RETHINKING AGENT SKILL ORCHESTRATION: DO WE NEED TO COMPILE SKILLS BEFORE EACH EXECUTION?

Anonymous authors
Paper under double-blind review

ABSTRACT

As agent skills become widely adopted, the prevailing practice is to retrieve relevant skills for each task and orchestrate them on the fly. When an agent repeatedly solves tasks of the same type, however, recompilng skills for every instance incurs extra LLM calls and latency, and discards the memory accumulated in earlier executions. Prior work compiles skills into dependency graphs or induces reusable skills from trajectories offline, yet still leaves orchestration to test time, performed anew for each task. We observe that successful executions of the same task type follow a highly consistent composition, so composition can be done once, at induction time. Building on this observation, we propose FlatSkill, which compiles an agent's skill-guided trajectories into one prefab per task type, pairing a slotted, task-level procedure with rules verified against the trajectories. At test time, the agent only selects the matching prefab and fills in its slots. On ALFWorld, ScienceWorld, WebShop and MedCalc-Bench, FlatSkill outperforms existing methods in all evaluation settings with DeepSeek-V3.2, while taking fewer interaction steps than the strongest baseline. Prefabs compiled by a stronger model can also be reused directly by a smaller one, demonstrating the benefit of amortizing skill composition across executions rather than repeating it for every task.

1 INTRODUCTION

Agent skills have been widely applied when solving complex tasks in interactive environments (Wang et al., 2023; 2024; Anthropic, 2025). A skill packages procedural knowledge, such as how to operate a tool or how to complete a recurring subtask, into a reusable unit that the agent loads when it is needed. As the containing of skill library grows to hundreds of thousands of skills (Liang et al., 2026), existing works treat using of skill as an orchestration problem that will arise immediately for every task. Before execution, agent will retrieve candidate skills from the library, select the relevant ones and compile them into some specific forms for the task. Recent methods make this orchestration increasingly elaborate. GraSP compiles the retrieved skills into a dependency graph and repairs it during execution (Xia et al., 2026b), Graph-of-Skills and HiSkill retrieve dependency-aware subgraphs of a skill graph (Liu et al., 2026a; Hao et al., 2026), and other methods decode a skill sequence or synthesize a new skill for each test task (Zhao et al., 2026; Wang et al., 2026).

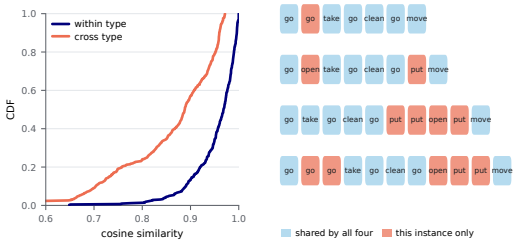


Figure 1: Skill-augmented trajectories share their structure within a task type.

The orchestration of each task is costly when an agent faces many instances of the same kind of task. Every execution needs retrieval, selection and compiling with additional LLM calls and end-to-end latency, although some orchestrations of earlier instances can be reused. The benefit brought by this process is also limited. When skills are evaluated in realistic settings, their gains degrade consis-

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

tently and approach the no-skill baseline in the most challenging scenarios (Liu et al., 2026b). Skills written to transfer across tasks provide too little guidance for any specific task, while skills tailored to one task fail to transfer (Zhang et al., 2026a). The agent therefore repeats similar orchestration steps for every task, and the experience gained in each execution is discarded once the task ends.

We argue that for recurring task types this orchestration does not need to be repeated. Our key observation is that the trajectories of a skill-augmented agent are highly structured within a task type. The left panel of Figure 1 compares each trajectory with its nearest neighbour. Neighbours of the same task type are far more similar than neighbours of other task types, and most same-type pairs exceed a cosine similarity of 0.9. The right panel shows where this similarity comes from. Four successful trajectories of one same task type differ in their exploratory detours and invalid attempts, yet they share one ordered sequence of successful state-changing actions. The composition that test-time orchestration rediscovers for every instance is therefore a property of the task type rather than of the individual task, and it can be extracted once from past executions and reused for all future instances of that type.

Building on this observation, we propose FlatSkill, which moves skill orchestration before test executions. FlatSkill collects the trajectories that a skill-augmented agent produces and groups them by task type. For each type, it reduces the successful trajectories to their state-changing actions, induces a procedure whose task-specific arguments become named slots, and keeps the candidate that best covers the observed action sequences. It then attaches rules that separate successful from failed trajectories, and keeps a rule only when the recorded outcomes support it. The result is a prefab, a task-level procedure that is assembled once and installed as is for every new instance. At test time, FlatSkill routes the task to its type from the task description, fills the slots of the matching prefab with the entities of the task, and hands the prefab to an unmodified ReAct agent (Yao et al., 2022b). No skill retrieval, graph construction or repair takes place during execution, and the whole library remains a flat table with one prefab per task type. Because a prefab is plain text, a prefab compiled by one model can be executed directly by another.

We evaluate FlatSkill on ALFWorld, ScienceWorld, WebShop and MedCalc-Bench (Shridhar et al., 2020; Wang et al., 2022; Yao et al., 2022a; Khandekar et al., 2024) with DeepSeek-V3.2 (Liu et al., 2025), Qwen3-235B and Qwen3-8B (Yang et al., 2025) as backbones. Our contributions are summarized as follows.

- We identify a redundancy in skill-based agents. Per-task orchestration recomputes a composition that is largely shared by all successful trajectories of a task type, and we verify this structure empirically.
- We propose FlatSkill, which compiles skill-augmented trajectories into one prefab per task type and reduces test-time orchestration to selecting a prefab and filling its slots.
- FlatSkill outperforms existing skill-based methods in almost all evaluation settings with DeepSeek-V3.2 and takes fewer interaction steps than the strongest baseline on three of the four benchmarks.
- Prefabs compiled by a stronger model can be reused directly by a smaller one, and controlled ablations show that the gains come mainly from selecting the right prefab, whereas test-time composition, search and repair add little.

2 PRELIMINARY

Agent skills. Agent skills (Anthropic, 2025; Xu & Yan, 2026) package procedural knowledge into modular, file-based units that a general-purpose LLM agent loads on demand. A skill $s_i = (m_i, b_i, \mathcal{R}_i)$ consists of lightweight metadata m_i that states its name and when to use it, an instruction body b_i such as a `SKILL.md` file, and optional resources $\mathcal{R}_i$ such as scripts, templates and reference documents. Given a skill library $\mathcal{S} = \{s_i\}_{i=1}^{N}$, skills are consumed through *progressive disclosure*. The initial context contains only the query and the metadata, $c^1 = [q; m_1, \ldots, m_N]$, and the agent decides at run time which skill to load. A trajectory $\tau = (c^1, a^1, e^1, \ldots, c^T, a^T)$ then unrolls as

$$a^t \sim \pi_\theta(\cdot \mid c^t), \quad c^{t+1} = c^t \oplus a^t \oplus e^t, \quad e^t = \begin{cases} b_i \text{ or } \rho \in \mathcal{R}_i, & \text{if } a^t = \texttt{load}(s_i), \\ \mathrm{Env}(a^t), & \text{otherwise}, \end{cases} \tag{1}$$

where $\oplus$ denotes context concatenation and $\mathrm{Env}(\cdot)$ returns the observation of a tool or environment action. Skills can be written by hand or learned from agent experience (Wang et al., 2023; Zhao et al., 2024; Xia et al., 2026a), yet their benefit depends on quality and selection (Li et al., 2026a).

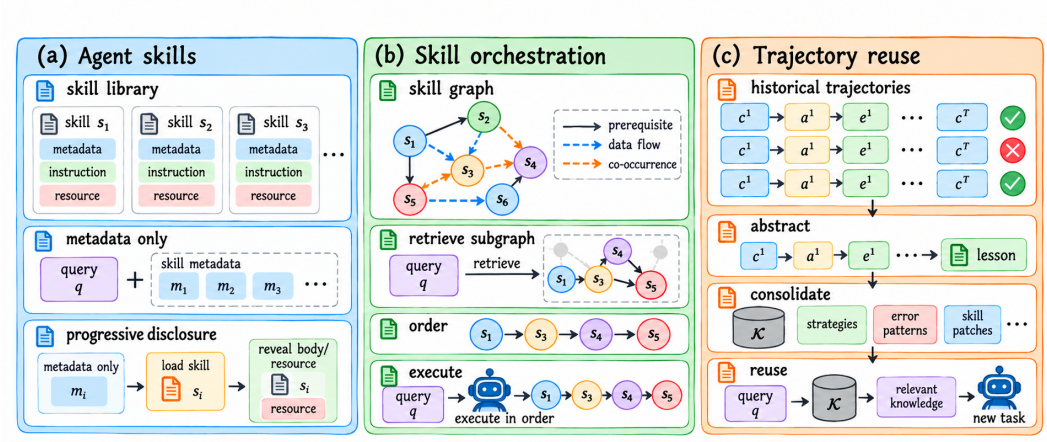


Figure 2: **Three paradigms of skill-based LLM agents.** (a) *Agent skills* expose only skill metadata in context and load instructions or resources on demand. (b) *Skill orchestration* organizes skills into a typed graph, retrieves a task-relevant subgraph for every query and executes it in dependency order. (c) *Trajectory reuse* abstracts successful and failed trajectories into consolidated knowledge, such as skill patches or error schemata, and selects the relevant part for every new query.

Skill orchestration. Skill orchestration methods (Li et al., 2026b; Hao et al., 2026; Xia et al., 2026b; Liu et al., 2026a) organize the library into a skill graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, whose nodes are skills and whose typed edges encode relations such as prerequisite, data flow and co-occurrence, as shown in Figure 2(b). Given a query q, the agent retrieves a task-relevant subgraph, orders it and executes the skills accordingly,

$$\mathcal{G}_q = \mathrm{Retrieve}(\mathcal{G}, q), \qquad (v_1, \ldots, v_K) = \mathrm{Order}(\mathcal{G}_q), \qquad a^t \sim \pi_\theta\big(\cdot \mid c^t, \mathcal{G}_q, v_k\big), \tag{2}$$

where $\mathrm{Order}(\cdot)$ is typically a topological sort and v_k is the currently active skill. These methods differ in how the graph is built and adapted, but all of them recompute $\mathrm{Retrieve}$ and Order for every query.

Trajectory reuse. Another line treats historical trajectories as a source of transferable knowledge (Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024; Yu et al., 2025). Given trajectories $\mathcal{T} = \{\tau_j\}_{j=1}^{M}$ with outcomes $r(\tau_j) \in [0, 1]$, these methods abstract lessons from individual trajectories, consolidate them into a knowledge store $\mathcal{K}$ and select the relevant part for each query, as shown in Figure 2(c),

$$\mathcal{K} = \mathrm{Consolidate}\Big(\big\{\mathrm{Abstract}\big(\tau_j, r(\tau_j)\big)\big\}_{j=1}^{M}\Big), \qquad a^t \sim \pi_\theta\big(\cdot \mid c^t, \mathrm{Select}(\mathcal{K}, q)\big). \tag{3}$$

In both paradigms, the pieces that serve a query are assembled for that query at test time, and the composition that the agent actually realized on a task type is never stored as a unit. FlatSkill stores exactly this composition, once for each task type.

3 FlatSkill

FlatSkill moves skill orchestration from test time to before test time. Figure 3 gives an overview. Given training tasks with task types, FlatSkill first runs a skill-augmented agent and extracts from

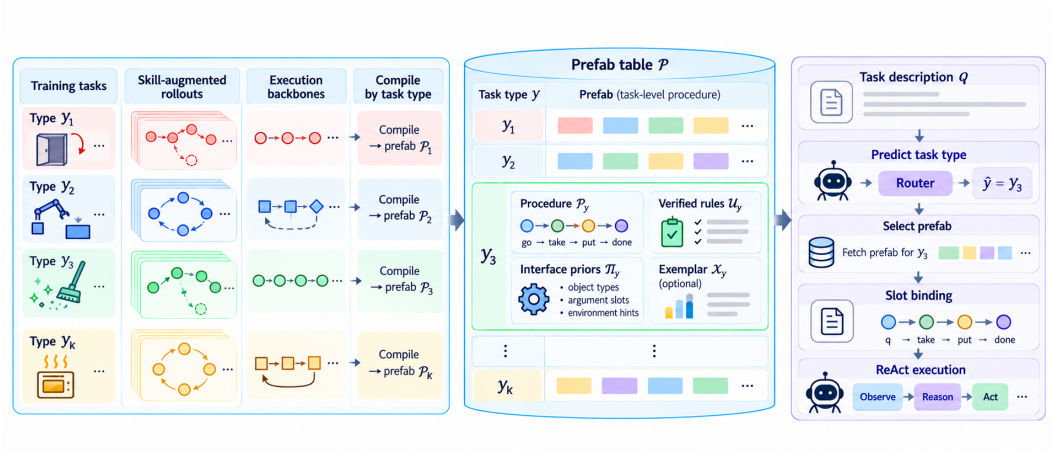


Figure 3: **Overview of FlatSkill. Left.** A skill-augmented agent solves training tasks, and each trajectory is reduced to an execution backbone of state-changing actions. **Middle.** The backbones of one task type are compiled into a prefab that pairs a slotted procedure with outcome-verified rules, interface priors and an optional exemplar. **Right.** At test time, a router selects one prefab, its slots are bound from the task description, and an unmodified ReAct agent executes it.

each trajectory the composition of actions that the agent actually realized (§3.1). Before any test task is seen, it then compiles the trajectories of each task type into a prefab, a slotted task-level procedure accompanied by verified rules and interface statistics, and stores all prefabs in a flat table (§3.2). At test time, the agent selects the prefab of the predicted task type, fills its slots and executes it without any further retrieval or composition (§3.3).

3.1 Skill-Augmented Trajectory Extraction

Collecting trajectories. Let $\mathcal{D} = \{(q_j, y_j)\}_{j=1}^{M}$ be a set of training tasks, where q_j is the task description visible to the agent and $y_j \in \mathcal{Y}$ is its task type. The task type is used only for compilation. It is read from benchmark metadata in our main experiments, and Appendix C.2 shows that clusters of the task descriptions perform comparably. We run the skill-augmented agent of Eq. (1) on every training task and record its trajectory τ_j together with the normalized outcome $r_j = r(\tau_j) \in [0, 1]$ returned by the environment. A trajectory is successful when $r_j \geq \gamma$. When the number of successful trajectories for a certain task type is insufficient, additional rollouts are sampled on its failed tasks and only newly verified successes are added. The pool $\mathcal{T}_y = \{\tau_j \mid y_j = y\}$ keeps both successes and failures, ensuring that information can be obtained from both sides.

Execution backbones. A skill-augmented trajectory mixes steps that load skills, steps that inspect the environment or fail, and steps that change the state of the environment. Only the last kind records the composition that the agent realized after reading its skills, so we reduce every trajectory to an *execution backbone*. Let $\mathcal{A}_{\mathrm{c}}$ be the set of state-changing action types and $\mathcal{A}_{\mathrm{n}}$ the set of navigation action types of an environment, both read from its action grammar. A step a^t is kept if it is valid and $a^t \in \mathcal{A}_{\mathrm{c}}$, or if it is a valid navigation action followed within four steps by a kept state-changing action. Every maximal run of dropped steps is replaced by a single `explore` marker, adjacent duplicates are merged, and trailing markers are removed. We write $\beta(\tau)$ for the resulting backbone and $v(\cdot)$ for the sequence of action types of a backbone or a procedure, and $v_{\mathrm{c}}(\tau)$ for the sequence of valid state-changing action types of a trajectory. Skill-loading steps never enter $\beta(\tau)$, so the knowledge contributed by the loaded skills appears in the backbone only through the actions it caused.

Consistency within a task type. The successful trajectories of one task type are highly consistent, which is the observation behind Figure 1. We quantify it by the agreement

$$\mathrm{Agr}(y) = \frac{2}{|\mathcal{T}_y^+|(|\mathcal{T}_y^+|-1)} \sum_{\tau<\tau'\in\mathcal{T}_y^+} \mathrm{F1}_{\mathrm{LCS}}\big(v_{\mathrm{c}}(\tau), v_{\mathrm{c}}(\tau')\big), \qquad \mathrm{F1}_{\mathrm{LCS}}(u,w) = \frac{2\,|\mathrm{LCS}(u,w)|}{|u|+|w|}, \tag{4}$$

where $\mathcal{T}_y^+ \subseteq \mathcal{T}_y$ is the set of successful trajectories of type y and LCS is the longest common subsequence. A high agreement means that the successful trajectories of a task type follow one ordered composition, which a single procedure can express.

3.2 Prefab Compilation

For every task type y, FlatSkill compiles its trajectories into a prefab

$$\mathcal{P}_y = \big(P_y,\ \mathcal{U}_y,\ \Pi_y,\ X_y\big), \tag{5}$$

where P_y is a slotted procedure, $\mathcal{U}_y$ a set of verified rules, Π_y a set of interface statistics and X_y an optional exemplar. All prefabs form a flat table $\mathcal{L} = \{(y, \mathcal{P}_y)\}_{y\in\mathcal{Y}}$ with one entry per task type and no edges between entries.

Procedure compilation. From the backbones of $\mathcal{T}_y^+$ we take a set $\mathcal{B}_y$ of multi distinct backbones, preferring those with the fewest state-changing actions so that detours which happened to succeed do not enter the procedure. An LLM receives these backbones and the action grammar and writes a numbered procedure with a maximum number of steps. Each step is one action in the exact syntax observed in the backbones, arguments that vary across backbones become named slots such as {`target`}, and a repeated action becomes one step with an observable stopping condition. The LLM writes three candidates, one greedy and two sampled, and we keep the candidate that best covers the observed compositions,

$$P_y = \arg\max_{P\in\{P^{(1)},P^{(2)},P^{(3)}\}} \frac{1}{|\mathcal{B}_y|} \sum_{b\in\mathcal{B}_y} \mathrm{F1}_{\mathrm{LCS}}\big(v(P), v(b)\big). \tag{6}$$

This mechanical selection penalizes a candidate that omits an observed phase or invents actions that no successful trajectory took.

Outcome-verified rules. A procedure describes the common path but not the conditions that separate success from failure. Then a proposer agent reads balanced samples of successful and failed trajectories of type y, together with aggregate statistics of both groups, and writes some imperative rules that are checkable from actions and observations. A separate judge labels a larger balanced sample of trajectories as following, violating or not applicable for each rule. With n_u^+ and n_u^- the numbers of trajectories that follow and violate rule u and $\bar{r}_u^+$ and $\bar{r}_u^-$ their mean outcomes, the rule is kept only if

$$n_u^+ \geq n_{\min}, \qquad n_u^- \geq n_{\min}, \qquad \bar{r}_u^+ - \bar{r}_u^- \geq \Delta_{\min}. \tag{7}$$

Rules proposed for the whole benchmark are verified in the same way and shared by all prefabs.

Interface statistics and exemplar. Π_y records, where the environment exposes locations, where successful trajectories of type y found the objects they used and, for environments with an explicit terminal action, how many steps successful trajectories took before issuing it. The step statistic is added to the prefab only when at least a quarter of the failed trajectories exhausted the step budget without terminating. A prefab whose procedure is weakly supported, with insufficient number of successful trajectories or a coverage below threshold in Eq. (6), additionally carries as X_y the backbone of the successful training task of type y whose description is most similar to the test task.

Verified refinement. The trajectories of the agent can miss a step that the agent never discovered. For task types whose prefab still leaves failures on training tasks, FlatSkill therefore proposes revised procedures from the failed trajectories and the gold labels of the same training tasks. Each candidate is executed by the agent on held-out training tasks of the type, and it replaces the current procedure

only if its verified reward is strictly higher. For the few types whose training trajectories almost never succeed, the procedure is written from the gold labels directly and verified in the same way. The refinement touches only training tasks and happens once, before any test task is seen.

3.3 Inference with Prefabs

Prefab selection. At test time the task type is unknown. FlatSkill fits a router g on the training pairs (q_j, y_j) as a TF-IDF centroid classifier over the task descriptions, and selects the prefab of the predicted type $\hat{y} = g(q)$.

Slot binding. The slots of $P_{\hat{y}}$ are filled by a single LLM call that reads the task description and the role description of every slot. A value is accepted only if it is a span of at most four words copied verbatim from q. Otherwise the slot keeps its role description and the agent resolves it during execution. We write $\mathcal{P}_{\hat{y}}(q)$ for the rendered prefab, which contains the bound procedure, the rules of $\mathcal{U}_{\hat{y}}$ and the shared rules, the statistics of $\Pi_{\hat{y}}$ and the exemplar when present.

Execution. The rendered prefab is placed in the initial context and an unmodified ReAct agent (Yao et al., 2022b) acts as usual,

$$c^1 = \big[q;\ \mathcal{P}_{g(q)}(q)\big], \qquad a^t \sim \pi_\theta(\cdot \mid c^t), \qquad c^{t+1} = c^t \oplus a^t \oplus \mathrm{Env}(a^t). \tag{8}$$

Compared with Eq. (1), no skill is loaded during execution. Compared with Eq. (2) and Eq. (3), the per-query $\mathrm{Retrieve}$, Order and Select are replaced by one classification and one slot-binding call, and no graph is traversed or repaired. The procedure guides rather than constrains the agent, which still reads every observation and may deviate when the environment requires it.

4 Experiments

4.1 Experimental Setup

Benchmarks. We use ALFWorld (Shridhar et al., 2020) for household manipulation, ScienceWorld (Wang et al., 2022) for science experiments, WebShop (Yao et al., 2022a) for product search and MedCalc-Bench (Khandekar et al., 2024) for clinical calculation, which differ in action space, horizon and reliance on external knowledge. We report the official seen and unseen splits of ALFWorld and ScienceWorld and the official test splits of WebShop and MedCalc-Bench. The score R is the success rate on ALFWorld, the average reward on ScienceWorld and WebShop, and the accuracy on MedCalc-Bench, all scaled to 100. S is the average number of interaction steps. Prefabs are compiled from the training splits only.

Baselines. We compare with three groups of methods. ReAct (Yao et al., 2022b) acts without external knowledge. Reflexion (Shinn et al., 2023) and ExpeL (Zhao et al., 2024) reuse experience through verbal reflections and extracted insights. ReAct+Skills appends skill documents retrieved from the SkillNet repository to the ReAct context, SkillNet (Liang et al., 2026) runs its released pipeline that retrieves skills and synthesizes a controller for each task, and GraSP (Xia et al., 2026b) compiles the retrieved skills into a per-task dependency graph with verification and local repair.

Implementation. We use DeepSeek-V3.2 (Liu et al., 2025), Qwen3-235B and Qwen3-8B (Yang et al., 2025) as backbones. Unless stated otherwise, the same model collects trajectories, compiles prefabs and executes them. Skill-augmented trajectories are collected by running ReAct+Skills on the training instances. All methods share the environment harness, the decoding temperature of zero and a budget of 30 interaction steps. Analyses in §4.4 and §4.5 use DeepSeek-V3.2 on selection pools drawn from the training distribution and disjoint from the mining instances, where the run-to-run spread is very low on ALFWorld, ScienceWorld, WebShop and MedCalc-Bench(Appendix C.8).

4.2 Main Results

Table 1 reports the results. **FlatSkill achieves the best score in almost all settings.** Averaged over the six settings, it scores 85.3, 83.3 and 72.3 with DeepSeek-V3.2, Qwen3-235B and Qwen3-8B,

Method	ALFWorld				WebShop		ScienceWorld				MedCalc	
	Seen		Unseen		Seen		Seen		Unseen		Seen	
	R↑	S↓	R↑	S↓	R↑	S↓	R↑	S↓	R↑	S↓	R↑	S↓
DeepSeek V3.2												
ReAct	68.6	19.1	65.7	19.5	29.7	24.8	71.8	17.5	70.0	19.0	76.5	2.9
Reflexion	70.1	18.2	73.6	18.0	35.3	22.4	72.4	16.3	67.8	17.9	85.8	2.8
ExpeL	67.9	18.9	76.1	17.4	29.2	24.0	74.9	16.0	74.1	17.5	80.5	2.4
ReAct + Skills	62.9	20.6	56.0	21.7	36.5	23.0	84.3	15.9	78.9	17.2	91.4	3.1
SkillNet	92.9	12.0	89.6	13.0	44.7	21.5	75.6	15.4	73.2	17.1	82.4	4.1
GraSP	80.6	14.5	83.6	14.8	46.2	17.8	84.9	11.9	81.3	12.5	78.7	4.3
FlatSkill	98.6	9.1	94.0	10.9	61.7	17.4	86.3	13.9	79.4	15.5	91.9	2.8
Qwen3-235B												
ReAct	55.4	20.5	53.8	21.0	28.6	22.4	54.8	19.6	50.6	20.8	66.6	2.5
Reflexion	61.8	19.0	60.2	19.5	32.4	20.6	63.6	17.7	59.4	18.9	76.8	2.5
ExpeL	63.2	18.2	62.4	17.8	30.8	19.2	69.4	15.3	63.8	16.2	69.9	2.3
ReAct + Skills	68.9	17.3	69.4	16.8	37.1	18.7	73.0	14.9	67.6	15.8	89.5	3.6
SkillNet	61.4	17.9	57.5	19.8	23.6	19.6	76.3	15.2	72.5	15.3	74.1	4.3
GraSP	77.9	14.7	82.8	13.9	42.3	18.2	74.5	14.8	69.6	15.8	68.5	4.2
FlatSkill	89.3	10.8	90.3	11.5	60.3	10.5	86.0	14.0	82.0	15.9	91.8	3.0
Qwen3-8B												
ReAct	47.1	22.5	37.3	24.4	15.1	23.2	39.4	19.2	35.9	20.6	48.2	1.9
Reflexion	41.5	21.1	36.4	21.8	17.0	10.3	34.2	19.8	38.3	20.1	59.5	1.9
ExpeL	39.2	18.6	42.9	18.1	23.7	9.7	39.6	17.4	40.5	17.8	53.3	1.8
ReAct + Skills	41.3	24.2	31.8	26.0	20.6	20.0	76.3	14.8	69.1	14.7	85.5	3.2
SkillNet	56.8	19.1	61.2	19.7	16.1	22.0	58.9	15.8	51.9	18.4	72.8	2.0
GraSP	54.2	10.9	59.1	7.7	28.4	8.6	31.4	19.1	32.2	19.6	61.5	3.6
FlatSkill	81.4	12.9	79.9	13.7	38.9	11.6	76.8	15.1	71.3	16.6	85.6	2.0

Table 1: Performance comparison across ALFWorld, WebShop, ScienceWorld, and MedCalc-Bench using different backbone models.

which exceeds the strongest baseline of each backbone by 8.9, 14.0 and 18.2 points and ReAct by 21.6, 31.7 and 35.1 points. The largest margins appear on WebShop and ALFWorld, where the tasks of one type share a long and stable composition. The only exception is the unseen split of ScienceWorld with DeepSeek-V3.2, where GraSP scores 81.3 against 79.4.

Per-task orchestration is inconsistent across benchmarks. SkillNet is the strongest baseline on ALFWorld with DeepSeek-V3.2 but trails ReAct+Skills by 8.7 points on ScienceWorld seen, GraSP is strong on ScienceWorld but falls to 78.7 on MedCalc-Bench, and ReAct+Skills scores below plain ReAct on ALFWorld with DeepSeek-V3.2 and Qwen3-8B. FlatSkill ranks first or second in every setting, because a prefab carries the composition that worked for the task type instead of a composition assembled from generic skills for each task.

Prefabs also shorten execution. FlatSkill takes fewer steps than the strongest baseline of the same setting in 12 of the 18 settings, and 9% fewer steps on average. On ALFWorld the reduction ranges from 16% to 32%. At test time, FlatSkill adds one routing decision and one slot-binding call to a ReAct episode, whereas SkillNet and GraSP retrieve skills and synthesize a controller or a dependency graph for every task.

4.3 Reusing Prefabs across Models and Trajectory Sources

A prefab compiled by a strong model transfers to smaller models. Because a prefab is plain text, we give the prefabs compiled by DeepSeek-V3.2 to the two Qwen3 models without any change. Figure 4(a) shows that Qwen3-8B improves in five of six settings, by 3.4 points on average and by 11.9 points on ALFWorld unseen, which exceeds what it achieves with prefabs compiled from its

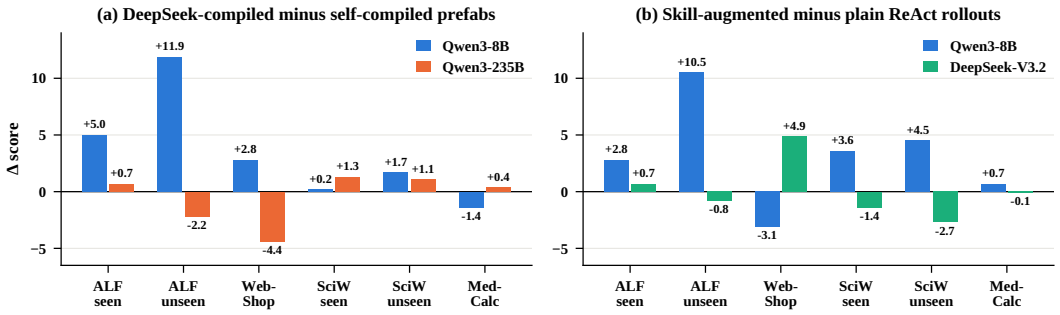


Figure 4: **Reusability of prefabs.** (a) Score of prefabs compiled by DeepSeek-V3.2 minus the score of prefabs compiled by the executing model itself. (b) Score of prefabs compiled from skill-augmented trajectories minus the score of prefabs compiled from plain ReAct trajectories.

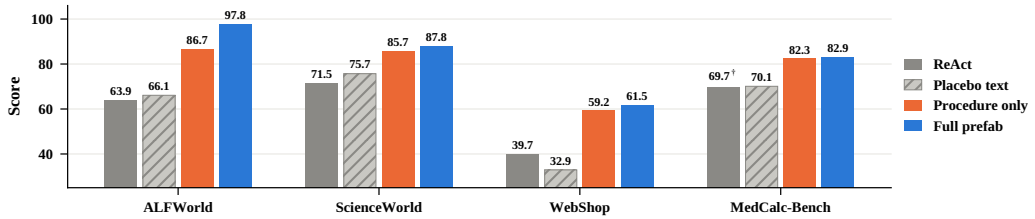


Figure 5: **Content against form** on the selection pools. The placebo matches the length, headings and numbered steps of a prefab but contains no information about the environment. † ReAct on MedCalc-Bench is measured on a separate held-out pool.

own trajectories. Qwen3-235B, whose own prefabs are already strong, changes by -0.5 points on average.

Skill-augmented trajectories exhibit more pronounced performance improvements for weaker models. Figure 4(b) compares prefabs compiled from the trajectories of ReAct+Skills with prefabs compiled from plain ReAct. For Qwen3-8B, skill-augmented trajectories improve almost all settings and the average by 3.2 points, with 10.5 points on ALFWorld unseen. For DeepSeek-V3.2 the two sources differ by 0.1 points on average. Skills help most when the model cannot discover the right composition on its own, and FlatSkill turns this help into a prefab that no longer needs the skills at test time.

4.4 Where Does the Gain Come From?

The gain comes from what a prefab says, not from its format. We replace the prefab with a placebo of the same length, heading structure and number of steps, whose steps give generic advice such as reading the task carefully. Figure 5 shows that the placebo changes ReAct by $+2.2$, $+4.2$ and -6.8 points on ALFWorld, ScienceWorld and WebShop, while the induced procedure alone exceeds the placebo by 20.6, 10.0, 26.3 and 12.2 points on the four benchmarks. The format is thus a small and inconsistent effect, and it even hurts on WebShop, where the real procedure must first recover this loss.

Selecting the right prefab matters more than any text inside it. Table 2 removes or replaces one component at a time. Routing a task to the prefab of another type costs 16.7 and 12.8 points, and serving one global prefab to all types costs 15.5 points on ScienceWorld, where the types differ most. Both losses exceed the combined effect of removing the procedure, the interface priors, the exemplar and the rules. Within the prefab, the procedure and its order matter. Shuffling the steps costs as much as removing them on ScienceWorld, so the prefab conveys an ordered composition

Table 2: **Removing or replacing one component** of the full prefab on the selection pools. Each column reports reward / average environment steps after the named change.

	Full prefab	Wrong type	One global	−Priors	−Procedure	Shuffled order	−Exemplar	−Rules
ALFWorld	**97.8/9.2**	81.1/14.9	92.2/11.9	92.8/11.0	95.0/9.6	95.6/9.7	97.2/9.3	98.3/9.3
ScienceWorld	**87.8/13.1**	75.0/16.8	72.4/17.2	87.2/13.2	85.7/15.4	85.5/13.6	86.7/13.1	88.6/13.5

rather than a bag of useful actions. Interface priors help where objects must be searched for, as in ALFWorld. The exemplar is attached to only 0 of 180 and 4 of 94 instances, and removing the verified rules changes the score within the run-to-run spread, so neither is a source of the gain.

4.5 Scaling the Flat Table

Graph-based methods motivate their structure by the difficulty of large skill sets. We therefore remove task types from the table while keeping the evaluation set fixed, so that tasks of a removed type must borrow the prefab of another type. **The flat table does not degrade as it grows.** Figure 6 shows that the score rises with the number of prefabs, with dips of at most 1.7 points, and peaks with the full table on both benchmarks. On ScienceWorld, whose 24 types differ widely, a single global prefab is no better than ReAct, and the full table adds 15.4 points on top of it. On ALFWorld, whose six types share one skeleton, a single prefab already recovers most of the gain, yet the full table still adds 5.6 points.

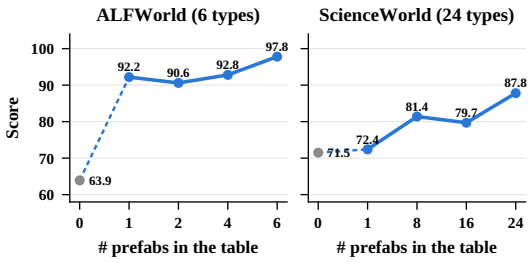


Figure 6: **Size of the prefab table.** Types are removed from the table while the evaluation set is fixed, so tasks of a removed type borrow another prefab. Zero prefabs is ReAct and one prefab is a single global prefab.

4.6 Iterating on Prefab-Augmented Trajectories

We replay the compiled prefabs on the training instances and compile a second round of prefabs from these prefab-augmented trajectories. On ScienceWorld with DeepSeek-V3.2, the second round raises the score from 86.3 to 89.0 on the seen split and from 79.4 to 84.4 on the unseen split. As shown in Figure 7, the second round either refines the existing procedure or replaces it when the original composition is wrong.

5 Conclusion

This paper questions a default of skill-based agents, namely that skills must be retrieved and orchestrated anew for every task. We observed that the trajectories of a skill-augmented agent share one composition within a task type, and we proposed FlatSkill, which compiles these trajectories once into a prefab for each task type and reduces test-time orchestration to selecting a prefab and filling its slots. Across four benchmarks and three backbones, FlatSkill achieves the best score in 17 of 18 settings with fewer interaction steps in most of them, its prefabs transfer from a strong model to a small one, and controlled analyses attribute the gain to selecting the right prefab rather than to the format of the prompt or to the machinery around it. FlatSkill relies on recurring task types whose successful executions agree on one composition. When a type mixes incompatible procedures, or when a long horizon branches in ways that no single procedure covers, per-task orchestration may still earn its cost, and identifying this regime from training trajectories alone is a natural direction for future work.

USE OF LARGE LANGUAGE MODELS

Large language models (LLMs) were used to assist with manuscript preparation, primarily for language polishing, grammatical correction, and improving the clarity and readability of the text. All scientific content, including the research design, methodology, experiments, analysis, and conclusions, was developed and verified by the authors. The authors reviewed and revised all LLM-assisted content and take full responsibility for the final manuscript.

REFERENCES

Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, 7(1):39–59, 1994.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

Marc-Antoine Allard, Arnaud Teinturier, Victor Xing, and Gautier Viaud. Experiential reflective learning for self-improving llm agents. *arXiv preprint arXiv:2603.24639*, 2026.

Anthropic. Equipping agents for the real world with agent skills. Anthropic Engineering Blog, `https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills`, 2025.

Samuel Jacob Chacko, James Hugglestone, Chashi Mahiul Islam, and Xiuwen Liu. When skills don't help: A negative result on procedural knowledge for tool-grounded agents in offensive cybersecurity. *arXiv preprint arXiv:2605.20023*, 2026.

Yuan Chang and Xiaoqi Chen. Naive prompt optimization: Rethinking the need for complex prompt search. *arXiv preprint arXiv:2608.27266*, 2026.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. Automanual: Constructing instruction manuals by llm agents via interactive environmental learning. *Advances in Neural Information Processing Systems*, 37:589–631, 2024.

Seonghyeon Cho and Chanjun Park. Skill following: Evaluating actual skill use in retrieval-enabled llm agents. *arXiv preprint arXiv:2609.00549*, 2026.

Jayanaka Dantanarayana, Savini Kashmira, Lingjia Tang, and Jason Mars. Sigil: Compiling agent skills into typed harnesses. *arXiv preprint arXiv:2607.27309*, 2026.

Gen Dong, Yanjie Gao, Liqun Li, Tianyin Xu, Yu Hua, and Fan Yang. Agent skills can be harmful: An empirical study of skill-induced failures in llm agents. *arXiv preprint arXiv:2608.11888*, 2026.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17490–17502, 2026.

Jinyuan Feng, Dongmin Li, Yiqun Chen, Yang Gao, Xing Chen, Huimu Wang, and Zhiqiang Pu. Coskill: Joint reinforcement learning of reasoning and meta-skill agents for hierarchical skill evolution. *arXiv preprint arXiv:2609.04865*, 2026.

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. Autoguide: Automated generation and selection of context-aware guidelines for large language model agents. *Advances in Neural Information Processing Systems*, 37: 119919–119948, 2024.

Zhongxin Guo, Danrui Qi, Hanwen Gu, Peng Cheng, and Yongqiang Xiong. Skill-disco: Distilling and compiling agent traces into reusable procedural skills. *arXiv preprint arXiv:2606.26669*, 2026.

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

Yu Hao, Jinxuan Cai, Qi Zhang, Yawen Li, Zhiqiang Zhang, Chuan Shi, and Cheng Yang. Hiskill: Empowering llm agents with hierarchical skill graphs. *arXiv preprint arXiv:2607.25853*, 2026.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

Wei-Jung Huang. Do llm-generated skills make better ai data scientists? a component ablation across data-science workflows. *arXiv preprint arXiv:2607.07504*, 2026.

Hamish Ivison, Junjie Oscar Yin, Rulin Shao, Teng Xiao, Nathan Lambert, and Hannaneh Hajishirzi. Tmax: A simple recipe for terminal agents. *arXiv preprint arXiv:2606.23321*, 2026.

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

Zhiyuan Jiang, Fangrui Huang, Hanwen Xing, Xander Wu, Yipeng Gao, Rui Cao, Mengdi Wang, Shilong Liu, and Yijiang Li. Demystifying agent skills: Why they work-until they don't. *arXiv preprint arXiv:2608.14036*, 2026b.

Prathyusha Jwalapuram, Hehai Lin, Chuyuan Li, Fangkai Jiao, Sudong Wang, Yifei Ming, Zixuan Ke, Chengwei Qin, Giuseppe Carenini, and Shafiq Joty. The illusion of multi-agent advantage. *arXiv preprint arXiv:2606.13003*, 2026.

Nikhil Khandekar, Qiao Jin, Guangzhi Xiong, Soren Dunn, Serina S Applebaum, Zain Anwar, Maame Sarfo-Gyamfi, Conrad W Safranek, Abid A Anwar, Andrew Zhang, et al. Medcalc-bench: Evaluating large language models for medical calculations. *Advances in Neural Information Processing Systems*, 37:84730–84745, 2024.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Ishant Kohar and Aswanth Krishnan. A benchmark for procedural memory retrieval in language agents. *arXiv preprint arXiv:2511.21730*, 2025.

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

Xiaoxiao Li. When single-agent with skills replace multi-agent systems and when they fail. *arXiv preprint arXiv:2601.04748*, 2026.

Xiaoyuan Li, Moxin Li, Keqin Bao, Yubo Ma, Wenjie Wang, Dayiheng Liu, and Fuli Feng. Skillgraph: Skill-augmented reinforcement learning for agents via evolving skill graphs. *arXiv preprint arXiv:2605.12039*, 2026b.

Zongxia Li, Zhongzhi Li, Yucheng Shi, Ruhan Wang, Junyao Yang, Zhichao Liu, Xiyang Wu, Anhao Li, Yue Yu, Ninghao Liu, et al. Long-horizon-terminal-bench: Testing the limits of agents on long-horizon terminal tasks with dense reward-based grading. *arXiv preprint arXiv:2607.08964*, 2026c.

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

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Dawei Liu, Zongxia Li, Hongyang Du, Xiyang Wu, Shihang Gui, Yongbei Kuang, and Lichao Sun. Graph-of-skills: Dependency-aware structural retrieval for massive agent skills. *arXiv preprint arXiv:2604.05333*, 2026a.

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

Renze Lou, Baolin Peng, Wenlin Yao, Qianhui Wu, Hao Cheng, Suman Nath, Wenpeng Yin, and Jianfeng Gao. The tool illusion: Rethinking tool use in web agents. *arXiv preprint arXiv:2604.03465*, 2026.

Yongliang Miao, Ziyang Yu, Liang Zhao, Bowen Zhu, and Hasibul Haque. Skilllens: Adaptive multi-granularity skill reuse for cost-efficient llm agents. *arXiv preprint arXiv:2605.08386*, 2026.

Kolby Nottingham, Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Sameer Singh, Peter Clark, and Roy Fox. Skill set optimization: Reinforcing language model behavior via transferable skills. *arXiv preprint arXiv:2402.03244*, 2024.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, 2024.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026.

Vishnu Sarukkai, Zhiqiang Xie, and Kayvon Fatahalian. Self-generated in-context examples improve llm agents for sequential decision-making tasks. *Advances in Neural Information Processing Systems*, 38:64392–64425, 2026.

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

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

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

Theodore R Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L Griffiths. Cognitive architectures for language agents. *arXiv preprint arXiv:2309.02427*, 2023.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, 2026.

Dat Tran and Douwe Kiela. Single-agent llms outperform multi-agent systems on multi-hop reasoning under equal thinking token budgets. *arXiv preprint arXiv:2604.02460*, 2026.

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

Jingxing Wang, Chenyu Zhou, Zhihui Fu, Jun Wang, Weiwen Liu, Weinan Zhang, and Jianghao Lin. Skills on the fly: Test-time adaptive skill synthesis for llm agents. *arXiv preprint arXiv:2605.16986*, 2026.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. Scienceworld: Is your agent smarter than a 5th grader? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11298, 2022.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. *arXiv preprint arXiv:2504.06821*, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

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

Tianle Xia, Lingxiang Hu, Yiding Sun, Ming Xu, Lan Xu, Siying Wang, Wei Xu, and Jie Jiang. Grasp: Graph-structured skill compositions for llm agents. *arXiv preprint arXiv:2604.17870*, 2026b.

Yiming Xiong, Shengran Hu, and Jeff Clune. Learning to continually learn via meta-learning agentic memory designs. *arXiv preprint arXiv:2602.07755*, 2026.

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

Xiaonan Xu and Wenjing Wu. Skill availability and presentation granularity in large-language-model agents: A controlled skillsbench study. *arXiv preprint arXiv:2605.31408*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024.

Ziyue Yang and Fan Ding. Signal or noise? a benchmark study of agent skills in web development. *arXiv preprint arXiv:2608.23067*, 2026.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022b.

Yifan Yu, Moyan Li, Shaoyuan Xu, Jinmiao Fu, Xinhai Hou, Fan Lai, and Bryan Wang. Correct: Condensed error recognition via knowledge transfer in multi-agent systems. *arXiv preprint arXiv:2509.24088*, 2025.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative ai by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025.

Qi Zhang, Zhaopeng Feng, Xiaonan Shi, Xiaomeng Hu, Chu Liu, Pengjun Xie, Xiaobin Wang, Jieping Ye, Bryan Hooi, Haobo Wang, et al. Skillcomposer: Learning to evolve agent skills for specification and generalization. *arXiv preprint arXiv:2606.06079*, 2026a.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026b.

Xing Zhang, Guanghui Wang, Yanwei Cui, Wei Qiu, Ziyuan Li, Bing Zhu, and Peiyang He. Prompt optimization is a coin flip: Diagnosing when it helps in compound ai systems. *arXiv preprint arXiv:2604.14585*, 2026c.

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

Xinyu Zhao, Zhen Tan, Vaishnav Tadiparthi, Nakul Agarwal, Kwonjoon Lee, Ehsan Moradi Pari, Hossein Nourkhiz Mahjoub, and Tianlong Chen. Generative skill composition for llm agents. *arXiv preprint arXiv:2606.32025*, 2026.

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

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

A Related Work

Skill libraries and where composition happens. Early skill libraries induce reusable procedures from an agent's own experience and expose them for reuse: Voyager accumulates executable code skills in Minecraft (Wang et al., 2023); ASI and SkillWeaver induce verified programmatic skills for web agents (Wang et al., 2025; Zheng et al., 2025); Skill Set Optimization extracts text skills from high-reward sub-trajectories (Nottingham et al., 2024); and Agent Workflow Memory (AWM) induces text workflows offline or online and places all workflows induced for a website into the agent's memory (Wang et al., 2024). Agent Skills packages procedures as `SKILL.md` folders from which the model selects by description (Anthropic, 2025); see Jiang et al. (2026a); Xu & Yan (2026) for surveys. A recent line of work moves the assembly of skills to test time. GraSP retrieves skills for each task, compiles them into a DAG with typed precondition–effect edges, verifies each node during execution and repairs failures locally (Xia et al., 2026b); Graph of Skills and HiSkill retrieve a dependency-aware sub-graph per task (Liu et al., 2026a; Hao et al., 2026); SkillLens mixes skill granularities and lets a verifier accept, decompose or rewrite each piece per task (Miao et al., 2026); SkillComposer decodes which skills to use and in what order (Zhao et al., 2026); and SkillTTA writes a fresh skill for every test task from retrieved training trajectories (Wang et al., 2026). SkillNet provides the large curated repository such pipelines draw from (Liang et al., 2026), and another line trains the skill library jointly with the policy by reinforcement learning (Xia et al., 2026a; Li et al., 2026b; Shi et al., 2026; Feng et al., 2026). Concurrently with our work, SkillDisCo distills parameterized control-flow sub-graphs from successful traces offline and compiles them into callable, verified Python skills (Guo et al., 2026), and SIGIL compiles prose skills into typed harnesses to enforce the prescribed steps (Dantanarayana et al., 2026). These methods differ in representation (text, code, graphs) but share a premise: skills are finer than tasks, so something (a planner, a graph walk, or the agent's own tool calls) must assemble them for every new task. We test that premise directly. Our skills are task-level, induced once per task type from the agent's own rollouts, and the only test-time decision is which one to use; we then measure what composition, structure and repair add on top of that choice.

Empirical studies of agent skills. Controlled evaluations increasingly find that skill gains are smaller, more conditional and harder to attribute than method papers report. SkillsBench finds that curated skills help but self-generated skills provide no benefit on average, and that focused bundles of at most three modules beat exhaustive ones (Li et al., 2026a); in software engineering, 39 of 49 public skills yield no improvement (Han et al., 2026), and seemingly relevant skills can make

agents omit required steps or over-verify (Dong et al., 2026). Retrieval is a recurring bottleneck: the precision of the skills an agent actually uses falls from 29.6% to 3.3% as the pool grows from 5 to 100 skills (Jiang et al., 2026b), selection accuracy collapses beyond a critical library size (Li, 2026), and aggregate gains can hide negative effects on exactly the tasks where a skill was retrieved (Cho & Park, 2026). Jiang et al. (2026b) also attribute most successful skill use to procedural anchoring rather than knowledge injection. Two studies used a length-matched placebo, the control we adopt: in data science, full skills are indistinguishable from their ablations and from task-irrelevant skill-formatted text (Huang, 2026), and in web development an equally long irrelevant skill reproduces most of the loss that skills cause for some models (Yang & Ding, 2026). It has also been hypothesized that strict, schema-validated environment feedback makes skills redundant (Chacko et al., 2026), and presentation granularity has been found not to matter once skills are available (Xu & Wu, 2026). Our analysis is consistent with the retrieval and anchoring findings, and it explains two apparent conflicts with our results. First, self-induced skills help in our setting although self-generated ones do not in SkillsBench, because ours are induced from the agent's rollouts; a procedure the model writes from its own knowledge, without rollouts, costs 8.9 and 5.1 points in our experiments. Second, the placebo effect depends on the benchmark, ranging from positive to clearly harmful, and across 84 task types the prefab's gain grows with observation length rather than shrinking.

Experience reuse and procedural memory. In the terms of CoALA, our skills are procedural memory (Sumers et al., 2023), and selecting them resembles the retrieve step of case-based reasoning (Aamodt & Plaza, 1994). Prior work reuses experience in several forms: verbal reflections for retrying the same task (Shinn et al., 2023); natural-language insights contrasted from successes and failures, together with retrieved trajectories (Zhao et al., 2024; Fu et al., 2024); trajectories retrieved as exemplars (Zheng et al., 2024; Sarukkai et al., 2026); manuals and rule systems (Chen et al., 2024); procedural scripts (Fang et al., 2026); distilled reasoning strategies (Ouyang et al., 2026); adaptive cheatsheets (Suzgun et al., 2026); and evolving context "playbooks" edited by a reflector and a curator (Zhang et al., 2026b). Agentic Plan Caching matches new requests to cached plan templates and adapts them with a small model (Zhang et al., 2026b), and CORRECT reuses distilled error schemata to localize failures in multi-agent systems (Yu et al., 2025). Whether abstraction beats raw experience remains contested. Accumulating an agent's own successful trajectories alone raises ALFWorld success from 73% to 89% (Sarukkai et al., 2026), and plain trajectory retrieval outperforms several abstraction-based memories in a meta-learned comparison (Xiong et al., 2026). Distilled strategies and heuristics, however, have been reported to beat raw trajectories (Ouyang et al., 2026; Allard et al., 2026) and to transfer better across vocabularies (Kohar & Krishnan, 2025). At a matched context budget we find abstraction and retrieval indistinguishable on ScienceWorld and retrieval ahead on ALFWorld. What both need is the right match: routing a task to another type's prefab costs more than removing its induced procedure. Relative to this literature, our contribution is not a new memory format but an account of which of these mechanisms carry the gain.

Searching and optimizing prompts and workflows. A large body of work searches over prompts, demonstrations or agent designs with textual feedback: OPRO, DSPy and MIPRO, TextGrad and GEPA optimize instructions and demonstrations (Yang et al., 2024; Khattab et al., 2023; Opsahl-Ong et al., 2024; Yuksekgonul et al., 2025; Agrawal et al., 2026); AFlow runs Monte Carlo tree search over code-represented workflows (Zhang et al., 2025); and ADAS searches over agent designs with a meta-agent (Hu et al., 2025). Recent diagnostics find that many optimization runs fall below the zero-shot prompt (Zhang et al., 2026c) and that a single revision loop can match complex prompt search (Chang & Chen, 2026). We reach the same conclusion for skills with the search budget held fixed: shortest skeleton, one-shot induction, iterative refinement, best-of-N and MCTS land within the run-to-run noise, the value proxy that the search maximizes does not predict the size of real improvements, and textual-gradient patches of a prefab leave both benchmarks unchanged.

Less is more in agent design. Our study belongs to a growing set of equal-budget re-examinations of agent complexity. A three-phase pipeline without an agent loop was competitive with software-engineering agents (Xia et al., 2024); automatically designed multi-agent systems underperform self-consistent chain-of-thought at up to ten times the cost (Jwalapuram et al., 2026); single agents match multi-agent systems once thinking tokens are equalized (Tran & Kiela, 2026); and LLM-synthesized tools help web agents only when the tool user is weaker than the tool author (Lou et al.,

Table 3: Constants used for prefab compilation and inference.

Constant	Value
Success threshold γ	0.75
Minimum successes before verified resampling	3
Navigation window for backbone extraction	4 steps
Backbones shown for procedure induction	at most 8
Procedure candidates	3
Sampling temperature for non-greedy candidates	0.7
Maximum procedure length	14 steps
Rules proposed per task type	at most 6
Minimum rule support on each side	3
Minimum outcome gap for rule acceptance	0.20
Termination statistic trigger	at least 25% of failures
Exemplar attachment criterion	≤ 5 successes or coverage < 0.6
Maximum slot value length	4 words

2026). We ask the analogous question for skill orchestration, and report every comparison against its benchmark's run-to-run noise.

B Implementation Details

Models. We use three agent models: DeepSeek-V3.2, Qwen3-235B-A22B-Instruct-2507, and Qwen3-8B. All models are served locally through OpenAI-compatible endpoints.

Decoding. All methods within the same comparison use the same decoding configuration: temperature 0, top-p 1, and seed 0. The maximum generation budget is 32,768 tokens. Mining and slot binding use the same default decoding policy. For procedure induction, the candidate set contains one greedy generation and additional sampled generations produced with temperature 0.7.

Episode limits. ALFWorld, ScienceWorld, and WebShop use a maximum of 30 environment interactions per episode, while MedCalc-Bench uses 15. For terminal-based tasks, we allow up to 60 interaction steps, with a 120-second timeout for each command. When an episode reaches its interaction limit, we use the reward reported by the environment at that point without retrying the episode. Crashed episodes are counted as failures.

Prefab compilation. Table 3 lists the constants used during offline prefab compilation and online slot binding. The core prompts are reproduced in Figures 8, 9, 10, and 11.

Compute. All experiments are conducted on a server equipped with eight NVIDIA H200 GPUs.

C Additional Results

Iterating on Prefab-Augmented Trajectories

We replay the compiled prefabs on the training instances and compile a second round of prefabs from these prefab-augmented trajectories. On ScienceWorld with DeepSeek-V3.2, the second round raises the score from 86.3 to 89.0 on the seen split and from 79.4 to 84.4 on the unseen split, which also exceeds the 81.3 achieved by GraSP.

Figure 7 aligns the first- and second-round procedures for the ten task types with the largest gains. In six types, the alignment retains a clear diagonal: the second round preserves the overall order while revising, inserting, or merging individual steps. In three types, few steps align, indicating that the second round replaces the original procedure with a different composition. Thus, recompilation can improve a prefab in two ways: refining a useful execution skeleton or re-inducing a procedure whose original composition was wrong. Both updates are made from training trajectories before test time.

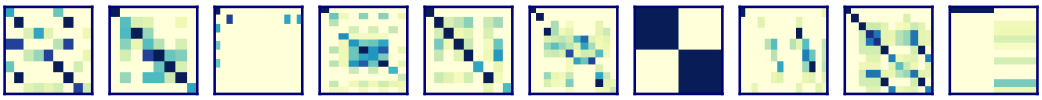


Figure 7: **Procedures of the first and second round** for the ten ScienceWorld task types with the largest gains. Rows are first-round steps, columns are second-round steps, and a darker cell means a larger word overlap.

Table 4: The software-engineering subset of LHTB. Each task is executed in its own container and evaluated by its task-specific verifier.

Task	Description
alp-paper-reproduction	Reproduce ALP-style lossless floating-point compression, including sample-based parameter selection, frame-of-reference encoding, exception handling, an ALP-RD fallback, and exact floating-point round-trip checks.
commit0-multilib-tdd	Reimplement TinyDB, sortedcontainers, and cachetools from projects whose function bodies have been replaced with stubs. The agent writes its own tests from the documentation and is evaluated with hidden tests.
foldseek-paper-reproduction	Reproduce a Foldseek-style protein-structure search pipeline, including PDB parsing, a 3Di-like structural alphabet, k-mer prefiltering, local alignment, and structure-based ranking.
great-expectations-audit	Build a multi-table data-quality pipeline with Great Expectations to clean and validate customer, order, payment, and refund records, quarantine anomalies, and reconcile financial totals.
langchain-version-migration	Migrate a legacy LangChain application to `langchain==1.3.4` while preserving its CLI, public Python API, historical-session replay, multi-turn memory, and dynamic routing.
riscv-core-debug	Diagnose and repair injected defects in an out-of-order RV32GCB SystemVerilog processor using targeted ISA tests, architectural tests, co-simulation, and formal verification.
unison-paper-reproduction	Reproduce a UNISON-style parallel network simulator with fine-grained logical-process partitioning, load-adaptive scheduling, and deterministic event execution, and validate it on fat-tree workloads.

C.1 Evaluation on Long-Horizon Terminal Tasks

Tasks and evaluation. To examine whether FlatSkill extends to more complex terminal workflows, we evaluate it on seven software-engineering tasks selected from Long-Horizon Terminal Benchmark (LHTB) (Li et al., 2026c). The tasks cover paper reproduction, library implementation, data auditing, dependency migration, and hardware debugging; Table 4 summarizes each task. Every task runs in an isolated Linux container and is graded by its task-specific verifier. We report reward as the percentage of verifier checks passed, macro-averaged with equal weight across the seven tasks. All methods use `deepseek-v32` and are allowed at most 60 environment interactions per task.

Skill library construction. The reference skill library for this terminal evaluation is derived from the software-engineering skill taxonomy in TMax-15K (Ivison et al., 2026). For each of its seven skill types, we provide the type name and its list of primitive skills to a language model and synthesize one Markdown skill document. The resulting library therefore contains seven documents, each organizing the primitives of one type into procedural guidance for the agent. This construction reads only the TMax-15K taxonomy; it does not use LHTB task descriptions, LHTB trajectories, or LHTB evaluation results.

Results. Table 5 reports macro-averaged reward and mean environment interactions. FlatSkill obtains the highest reward, 35.44, which is 2.60 percentage points above GraSP and 9.74 points above ReAct. Its mean interaction count is 52.7, compared with 58.3 for GraSP. These results suggest that a task-type prefab can remain useful in longer terminal workflows, although the absolute

Table 5: **Results on seven selected LHTB tasks.** Each entry reports macro-average reward (%) / mean environment steps. All methods use `deepseek-v32` with a 60-step limit.

Method	ReAct	Reflexion	ExpeL	ReAct + Skills	SkillNet	GraSP	FlatSkill
Reward / steps	25.7/47.9	24.53/57.1	20.87/49.7	30.40/48.3	31.99/55.7	32.84/58.3	**35.44**/52.7

reward also shows that these tasks remain difficult. Interaction counts alone should not be interpreted as execution efficiency: failed or prematurely terminated trajectories may also contain fewer steps.

C.2 Predefined Task Types versus Clustering

The main experiments use task types predefined by each benchmark. We examine whether these types are necessary by instead clustering the training task descriptions and treating each cluster as a task type. We then compile one prefab per cluster using the same procedure as for the predefined types. For a controlled comparison, neither arm in Table 6 uses verified refinement; consequently, the ScienceWorld result for predefined types differs from that of the full method in the main experiments. Clustering preserves nearly all of the performance obtained with predefined types: the score changes from 97.8 to 95.0 on ALFWorld and from 84.6 to 85.3 on ScienceWorld. Both cluster-based results also exceed the corresponding single-global-prefab baselines. These results suggest that FlatSkill can form useful groups directly from training task descriptions when task types are not provided.

Table 6: **Predefined task types versus clusters of training task descriptions.** Parentheses indicate the number of clusters. Neither prefab variant uses verified refinement.

	Predefined types	Clusters	One global prefab	ReAct
ALFWorld	97.8	95.0 (12)	92.2	63.9
ScienceWorld	84.6	85.3 (24)	72.4	71.5

C.3 Prefabs against Retrieved Raw Trajectories

A simple alternative to compiling a prefab is to retrieve successful trajectories and place them directly in the agent's prompt. We retrieve the top eight trajectories for each evaluation task, matching the number of trajectories during prefab compilation. Retrieval is performed either within the routed task type or across the full training pool. For comparison, "compiled procedure only" provides the induced sequence of steps without the other prefab components.

On ALFWorld, both retrieval variants outperform the compiled procedure alone: reward rises from 86.7 to 90.0 with within-type retrieval and to 92.2 with global retrieval. On ScienceWorld, within-type retrieval scores 86.8, compared with 85.7 for the procedure, while global retrieval also scores 85.7. The full prefab achieves the highest reward on both benchmarks, reaching 97.8 on ALFWorld and 87.8 on ScienceWorld. These results show that raw successful trajectories are a strong source of guidance. They also show that the complete prefab performs better than either the procedure alone or the top-eight retrieval baselines.

Table 7: Retrieved raw trajectories against the compiled procedure. Tokens are the length of the injected text on ALFWorld.

Injected into the prompt	ALFWorld		ScienceWorld	
	R↑	S↓	R↑	S↓
Nothing (ReAct)	63.9	19.5	71.5	18.1
Compiled procedure only	86.7	14.0	85.7	14.7
8 trajectories of the routed type	90.0	13.0	86.8	15.2
8 trajectories retrieved globally	92.2	12.3	85.7	14.4
Full prefab	**97.8**	**9.2**	**87.8**	**13.1**

C.4 Searching over Prefabs

If test time orchestration is replaced by compilation, one question is whether a more expensive search during compilation yields better prefabs. We compare five ways of producing the procedure for each task type. Table 8 shows no consistent gain in reward or reduction in interaction steps from additional search. On ALFWorld, MCTS reaches the highest reward of 97.8, but the one shot procedure uses fewer steps. On ScienceWorld, one shot compilation achieves the highest reward of 86.1, while the shortest successful backbone uses the fewest steps. The differences in reward are small relative to the variation between runs. In our experiments, MCTS usually expands along a single chain because many candidates receive the maximum verification reward, leaving little signal for choosing among branches.

Table 8: Reward and mean interaction steps for different ways of producing procedures.

	ALFWorld		ScienceWorld	
Procedure produced by	R ↑	S ↓	R ↑	S ↓
Shortest successful backbone	96.7	9.4	85.5	12.9
One shot compilation	97.2	9.0	86.1	13.3
Iterative refinement	96.7	9.0	85.9	13.6
Best of N with verified value	97.2	9.1	84.8	13.6
MCTS	97.8	9.4	85.6	13.4

C.5 Sensitivity to Compilation Hyperparameters

We change one compilation hyperparameter at a time on the ScienceWorld selection pool, namely the success threshold γ, the number of backbones shown to the LLM and the order in which backbones are chosen.

Table 9: Sensitivity to compilation hyperparameters on the ScienceWorld selection pool.

Setting	ScienceWorld
Default ($\gamma = 0.75$, 8 backbones, fewest actions first)	87.8
Success threshold $\gamma = 0.5$	88.3
16 backbones shown to the LLM	88.3
Highest-reward backbones first	87.7

C.6 Robustness of Prefab Selection

FlatSkill selects a prefab with a router fitted on training task descriptions. We test two stresses. First, every evaluation task description is rewritten by an LLM that keeps the entities verbatim, and only the router reads the rewritten text. Second, the task types of two benchmarks are merged into one table, and we measure how often a task is routed into a different benchmark.

Table 10: Prefab selection under rewritten task descriptions and a merged table of all benchmarks.

	Same type after rewriting	Score before	Score after	Routed to another benchmark
ALFWorld	163/180 (90.6%)	97.8	96.1	0.0%
ScienceWorld	93/94 (98.9%)	87.8	87.1	0.0%

C.7 Replacing Components with Alternatives

Table 2 removes components. Here each component is replaced by an alternative instead, which separates the cost of missing a component from the cost of a wrong one. The procedure is replaced by one written by the model from its own knowledge without any trajectory, or by the procedure before verified refinement, the router is replaced by the oracle task type, and the exemplar is replaced by a random successful trajectory of the same type.

Table 11: Replacing one component of the full prefab on the selection pools.

Substitution	ALFWorld	ScienceWorld
Full prefab	97.8	87.8
Router → oracle task type	96.7	86.4
Procedure → written from model knowledge	88.9	82.7
Procedure → before verified refinement	97.8	84.6
Exemplar → random same-type trajectory	96.1	86.7

C.8 Run-to-Run Variance

To assess variation between runs, we evaluate the same configuration twice with temperature zero. Table 12 reports the absolute difference in reward between the two runs. Across the tested configurations, the difference is 1.1–1.6 points on ALFWorld, 0.0–1.1 points on ScienceWorld, 0.2 points on WebShop, and 0.4–0.6 points on MedCalc-Bench. We use these observed differences as a reference when interpreting small changes in the analysis experiments.

Table 12: Absolute difference in reward between two runs of the same configuration with temperature zero. Ranges summarize the tested configurations.

Benchmark	Reward type	Difference in reward
ALFWorld	Binary success	1.1–1.6
ScienceWorld	Partial credit	0.0–1.1
WebShop	Partial credit	0.2
MedCalc-Bench	Binary correctness	0.4–0.6

C.9 Offline Compilation Cost and Amortization

FlatSkill incurs a one-time cost to compile training trajectories into a reusable prefab table, while ReAct+Skills loads skills during each execution. We measure this tradeoff on ScienceWorld using the complete table of 24 task types and all 211 evaluation episodes. Input and output tokens are counted with the same method.

At test time, FlatSkill uses an average of 32.6k input tokens and 0.7k output tokens per episode, for a total of 33.3k tokens. ReAct+Skills uses 98.1k input tokens and 1.1k output tokens, for a total of 99.2k tokens. FlatSkill therefore uses 65.9k fewer tokens per episode in this comparison. Compiling the complete ScienceWorld prefab table requires 205 offline LLM calls, comprising 220.5k input tokens and 20.6k output tokens, or 241.1k tokens in total.

Let C_{off} denote the one-time cost of compiling the complete table, and let C_{FS} and C_{base} denote the mean online token costs per episode of FlatSkill and ReAct+Skills. After N episodes, the amortized token cost per episode of FlatSkill is

$$C_{\text{amort}}(N) = C_{\text{FS}} + \frac{C_{\text{off}}}{N}. \tag{9}$$

The compilation cost is recovered when the cumulative online savings exceed C_{off}. For ScienceWorld, the break-even point is

$$N^* = \frac{C_{\text{off}}}{C_{\text{base}} - C_{\text{FS}}} = \frac{241.1}{99.2 - 33.3} = 3.66. \tag{10}$$

Thus, even when the cost of compiling all 24 task types is charged before any evaluation episode, FlatSkill recovers that cost during the fourth episode. Across all 211 episodes, its total estimated token use, including compilation, is 7.27 million tokens, compared with 20.93 million tokens for ReAct+Skills.

D Algorithm

The pipeline reads mining rollouts consisting of task text, task labels, rewards, actions, and observations, and compiles a flat table that maps each task family to a prefab. No benchmark-specific

Algorithm 1 Mining the flat prefab table (offline)

Require: mining rollouts R, success threshold γ, procedure candidate set size N_{cand}, backbone sampling budget B

Verified resampling: for each task family with insufficient successful rollouts, re-run the base agent on failed mining instances and retain only newly verified successes according to the training reward
Router: fit a TF-IDF centroid classifier from agent-visible task descriptions to task-family labels
for all task-family labels ℓ **do**
    Execution backbones: reduce each rollout of ℓ to its valid state-changing actions; retain navigation only when it leads to a retained state-changing action, and collapse other stretches into an exploration marker
    Procedure induction: select up to B successful rollouts with short execution backbones, generate N_{cand} slotted procedure candidates with the LLM, and retain the candidate with the highest mean LCS-F1 against the observed backbones
    Outcome-verified rules: propose checkable rules from balanced successful and failed rollouts and their outcome statistics; use an independent LLM judge to label additional rollouts as following, violating, or not applicable, and retain only rules satisfying the required support and outcome-margin criteria
    Interface statistics: estimate environment-exposed priors and interaction statistics from successful rollouts when such information is available through the adapter
    Optional exemplar: when the induced procedure has insufficient successful support or insufficient backbone coverage, attach a representative successful training rollout of the task family
    assemble the initial prefab $T[\ell] \leftarrow (\text{procedure}, \text{rules}, \text{priors}, \text{example})$
end for
Label-verified refinement: for task families whose prefabs still fail on training tasks, propose revised procedures from failed rollouts and their training labels; replay each revision on held-out training instances and replace the current procedure only when its verified reward improves
return flat table $T : \ell \mapsto (\text{procedure}, \text{rules}, \text{priors}, \text{example})$

Algorithm 2 Prefab hydration and execution (online)

Require: task description q, flat table T, base system prompt P

$\ell \leftarrow \textsc{Route}(q)$ ▷ TF-IDF nearest-centroid lookup; no LLM invocation
$\sigma \leftarrow \textsc{BindSlots}(T[\ell].\text{procedure}, T[\ell].\text{slots}, q)$ ▷ bind slots using values copied from the task description
$C \leftarrow \textsc{Render}(T[\ell].\text{procedure}, \sigma, T[\ell].\text{priors}, T[\ell].\text{rules}, T[\ell].\text{example})$
run the unmodified ReAct loop with initial context $P \parallel C$ until the environment terminates or the interaction budget is exhausted

procedure is written by hand beyond the adapter interface, which provides rollout loading, the environment action grammar, reward normalization, action validity tests, and environment-specific interface statistics. All numerical hyperparameters used below are specified separately in the implementation details.

At inference time, slot binding is the only additional LLM-based compilation operation. Routing is implemented as a nearest-centroid lookup, while the procedure, verified rules, interface statistics, and optional exemplar are read directly from the precompiled flat table. No skill retrieval, graph construction, per-task composition, or online repair is performed during execution.

E Prompt Templates

FlatSkill uses a small set of structured prompts during prefab compilation and inference. The prompts correspond directly to the main stages described in Section 3: (i) inducing a reusable procedure from successful execution backbones, (ii) proposing and verifying outcome-related rules from successful and failed trajectories, and (iii) binding procedure slots to a concrete task at inference

Procedure induction (mining)

You are an agent-programming assistant that extracts reusable procedures from successful interaction trajectories.
Your goal is to write one reusable **PROCEDURE** for a single task family in an interactive text environment.
You are given several successful training rollouts from this family. Each rollout has already been reduced to its causal actions. The marker `(explore / inspect)` denotes collapsed search or inspection behavior.
Requirements:

- Write a numbered procedure with one environment action per step.
- Write each action in backticks using the **exact action syntax** observed in the rollouts. Use only action forms that occur in the provided rollouts.
- Arguments that vary across rollouts must become role-based placeholders in braces, e.g., `{target}`. Arguments that remain constant across rollouts should stay verbatim.
- If an action repeats, compress it into one step using `repeat until <observable condition>`.
- Do not create a numbered step for `(explore / inspect)`. Instead, fold the search into the step that requires the object, e.g., "find X, then `pick up X`".
- Add a short parenthetical precondition or branch only when it is directly visible in the provided rollouts.
- Do not invent unsupported actions or steps.
- Use at most 14 numbered steps.

After the procedure, output a line `SLOTS:`, followed by one line for each placeholder:
`{name}: what it refers to in the task description`
Output only the procedure and the SLOTS block.

Figure 8: Procedure induction prompt. It is applied once per task family to successful execution backbones. Multiple procedure candidates are sampled, and FlatSkill retains the candidate whose action-type sequence best covers the observed backbones according to LCS-F1.

time. We provide the exact templates below for reproducibility. The task trajectories, action grammar, candidate rules, and task descriptions are inserted into the corresponding placeholders when the prompts are instantiated.

Procedure induction. The first prompt converts multiple successful execution backbones of the same task family into a single reusable procedure. The prompt explicitly asks the model to preserve the observed action syntax, abstract task-specific arguments into named slots, and avoid introducing actions that were not supported by the trajectories.

Rule proposal. The induced procedure captures the execution path shared by successful trajectories, but it does not necessarily capture conditions that distinguish successful from failed executions. FlatSkill therefore uses a second prompt to propose short behavioral rules from contrasts between the two groups.

Rule verification. Because the proposed rules are generated by an LLM, FlatSkill evaluates them independently on additional trajectories. The verifier judges only whether the observed behavior follows a rule, without using the trajectory's final success or failure as evidence.

Rule proposal (mining)

You are a trajectory analyst studying why agents succeed or fail in an interactive text environment.
You are given successful and failed training rollouts from the **same task family**. Each rollout contains its actions, observations, and an `OUTCOME` field.
Your goal is to identify concise behavioral rules that could be followed by a future agent.
Requirements:

- Propose at most 6 short imperative **RULES**.
- Each rule should plausibly explain a behavioral difference between successful and failed rollouts.
- Use both the rollout actions and the `OUTCOME` information when proposing rules.
- Every rule must be checkable using only the actions and observations shown in a rollout.
- Do not restate the task goal.
- Avoid generic advice that is not grounded in observable behavior.

Return the result as a JSON list of strings.

Figure 9: Rule proposal prompt. Successful and failed trajectories of the same task family are shown together. The proposed rules are not used directly; they are retained only after the independent verification step in Figure 10.

Rule verification (mining)

You are an impartial verifier of behavioral rules for trajectories in an interactive text environment.
You are given one rollout and a set of candidate **RULES**.
For each rule, determine whether the rollout:

- `follows`: the relevant situation occurs and the rollout obeys the rule;
- `violates`: the relevant situation occurs and the rollout contradicts the rule;
- `na`: the situation described by the rule never arises in this rollout.

Judge using **only the actions and observations shown in the rollout**. Ignore whether the task ultimately succeeded or failed.
Return JSON in the following form:
`{"R1": "...", "R2": "...", ...}`

Figure 10: Rule verification prompt. Each candidate rule is evaluated independently on held-out trajectories. FlatSkill retains a rule only when following versus violating it is associated with a sufficient difference in observed outcomes.

Slot binding at inference time. After routing a test task to a prefab, FlatSkill performs a single lightweight LLM call to instantiate the task-specific arguments of the procedure. The binder is intentionally restricted to copying values from the task description rather than generating new entities.

Slot binding (inference)

You are a slot-binding assistant that instantiates a reusable procedure for one concrete task.
You are given:

- the task description;
- the placeholders used by the selected procedure;
- a short description of the role of each placeholder.

Fill each placeholder using **only information explicitly present in the task description**. Copy object names and other task entities exactly as written.
If the task description does not determine the value of a placeholder, return `null`. Do not infer or invent missing values.
Return JSON in the following form:
`{placeholder: value, ...}`

Figure 11: Slot-binding prompt. This is the only LLM-based compilation call used at inference time: the slots of the selected prefab are instantiated directly from the test-task description before the prefab is passed to the execution agent.

F CASE STUDY: BUYING A PAIR OF PETITE WIDE-LEG PANTS IN WEBSHOP

Case Study: *Find me wide leg, machine wash women's pants with color: chambray stripes, and size: small petite, and price lower than 60.00 dollars* (WebShop, session 7097)

Step 1: Routing. The task text is matched against the flat table of task families. WebShop is a single-family benchmark, so routing returns that family's prefab directly; no retrieval over a skill library and no plan compilation take place. What the agent receives is one procedure, the mined location priors, and three worked purchases from training instances.

Step 2: The routed prefab (abridged; mined from the training rollouts, never task-specific):

1. `search[<keywords>]` using relevant terms from the query.
2. If results appear, scan the list and `click[<product id>]` for the most relevant product.
3. If no matching product is found *or the clicked product is incorrect*, `click[Back to Search]` and refine.
4. On the product page, identify all required options (colour, size) from the query.
5. For each required option, `click[<option value>]` exactly as displayed.
9. Once satisfied with the product and options, `click[Buy Now]`.

Step 3: Execution. Five actions, no backtracking:

1. `search[wide leg machine wash women's pants chambray stripes small petite under 60]` ✓
2. `click[B01DBAYRRC]` — the goal product, taken from the first result list. ✓
3. `click[small petite]` ✓
4. `click[chambray stripes]` ✓
5. `click[Buy Now]` ✓ **Reward** 1.00 **in 5 steps.**

Comparison. All three baselines score 0.00 on this session and all three exhaust the 30-step budget without buying anything.

- **ReAct** issues six different searches for the same query, rewording it each time ("`...under $60`", "`chambray stripe petite pants`", "`""chambray stripe" wide leg pants`"), opens two wrong products, and returns to the search box after each.
- **ReAct + Skills** behaves the same way with the retrieved documents in context: five searches, four products opened, `click[Back to Search]` after every one.
- **SkillNet** opens the right product page once (`B08JVMV64R` is not the goal, but the listing containing the goal was on screen), then leaves it to inspect `Description` and `< Prev`, returns to search, and never comes back.

The difference is not knowledge of the domain — every method can read the same product pages — but a commitment rule. The prefab's step 2 says to click the most relevant product *from the results already on screen*, and its step 3 permits a new search only when the chosen product turns out to be wrong. The baselines keep rewriting the query instead of opening a candidate and checking it. This is the single largest per-session gap in the benchmark: eight of the two hundred sessions have at 1.00 with all three baselines at 0.00, and every one of them has this shape.

G Failure Analysis

An action form the environment never accepts. ALFWorld's grammar has `move {obj} to {recep}` and no `put`. Across 600 mining rollouts, `put` was issued and rejected every single time; it accounts for 56% of all rejected actions, and 303 of the 357 successful rollouts (85%) tried it at least once before recovering. This is why mining the failed rollouts adds $+10.0$ points on ALFWorld when it is the only thing injected and $+0.5$ on top of the procedure: the successful rollouts already teach the same fact positively, by using `move`.

Refusing to act. On WebShop, Qwen3-235B under SkillNet emits prose where an action belongs — "(No valid action can be taken; no product meets all requirements.)" — for 45.3% of its actions, against 1.2% for DeepSeek-V3.2, and repeats it until the step limit. Under the routed procedure the rate falls to 11.0%, the limit is reached by 21.0% of episodes instead of 52.0%, and 80.0% of episodes reach a purchase instead of 52.5%. Most of that model's WebShop gain is the procedure pinning the action form, not a better shopping policy, and the cell should be read that way.

Searching instead of committing. The largest per-session gaps on WebShop share one shape: the baselines rewrite the query instead of opening a candidate. On the eight sessions where the routed procedure scores 1.00 and all three baselines score 0.00, each baseline issues five or six different searches for the same request and exhausts its budget without a purchase. Appendix F traces one of them action by action.