.6939	-0.0861	0.8143	-0.0368	0.8670	0.7775
w/o extractor	0.6800	-0.1000	0.8001	-0.0510	0.8755	0.7501
w/o refiner	0.6939	-0.0861	0.8046	-0.0465	0.8730	0.7548
w/o refactor	0.5400	-0.2400	0.7796	-0.0715	0.8581	0.7339
w/o credit filter	0.7347	-0.0453	0.8135	-0.0376	0.8850	0.7647

Table 3: BFCL-v3 component ablations on the held-out test set.

Reusable structure requires cross-trace evidence. We find that removing refactor causes the largest degradation, making it the most important skill generation component in BFCL-v3. This supports the role of the overlap graph: reusable procedures and experiences are more reliably identified from shared evidence than from isolated traces.

Local proposal and posterior consolidation are complementary. Removing the extractor also hurts performance, even with refactoring enabled. Despite with least evidence, the human engineer's prior on reusability can be a strong source of performance gain complementing cross-trace evidence from real use cases.

Post-hoc evaluation and maintenance complete the closed-loop skill generation procedure. Removing either the refiner or the filter mechanism shows considerable degradation on performance. This suggests that generated skills require further test and maintenance to function well in future use. The refiner shows higher utility against the filter due to enriched operations on skill semantics, but filtering out harmful skills serves as a important role as well.

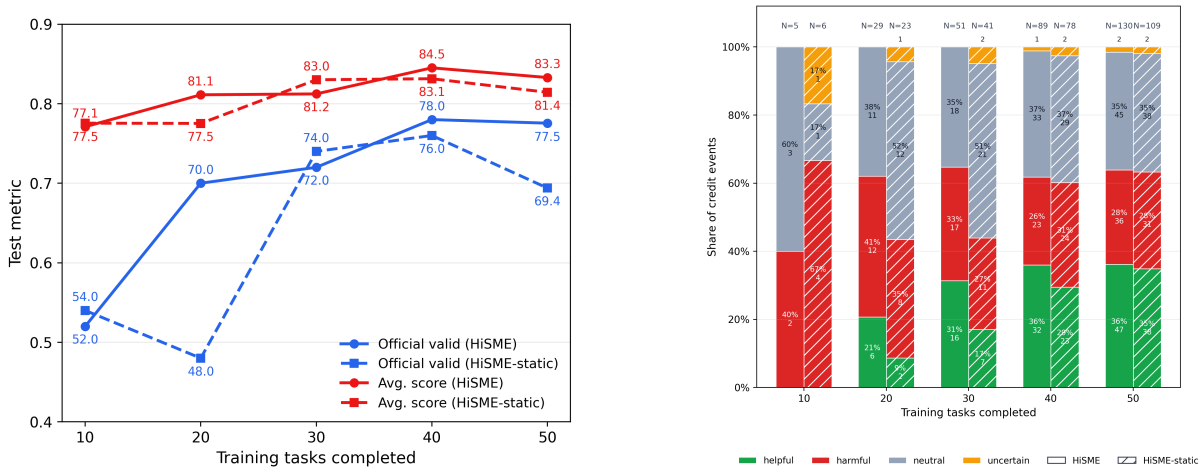
4.4 Process Evaluation

We further study the dynamic during evolving of HiSME and how it differs from static algorithm. Specifically, we save and analysis the intermediate skill libraries every fixed steps. We first test their performance on held-out test set. Then we perform credit assignment with the same pipeline during training on these skills to see their specific contribution to the final results.

As shown in Figure 2(a), both of our evolving algorithms consistently improves test performance as evolving proceeds. Meanwhile, the test performances of meta-evolving overall surpass skill evolving during training. This suggests that evolving on skill evolving algorithm itself can improve the skill quality during most of the training process. In Figure 2(b), both algorithms produce increasing helpful skills and decreasing harmful skills, while meta-evolving again surpasses static skill evolving by enhancing more and harming less performances of the executor.

4.5 Meta Test

Results above suggest that HiSME generates better skill library compared with static algorithm. We further explore whether the produced meta-skills bring transferable experience for future skill evolv-



(a) (b)

Figure 2: Process evaluation results compared between HiSME and HiSME-static. (a) Test result with skills accumulated during training. (b) Analyses on credits of each skills on test time in four categories.

ing. We take the role-level meta-skills learned by a completed HiSME run, freeze them, and use them as initialization for a new static evolving run. All other settings are aligned with cold-start HiSME-static. This setting tests whether the learned evolving rules can improve a future evolving process.

The results are shown in Table 4. Frozen meta-skills substantially improve `official_valid` over cold-start static evolving, nearly matching full HiSME on this metric, while maintaining close `avg_score`. This indicates strong but not full transfer: the learned role guidance helps the next static run produce more officially valid complete traces, but does not fully preserve the fine-grained tool-call quality of cold-start static evolving.

4.6 Case Studies

We include two qualitative studies to inspect whether HiSME's improvements come from grounded maintenance rather than accidental prompt expansion. Appendix E gives the detailed evidence.

Meta-skill evolution is grounded in observed candidate behavior. We trace five learned extractor meta-skills back to concrete evidence from BFCL-v3 candidate groups. The learned rules do not merely restate generic prompting advice. They distinguish schema-grounded contracts and cross-turn state dependencies from brittle rules triggered by surface words such as "urgent" or by stylistic preferences such as message brevity. This supports our interpretation that meta-evolving learns operational constraints for future skill generation from real skill outcomes.

Skill version updates make targeted semantic changes. We examines how skills are actually revised. Here are two representative skills: a TradingBot order-confirmation workflow and a VehicleControlAPI brake-before-start rule. In both cases, HiSME keeps the useful invariant but changes the condition under which it should apply: one update narrows a workflow trigger to immediately preceding order placement, while the other specifies the exact brake-pedal argument required by the API. This shows that skill evolution is not unrestricted rewriting; it uses task evidence to make local, testable changes to scope or contract.

5 Related Work

5.1 Static and RL-Based Skill Evolving

Trace-to-skill extraction. A first line of work studies how to convert agent experience into reusable external skills. Trace2Skill distills trajectory-local lessons into transferable skills (Ni et al., 2026), while SkillX organizes planning knowledge, functional skills, and atomic skills into a multi-level skill base (Wang et al., 2026a). Related systems such as SkillForge, SkillGen, and SkillMaster further explore memory-like skill stores, domain-specific support skills, inference-time skill synthesis, and autonomous skill editing (Liu et al., 2026b; Ma et al., 2026a; Yang et al., 2026).

Skill repositories and lifecycle control. A second line focuses on how skills should be stored, verified, selected, and maintained after creation. Co-

Method	official_valid ↑	avg_score ↑	call recall ↑	call precision ↑	total tok. ↓
static from scratch	0.6939	**0.8143**	0.8670	**0.7775**	**69865.7**
static from meta	**0.7708**	0.8020	**0.8847**	0.7440	70805.1

Table 4: BFCL-v3 meta-test results. Frozen-meta static uses meta-skills learned by HiSME as fixed role guidance for a new static evolving run, while disabling online meta-skill updates. Deltas are computed against cold-start static evolving.

EvoSkills and SkillClaw emphasize verifier-guided or repository-level skill evolving (Zhang et al., 2026; Ma et al., 2026b). SkillLens studies hierarchical and multi-granularity reuse (Miao et al., 2026). SkillMOO optimizes skill bundles under multiple objectives such as pass rate and cost (Gong et al., 2026), while PSN and SkillsVote study maturity-aware updates, rollback, recommendation, and evidence-gated lifecycle governance (Shi et al., 2026a; Liu et al., 2026a). These works show that useful skills require long-term repository management, not merely one-shot extraction.

RL-based and multi-agent skill adaptation. A third line couples skills with reinforcement learning or multi-agent adaptation. SkillRL, Skill1, D2Skill, XSkill, SkillMAS, and OpenClaw-RL use downstream rewards, verbal feedback, or agent restructuring to improve skill selection, utilization, distillation, or multi-agent behavior (Xia et al., 2026; Shi et al., 2026b; Tu et al., 2026; Jiang et al., 2026; Pan et al., 2026; Wang et al., 2026b). These methods demonstrate that skill-related adaptation can be optimized from task outcomes, although the learned executor, policy, or multi-agent controller is usually the central object of optimization.

HiSME is related to all three directions. It also extracts and maintains skills, but focuses on using later skill outcomes to guide the runtime evolving procedure itself.

5.2 Text-Based Optimization

Prompt-level search. OPRO formulates optimization as a natural-language search process in which the language model proposes new candidate prompts or solutions from previous trials (Yang et al., 2023). This line shows that textual hypotheses can be iteratively improved without gradient updates to model parameters.

Textual gradients and critiques. ProTeGi introduces natural-language "gradients" that describe how a prompt should change (Pryzant et al., 2023), and TextGrad generalizes this idea into a framework where textual feedback plays a role analogous to differentiation (Yuksekgonul et al., 2024). These methods use language feedback to revise textual objects rather than dense parameters.

Agent reflection and verbal reinforcement. Reflexion uses verbal feedback from prior trajectories to improve later agent behavior (Shinn et al., 2023). Similar reflection-style methods treat natural-language memory as a compact carrier of past failure modes and repair strategies.

HiSME adopts the same broad premise that text can be an optimization medium. Its optimized object, however, is a persistent skill repository and the role-level policy that creates, revises, and filters skills.

6 Conclusion

We introduce HiSME, a lightweight and efficient skill meta-evolving framework. By modeling skill evolving as a multi-level higher-order optimization problem over the utility of the final execution outcome, the skill and skill maintenance algorithms themselves can be automatically and adaptively tuned. Meta-skills are shown to be a simple and effective adaptive optimization method for LLM agent algorithms. The experiments not only demonstrate the advantages of the skills produced by our algorithm, but also reflect the advantages of meta-skills in downstream task adaptation.

Limitations

Inaccurate credit The framework still depends on LLM-based credit assignment, bundle maintenance, and refinement. When those judgments are wrong, useful skills can be over-pruned or harmful skills can survive longer than they should.

Extra train cost The system introduces extra runtime cost because maintenance and meta-maintenance require additional model calls. This cost is acceptable when skills are reused many times, but it is a real tradeoff in short-lived or low-reuse settings.

Limited skill format We only consider rules-of-thumb as meta-skills. More enriched formats, like

callable functions and workflows, are possibly beneficial for meta evolving. Furthermore, higher levels of evolution are also possible for automatically optimizing meta-evolving framework. We leave these possibilities for future works to explore.

References

Linxi (Jim) Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andy Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. 2022. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *ArXiv*, abs/2206.08853.

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

Guanyu Jiang, Zhaochen Su, Xiaoye Qu, and Yi R. Fung. 2026. Xskill: Continual learning from experience and skills in multimodal agents.

Hongyi Liu, Haoyan Yang, Tao Jiang, Bo Tang, Feiyu Xiong, and Zhiyu Li. 2026a. Skillsvote: Lifecycle governance of agent skills from collection, recommendation to evolution. *Preprint*, arXiv:2605.18401.

Xingyan Liu, Xiyue Luo, Linyu Li, Gang Huang, Jianfeng Liu, and Hongli Qiao. 2026b. Skillforge: Forging domain-specific, self-evolving agent skills in cloud technical support.

Yuchen Ma, Yue Huang, Han Bao, Haomin Zhuang, Swadheen Shukla, Michel Galley, Xiangliang Zhang, and Stefan Feuerriegel. 2026a. Skillgen: Verified inference-time agent skill synthesis.

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

Yongliang Miao, Ziyang Yu, Liang Zhao, Bowen Zhu, and Hasibul Haque. 2026. Skilllens: Adaptive multi-granularity skill reuse for cost-efficient llm agents.

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

Shuaidong Pan, Yixian Liu, Jiaye Gao, Te Gao, Weiwen Liu, Jianghao Lin, Zhihui Fu, Jun Wang, Weinan Zhang, and Yong Yu. 2026. Skillmas: Skill co-evolution with llm-based multi-agent system.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 48371–48392. PMLR.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with "gradient descent" and beam search. In *Conference on Empirical Methods in Natural Language Processing*.

Haochen Shi, Xingdi Yuan, and Bang Liu. 2026a. Evolving programmatic skill networks. *ArXiv*, abs/2601.03509.

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

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning.

Songjun Tu, Chengdong Xu, Qichao Zhang, Yaocheng Zhang, Xiangyuan Lan, Linjing Li, and Dongbin Zhao. 2026. Dynamic dual-granularity skill bank for agentic rl. *Preprint*, arXiv:2603.28716.

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

Yinjie Wang, Xuyang Chen, Xiaolong Jin, Mengdi Wang, and Ling Yang. 2026b. Openclaw-rl: Train any agent simply by talking.

Peng Xia, Jianwen Chen, Han 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.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2023. Large language models as optimizers. *ArXiv*, abs/2309.03409.

Min Yang, Jing Piao, Xuanye Xia, Xiaochong Lan, Jiajun Chen, Yongshun Gong, and Yong Li. 2026. Skillmaster: Toward autonomous skill mastery in llm agents.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. Textgrad: Automatic "differentiation" via text.

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. *Preprint*, arXiv:2604.01687.

A Benchmark Details

A.1 BFCL-v3: Function Calling and Multi-Turn Tool Use

Berkeley Function Calling Leaderboard (BFCL) was introduced by the UC Berkeley Gorilla team as a benchmark for evaluating whether language models can select and invoke external functions correctly (Patil et al., 2025). The official BFCL description frames function calling as the ability to map a natural language request into one or more valid API/tool invocations, including correct function names, argument schemas, values, ordering, and cases where no function should be called. The benchmark covers serial and parallel function calls, multiple programming languages, AST-based evaluation, executable evaluation, and later versions add more realistic multi-turn and multi-step agentic settings.

In this paper, we use BFCL-v3 as the primary benchmark for online skill evolving because it stresses exactly the type of behavior a skill repository is supposed to improve: repeated argument binding, ordering-sensitive workflows, schema obedience, state tracking across turns, and tool-use recovery after errors. A BFCL task is not simply a question-answer pair. It provides a conversation, a tool environment, and an official expected call trace. The model must produce calls that are both semantically appropriate and compatible with the tool schema. This makes BFCL a useful setting for measuring whether a skill helps the executor call tools more reliably, or whether it pollutes the prompt and causes extra or malformed calls.

Our local BFCL related-task split uses a curated subset of BFCL-v3 multi-turn tasks. The current recommended setting is a deterministic 50/50 split: 50 training tasks for online evolving and 50 held-out tasks for frozen-store evaluation. Training is serial because the skill repository, overlap graph, and feedback memory are mutated after each task. Held-out evaluation is read-only and can run concurrently. The related-task manifest ranks examples by repeated tool-family patterns, lookup/action mixtures, path length, and multi-turn argument-binding failure modes.

One representative BFCL training task is `multi_turn_base_118`, a TradingBot workflow. The user first asks to inspect a stock watchlist, then remove the first stock, then buy 100 shares of AAPL at the current market price, then inspect the latest order, and finally cancel that order. The expected trace is:

```
turn 1: get_watchlist()
turn 2:
↪  remove_stock_from_watchlist(symbol='NVDA')
turn 3: get_stock_info(symbol='AAPL')
        place_order(order_type='Buy',
        ↪  symbol='AAPL', price=227.16,
        ↪  amount=100)
turn 4: get_order_details(order_id=12446)
turn 5: cancel_order(order_id=12446)
```

This example tests several reusable behaviors. The executor must bind the watchlist item `NVDA` from the first call into the remove operation; it must lookup the current AAPL price before placing a market-price order; and it must carry the returned order id into later inspection and cancellation calls. A useful skill for this region is not "always buy AAPL", but a narrower workflow rule: when a trading request asks for current-market execution, first retrieve the stock information, then place the order using the observed price, and preserve the generated order identifier for subsequent order-management turns. This is why our credit assigner creates focused bundle fragments around the relevant tool-call subsequence instead of turning the entire conversation into a single coarse regression case.

Another held-out example, `multi_turn_base_51`, combines two tool families: vehicle control and messaging. The expected trace locks all doors, starts the engine only after pressing the brake pedal, checks tire pressure and finds a nearby tire shop, then logs into the message API and sends a specific message from `USR001` to `USR002`. This tests cross-domain tool selection and precondition ordering. A skill that teaches "press brake before startEngine" can help the vehicle subtask, but it should not influence the later message API turn. This motivates our attribution-scope field in credit assignment and our dependency-aware refiner: a local vehicle workflow should be refined or tested on the vehicle fragment, not on the unrelated messaging fragment.

A.2 MineDojo: Long-Horizon Embodied Control in Open-World Minecraft

MineDojo is an embodied-agent benchmark built on Minecraft for open-ended, language-conditioned interaction in procedurally generated 3D worlds (Fan et al., 2022). We use MineDojo to evaluate long-horizon embodied resource collection and crafting. Unlike BFCL-v3, where the main challenge is to produce valid symbolic tool calls,

MineDojo requires the executor to interact with a live environment over many rounds: it must observe the local world state, move and rotate the camera, mine or attack blocks, collect dropped items, craft intermediate objects, equip tools, and recover from failed low-level actions.

Our setting differs from several Minecraft-based agent benchmarks and systems that expose higher-level Mineflayer-style JavaScript APIs. Such interfaces often provide semantically meaningful helper functions for navigation, block collection, or crafting, which can hide much of the low-level control problem. In contrast, our MineDojo executor operates through the environment's atomic Minecraft action space, including movement, camera control, attack, use, craft, equip, place, and destroy. Thus, a successful trajectory must compose primitive grounded actions rather than call a prebuilt semantic routine such as collecting a block or navigating to an object.

We also do not use a visual language model in the MineDojo experiments. Although MineDojo can provide egocentric RGB observations, our evaluation uses a purely text-based interaction interface. To make this possible, we modify the environment interface to expose MineDojo's ray-based perception signal. The ray channel reports visible blocks and entities in the agent's field of view. This gives the LLM executor grounded local affordances without relying on image recognition or providing a global map.

Our local MineDojo split focuses on programmatic Harvest and Tech Tree tasks, where success can be measured from the final inventory or item-use state. We use 30 training cases for online evolving and 30 held-out test cases for frozen-store evaluation. Both splits are organized into batches of five cases.

We use a dynamic CodeAct-style executor for MineDojo. At each round, the client converts the current environment state into a textual prompt containing the task, previous execution result, cached Minecraft knowledge queries, and compact ray-based world state. The LLM then either issues an auxiliary knowledge query, such as a recipe or item lookup, or writes executable Python control code. The code defines a `run(controller, obs)` function that operates through MineDojo's primitive controller actions. After execution, the resulting logs, errors, inventory changes, and updated observation are returned to the next round, allowing the LLM to iteratively repair failed attempts and continue the embodied task. Because this interface uses primitive MineDojo actions and text-only ray observations, it is not directly comparable to Minecraft agents that rely on Mineflayer-style semantic APIs or visual-language perception.

B Retrieval Implementation Details

Retrieval is used only to expose a small number of relevant active skills to the frozen executor. For each turn, HiSME builds a query from the current user request, the dialogue state, recent tool errors, and the previous assistant message. Each active skill is represented by a compact retrieval view containing its name, description, trigger conditions, allowed tools, domains, and a short body summary. We score each skill by a mixture of sparse lexical overlap, semantic similarity, tool/domain compatibility, and lifecycle trust:

$$\begin{aligned}\rho(q,s) =& \lambda_1 \rho_{\text{sparse}}(q,s) + \lambda_2 \rho_{\text{emb}}(q,s) \\ &+ \lambda_3 \rho_{\text{tool}}(q,s) + \lambda_4 \rho_{\text{trust}}(s).\end{aligned} \tag{20}$$

Only active skills are exposed during held-out evaluation. Trial skills may be exposed during training with lower priority, but archived and disabled skills are never retrieved. The top-K skills are rendered as prompt-only context in BFCL; they are not converted into callable tools in this setting.

C Refactorer Implementation based on Overlap-Graph

Overlap-graph refactoring is used to discover reusable structure that is weakly supported by a single trace but stable across related traces or existing skills. HiSME constructs a graph whose nodes are projected trace segments and skill summaries. A segment contains the task fragment, relevant tool calls, errors, state transitions, and outcome. For each node pair (u, v), the edge weight combines sparse n-gram overlap, embedding similarity, and error-text overlap:

$$\begin{aligned}w(u,v) =& \alpha\, \text{sparse}(u,v) + \beta\, \text{emb}(u,v) \\ &+ \gamma\, \text{err}(u,v).\end{aligned} \tag{21}$$

Edges below threshold are removed. We then search for small strict cliques with at least two distinct source tasks unless the group is explicitly a skill-revision group. Each accepted group must have an explainable purity signal: shared tools, shared precondition failure, shared argument-binding pattern, or semantically aligned task structure. The refactorer receives only these high-purity

groups, which reduces the risk of merging traces that merely share superficial vocabulary.

The numerical graph parameters are benchmark-level implementation choices rather than the source of the main result. For the BFCL runs reported in this draft, we use $\alpha = 0.45$, $\beta = 0.35$, $\gamma = 0.20$, an internal error-overlap weight of 1.7, and a minimum edge weight of 0.18. In our experiments, small changes to these values do not alter the qualitative conclusion; they mainly affect which candidate groups are sent to the refactorer.

D SkillX Executor Alignment on BFCL

For the BFCL comparison, we additionally evaluated whether SkillX should be reported under our executor for stricter runtime alignment. This variant reuses the same SkillX library and retrieval context but injects it through our BFCL executor. As shown in Table 5, this executor-aligned variant performs worse than the SkillX official-style runner on official validity and average score. We therefore report the stronger SkillX official-style result in the main table, which is conservative with respect to our method.

E Case Studies

We provide the source evidence behind the qualitative studies discussed in Section 4.6. The first study inspects five learned extractor meta-skills. The second study follows one concrete skill across successive maintenance snapshots.

E.1 Meta-skill Evidence

Table 6 lists five extractor meta-skills and the evidence that led to them. The evidence comes from candidate-group feedback in the BFCL run `bfcl_meta_prompt_balanced_trl_50_50_20260523`. These cases show that the learned role guidance is grounded in observed helpful and harmful skill outcomes.

E.2 Skill Evolution Trace

Tables 7 and 8 show two cases where the skill body itself changes. They illustrate two common update types: *narrow trigger* updates that restrict when a workflow applies, and *specify argument* updates that make an API contract exact.

F Algorithm Pseudocode

SkillX variant	official_valid ↑	avg_score ↑	recall ↑	precision ↑	total tok. ↓
Official-style runner	0.6600	0.7689	0.8837	0.7026	74664.9
Through our executor	0.5000	0.7545	0.8364	0.7007	68850.4

Table 5: BFCL executor-alignment check for SkillX. Both rows use the same 50/50 shuffled split and SkillX library; the main table reports the stronger official-style SkillX result.

Meta-skill	Supporting evidence
Extract parameter contracts only when multiple independent traces confirm the pattern; reject stylistic preferences unless they encode API requirements or prevent errors.	An invoice-parameter skill received positive credit when the model omitted an unnecessary `insurance_id`. In contrast, a support-message brevity skill received harmful credit because its guidance likely blocked a required support call for a vague user request.
Do not extract workflow guardrails triggered by sentiment keywords or question patterns; require verifiable state, API preconditions, or cross-turn data dependencies.	Travel support candidates triggered by words such as "urgent" or "problem" produced harmful or unsupported behavior. The feedback indicated that surface wording alone did not define a reusable workflow condition.
Extract workflow guardrails for API preconditions only when at least two independent traces confirm the precondition and show successful application.	Later candidate groups showed that single-trace API-precondition workflows could be context-sensitive. The extractor update therefore required repeated positive evidence before turning such observations into reusable rules.
Do not extract skills that skip verification or status-check steps unless multiple positive traces confirm the optimization is safe.	Vehicle and travel candidates that encouraged skipping checks caused extra calls or wrong arguments when the skipped state was needed to compute a valid action. The learned rule preserves verification unless evidence shows it is safely redundant.
Merge workflow guardrail candidates that describe identical precondition sequences even if phrasing differs slightly; retain the clearer version.	Candidate groups repeatedly contained near-duplicate variants, such as multiple invoice-omission skills or multiple brake-before-start vehicle skills. Competition and credit identified that these variants encoded the same underlying contract.

Table 6: Examples of learned extractor meta-skills and their grounding evidence.

Version	Update	Explanation	Triggering event or update rationale
v1	New refactor skill.	`trading_order_confirmation_workflow`: after `place_order` returns an `order_id`, if the immediately following turn asks to retrieve, check, or confirm order details without a new id, call `get_order_details` with the resolved id.	Refactoring found repeated TradingBot trajectories where an order id produced by `place_order` must be reused in the next turn. Positive evidence includes `multi_turn_base_118`, where the workflow produced the correct order-detail lookup.
v2	Narrow trigger.	The revised body makes the trigger explicit: apply only when `place_order` was called in the immediately prior turn of the current task session and no intervening turn occurred. Do not apply merely because the user mentions a past order.	Harmful evidence from `multi_turn_base_143` showed workflow pollution: the task asked about a remembered past order without a prior `place_order` in the current session. The skill biased the model toward an inapplicable confirmation workflow, so the refiner narrowed the precondition.

Table 7: Skill-body update for `trading_order_confirmation_workflow`.

Version	Update	Explanation	Triggering event or update rationale
v1	New extractor skill.	`vehicle_brake_before_engine_start`: before `startEngine(ignitionMode='START')`, call `pressBrakePedal`. The initial rule captures the brake-before-start ordering but does not emphasize the exact pedal position.	Extracted from VehicleControlAPI evidence where engine start requires brake engagement before ignition. Subsequent positive credit showed the workflow aligned with correct engine-start traces.
v2	Specify argument.	The revised body adds the exact contract: `pedalPosition` must be exactly `1.0`, and the rule should not apply if the brake is already confirmed pressed at `1.0`.	A maintenance test showed the model could press the brake only partially, using `pedalPosition=0.5`. The refiner therefore changed the skill body from an ordering rule into an exact argument contract for the brake step.

Table 8: Skill-body update for `vehicle_brake_before_engine_start`.

Algorithm 1 HiSME Online Skill Meta-Evolving

Input: training tasks $X_{\text{train}} = \{x_1, \ldots, x_T\}$, frozen executor Exe_ϕ, initial library S_0, role rules M_0
Output: active skill library S_T^{active} and role rules M_T

Initialize credit table C, overlap graph G, and role buffers $\mathcal{D}^{\text{ext}}, \mathcal{D}^{\text{rfg}}, \mathcal{D}^{\text{rfn}}$
for $t = 1$ to T **do**
  $\tau_t \leftarrow \textsc{ExecuteWithRetrievedSkills}(\text{Exe}_\phi, x_t, S_{t-1}, K)$
  $U_t \leftarrow \textsc{EvaluateTrace}(x_t, \tau_t)$
  $z_t \leftarrow \textsc{Project}(\tau_t, U_t)$
  $G \leftarrow \textsc{UpdateOverlapGraph}(G, z_t, S_{t-1})$
  $E_t \leftarrow$ skills retrieved, injected, or explicitly used in τ_t
  $Y_t \leftarrow \textsc{AssignCredit}(\tau_t, U_t, E_t)$
  $C \leftarrow \textsc{UpdateCreditTable}(C, Y_t)$
  $G_t^{\text{ext}} \leftarrow \textsc{ExtractCandidates}(\tau_t, S_{t-1}, M_{\text{ext}})$
  $S_t \leftarrow S_{t-1} \cup G_t^{\text{ext}}$
  if $t \bmod k_{\text{micro}} = 0$ **then**
    $S_t \leftarrow \textsc{MicroMaintain}(S_t, C, Y_t, M_{\text{rfn}})$
  end if
  if $t \bmod k_{\text{macro}} = 0$ **then**
    $S_t \leftarrow \textsc{MacroMaintain}(S_t, G, C, M_{\text{rfg}}, M_{\text{rfn}})$
    $(\mathcal{D}^{\text{ext}}, \mathcal{D}^{\text{rfg}}, \mathcal{D}^{\text{rfn}}) \leftarrow \textsc{BuildRoleFeedback}(S_t, C)$
    $M_{\text{ext}} \leftarrow \textsc{UpdateRoleRules}(\text{ext}, M_{\text{ext}}, \mathcal{D}^{\text{ext}})$
    $M_{\text{rfg}} \leftarrow \textsc{UpdateRoleRules}(\text{rfg}, M_{\text{rfg}}, \mathcal{D}^{\text{rfg}})$
    $M_{\text{rfn}} \leftarrow \textsc{UpdateRoleRules}(\text{rfn}, M_{\text{rfn}}, \mathcal{D}^{\text{rfn}})$
  end if
end for
return $(S_T^{\text{active}}, M_T)$

Algorithm 2 Execution with Retrieved Skills

Input: executor Exe_ϕ, task x, skill library S, retrieval size K
Output: trace τ

$\tau \leftarrow \emptyset$
for turn j in x **do**
  $q_j \leftarrow \textsc{BuildRetrievalQuery}(x, j, \tau)$
  $R_j \leftarrow \mathrm{TopK}_{s \in S_{\mathrm{active}}}\, \rho(q_j, s)$
  $p_j \leftarrow \textsc{NativePrompt}(x, j, \tau) \oplus \textsc{RenderSkills}(R_j)$
  $(a_j, e_j) \leftarrow \mathrm{Exe}_\phi(p_j)$
  $\tau \leftarrow \tau \cup \{(j, q_j, R_j, a_j, e_j)\}$
end for
return τ

Algorithm 3 Overlap-Graph Refactoring

Input: graph state G, skill library S, refactorer rules M_{rfg}
Output: updated skill library S'

$V \leftarrow$ trace segments and active or pending skill nodes
for all candidate pairs $(u, v) \in V \times V$ **do**
  $w(u, v) \leftarrow \alpha\, \mathrm{sparse}(u, v) + \beta\, \mathrm{emb}(u, v) + \gamma\, \mathrm{err}(u, v)$
  if $w(u, v) \geq \eta$ **then**
    add weighted edge (u, v) to G
  end if
end for
$\mathcal{C} \leftarrow \textsc{StrictCliqueSearch}(G, C_{\min}, C_{\max})$
for all $C \in \mathcal{C}$ with explainable purity **do**
  $P_C \leftarrow$ clique evidence, selected segments, related skills, and M_{rfg}
  $G_C \leftarrow \textsc{LLMRefactorer}(P_C)$
  for all $s \in G_C$ **do**
    if s is supported by all true instances and $\textsc{Gate}(s)$ passes **then**
      add s to S with source role REFACTORER
    end if
  end for
end for
return S

Algorithm 4 Credit, Bundle Maintenance, and Refinement

Input: trace τ, metrics U, exposed skills E, skill library S
Output: updated skill library S' and credit table C'

$Y \leftarrow \text{LLMCreditAssigner}(\tau, U, E)$
for all $y_{t,s} \in Y$ **do**
  update exposure, usage, helpful, harmful, neutral, and uncertain counts in C_s
  if $y_{t,s}$ implies a new bundle case **then**
    patch $B(s)$ with a minimal positive, negative, or integration fragment
  end if
  if $y_{t,s}$ or bundle failure requires refinement **then**
    $G_{\text{rfn}} \leftarrow \text{LLMRefiner}(s, B(s), C_s, M_{\text{rfn}})$
    for all $s' \in G_{\text{rfn}}$ **do**
      reset $C_{s'}$, record parent$(s') = s$, and run $\text{Gate}(s')$
    end for
    replace, rollback, disable, or archive s according to revision evidence
  end if
end for
for all skills s with sufficient harmful evidence and insufficient helpful evidence **do**
  disable or archive s
end for
return (S, C)

Algorithm 5 Role-Specific Meta-Rule Update

Input: role r, current rules M_r, role feedback buffer $\mathcal{D}^r$
Output: updated role rules M'_r

$\mathcal{E} \leftarrow \{d_s \in \mathcal{D}^r : \text{mature}(s)\}$
$F \leftarrow$ compact evidence from $\mathcal{E}$, including skill signatures, credit, bundle outcomes, lifecycle decisions, and parent–revision comparisons
$R \leftarrow \text{LLMMetaMaintainer}(r, M_r, F)$
if R contains parseable analysis, summary, and rule sections **then**
  $M'_r \leftarrow$ at most five normalized concise rules from R
else
  $M'_r \leftarrow M_r$
end if
return M'_r