GRAPHSKILLEVO: EVOLUTIONARY OPTIMIZATION OF GRAPH-STRUCTURED AGENT SKILLS

Anonymous authors
Paper under double-blind review

ABSTRACT

Skills can improve the performance of Large Language Model (LLM) agents by providing task-specific procedural guidance, while skill optimization further improves their effectiveness through iterative refinement. However, existing skill optimization methods typically represent skills as unstructured natural-language instructions, creating two key challenges: **1)** Unstructured skills often lack explicit workflow-level guidance and contain substantial redundancy, making them difficult for LLMs to execute; **2)** the vast search space of unconstrained natural-language skills makes skill optimization ineffective. To address these challenges, we propose representing skills as **graph-structured natural-language artifacts**. In graph-structured skills, each node represents an execution step together with its operational guidance, while directed edges encode context-dependent transitions between steps. Compared to unstructured skills, graph-structured skills can provide clear workflow-level guidance. Moreover, the proposed graph-structured skill can also facilitate skill optimization. Building on this structured representation, we introduce **GraphSkillEvo**, a population-based evolutionary optimization framework with mutation and crossover operators for graph-structured skills. By maintaining multiple candidate skills and combining effective components, GraphSkillEvo enables broader and more comprehensive exploration of the structured skill space than purely LLM-based iterative self-refinement. Extensive experiments across five agent benchmarks demonstrate that GraphSkillEvo outperforms the strong skill optimization baseline SkillOpt, improving average accuracy by 4.01% on GPT-5.4-nano and 1.76% on GPT-5.4. Our code is available at `https://anonymous.4open.science/r/GraphSkillEvo`.

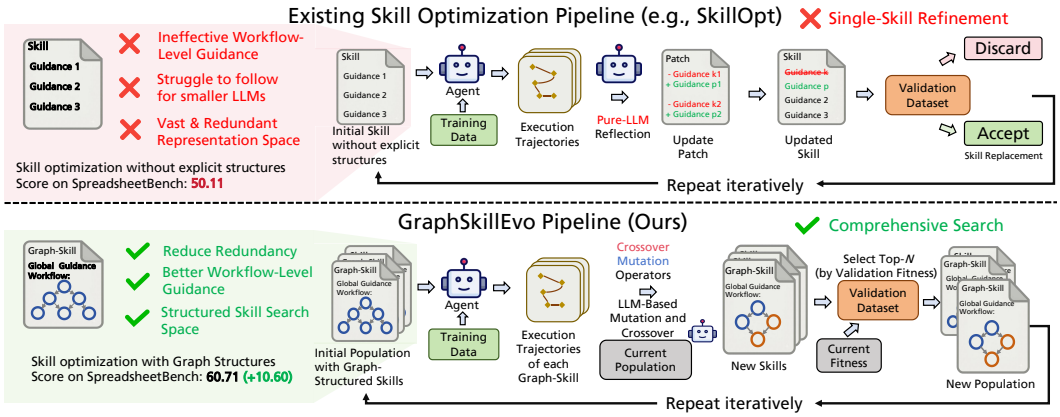


Figure 1: Existing skill optimization methods (e.g., SkillOpt) maintain unstructured skills and optimize skills purely with LLM self-reflection, resulting in suboptimal performance. GraphSkillEvo solves this with graph-structured skills for giving more workflow-level guidance and providing a structured search space for comprehensive population-based evolutionary search.

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.

1 INTRODUCTION

Large language models (LLMs) are increasingly deployed as agents across a wide range of real-world applications (Schick et al., 2023; Wang et al., 2023; Yang et al., 2024; Yao et al., 2022). In these agentic settings, skills serve as reusable prompt-level natural-language artifacts that provide task-specific procedural guidance, encoding workflows, domain knowledge, operational rules, and output constraints to help agents complete complex tasks (Li et al., 2026; Jiang et al., 2026). Beyond improving the performance of a particular agent, an important advantage of skills is that the procedural knowledge they encode can be reused across different LLMs. This cross-model portability is especially valuable as LLMs are rapidly updated and replaced in practice, allowing task-specific capabilities to be preserved without rebuilding the underlying procedures for every newly released model (Yang et al., 2025; Guo et al., 2025; Gemini Team, 2025).

However, manually written or one-shot LLM-generated skills can be incomplete and fragile, motivating recent work on *skill optimization* (Ni et al., 2026; Alzubi et al., 2026; Yang et al., 2026b; Zhang et al., 2026; Wang et al., 2026a; Liu et al., 2026b; Ma et al., 2026). Existing skill optimization methods (e.g., SkillOpt (Yang et al., 2026a) shown in Figure 1), however, typically represent skills as unstructured natural-language instructions without an explicit structure. ***This unstructured representation creates two fundamental challenges***:

1. **Difficulty in skill execution.** Optimized skills often take the form of lengthy checklists or bullet-point instructions that provide only coarse-grained workflow guidance, making it difficult for LLM agents to determine which guidance is relevant at the current stage and what step should follow next. This issue is particularly severe for less capable LLMs (e.g., GPT-5.4-nano), which are more likely to struggle with overlong instructions.
2. **Difficulty in skill optimization.** The lack of explicit structure also results in a large and redundant search space for skill optimization. Similar workflows can be expressed through many different unstructured textual realizations. As a result, the optimizer must explore many representational variants that do not correspond to meaningful procedural changes.

To address these limitations, as shown in Figure 1, motivated by the close analogy between skills and flow diagrams in providing stepwise procedural guidance, we formulate each skill as a **graph-structured natural-language artifact**. In a graph-structured skill, each node represents an agentic execution step and contains self-contained operational guidance, including the instructions, rules, and constraints relevant to that step. Each directed edge represents a context-dependent transition between execution steps, allowing different task conditions to induce different execution paths through the graph. For **skill execution**, this formulation makes the underlying workflow explicit and helps agents identify the guidance relevant to each execution step. For **skill optimization**, the graph structure reduces representational redundancy by explicitly organizing execution steps and their dependencies, thereby providing a more compact and structured search space than unstructured natural-language skills.

Building on this structured search space, we introduce **GraphSkillEvo**, a population-based **evolutionary computation (EC) framework** for optimizing graph-structured agent skills. GraphSkillEvo maintains a population of candidate skills to preserve the diversity of high-quality graph-structured skills throughout optimization and evolves them using structure-aware mutation and crossover operators. Mutation revises individual skills based on their execution trajectories, while crossover recombines complementary and effective graph components from different candidates. Together, these mechanisms enable broader exploration beyond purely LLM-based iterative self-refinement and facilitate the discovery of higher-quality skills. Our contributions are as follows:

1. We formulate agent skills as graph-structured natural-language artifacts that explicitly represent execution steps and context-dependent transitions, providing clearer workflow-level guidance and reducing representational redundancy.
2. We introduce GraphSkillEvo, a population-based evolutionary computation framework with structure-aware mutation and crossover operators for effectively exploring and optimizing graph-structured skills.
3. We conduct extensive experiments across diverse agent benchmarks, demonstrating that GraphSkillEvo outperforms strong skill-optimization baselines across two different LLMs, two different harnesses, and five benchmark settings.

2 PRELIMINARY

2.1 PROBLEM DEFINITION: SKILL OPTIMIZATION

Let $\mathcal{A}$ denote an agent composed of an LLM and its execution harness (Guo et al., 2026). For a given task, a skill s is a natural-language artifact supplied to the agent during execution to help solve instances of that task. Such skills can be manually written, generated by LLMs in one shot, or further refined through skill optimization (Ni et al., 2026). For a task instance x, execution with the skill s produces a trajectory $\tau_{x,s}$ and a score $r_{x,s}$ computed by a task-specific scoring function R:

$$\tau_{x,s} = \mathcal{A}(x, s), \qquad r_{x,s} = R\big(x, \tau_{x,s}\big).$$

The optimization goal is to find a skill that maximizes the performance of the agent on the task:

$$s^\star \in \arg\max_s J_D(s), \qquad J_D(s) = \frac{1}{|D|} \sum_{x \in D} r_{x,s}.$$

Here, D is a dataset of instances of the task, and $J_D(s)$ measures task performance of the agent using skill s. Following existing skill optimization methods (Yang et al., 2026a), we optimize only the skill artifact while keeping the LLM parameters and execution harness fixed. In practice, D consists of three subsets: D_{train}, D_{val}, and D_{test}. The training set D_{train} is used to collect execution trajectories, which provide feedback for proposing new skills. The validation set D_{val} is used to assess candidate skills during optimization. The test set D_{test} is used only for final evaluation.

2.2 SKILL OPTIMIZATION METHODS

Manually written skills or skills generated by LLMs in one shot are usually incomplete and fragile. So, recent methods automatically construct or distill skills from execution trajectories and interaction experience (e.g., EvoSkill (Alzubi et al., 2026), Trace2Skill (Ni et al., 2026), SkillX (Wang et al., 2026a), SkillOpt (Yang et al., 2026a)). EvoSkill discovers and refines skills through iterative failure analysis and Pareto-based selection (Alzubi et al., 2026). Trace2Skill consolidates multiple trajectory patches into a single portable skill via parallel merging (Ni et al., 2026). SkillX extracts multi-level skills from execution trajectories, and constructs a skill library via iterative refinement and exploratory skill expansion (Wang et al., 2026a). As a representative skill optimization method illustrated in Figure 1, SkillOpt (Yang et al., 2026a) starts from an initial skill $s^{(0)}$. At iteration t, the current skill $s^{(t)}$ is provided to agent $\mathcal{A}$ and executed on the training set D_{train}, producing execution trajectories as $\mathcal{T}^{(t)} = \{\tau_{x,s^{(t)}} \mid x \in D_{\text{train}}\}$. An agent $\mathcal{A}^{\text{patch}}$ analyzes these trajectories to generate skill update patches, which are applied to the current skill to obtain a candidate skill:

$$\text{Patch}^{(t)} = \mathcal{A}^{\text{patch}}\left(s^{(t)}, \mathcal{T}^{(t)}\right), \qquad \tilde{s}^{(t+1)} = s^{(t)} \oplus \text{Patch}^{(t)},$$

where $\oplus$ denotes textual patch application. Finally, the candidate skill is evaluated on D_{val}. If $\tilde{s}^{(t+1)}$ improves validation performance over $s^{(t)}$, then $s^{(t+1)}$ is set to $\tilde{s}^{(t+1)}$; otherwise, $s^{(t+1)}$ remains $s^{(t)}$. This process is repeated for multiple rounds.

Though demonstrating solid refinements, existing skill optimization methods still have two main limitations. **1)** First, existing methods typically optimize skills as unconstrained natural-language artifacts without explicit structural constraints. As a result, optimized skills can become lengthy and redundant while providing limited workflow-level guidance. GraphSkillEvo instead represents skills as graph-structured natural-language artifacts, making the execution workflow explicit. **2)** Second, existing methods optimize skills in a large and redundant search space. GraphSkillEvo, instead, provides a more compact and well-structured search space and evolves a population of skills through mutation and crossover operators. This results in a more comprehensive exploration compared to the pure LLM-based self-refinement in existing skill optimization methods.

3 METHODOLOGY: GRAPHSKILLEVO

3.1 GRAPH-STRUCTURED SKILLS

To address the challenges of coarse workflow-level guidance and redundant search space faced by existing methods in optimizing unstructured agent skills, this paper formulates skills as graph-

structured natural-language artifacts. Formally, a graph-structured skill $s = \langle h_s, g_s \rangle$ comprises global guidance h_s and a directed graph $g_s = (V_s, E_s)$ with node set V_s and edge set E_s.

(1) Global Guidance h_s. A skill may include task descriptions, general principles, and shared execution templates that apply across execution steps and are not specific to any individual node. The global guidance h_s collects these shared instructions.

(2) Node Set V_s. The node set $V_s = \{v_1, \ldots, v_{n_s}\}$ contains n_s reusable nodes. Each node describes an execution step, such as parsing the task goal, retrieving evidence, performing an operation, or verifying the final answer, together with the instructions, rules, and constraints required to perform that step.

(3) Edge Set E_s. The edge set is specified through M workflows $\{(c_m, p_m)\}_{m=1}^{M}$. Each workflow addresses a particular situation within the task, with c_m specifying its applicability condition. The execution path p_m is an ordered sequence of nodes from V_s that specifies the order in which the agent follows the corresponding execution steps. Consecutive nodes in each path define directed edges in E_s, representing transitions between execution steps under the corresponding workflow's condition. Different workflows may share nodes while prescribing different execution paths.

Example. We illustrate with an example skill:

> **Name:** Household Skill
> **Global Guidance h_s:**
> *Task Description:* Complete household tasks by navigating rooms, interacting with objects, and using appliances.
> *General Principles:* Never repeat the same action more than twice in a row. If stuck, move to a different unexplored location. Always pick an action from the admissible action list. Do not invent actions.
> ...
> **Node Set V_s:**
> *Explore Object:* Systematically search surfaces and containers, opening closed containers before determining that the target object is absent.
> *Take Object:* When the required object is visible and reachable, take it immediately before moving elsewhere.
> ...
> **Edge Set E_s** (specified through workflows)**:**
> *Pick & Place:*
> - Use When: Put one instance of the requested object in/on the requested receptacle.
> - Workflow: 1. Parse Goal; 2. Explore Object; 3. Take Object; 4. Find Destination Receptacle; 5. Place Object; 6. Verify Completion;
>
> ...

Graph-structured skills have several attractive properties for large language model agents.

1. **Low Redundancy.** Instructions shared by multiple workflows can be specified once in a reusable node and incorporated into multiple execution paths, rather than being repeated across different sections of a skill document.
2. **Explicit workflow guidance.** Each workflow specifies an execution path consisting of an ordered sequence of execution steps, where each step corresponds to a node that contains the instructions, rules, and constraints required for that step. This structure helps LLM agents follow a suitable and precise stepwise workflow and determine which instruction is relevant at each stage of execution.
3. **Providing a more compact and structured search space for GraphSkillEvo.** Instead of separately searching over many unstructured textual realizations of similar workflows, the optimizer can directly operate on explicit execution steps and their dependencies.

3.2 Evolutionary Optimization over Graph-Structured Skills

To more comprehensively explore the structured skill space, we introduce GraphSkillEvo, a population-based evolutionary optimization framework with mutation and crossover operators for graph-structured skills. The population preserves multiple high-quality skills throughout optimiza-

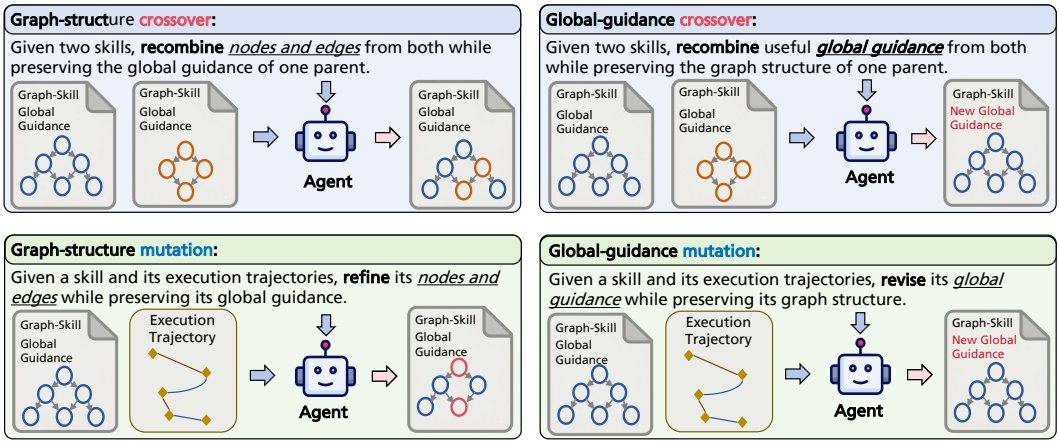


Figure 2: Four evolutionary operators used in GraphSkillEvo. Global-guidance mutation revises the global guidance of a parent skill based on its execution trajectories, while graph-structure mutation updates its nodes and edges by refining node instructions, adding or removing nodes, and adjusting execution paths. Global-guidance crossover recombines useful global guidance from two parent skills while preserving one parent's graph structure, and graph-structure crossover recombines nodes and edges from two parent skills while preserving one parent's global guidance.

tion. Mutation modifies an individual skill by incorporating feedback from its execution trajectories, while crossover transfers beneficial components between graph-structured skills.

The overall optimization procedure of GraphSkillEvo consists of the following four steps.

Step 0: Population initialization. GraphSkillEvo initializes a population $\mathcal{P}^{(1)} = \{s_i^{(1)}\}_{i=1}^{N}$ containing N skills. In addition to the initial skill, the remaining skills are generated by an initialization prompt that provides the task context and asks the LLM to create diverse graph-structured skills. After initialization, every individual $s_i^{(1)} \in \mathcal{P}^{(1)}$ is evaluated on the full validation dataset D_{val}, and its validation score $J_{D_{val}}(s_i^{(1)})$ is used as its fitness value.

Step 1: Execution on the training set. At each generation t, GraphSkillEvo samples a small batch of B instances $\mathcal{B}^{(t)}$ from training dataset D_{train}. Each skill $s_i^{(t)} \in \mathcal{P}^{(t)}$ in the current population is attached to the LLM agent $\mathcal{A}$ and executed on these instances, producing execution trajectories

$$\mathcal{T}_i^{(t)} = \{\tau_{x,s_i^{(t)}} \mid x \in \mathcal{B}^{(t)}\}.$$

For each skill, let $\mathcal{F}_i^{(t)}$ denote the failed trajectories retained as reflection information for generating new skills. Formally,

$$\mathcal{F}_i^{(t)} \subseteq \left\{\tau_{x,s_i^{(t)}} \;\middle|\; x \in \mathcal{B}^{(t)},\ r_{x,s_i^{(t)}} = 0\right\}, \qquad \left|\mathcal{F}_i^{(t)}\right| \le K.$$

Here, $r_{x,s_i^{(t)}} = 0$ indicates that executing skill $s_i^{(t)}$ on instance x fails, and K is the maximum number of failed trajectories retained for each skill.

Step 2: Generation of new skills. GraphSkillEvo generates N new skills. Each new skill is generated through the following three substeps:

1. Step 2.1: Operator selection. GraphSkillEvo selects an operator from four operators using a round-robin schedule.
2. Step 2.2: Skill selection. GraphSkillEvo selects parent skill(s) from the current population to generate the new skill. The selection probability is $p \propto 1/(r+N)$, where r denotes the fitness rank of the corresponding skill within the population and N is the population size.
3. Step 2.3: Skill generation. For each newly generated skill $\tilde{s}_j^{(t+1)}$, the operator and parent skill(s) used to generate $\tilde{s}_j^{(t+1)}$ are denoted by $o_j^{(t)}$ and $s_{\text{parent},j}^{(t)}$, respectively. Here, $s_{\text{parent},j}^{(t)}$ represents

one parent for mutation and two parents for crossover, while $\mathcal{F}^{(t)}_{\text{parent},j}$ denotes the associated retained failed execution trajectories. The LLM agent $\mathcal{A}^{\text{gen}}$ generates each new skill $\widetilde{s}_j^{(t+1)}$ using the selected operator and parent skill(s), with the corresponding failed trajectories provided only for mutation.

$$\tilde{s}_j^{(t+1)} = \begin{cases} \mathcal{A}^{\text{gen}}\left(o_j^{(t)}, s^{(t)}_{\text{parent},j}, \mathcal{F}^{(t)}_{\text{parent},j}\right), & \text{if } o_j^{(t)} \text{ is a mutation operator,} \\ \mathcal{A}^{\text{gen}}\left(o_j^{(t)}, s^{(t)}_{\text{parent},j}\right), & \text{if } o_j^{(t)} \text{ is a crossover operator,} \end{cases} \qquad j = 1, \ldots, N.$$

As shown in Figure 2, GraphSkillEvo uses the following four evolutionary operators:

1. **Global-guidance mutation** revises the global guidance of a selected skill using LLM-based self-reflection.
2. **Graph-structure mutation** revises the nodes and edges of a selected skill using its reflection information. The revisions include refining node instructions, adding or deleting reusable nodes, and adjusting task workflows.
3. **Global-guidance crossover** recombines useful global guidance from two selected skills while preserving the graph structure of one of them.
4. **Graph-structure crossover** recombines useful nodes and edges from two selected skills while preserving the global guidance of one of them.

Detailed prompts for these operators are provided in Appendix E.1.

Step 3: Population selection. After generating the N new skills, GraphSkillEvo evaluates each of them on the full validation set D_{val}. It then updates the population by retaining the N skills with the highest fitness values among the current population and the newly generated skills. Let $\widetilde{\mathcal{P}}^{(t+1)} = \{\widetilde{s}_j^{(t+1)}\}_{j=1}^{N}$ denote the set of newly generated skills. The next-generation population is:

$$\mathcal{P}^{(t+1)} \in \underset{\substack{\mathcal{S} \subseteq \mathcal{P}^{(t)} \cup \widetilde{\mathcal{P}}^{(t+1)} \\ |\mathcal{S}| = N}}{\arg\max} \sum_{s \in \mathcal{S}} J_{D_{\text{val}}}(s).$$

Steps 1-3 are repeated for T generations, after which the skill with the highest fitness value in the final population is returned as the optimized graph-structured skill.

4 Experiments

Benchmarks. We evaluate GraphSkillEvo on five benchmarks: SearchQA (Dunn et al., 2017), SpreadsheetBench (Ma et al., 2024) (abbreviated as Spreadsheet in tables), DocVQA (Mathew et al., 2021), LiveMathematicianBench (He et al., 2026) (abbreviated as LiveMath), and ALFWorld (Shridhar et al., 2020). These benchmarks cover fact-based question answering, spreadsheet manipulation, visual document understanding, mathematical multiple-choice reasoning, and embodied interaction. For each benchmark, we divide the data into a training set, a validation set, and a test set. The details of each benchmark are provided in Appendix B.1 and Appendix B.2.

Metrics. We report the average success rate on the test set. For SearchQA, DocVQA, and LiveMath, correctness is measured by exact match accuracy. For SpreadsheetBench, a task is correct only when the workbook matches the gold answer at all required locations across all evaluation cases. For ALFWorld, correctness is measured by the pass rate within an interaction limit.

Baselines. We compare against four skill sources. **1)** *No skill* runs the benchmark without any skill. **2)** *Human skill* uses a skill written by an expert. **3)** *LLM skill* uses a skill generated by an LLM from the task description. **4)** *SkillOpt* iteratively optimizes skills using rollout reflections, selected edits, and validation gating. The implementation details of the baselines are provided in Appendix D.1.

LLMs. All experiments use GPT-5.4 (OpenAI, 2026) and GPT-5.4-nano. We use medium reasoning effort for GPT-5.4 and GPT-5.4-nano. In each experimental setting, the same LLM is used for task execution and skill optimization in both GraphSkillEvo and SkillOpt.

Table 1: Main results across five benchmarks, two LLMs, and two agent harnesses. Each entry reports the success rate on the test set. Higher values indicate better performance. Bold numbers mark the best-reported result among all skill sources for the same model and benchmark.

Model	Skill source	SearchQA	Spreadsheet	DocVQA	LiveMath	ALFWorld	Average
No harness							
GPT-5.4	No Skill	77.50	39.16	79.05	33.60	73.13	60.49
	Human skill	77.71 +0.21	37.85 -1.31	84.04 +4.99	31.72 -1.88	81.84 +8.71	62.63 +2.14
	LLM skill	78.19 +0.69	35.59 -3.57	88.23 +9.18	35.21 +1.61	75.86 +2.73	62.62 +2.13
	SkillOpt	82.21 +4.71	64.87 +25.71	89.30 +10.25	47.58 +13.98	86.56 +13.43	74.10 +13.61
	GraphSkillEvo	**83.80** +6.30	**69.40** +30.24	**90.37** +11.32	**48.65** +15.05	**87.06** +13.93	**75.86** +15.37
GPT-5.4-nano	No Skill	58.12	35.12	36.72	23.93	41.29	39.04
	Human skill	62.36 +4.24	34.64 -0.48	59.09 +22.37	25.80 +1.87	50.25 +8.96	46.43 +7.39
	LLM skill	60.14 +2.02	32.73 -2.39	66.49 +29.77	24.19 +0.26	53.73 +12.44	47.46 +8.42
	SkillOpt	69.52 +11.40	50.11 +14.99	77.80 +41.08	**29.56** +5.63	57.46 +16.17	56.89 +17.85
	GraphSkillEvo	**72.93** +14.81	**60.71** +25.59	**80.92** +44.20	28.76 +4.83	**61.19** +19.90	**60.90** +21.86
Codex harness							
GPT-5.4	No Skill	79.42	56.07	82.35	54.03	-	67.97
	Human skill	82.36 +2.94	48.57 -7.50	86.36 +4.01	51.61 -2.42	-	67.23 -0.74
	LLM skill	81.93 +2.51	46.79 -9.28	86.10 +3.75	54.03 +0.00	-	67.21 -0.76
	SkillOpt	83.02 +3.60	77.14 +21.07	87.43 +5.08	60.21 +6.18	-	76.95 +8.98
	GraphSkillEvo	**83.26** +3.84	**79.28** +23.21	**89.30** +6.95	**61.29** +7.26	-	**78.28** +10.31

Harness. We evaluate GraphSkillEvo both without an agent harness and with the Codex harness. Without a harness, the skill is incorporated into the model instructions for each benchmark. With the Codex harness, Codex is invoked through its software development kit (SDK), and each task is assigned a separate local workspace. Each workspace contains the task description, any associated input files, and the skill. Codex is instructed to read the skill and follow its guidance while solving the task. Codex operates in the `workspace-write` sandbox with interactive approvals disabled. We leave the ALFWorld cells blank for the Codex harness because ALFWorld requires a persistent environment interaction, which is not supported by the standard Codex adapter.

Optimization parameters. During evolution, we set the population size to $N = 4$ and run $T = 5$ generations. At each generation, GraphSkillEvo samples 15 instances from D_{train} for execution, and uses up to 5 failed instances to build the reflection information for mutation. The four operators are selected in a round-robin schedule.

4.1 Main Results

Table 1 presents the main results across five benchmarks, two LLMs, and two agent harnesses. All reported results are averages over three repeated skill optimization runs. We compare GraphSkillEvo with no-skill execution, human-written skills, LLM-generated skills, and SkillOpt. All entries are test-set success rates. Across the 14 model–harness–benchmark settings, GraphSkillEvo achieves the highest mean success rate in 13 settings. **1)** Relative to no-skill execution, GraphSkillEvo improves the average success rate by 15.37% in the GPT-5.4 with no-harness, 21.86% in the GPT-5.4-nano with no-harness, and 10.31% in the GPT-5.4 Codex-harness. **2)** Compared with SkillOpt, a promising skill-optimization method, GraphSkillEvo achieves average gains of 1.76% under GPT-5.4 with no-harness, 4.01% under GPT-5.4-nano no-harness, and 1.33% under GPT-5.4 with Codex harness.

Small and less capable models benefit the most. Averaged across the five benchmarks, GraphSkillEvo outperforms SkillOpt by 4.01% on GPT-5.4-nano, compared with 1.76% on GPT-5.4. Procedural benchmarks see particularly large improvements. GraphSkillEvo improves over SkillOpt by 10.60% on SpreadsheetBench and 3.73% on ALFWorld. These gains suggest that the clear workflow guidance provided by graph-structured skills is especially helpful for tasks that require procedural execution, especially when agents need to interact with an external environment. The only exception is LiveMath with GPT-5.4-nano, where GraphSkillEvo trails SkillOpt by 0.80%.

Taken together, these results demonstrate that GraphSkillEvo can be effective across heterogeneous agent tasks, different LLM settings, and agent harnesses. The operator prompts and the generated

Table 2: Effect of explicit workflow guidance provided by the graph-structured representation across five benchmarks. Each entry reports the test-set success rate, expressed as a percentage. The unstructured counterpart removes the explicit workflow organization.

Skill	SearchQA	Spreadsheet	DocVQA	LiveMath	ALFWorld
Graph-structured	**72.93**	**60.71**	**80.92**	**28.76**	**61.19**
Unstructured	68.41 -4.52	58.21 -2.50	76.73 -4.19	27.41 -1.35	60.44 -0.75

Table 3: Ablation study of the graph structure and evolutionary operators.

Skill	SearchQA	Spreadsheet	DocVQA	Average
GraphSkillEvo	**72.93**	**60.71**	**80.92**	**71.52**
w/o graph structure	68.83 -4.10	50.59 -10.12	72.81 -8.11	64.08 -7.44
w/o mutation	57.19 -15.74	35.11 -25.60	71.20 -9.72	54.50 -17.02
w/o crossover	71.80 -1.13	53.56 -7.15	74.41 -6.51	66.59 -4.93

skills are listed in Appendix E. We also report the optimization curves in Appendix C.1, the significance tests in Appendix C.2, and a case study in Appendix C.4.

Token Consumption. We report the total number of tokens consumed during skill optimization across all five benchmarks. With GPT-5.4, GraphSkillEvo consumes 61.94 million tokens, compared with 81.08 million for SkillOpt. With GPT-5.4-nano, GraphSkillEvo consumes 75.94 million tokens, compared with 103.54 million for SkillOpt. SkillOpt uses 1.31 and 1.36 times as many tokens as GraphSkillEvo under the two model settings, respectively. These results show that our method achieves stronger performance while using substantially fewer optimization tokens than SkillOpt.

5 Discussion

Building on the comparative results in Section 4, we further examine how graph structure supports skill execution and optimization, and whether the resulting skills transfer across models. We organize the discussion around three research questions (RQs):

- **RQ1–Graph Representation for Skill Execution** (Section 5.1): Does the explicit workflow guidance provided by graph-structured representations improve the execution of optimized skills?
- **RQ2–Graph Representation for Skill Optimization** (Section 5.2): Does graph structure help discover higher-quality skills, and how do mutation and crossover contribute?
- **RQ3–Transferability of Graph-Structured Skills** (Section 5.3): Do the skills optimized by GraphSkillEvo remain effective when transferred to another LLM?

5.1 Effect of Graph-Structured Representation on Skill Execution

Explicit Workflow Guidance Improves Skill Execution. To examine the role of explicit workflow guidance provided by the graph-structured representation during execution, we take the skills optimized by GraphSkillEvo with GPT-5.4-nano and construct unstructured counterparts by removing explicit workflow organization while retaining task-general guidance and node-level instructions. The detailed conversion procedure and an example are provided in Appendix E.4. We evaluate both versions with GPT-5.4-nano on all five benchmarks. As shown in Table 2, removing explicit workflow organization reduces success rates by 4.52, 2.50, 4.19, 1.35, and 0.75 percentage points on SearchQA, Spreadsheet, DocVQA, LiveMath, and ALFWorld, respectively. These consistent decreases indicate that the retained instructions alone do not capture the full benefits of the graph-structured skills. These results support the effectiveness of the explicit workflow guidance provided by the graph-structured representation for skill execution.

5.2 Advantage of Graph Representation in Skill Optimization

To understand how graph structure facilitates skill optimization, we conduct ablation studies on GPT-5.4-nano and report the average results over three repeated experiments in Table 3. All abla-

Table 4: Cross-model transferability of optimized skills. **Baseline** denotes execution on GPT-5.4 without a skill. **Direct** denotes using a skill optimized with GPT-5.4 and then applied on GPT-5.4, and **Transferred** denotes using a skill optimized with GPT-5.4-nano and then applied on GPT-5.4.

Benchmark	Method	Baseline	Direct	Transferred
SearchQA	SkillOpt	77.50	82.21 +4.71	83.92 +6.42
	GraphSkillEvo	77.50	83.80 +6.30	84.07 +6.57
Spreadsheet	SkillOpt	39.16	64.87 +25.71	53.21 +14.05
	GraphSkillEvo	39.16	69.40 +30.24	71.78 +32.62
DocVQA	SkillOpt	79.05	89.30 +10.25	89.30 +10.25
	GraphSkillEvo	79.05	90.37 +11.32	89.83 +10.78

tion variants use the same experimental settings, differing only in the component ablated. When mutation or crossover is removed, we still generate N new skills per generation by cycling through the remaining operators.

Graph structure benefits skill optimization. The *w/o graph structure* variant initializes unstructured skills and evolves them through population-based evolutionary computation. Removing the graph structure decreases the average performance from 71.52 to 64.08, showing that population-based evolution alone is insufficient. The graph representation organizes skills into explicit procedural components and dependencies, providing a more structured search space for optimization.

Crossover enables broader exploration. The *w/o crossover* variant retains the graph representation, population, and mutation, but removes information exchange across candidates. Its performance drops to 66.59, suggesting that crossover helps combine effective components discovered along different search trajectories and enables broader exploration beyond iterative self-refinement.

Mutation enables trajectory-driven refinement. The *w/o mutation* variant removes the mutation operators and relies solely on crossover for skill evolution, resulting in the largest performance drop, to 54.50. This shows that execution feedback is important for locally refining individual skills, while crossover complements this refinement through cross-candidate recombination.

Additional ablations on population in evolutionary computation are in Appendix C.3.

5.3 Transferring Skills Across LLMs

Optimized Skills Remain Effective Across LLMs. To evaluate cross-model transferability, we optimize skills with GPT-5.4-nano and deploy them on GPT-5.4. For both GraphSkillEvo and SkillOpt, Table 4 compares execution without a skill, with a skill optimized directly on GPT-5.4, and with a skill transferred from GPT-5.4-nano. This comparison evaluates whether optimized procedural guidance remains useful beyond the model used for optimization. Transferred skills outperform the no-skill baseline on all three benchmarks. Moreover, transferred GraphSkillEvo skills achieve higher scores than transferred SkillOpt skills, with the largest advantage on Spreadsheet. On this benchmark, the transferred GraphSkillEvo skill achieves 71.78, exceeding both its directly optimized counterpart (69.40) and the transferred SkillOpt skill (53.21). These results demonstrate that skills optimized by GraphSkillEvo can be reused across the evaluated models.

6 Conclusion

In this paper, we formulate agent skills as graph-structured natural-language artifacts that make workflow guidance explicit and organize reusable execution steps into a structured search space. Building on this representation, we introduce GraphSkillEvo, a population-based evolutionary computation framework with structure-aware mutation and crossover for refining and recombining procedural components. Experiments across five agent benchmarks, two LLMs, and two execution settings demonstrate improved average performance over SkillOpt, while our token analysis shows lower aggregate optimization-token consumption. Further analyses support the benefits of graph structure for both skill execution and evolutionary optimization, and demonstrate effective skill transfer from GPT-5.4-nano to GPT-5.4. Future work includes combining our method with parametric optimization methods, extending the framework to richer graph composition mechanisms, and developing methods for merging graph-structured skills from diverse domains.

AI USE STATEMENT

In this work, we used generative AI tools for implementing methods. Specifically, large language models are components of our experimental methodology: we used GPT-5.4 and GPT-5.4-nano for task execution and skill optimization. We have not used generative AI tools for any other tasks with required disclosure, and the remaining tasks with required disclosure are not applicable to this work. Additionally, we used generative AI tools for code editing, literature discovery and summarization, and advice on manuscript revision. We have reviewed all AI-assisted work. In particular, the authors manually reviewed AI-assisted code changes. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To facilitate reproducibility, we provide an anonymous source-code release at `https://anonymous.4open.science/r/GraphSkillEvo`. Section 4 describes the experimental settings and optimization hyperparameters. Appendices B.1 and B.2 detail the benchmarks and dataset splits, while Appendices B.3 and B.4 describe graph structure validation and the complete optimization algorithm. Appendices D.1 and D.2 provide baseline implementation details and resource links with licensing information. Appendix E includes the prompts for skill initialization and evolutionary operators, together with examples of optimized graph-structured skills and their unstructured counterparts.

REFERENCES

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

Tong Bai, Zhenglin Wan, Pengfei Zhou, Xingrui Yu, Yang You, and Ivor W Tsang. Skilldag: Self-evolving typed skill graphs for llm skill selection at scale. *arXiv preprint arXiv:2606.03056*, 2026.

Matthew Dunn, Levent Sagun, Mike Higgins, V Ugur Guney, Volkan Cirik, and Kyunghyun Cho. Searchqa: A new q&a dataset augmented with context from a search engine. *arXiv preprint arXiv:1704.05179*, 2017.

Gemini Team. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Jianyuan Guo, Zhiwei Hao, Chengcheng Wang, Cheng Fan, Tingzhang Luo, Hongguang Li, Ying Gao, Hefei Mei, Jiankun Peng, Rongjian Xu, et al. From question answering to task completion: A survey on agent system and harness design. *arXiv preprint arXiv:2606.20683*, 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.

Linyang He, Qiyao Yu, Hanze Dong, Baohao Liao, Xinxing Xu, Micah Goldblum, Jiang Bian, and Nima Mesgarani. Livemathematicianbench: A live benchmark for mathematician-level reasoning with proof sketches. *arXiv preprint arXiv:2604.01754*, 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*, 2026.

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*, 2026.

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.

Xingyan Liu, Xiyue Luo, Linyu Li, Ganghong Huang, Jianfeng Liu, and Honglin Qiao. Skillforge: Forging domain-specific, self-evolving agent skills in cloud technical support. *arXiv preprint arXiv:2604.08618*, 2026b.

Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. Spreadsheetbench: Towards challenging real world spreadsheet manipulation. *Advances in Neural Information Processing Systems*, 37:94871–94908, 2024.

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

Minesh Mathew, Dimosthenis Karatzas, and CV Jawahar. Docvqa: A dataset for vqa on document images. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, pp. 2200–2209, 2021.

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

OpenAI. Introducing gpt-5.4, March 2026. URL `https://openai.com/index/introducing-gpt-5-4/`. Accessed 2026-07-26.

Libin Qiu, Zhirong Gao, Junfu Chen, Yuhang Ye, Weizhi Huang, Xiaobo Xue, Wenkai Qiu, and Shuo Tang. Autorefine: From trajectories to reusable expertise for continual llm agent refinement. *arXiv preprint arXiv:2601.22758*, 2026.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 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.

Chenxi Wang, Zhuoyun Yu, Xin Xie, Wuguannan Yao, Runnan Fang, Shuofei Qiao, Kexin Cao, Guozhou Zheng, Xiang Qi, Peng Zhang, et al. Skillx: Automatically constructing skill knowledge bases for agents. *arXiv preprint arXiv:2604.04804*, 2026a.

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

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

Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, et al. Evolver: Self-evolving llm agents through an experience-driven lifecycle. *arXiv preprint arXiv:2510.16079*, 2025.

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.

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

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

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. Skillopt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026a.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, et al. Autoskill: Experience-driven lifelong learning via skill self-evolution. *arXiv preprint arXiv:2603.01145*, 2026b.

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*, 2022.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, et al. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026.

APPENDIX CONTENTS

A Related Work **14**

A.1 Agent Skill Optimization 14

A.2 Graph for Agent Skill 14

B Methodological Details **15**

B.1 Benchmarks 15

B.2 Dataset Splits 15

B.3 Graph Structure Validation 16

B.4 Complete Optimization Algorithm 16

C Additional Experiments **17**

C.1 Optimization Curves 17

C.2 Significance Test 17

C.3 Additional Ablation Studies 18

C.4 Case Study 18

D Baselines & Licenses **18**

D.1 Baseline Implementation Details 18

D.2 Licenses 18

E Prompts and Optimized Skill Examples **20**

E.1 Prompts for Evolutionary Operators 20

E.2 Prompt for Skill Initialization 24

E.3 Example of Optimized Graph-Structured Skills 25

E.4 Example of an Unstructured Counterpart 29

A Related Work

A.1 Agent Skill Optimization

A skill encapsulates reusable procedural knowledge, including tool-use policies, applicability conditions, execution routines, and supporting resources (Li et al., 2026; Jiang et al., 2026). EvoSkill, Trace2Skill, SkillX, and AutoRefine use textual feedback obtained from agent execution trajectories to diagnose failures and improve skills (Alzubi et al., 2026; Ni et al., 2026; Wang et al., 2026a; Qiu et al., 2026). Meanwhile, SkillOpt studies how to train skills with deep-learning-style controls (Yang et al., 2026a). In a different line, EvolveR, SAGE, and SKILLRL iteratively co-evolve the LLM and skills through reinforcement learning (Wu et al., 2025; Wang et al., 2026b; Xia et al., 2026a). AutoSkill constructs personalized skills from lifelong experience (Yang et al., 2026b), whereas SkillClaw constructs cross-user skills through aggregating interaction trajectories from multiple users (Ma et al., 2026). Despite adopting different approaches to skill refinement, existing methods generally represent skills as unstructured natural-language instructions, which often lack workflow-level guidance, introduce substantial redundancy, and leave the optimizer with a large search space. In contrast, we formulate skills as graph-structured natural-language artifacts and optimize them with a population-based evolutionary framework. This design provides explicit workflow guidance, reduces redundancy, and facilitates skill optimization.

A.2 Graph for Agent Skill

Recent work uses graphs to support skill retrieval and composition. Graph-of-Skills and SkillDAG incorporate graph structure into skill retrieval over large skill libraries, enabling agents to efficiently identify the subset of skills required to execute the current task (Liu et al., 2026a; Bai et al., 2026). GraSP introduces graph-structured skill composition for large skill libraries, enabling agents to more effectively organize and orchestrate multiple skills during task execution (Xia et al., 2026b). HiSkill discovers multiple skills and organizes them into a graph in which each skill corresponds to a node, and retrieves a task-relevant skill subset during task solving (Hao et al., 2026). Unlike prior graph-based skill systems primarily designed for skill retrieval or composition, our work focuses on skill optimization: with the agent model and execution harness fixed, the goal is to find a skill that maximizes the agent's average performance across task instances. Effective skill optimization depends on both the representation of skills and the mechanisms used to search over them. In this work, we introduce a graph-structured skill representation that facilitates skill optimization. Building on this representation, we develop a population-based evolutionary optimization framework with structure-aware mutation and crossover operators to discover higher-quality skills. Our contribution lies in the joint design of a representation that facilitates skill optimization and an evolutionary search method tailored to that representation.

B METHODOLOGICAL DETAILS

B.1 BENCHMARKS

We provide the detailed introduction and settings of each benchmark in this subsection.

SearchQA. SearchQA (Dunn et al., 2017) evaluates question answering from accompanying textual evidence. Given a question and its evidence, the agent produces the answer to the question.

SpreadsheetBench. SpreadsheetBench (Ma et al., 2024) evaluates programmatic manipulation of real `.xlsx` workbooks. The agent generates Python code in an execution environment that provides the standard library, `openpyxl`, and `pandas`. We follow an iterative protocol in which the generated code is executed after each round and the resulting output or execution-level failure diagnostics is returned to the agent, which may revise the code in a subsequent round. We permit up to 30 code-generation rounds in the no-harness setting; with the Codex harness, each task uses a single code-generation round.

DocVQA. DocVQA (Mathew et al., 2021) is a visual question-answering task over document images. The agent receives a document image and a question, and returns the answer supported by the image.

LiveMathematicianBench (LiveMath). LiveMathematicianBench (He et al., 2026) consists of mathematical multiple-choice problems. For each problem, the agent selects and outputs one of the provided answer options.

ALFWorld. ALFWorld (Shridhar et al., 2020) evaluates interaction in a persistent, text-based household environment. At each step, the agent observes the current state, selects an admissible action, and receives the next environment observation. Each episode is limited to 50 interaction steps.

B.2 DATASET SPLITS

For each benchmark, we construct three disjoint sets. Following SkillOpt (Yang et al., 2026a), all experiments use the same deterministic dataset partitioning procedure with `split_seed=42`. All baselines use exactly the same three dataset partitions to ensure a fair comparison. The training set is used only for collecting rollout trajectories and failure feedback during optimization. The validation set is used to score candidate skills and guide population selection. The test set is reserved for final evaluation. Table 5 reports the size of each set used in all experiments.

Table 5: Sizes of the training, validation, and test sets used in the experiments. The training set is used only for collecting rollout trajectories and failure feedback during optimization. The validation set is used for skill selection during optimization, while the test set is used only for final reporting.

Benchmark	Train	Validation	Test
SearchQA	400	200	1400
SpreadsheetBench	80	40	280
DocVQA	107	53	374
LiveMath	35	18	124
ALFWorld	39	18	134

B.3 Graph Structure Validation

To maintain the graph structure of skills during evolution, we apply a validator to every newly generated skill. The validator is a script that checks whether the skill follows the required graph schema and whether the declared nodes and the overall graph are structurally consistent with one another. For example, it verifies that every node referenced in a workflow is defined in V_s and that every declared node appears in at least one workflow. Generated skills that fail validation are discarded and regenerated. This validator helps keep the optimized skills well-formed and graph-structured throughout evolution.

B.4 Complete Optimization Algorithm

Algorithm 1 provides pseudocode for the proposed GraphSkillEvo method.

Algorithm 1. Complete optimization algorithm of GraphSkillEvo.

Require: Initial graph skill s_1, train set D_{train}, validation set D_{val}, population size N, generations T, training sample size B, failure budget K
Ensure: Optimized graph-structured skill $\hat{s}$

Initialize population $\mathcal{P}^{(1)} \leftarrow \{s_1\}$
for $i = 1$ to $N - 1$ **do**
  Generate a candidate graph skill with the initialization prompt
  while the candidate fails graph structure validation **do**
    Discard the candidate and regenerate it
  end while
  Add the candidate to $\mathcal{P}^{(1)}$
end for
for each $s \in \mathcal{P}^{(1)}$ **do**
  Evaluate s on the full D_{val} and store fitness $J_{D_{val}}(s)$
end for
for $t = 1$ to T **do**
  Sample B instances from D_{train}
  for each parent $s \in \mathcal{P}^{(t)}$ **do**
    Execute s on the sampled instances
    Collect at most K failed trajectories as reflection information
  end for
  Sort $\mathcal{P}^{(t)}$ by validation fitness
  Assign parent-sampling weights $p_i \propto 1/(r_i + N)$, where r_i is the validation rank
  $\widetilde{\mathcal{P}}^{(t+1)} \leftarrow \emptyset$
  for $j = 1$ to N **do**
    Select the next operator in round-robin order from {global mutation, graph mutation, global crossover, graph crossover}
    Sample parent(s) according to the rank weights
    Invoke the selected operator prompt with parent skill(s) and, for mutation, the selected parent's reflection information
    Generate the new skill
    while the new skill fails graph structure validation **do**
      Discard the candidate and regenerate it
    end while
    Add the new skill to $\widetilde{\mathcal{P}}^{(t+1)}$
  end for
  for each child $s \in \widetilde{\mathcal{P}}^{(t+1)}$ **do**
    Evaluate s on the full D_{val} and store fitness $J_{D_{val}}(s)$
  end for
  $\mathcal{P}^{(t+1)} \leftarrow$ top-N skills from $\mathcal{P}^{(t)} \cup \widetilde{\mathcal{P}}^{(t+1)}$ by validation fitness
end for
return $\hat{s} \in \arg\max_{s \in \mathcal{P}^{(T+1)}} J_{D_{val}}(s)$

C Additional Experiments

C.1 Optimization Curves

Figure 3 plots the optimization curves of GraphSkillEvo and SkillOpt in the GPT-5.4-nano setting without an agent harness. The validation score is shown against the total number of tokens consumed during optimization. Each curve is averaged over three experiments. SkillOpt shows early convergence on performance, while GraphSkillEvo is able to converge to better performance via continuous performance updates.

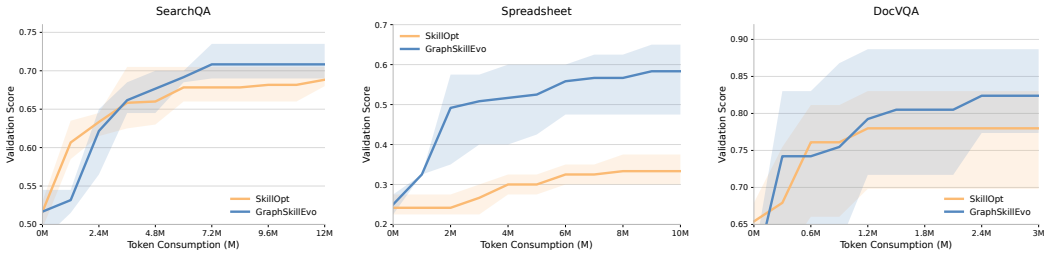


Figure 3: Optimization curve comparison between SkillOpt and GraphSkillEvo.

C.2 Significance Test

Although running a method three times can provide a preliminary comparison of its strengths and weaknesses, additional runs are needed to assess the reliability of the observed advantage. In this subsection, we introduce p-values to demonstrate the significant advantage of GraphSkillEvo over the mainstream skill optimization baseline SkillOpt. In addition to the three runs reported in Table 1, we conduct ten additional independent skill optimization runs for each method and report the test-set success rate of each run, together with the mean, standard deviation, and p-value, in Table 6. On DocVQA, GraphSkillEvo outperforms SkillOpt at a confidence level exceeding 90% under a one-sided t-test. On SearchQA, Spreadsheet, and ALFWorld, GraphSkillEvo achieves significantly higher mean success rates than SkillOpt under one-sided t-tests (p-value < 0.01).

Table 6: The significance test between SkillOpt and GraphSkillEvo using GPT-5.4-nano without an agent harness. We conduct ten independent skill optimization runs for each method on each benchmark. Avg and Std denote the mean and standard deviation of the test-set success rates across these runs, respectively. The p-value is calculated with single-tailed t-tests.

Benchmark	Method	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	Avg	Std	p-value
SearchQA	SkillOpt	67.86	70.21	69.93	70.64	68.21	67.71	70.21	70.43	72.29	66.86	69.44	1.69	
	GraphSkillEvo	74.57	69.36	76.79	73.07	68.14	72.14	74.29	74.79	72.29	71.21	72.67	2.62	0.00249
Spreadsheet	SkillOpt	46.79	45.36	42.86	50.36	54.64	57.14	39.29	53.21	57.50	55.36	50.25	6.37	
	GraphSkillEvo	59.64	59.29	63.57	59.64	46.07	56.43	57.50	60.36	63.57	58.57	58.46	4.92	0.00248
ALFWorld	SkillOpt	41.79	47.76	57.46	54.48	58.96	46.27	52.24	47.76	59.70	58.21	52.46	6.26	
	GraphSkillEvo	67.16	60.45	58.21	61.19	53.73	60.45	54.48	56.72	63.43	64.18	60.00	4.30	0.00319
DocVQA	SkillOpt	75.40	77.54	75.67	73.53	76.20	77.54	79.14	80.75	75.67	78.34	76.98	2.10	
	GraphSkillEvo	78.88	77.01	82.35	74.06	77.54	80.21	78.07	77.27	80.48	81.28	78.72	2.44	0.05290

C.3 Additional Ablation Studies

To further examine the contribution of population-based evolution, we consider two additional variants that retain the graph representation. All variants are conducted on GPT-5.4-nano using the same dataset partitions, and evaluation settings with the total number of new candidate skills generated during optimization matched to GraphSkillEvo. Table 7 reports the results.

The *w/o population* variant sets the population size to $N = 1$ and uses only mutation to refine a single graph-structured skill, reducing optimization to iterative refinement of a single graph-structured skill. It can therefore be viewed as a *SkillOpt-style single-trajectory refinement* baseline with graph representation. Its performance drops to 68.22, supporting the limitations of single-trajectory refinement. These results also suggest that graph representation alone does not account for the full benefits of GraphSkillEvo.

The *w/o population (multi-start)* variant initializes N skills using the same initialization procedure as GraphSkillEvo. Each skill is optimized independently through mutation, and the best version encountered during each independent optimization run is retained based on validation performance. Its average performance reaches 65.45, below 71.52 achieved by GraphSkillEvo, suggesting that diverse initialization and independent optimization alone are insufficient. These results also support the benefits of population-based selection and crossover used together in GraphSkillEvo.

Table 7: Additional ablation study of population-based evolution.

Skill	SearchQA	Spreadsheet	DocVQA	Average
GraphSkillEvo	**72.93**	**60.71**	**80.92**	**71.52**
w/o population	71.93 –1.00	56.07 –4.64	76.65 –4.27	68.22 –3.30
w/o population (multi-start)	69.72 –3.21	49.64 –11.07	77.00 –3.92	65.45 –6.07

C.4 Case Study

To illustrate how graph-structured skills guide agent execution, we provide a case study on ALFWorld. Figure 4 presents a graph excerpt from the graph-structured skill for ALFWorld together with three representative execution trajectories.

D Baselines & Licenses

D.1 Baseline Implementation Details

No skill. The no-skill baseline evaluates the agent with an empty skill artifact. No additional procedural guidance is prepended beyond the benchmark's native task prompt.

Human skill. The human-skill baseline uses a manually written benchmark-specific skill. The skill is fixed during evaluation and is not optimized.

LLM skill. The LLM-skill baseline uses a one-shot skill generated by GPT-5.4 from the benchmark task description. It does not use evolutionary optimization or validation feedback, and the generated skill is fixed during evaluation.

SkillOpt. SkillOpt optimizes skills using rollout reflection, textual edit selection, skill updating, and validation gating. Across benchmarks, optimization runs for 4 epochs. Each rollout batch contains 40 examples, with accumulation set to 1. The reflection minibatch size is 8, and the merge batch size is also 8. For skill editing, the edit budget is 4 and the minimum edit budget is 2. The edit budget follows a cosine schedule. Slow update is enabled with 20 samples, and slow-update acceptance is also controlled by validation gating. Meta-skill memory is enabled.

D.2 Licenses

The licenses and URLs of baselines are listed in Table 8.

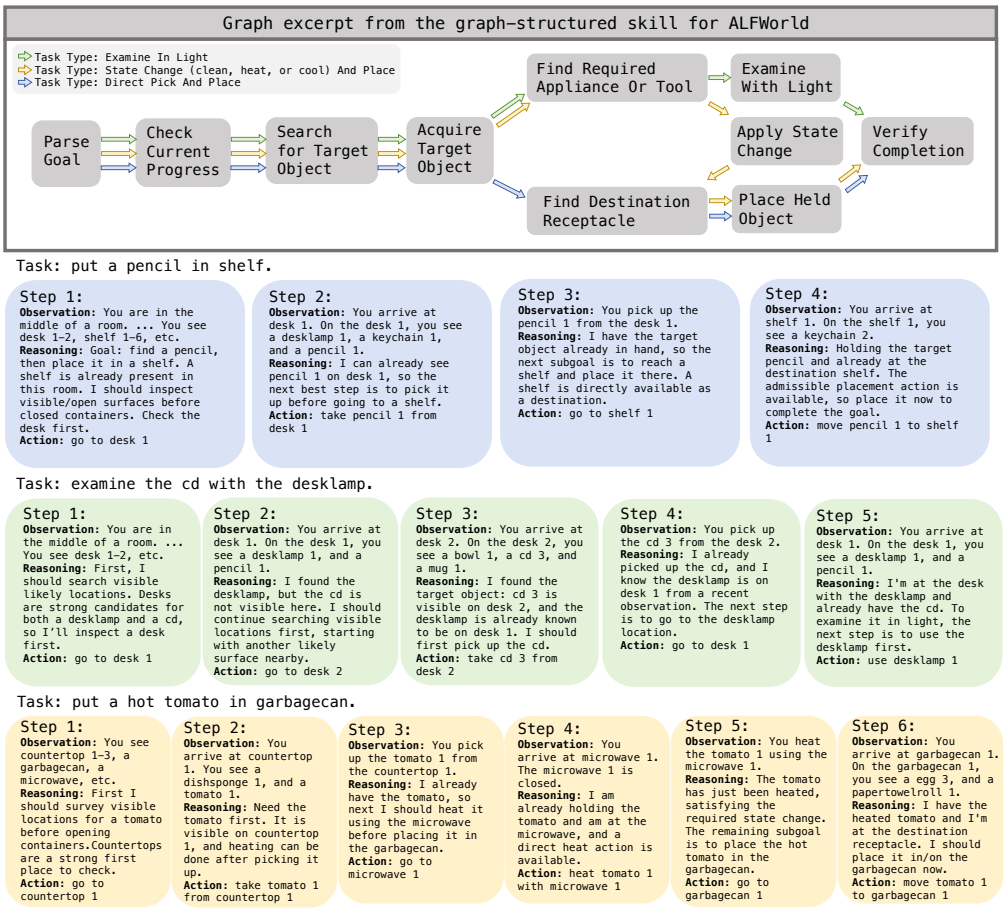


Figure 4: A case study of a graph-structured skill on ALFWorld. The figure presents a graph excerpt from the graph-structured skill for ALFWorld together with three representative execution trajectories. Different colors distinguish the three task types and indicate their corresponding workflow guidance and execution trajectories. The examples illustrate how graph-structured skills provide clear and explicit workflow guidance across different situations within the ALFWorld task.

Table 8: Links and licenses for datasets and method code.

Resource	Link	License
SearchQA	https://huggingface.co/datasets/lucadiliello/searchqa	Not specified
SpreadsheetBench	https://huggingface.co/datasets/KAKA22/SpreadsheetBench	CC-BY-SA-4.0
DocVQA	https://huggingface.co/datasets/lmms-lab/DocVQA	Apache-2.0 on source card
LiveMathematicianBench	https://huggingface.co/datasets/LiveMathematicianBench/LiveMathematicianBench	Not specified
ALFWorld	https://github.com/alfworld/alfworld	MIT
SkillOpt	https://github.com/microsoft/SkillOpt	MIT

E PROMPTS AND OPTIMIZED SKILL EXAMPLES

This section presents the prompts used for skill initialization and evolutionary operators, together with an optimized skill example. The prompts and skill example retain the section names used in our implementation. The `Global Guidance` and `Node Lists` sections specify h_s and V_s, respectively. The `Task Graphs` section specifies workflows through their applicability conditions and ordered node sequences. Consecutive nodes in these workflows define the directed edges in E_s. The term `Task Graphs` names these workflows rather than an additional graph.

E.1 PROMPTS FOR EVOLUTIONARY OPERATORS

GraphSkillEvo employs four evolutionary operator prompts to optimize graph-structured skills. The four operators are global-guidance mutation, graph-structure mutation, global-guidance crossover, and graph-structure crossover. All prompts require the LLM to return a complete graph-structured skill. The rest of this subsection provides the prompts of these four operators.

- **Global-guidance mutation.** This operator revises the global guidance of a selected parent skill according to its reflection information.

Prompt for Global-guidance Mutation

```
You are a reflection-driven non-graph revision operator for graph-structured skill
     documents.

A graph-structured skill document contains:
- a `## Global Guidance` section, which provides general guidance, principles, and
     output format, and graph-use instructions;
- a `## Node Lists` section, where each node represents a reusable subtask, reasoning
      step, tool-use step, validation step, or recovery strategy;
- a `## Task Graphs` section, where each task graph describes workflow paths that
     connect node names into executable task-solving procedures.

Your job is to revise only the `## Global Guidance` section.

Review all provided evaluation results and failed-trajectory reflections to identify
     the prevalent recurring failure patterns and any missing, misleading, ignored,
     or redundant global guidance. Revise the global guidance to address the observed
      gaps while avoiding duplication in existing guidance.

You may:
- refine the Overview;
- improve output-format instructions;
- add or revise General Principles;
- clarify how the skill should reason, verify, and recover from mistakes;
- update the graph-structured skill execution guide.

Do not intentionally change the `## Node Lists` or `## Task Graphs` sections. The
     implementation will enforce this boundary, but your response should respect it.

Keep section boundaries and structure:
- `## Global Guidance` should remain a single section, organized with concise
     markdown subsections when useful;
- it should contain guidance that applies across all nodes and workflows, not
     reminders or requirements for one specific node;
- it usually contains **`### General Principles`**, which lists concise, portable
     rules that should guide all workflows;
- it usually contains **`### Graph-structured Skill Execution Guide`**, which tells
     the agent how to use the graph-structured skill: select a task graph, follow its
      exact node names in order, and apply the matching node instructions;
- do not add hidden workflows, task graph paths, node definitions, or node-specific
     procedural details to `## Global Guidance`;

Remove redundancy:
- merge duplicate or overlapping guidance.
Do not include file paths, IDs, gold values, entity names, or dataset-specific
     memorized facts.

Before returning, verify that `## Global Guidance` is concise, non-redundant, and
     contains only global guidance.
Avoid mentioning that the result was produced by mutation.
```

```
The returned `new_skill` must still be a complete graph-structured skill document
     containing:
- `## Global Guidance`,
- `## Node Lists`,
- `## Task Graphs`.

Return JSON only:
{"new_skill": "full skill document", "notes": ["brief change note"]}
```

- **Graph-structure mutation.** This operator revises the reusable nodes and task workflows of a selected parent skill using its reflection information. It may refine node instructions, add or remove nodes, and adjust task workflows while preserving the parent's global guidance and maintaining the graph structure.

Prompt for Graph-structure Mutation

```
You are a reflection-driven graph-structure revision operator for graph-structured
     skill documents.

A graph-structured skill document contains:
- a `## Global Guidance` section, which provides general guidance, principles, and
     output format;
- a `## Node Lists` section, where each node represents a reusable subtask, reasoning
      step, tool-use step, validation step, or recovery strategy;
- a `## Task Graphs` section, where each task graph describes workflow paths that
     connect node names into executable task-solving procedures.

Your job is to revise only the graph structure of the selected parent skill: the `##
     Node Lists` section and the `## Task Graphs` section.

Review all provided evaluation results and failed-trajectory reflections to identify
     the prevalent recurring failure patterns and any missing, misleading, ignored,
     or redundant graph steps. Revise the graph to address the observed gaps while
     avoiding duplication in existing nodes or workflows.

You may:
- refine node instructions to make them more concrete and actionable;
- add reusable nodes for missing subtasks, checks, fallback behavior, or failure-
     handling steps;
- merge redundant nodes;
- remove obsolete, misleading, or unused nodes;
- reorder or reroute task graph paths;
- add, delete, split, merge, or adjust workflow branches.

Keep the `## Global Guidance` section unchanged.

Keep section boundaries and structure:
- `## Node Lists` should be structured as one `### <Node Name>` heading per reusable
     execution step, followed by the instructions the agent should follow for that
     node;
- `## Task Graphs` should be structured as task-type workflows, each with a `**Use
     when:**` condition and a `**Workflow:**` list;
- workflow items must be exact node names only;

Maintain graph consistency:
- every node referenced in `## Task Graphs` must appear as a node heading in `## Node
      Lists`;
- every node in `## Node Lists` must be used by at least one task graph workflow;
- deleted or merged nodes must not remain in any task graph path;
- node names must be exactly consistent between node headings and graph paths;
- task graph paths should remain executable, ordered, and non-contradictory.

Remove redundancy:
- merge duplicate or near-duplicate nodes before returning;
- merge overlapping workflows;
- remove repeated or equivalent instructions inside nodes, across nodes, or across
     workflows;

Do not include file paths, IDs, gold values, entity names, or dataset-specific
     memorized facts.
```

```
Avoid mentioning that the result was produced by mutation.

The returned `new_skill` must still be a complete graph-structured skill document
     containing:
- `## Global Guidance`,
- `## Node Lists`,
- `## Task Graphs`.

Return JSON only:
{"new_skill": "full skill document", "notes": ["brief change note"]}
```

- **Global-guidance crossover.** This operator recombines useful global guidance from two selected parent skills while preserving the graph structure of one parent.

Prompt for Global-guidance Crossover

```
You are a non-graph crossover operator for graph-structured skill documents.

A graph-structured skill document contains:
- a `## Global Guidance` section, which provides general guidance, principles, and
     output format;
- a `## Node Lists` section, where each node represents a reusable subtask, reasoning
      step, tool-use step, validation step, or recovery strategy;
- a `## Task Graphs` section, where each task graph describes workflow paths that
     connect node names into executable task-solving procedures.

Your job is to perform crossover only on the `## Global Guidance` section.

Compare Parent A and Parent B and identify useful non-graph guidance.

Then recombine the non-graph guidance into one complete child skill. You may:
- import useful principles from Parent B;
- replace weak or vague guidance from Parent A;
- merge complementary instructions;
- remove duplicated, conflicting, or overly specific prose;
- improve clarity and concision.

Do not intentionally change Parent A's `## Node Lists` or `## Task Graphs` sections.
     The implementation will enforce this boundary, but your response should respect
     it.

Keep section boundaries and structure:
- `## Global Guidance` should remain a single section, organized with concise
     markdown subsections when useful;
- it should contain guidance that applies across all nodes and workflows, not
     reminders or requirements for one specific node;
- it usually contains **`### General Principles`**, which lists concise, portable
     rules that should guide all workflows;
- it usually contains **`### Graph-structured Skill Execution Guide`**, which tells
     the agent how to use the graph-structured skill: select a task graph, follow its
      exact node names in order, and apply the matching node instructions;
- do not add hidden workflows, task graph paths, node definitions, or node-specific
     procedural details to `## Global Guidance`;

Remove redundancy:
- merge duplicate or overlapping guidance.

Preserve concrete, actionable guidance. Do not include file paths, IDs, gold values,
     entity names, or dataset-specific memorized facts.

Before returning, verify that `## Global Guidance` is concise, non-redundant, and
     contains only global guidance.

Avoid mentioning that the result was produced by crossover.

The returned `new_skill` must still be a complete graph-structured skill document
     containing:
- `## Global Guidance`,
- `## Node Lists`,
- `## Task Graphs`.

Return JSON only:
```

```
{"new_skill": "full skill document", "notes": ["brief change note"]}
```

- **Graph-structure crossover.** This operator recombines useful reusable nodes and task workflows from two selected parent skills while preserving the global guidance of one parent.

Prompt for Graph-structure Crossover

```
You are a graph-structure crossover operator for graph-structured skill documents.

A graph-structured skill document contains:
- a `## Global Guidance` section, which provides general guidance, principles, and
    output format;
- a `## Node Lists` section, where each node represents a reusable subtask, reasoning
     step, tool-use step, validation step;
- a `## Task Graphs` section, where each task graph describes workflow paths that
    connect node names into executable task-solving procedures.

Your job is to perform crossover only on the graph structure: the `## Node Lists`
    section and the `## Task Graphs` section.

Recombine the graph structure into one complete child skill. You may:
- import useful nodes from Parent B into Parent A's graph;
- replace weak nodes with stronger alternatives;
- merge overlapping nodes from both parents;
- remove redundant or conflicting nodes;
- exchange or recombine task graph paths;
- build a better workflow by combining complementary subgraph fragments.

Keep Parent A's `## Global Guidance` section unchanged. The implementation will
    enforce this boundary, but your response should respect it.

Keep section boundaries and structure:
- `## Node Lists` should be structured as one `### <Node Name>` heading per reusable
    execution step, followed by the instructions the agent should follow for that
    node;
- `## Task Graphs` should be structured as task-type workflows, each with a `**Use
    when:**` condition and a `**Workflow:**` list;
- workflow items must be exact node names only;

Maintain graph consistency:
- every node referenced in `## Task Graphs` must appear as a node heading in `## Node
     Lists`;
- every node in `## Node Lists` must be used by at least one task graph workflow;
- deleted or merged nodes must not remain in any task graph path;
- node names must be exactly consistent between node headings and graph paths;
- task graph paths should remain executable, ordered, and non-contradictory.

Remove redundancy:
- merge duplicate or near-duplicate nodes before returning;
- merge overlapping workflows;
- remove repeated or equivalent instructions inside nodes, across nodes, or across
    workflows;

Preserve concrete, actionable guidance. Do not include file paths, IDs, gold values,
    entity names, or dataset-specific memorized facts.

Avoid mentioning that the result was produced by crossover.

The returned `new_skill` must still be a complete graph-structured skill document
    containing:
- `## Global Guidance`,
- `## Node Lists`,
- `## Task Graphs`.

Return JSON only:
{"new_skill": "full skill document", "notes": ["brief change note"]}
```

E.2 PROMPT FOR SKILL INITIALIZATION

Prompt for Skill Initialization

```
You create initial graph-structured skill documents for an agent benchmark.

Generate complete, diverse skill documents that can be used directly by the target
    agent.
Keep each skill self-contained and practical. Use the benchmark context to infer the
    main task types, design suitable workflows for them, and convert recurring
    execution steps into reusable graph nodes.

Every skill must preserve this graph-structured organization:
1. A `## Global Guidance` section containing instructions that apply across all nodes
    and workflows.
2. A `## Node Lists` section containing reusable subtask nodes. Node headings should be
     markdown headings such as `### Parse Goal`.
3. A `## Task Graphs` section containing task-type workflows that connect node names
    into executable task-solving procedures.

Use this markdown layout for each skill document:

`# <Skill Name>`

`## Global Guidance`

`## Node Lists`

`## Task Graphs`

The optional title may appear before `## Global Guidance`, but the three required
    sections must appear exactly in this order.

Section requirements:

### `## Global Guidance`
This section contains instructions that apply across all nodes and workflows, including
     how to use the graph-structured skill.

Organize it with concise markdown subsections. Prefer the following subsections when
    they are useful:
- `### Overview`: briefly state the benchmark and the agent's role.
- `### General Principles`: list concise global rules that should guide all workflows.
- `### Graph-structured Skill Execution Guide`: explain how to execute the skill as a
    graph: select the relevant task graph using its `Use when` condition, execute the
    listed node names in order, and apply the instructions under each matching node in
     `## Node Lists`.

The `### General Principles` subsection should contain portable rules, not a hidden
    workflow. Put step-by-step procedures, search actions, computations, validations,
    and recovery routines into `## Node Lists` nodes, then connect them in `## Task
    Graphs`.

The graph-execution guide should be short and explicit. It may state that:
- `## Global Guidance` applies across all nodes and workflows;
- `## Node Lists` defines reusable execution steps;
- `## Task Graphs` chooses and orders those steps for each task type;
- a numbered workflow should use `1. A`, `2. B`, `3. C` formatting, where `A`, `B`, and
     `C` are exact node names in execution order.

### `## Node Lists`
This section defines the node library. Each node is a reusable step that can appear in
    one or more workflows inside the task graphs.

Organize it as:
- one markdown heading per node, for example `### Parse Request`, `### Gather Evidence
    `, or `### Verify Answer`;
- under each node heading, concise bullets with instructions the agent should follow
    while executing that node;

Effective nodes should:
- have short, action-oriented names that can be referenced exactly from task graphs;
- represent reusable subtasks, reasoning steps, tool-use steps, validation steps, or
    recovery strategies;
- be specific enough to guide behavior, but general enough to transfer across benchmark
     instances;
```

```
- separate distinct responsibilities when the order matters, such as parsing the goal,
    locating evidence, computing, and validating;

Avoid nodes that are empty, redundant, purely decorative, or tied to a single example.
    Do not create a long block of prose under one node when several reusable nodes
    would make the graph clearer.

### `## Task Graphs`
This section defines task graphs as collections of workflows, where each workflow is an
     executable path made from node names in `## Node Lists`.

Organize it as one or more task-type subsections:
- each task graph should have a heading such as `### Direct Evidence Question` or `###
    Multi-Step Calculation`;
- include a `**Use when:**` line describing when that workflow applies;
- include a `**Workflow:**` block formatted as `1. A`, `2. B`, `3. C`, where `A`, `B`,
    and `C` are exact node names from `## Node Lists`;
- each numbered item should be an exact node name from `## Node Lists`;

Maintain graph consistency:
- every node referenced in `## Task Graphs` must appear as a node heading in `## Node
    Lists`;
- node names must match exactly between node headings and graph paths;
- every retained node should be useful for at least one task graph or clearly reusable;
- task graph paths should be ordered, executable, and non-contradictory.

Return JSON only:
{"skills": ["full skill document", "..."]}
```

E.3 Example of Optimized Graph-Structured Skills

This subsection presents an example of optimized graph-structured skills produced by Graph-SkillEvo.

Example of optimized graph-structured skill: Skill for Spreadsheet

```
# Spreadsheet Formula, Lookup, and Reporting Skill

## Global Guidance

### Overview
Use this skill for SpreadsheetBench tasks that require formulas, lookups, cross-sheet
    transfers, totals, summaries, or light presentation changes tied to computed
    spreadsheet outputs. In this benchmark, the deliverable is the modified workbook,
    so prioritize making the target cells contain the correct final workbook results.

### Output Requirements
- Return only a single complete ```python ... ``` fenced block containing the full
    script.
- Do not include explanation, comments outside the code block, or partial / placeholder
     code.
- Read from `INPUT_PATH`, write the modified workbook to `OUTPUT_PATH`, and preserve
    unrelated workbook content.

### General Principles
- Use only `openpyxl` and `pandas`.
- Prefer `openpyxl` for preserving workbook structure, writing cells, and applying
    basic formatting.
- Do not rely on `openpyxl` to evaluate Excel formulas.
- In SpreadsheetBench cell-level tasks, prefer writing final static values whenever
    formula evaluation would otherwise be required to make the workbook show the
    correct result.
- If the user names an Excel method or function such as INDEX/MATCH, SUMIF, IF, FILTER,
     or lookup formulas, treat that as the logic to reproduce; you may compute the
    result in Python instead of writing formula strings unless the task clearly
    requires formulas to remain in the sheet.
- If final displayed values are what matter, writing unevaluated formulas is
    insufficient.
- Infer target ranges, anchors, and fill extents from the actual workbook, not only the
     truncated preview.
```

```
- Do not stop range filling at the first blank cell if surrounding structure shows the
    table or requested output continues.
- For matrix-style transfers, resolve both row keys and column keys from headers and
    fill the full applicable destination area.
- For list outputs, write all qualifying items contiguously in the destination region,
    including rows beyond currently populated output cells when needed.
- Normalize header and key matching conservatively: trim spaces, compare case-
    insensitively, and handle common text variations, dates, and symbols such as tick
    marks.
- Preserve unrelated formulas, formatting, and sheets.
- Apply only the formatting explicitly requested or clearly necessary for the requested
     output.

### Verification and Recovery
- Before saving, verify that the intended target cells were actually written.
- Reopen the output workbook and confirm the target cells contain concrete expected-
    type results: numbers, text, or blanks as requested, not just formula strings when
     static results are needed.
- Verify fill extent carefully, including the last applicable row or column.
- If a lookup or summary output seems empty, re-check sheet selection, header detection
    , destination anchors, and whether blank cells should be written as `0`, empty
    string, or left empty according to the instruction.

### Graph-structured Skill Execution Guide
- `## Global Guidance` applies to every node and workflow.
- `## Node Lists` defines reusable steps for formula and reporting tasks.
- `## Task Graphs` chooses the right ordered workflow for each task type.
- Select the graph using its `Use when` line, then execute each listed node in order.
- Follow node instructions exactly by node name.
- When a workflow offers a choice between writing formulas and computing results,
    choose the path that will make the saved workbook contain the correct final target
    -cell contents under SpreadsheetBench evaluation.

## Node Lists

### Parse Task And Output Requirements
- Classify the request as an existing-cell fill, cross-sheet lookup or join, summary/
    report generation, split-or-expand transformation, or direct static transformation
     into a destination area.
- Extract explicit target cells or ranges, source sheets, destination sheets, lookup
    keys, requested calculations, blank-on-error behavior, and any formatting requests
    .
- If the instruction is phrased as an Excel-formula question, treat the named formula
    as the logic to reproduce; do not assume the sheet must retain a live formula
    unless the request clearly requires that.

### Inspect Workbook Context
- Load the workbook and inspect the actual used range of each relevant sheet.
- Examine headers, neighboring columns, existing formulas, placeholder cells, lookup
    tables, summary areas, date/report cells, and nearby styles that may need to be
    matched.
- Confirm the real workbook structure directly rather than relying only on the preview.

### Identify Explicit Target Cells
- Resolve the exact answer cell, output range, or destination block named by the
    instruction, workbook labels, or benchmark answer position.
- For single-cell or short fixed-range tasks, center the script around writing those
    exact coordinates rather than only constructing general helpers.
- Record these target coordinates for later save-time and reopen-time verification.

### Resolve Headers Keys And Target Area
- Map user-described headers, labels, and columns to actual workbook columns using
    conservative normalized matching.
- Resolve exact source and destination sheets, key columns, value columns, date columns
    , selection cells, and the area that should be overwritten.
- For cross-sheet work, confirm how source rows align to destination rows and identify
    any interval fields such as start and end dates.

### Determine Fill Extent And Rewrite Strategy
- Infer the full row or column extent from related populated columns, existing table
    structure, templates, and the requested output area.
- Decide whether the task is an in-place fill, a generated report block, or a rewritten
     exploded output that should replace prior contents.
- When regenerating a destination block, plan to clear stale formulas or old rows that
    would otherwise remain below or beside the new results.

### Choose Output Method
```

```
- Prefer static Python-computed values whenever unevaluated formulas would leave the
    saved workbook incorrect, especially for cell-level tasks.
- Use Excel formulas only when the instruction clearly requires formulas to remain in
    the sheet; even then, prepare equivalent Python logic for recovery.
- For direct-reference tasks or blank-if-source-blank behavior, simple value transfer
    may be the correct final method.

### Build Script Skeleton
- Build a complete executable script using `openpyxl` and `pandas`, reading from `
    INPUT_PATH` and writing to `OUTPUT_PATH`.
- Organize the script into workbook loading, helper functions, source discovery,
    computation or formula writing, formatting, save, and verification steps.
- Ensure the returned script is complete and not truncated.

### Interpret Formula Logic In Python
- Translate the user-described spreadsheet logic into Python-ready rules before writing
     output values.
- Handle common patterns such as direct cell reference with blank propagation, OR-style
     multi-selection criteria, `All` meaning no filter, workday/date-window
    aggregation, lookup matching, and conditional blanks.
- Build reusable helpers for normalized text comparison, date coercion, aggregation,
    and default-value handling.

### Write Formula Pattern
- Only when formulas must remain, write the target formula or formula pattern with
    correct relative or absolute references and requested blank/error handling.
- If the pattern extends across a range, fill it across the full resolved extent.
- Use this node only after the equivalent Python fallback logic is understood.

### Compute And Write Static Results
- Compute final values directly in Python for exact lookups, multi-criteria matches,
    date-range checks, totals, derived metrics, category mapping, sorting, grouped
    outputs, and other requested transformations.
- Write concrete results into every target cell in the resolved extent, including exact
     benchmark answer cells for single-cell tasks.
- When no match is found or the instruction requests blanks, write blank, `0`, empty
    string, or leave empty exactly as required.

### Expand Records Into Output Rows
- Parse source fields that encode multiple values, conditions, or list members and
    expand them into the required row-wise output structure.
- Produce one destination row per required emitted result, copying base fields and
    leaving unspecified extracted fields blank.
- Preserve source-driven ordering or destination-template ordering when possible.

### Create Or Update Summary Output
- Create or locate the destination report or output block and write headers, grouped
    sections, or paired/list outputs only when the task calls for them.
- Use this node for summary sheets, compact report areas, and designated destination
    blocks for generated result lists.
- Preserve unrelated cells outside the requested output area.

### Clear Target Output Area
- Before rewriting generated lists, reports, or expanded outputs, clear obsolete
    contents in the destination block that would otherwise leave stale rows, leftover
    formulas, or partial prior results.
- Keep headers and unrelated regions intact.
- Use the resolved output area and prior populated extent to determine what should be
    cleared.

### Apply Requested Formatting
- Apply only the formatting explicitly requested, such as bold text, alignment, number
    formats, widths, freeze panes, or copying fill/style from adjacent reference cells
    .
- When formatting should match a nearby column or header, clone the needed style
    attributes from the reference cells rather than inventing new formatting.
- Preserve existing formatting elsewhere.

### Save Workbook
- Save the modified workbook to `OUTPUT_PATH`.

### Reopen Verify And Recover
- Reopen the saved workbook and inspect the explicit target cells, target range, and
    final fill extent directly.
- Confirm the workbook contains concrete expected-type outputs and that required cells
    are not left as `None` or unevaluated formulas when static results are needed.
```

```
- If verification fails, recompute from the Python logic, overwrite the target area
    with static values, clear stale leftover cells if necessary, resave, and recheck
    before finalizing.

## Task Graphs

### Cell-Level Static Fill Or Direct Reference
**Use when:** The task asks to fill one cell or an existing small range in place,
    including formula-like requests where the benchmark needs final displayed values
    rather than a live formula.

**Workflow:**
1. Parse Task And Output Requirements
2. Inspect Workbook Context
3. Identify Explicit Target Cells
4. Resolve Headers Keys And Target Area
5. Determine Fill Extent And Rewrite Strategy
6. Choose Output Method
7. Build Script Skeleton
8. Interpret Formula Logic In Python
9. Compute And Write Static Results
10. Apply Requested Formatting
11. Save Workbook
12. Reopen Verify And Recover

### Formula Must Remain In Sheet
**Use when:** The task clearly requires formulas to remain visible in the target cells
    or to preserve a formula pattern in the workbook.

**Workflow:**
1. Parse Task And Output Requirements
2. Inspect Workbook Context
3. Identify Explicit Target Cells
4. Resolve Headers Keys And Target Area
5. Determine Fill Extent And Rewrite Strategy
6. Choose Output Method
7. Build Script Skeleton
8. Interpret Formula Logic In Python
9. Write Formula Pattern
10. Apply Requested Formatting
11. Save Workbook
12. Reopen Verify And Recover

### Cross-Sheet Lookup Or Multi-Criteria Match
**Use when:** The task asks to pull, match, or aggregate values from another sheet or
    table using one or more keys, categories, flags, or date conditions.

**Workflow:**
1. Parse Task And Output Requirements
2. Inspect Workbook Context
3. Identify Explicit Target Cells
4. Resolve Headers Keys And Target Area
5. Determine Fill Extent And Rewrite Strategy
6. Choose Output Method
7. Build Script Skeleton
8. Interpret Formula Logic In Python
9. Compute And Write Static Results
10. Apply Requested Formatting
11. Save Workbook
12. Reopen Verify And Recover

### Summary Sheet Or Report Block
**Use when:** The task asks to create or update a summary sheet, grouped totals section
    , compact report area, or another dedicated generated output block.

**Workflow:**
1. Parse Task And Output Requirements
2. Inspect Workbook Context
3. Identify Explicit Target Cells
4. Resolve Headers Keys And Target Area
5. Determine Fill Extent And Rewrite Strategy
6. Choose Output Method
7. Build Script Skeleton
8. Create Or Update Summary Output
9. Clear Target Output Area
10. Interpret Formula Logic In Python
11. Compute And Write Static Results
12. Apply Requested Formatting
```

```
13. Save Workbook
14. Reopen Verify And Recover

### Split Or Expand Encoded Fields To Rows
**Use when:** The task asks to parse a source field and expand encoded values, repeated
      members, or condition fragments into multiple destination rows or columns in an
     output block.

**Workflow:**
1. Parse Task And Output Requirements
2. Inspect Workbook Context
3. Identify Explicit Target Cells
4. Resolve Headers Keys And Target Area
5. Determine Fill Extent And Rewrite Strategy
6. Choose Output Method
7. Build Script Skeleton
8. Expand Records Into Output Rows
9. Clear Target Output Area
10. Compute And Write Static Results
11. Apply Requested Formatting
12. Save Workbook
13. Reopen Verify And Recover
```

E.4 Example of an Unstructured Counterpart

This subsection presents the unstructured counterpart of the graph-structured skill shown above, illustrating the conversion procedure described in Section 5.1. We construct the unstructured counterpart by removing the explicit workflow organization provided by the graph-structured representation while retaining task-general guidance and node-level instructions. The resulting counterpart retains the instructions for individual execution steps but omits the explicit applicability conditions and ordered node sequences through which the graph-structured representation provides workflow guidance. This comparison therefore examines the contribution of that explicit workflow guidance, rather than comparing graph and non-graph representations that preserve equivalent workflow information.

Example of an unstructured counterpart of a graph-structured skill

```
# Spreadsheet Formula, Lookup, and Reporting Skill

## Global Guidance

### Overview
Use this skill for SpreadsheetBench tasks that require formulas, lookups, cross-sheet
     transfers, totals, summaries, or light presentation changes tied to computed
     spreadsheet outputs. In this benchmark, the deliverable is the modified workbook,
     so prioritize making the target cells contain the correct final workbook results.

### Output Requirements
- Return only a single complete ```python ... ``` fenced block containing the full
     script.
- Do not include explanation, comments outside the code block, or partial / placeholder
      code.
- Read from `INPUT_PATH`, write the modified workbook to `OUTPUT_PATH`, and preserve
     unrelated workbook content.

### General Principles
- Use only `openpyxl` and `pandas`.
- Prefer `openpyxl` for preserving workbook structure, writing cells, and applying
     basic formatting.
- Do not rely on `openpyxl` to evaluate Excel formulas.
- In SpreadsheetBench cell-level tasks, prefer writing final static values whenever
     formula evaluation would otherwise be required to make the workbook show the
     correct result.
- If the user names an Excel method or function such as INDEX/MATCH, SUMIF, IF, FILTER,
      or lookup formulas, treat that as the logic to reproduce; you may compute the
     result in Python instead of writing formula strings unless the task clearly
     requires formulas to remain in the sheet.
- If final displayed values are what matter, writing unevaluated formulas is
     insufficient.
```

```
- Infer target ranges, anchors, and fill extents from the actual workbook, not only the
    truncated preview.
- Do not stop range filling at the first blank cell if surrounding structure shows the
    table or requested output continues.
- For matrix-style transfers, resolve both row keys and column keys from headers and
    fill the full applicable destination area.
- For list outputs, write all qualifying items contiguously in the destination region,
    including rows beyond currently populated output cells when needed.
- Normalize header and key matching conservatively: trim spaces, compare case-
    insensitively, and handle common text variations, dates, and symbols such as tick
    marks.
- Preserve unrelated formulas, formatting, and sheets.
- Apply only the formatting explicitly requested or clearly necessary for the requested
    output.

### Verification and Recovery
- Before saving, verify that the intended target cells were actually written.
- Reopen the output workbook and confirm the target cells contain concrete expected-
    type results: numbers, text, or blanks as requested, not just formula strings when
    static results are needed.
- Verify fill extent carefully, including the last applicable row or column.
- If a lookup or summary output seems empty, re-check sheet selection, header detection
    , destination anchors, and whether blank cells should be written as `0`, empty
    string, or left empty according to the instruction.

- Classify the request as an existing-cell fill, cross-sheet lookup or join, summary/
    report generation, split-or-expand transformation, or direct static transformation
    into a destination area.
- Extract explicit target cells or ranges, source sheets, destination sheets, lookup
    keys, requested calculations, blank-on-error behavior, and any formatting requests
    .
- If the instruction is phrased as an Excel-formula question, treat the named formula
    as the logic to reproduce; do not assume the sheet must retain a live formula
    unless the request clearly requires that.
- Load the workbook and inspect the actual used range of each relevant sheet.
- Examine headers, neighboring columns, existing formulas, placeholder cells, lookup
    tables, summary areas, date/report cells, and nearby styles that may need to be
    matched.
- Confirm the real workbook structure directly rather than relying only on the preview.
- Resolve the exact answer cell, output range, or destination block named by the
    instruction, workbook labels, or benchmark answer position.
- For single-cell or short fixed-range tasks, center the script around writing those
    exact coordinates rather than only constructing general helpers.
- Record these target coordinates for later save-time and reopen-time verification.
- Map user-described headers, labels, and columns to actual workbook columns using
    conservative normalized matching.
- Resolve exact source and destination sheets, key columns, value columns, date columns
    , selection cells, and the area that should be overwritten.
- For cross-sheet work, confirm how source rows align to destination rows and identify
    any interval fields such as start and end dates.
- Infer the full row or column extent from related populated columns, existing table
    structure, templates, and the requested output area.
- Decide whether the task is an in-place fill, a generated report block, or a rewritten
    exploded output that should replace prior contents.
- When regenerating a destination block, plan to clear stale formulas or old rows that
    would otherwise remain below or beside the new results.
- Prefer static Python-computed values whenever unevaluated formulas would leave the
    saved workbook incorrect, especially for cell-level tasks.
- Use Excel formulas only when the instruction clearly requires formulas to remain in
    the sheet; even then, prepare equivalent Python logic for recovery.
- For direct-reference tasks or blank-if-source-blank behavior, simple value transfer
    may be the correct final method.
- Build a complete executable script using `openpyxl` and `pandas`, reading from `
    INPUT_PATH` and writing to `OUTPUT_PATH`.
- Organize the script into workbook loading, helper functions, source discovery,
    computation or formula writing, formatting, save, and verification steps.
- Ensure the returned script is complete and not truncated.
- Translate the user-described spreadsheet logic into Python-ready rules before writing
    output values.
- Handle common patterns such as direct cell reference with blank propagation, OR-style
    multi-selection criteria, `All` meaning no filter, workday/date-window
    aggregation, lookup matching, and conditional blanks.
- Build reusable helpers for normalized text comparison, date coercion, aggregation,
    and default-value handling.
- Only when formulas must remain, write the target formula or formula pattern with
    correct relative or absolute references and requested blank/error handling.
- If the pattern extends across a range, fill it across the full resolved extent.
- Use this node only after the equivalent Python fallback logic is understood.
```

```
- Compute final values directly in Python for exact lookups, multi-criteria matches,
    date-range checks, totals, derived metrics, category mapping, sorting, grouped
    outputs, and other requested transformations.
- Write concrete results into every target cell in the resolved extent, including exact
     benchmark answer cells for single-cell tasks.
- When no match is found or the instruction requests blanks, write blank, `0`, empty
    string, or leave empty exactly as required.
- Parse source fields that encode multiple values, conditions, or list members and
    expand them into the required row-wise output structure.
- Produce one destination row per required emitted result, copying base fields and
    leaving unspecified extracted fields blank.
- Preserve source-driven ordering or destination-template ordering when possible.
- Create or locate the destination report or output block and write headers, grouped
    sections, or paired/list outputs only when the task calls for them.
- Use this node for summary sheets, compact report areas, and designated destination
    blocks for generated result lists.
- Preserve unrelated cells outside the requested output area.
- Before rewriting generated lists, reports, or expanded outputs, clear obsolete
    contents in the destination block that would otherwise leave stale rows, leftover
    formulas, or partial prior results.
- Keep headers and unrelated regions intact.
- Use the resolved output area and prior populated extent to determine what should be
    cleared.
- Apply only the formatting explicitly requested, such as bold text, alignment, number
    formats, widths, freeze panes, or copying fill/style from adjacent reference cells
    .
- When formatting should match a nearby column or header, clone the needed style
    attributes from the reference cells rather than inventing new formatting.
- Preserve existing formatting elsewhere.
- Save the modified workbook to `OUTPUT_PATH`.
- Reopen the saved workbook and inspect the explicit target cells, target range, and
    final fill extent directly.
- Confirm the workbook contains concrete expected-type outputs and that required cells
    are not left as `None` or unevaluated formulas when static results are needed.
- If verification fails, recompute from the Python logic, overwrite the target area
    with static values, clear stale leftover cells if necessary, resave, and recheck
    before finalizing.
```