FROM SKILLS TO STAGES: STAGE-AWARE GRAPH RETRIEVAL FOR RELEVANT SKILLS IN LARGE-SCALE AGENT SKILL LIBRARIES

Anonymous authors
Paper under double-blind review

ABSTRACT

Agents based on Large Language Models (LLMs) increasingly rely on reusable skills. As these agents become deeply integrated with personal applications, web browsers, and various interactive interfaces, the scale of skill libraries has expanded to thousands of entries. Currently, numerous methods have attempted to address this challenge through skill retrieval. However, existing approaches treat each skill as an opaque monolith, matching solely based on holistic descriptions or global features. Since a skill actually consists of multiple execution stages—each with its own independent goals, preconditions, and execution outcomes—compressing this internal structure into a single representation makes it impossible to distinguish between skills that have similar descriptions but different execution workflows. To address the aforementioned limitations, we propose the Stage-aware Skill Graph (SSG), which explicitly models two node hierarchies during graph construction: the skill level and the internal stage level. During inference, we introduce a stage-aware retrieval mechanism that combines stage seed selection, stage-aware PageRank, and stage-aware query rewriting to retrieve complementary skills for different stages of task execution. Experiments on ALFWorld, SkillsBench, and AgentSkillOS with three representative models demonstrate the effectiveness of our approach. SSG achieves a task success rate of 76.4% on ALFWorld, outperforming the strongest baseline by 8.5 percentage points, and achieves up to a 7.8 percentage-point improvement over the strongest baseline on SkillsBench. Across all three benchmarks and model settings, SSG consistently improves end-to-end task performance, highlighting the benefit of stage-aware skill retrieval for multi-stage task execution.

1 INTRODUCTION

Large language model (LLM) agents increasingly rely on reusable skills that package procedures for recurring tasks (Wang et al., 2023; Li et al., 2026c). As agents integrate deeply with personal applications, web browsers, and various interactive interfaces, skill libraries have expanded to thousands of entries (Kang et al., 2026; Li et al., 2026c). To manage this scale, agents typically retrieve relevant skills on demand. Existing approaches support this selection through vector similarity (Qin et al., 2024), capability hierarchies (Li et al., 2026a; Meng et al., 2026), or graph relations between skills (Liu et al., 2026a; Bai et al., 2026).

However, these approaches typically represent and match skills as **holistic entities**, creating a granularity mismatch between task intent and skill representation. Complex tasks often consist of multiple local execution stages, while the relevant evidence may exist only within a particular stage of a skill. Matching against the entire skill can therefore dilute such local signals, causing semantically similar skills to be confused and leading to incorrect retrieval and execution failures.

As illustrated in Fig. 1, consider the task "Analyze penguin morphological data, identify species-specific characteristics, produce a report, and create conference slides." The task can be decomposed into four stages: data analysis, visualization, report generation, and slide creation. Some intermediate stages, such as scientific visualization, are necessary for completing the task but are not explicitly specified in the task description. Holistic skill matching can overlook such local ex-

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.

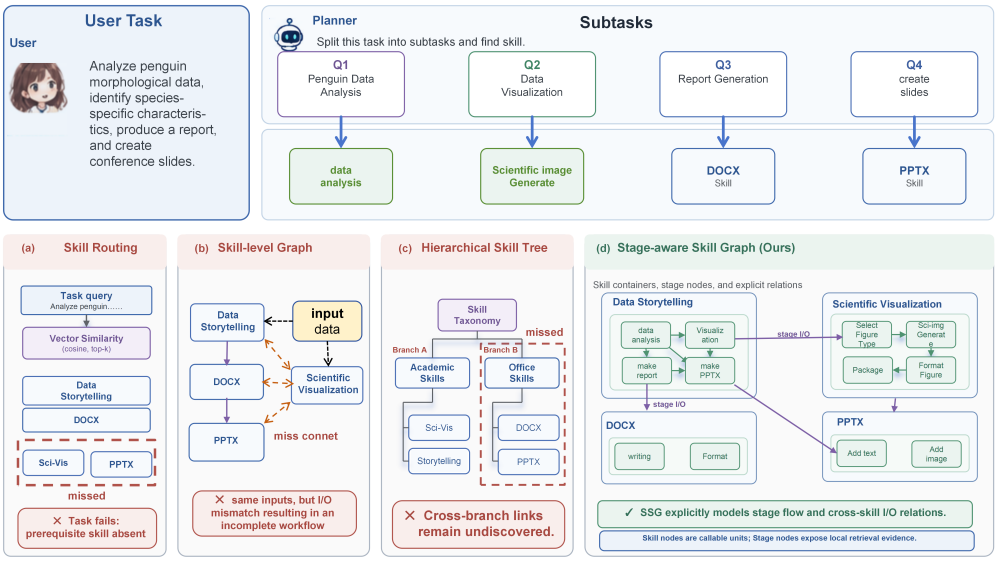

Figure 1: Comparison of skill organization and retrieval paradigms for the task "Analyze penguin morphological data and create a report and conference slides." The planner decomposes the task into four stages, Q1–Q4, corresponding to data analysis, visualization, report generation, and slide creation. Existing approaches exhibit different limitations: (a) *Skill Routing* performs holistic semantic matching and can overlook intermediate stages that are necessary for task completion; (b) *Skill-level Graph* models relations between skills but does not explicitly represent internal stages, making stage-level evidence difficult to exploit; (c) *Hierarchical Skill Tree* captures hierarchical relations but provides limited support for cross-branch stage dependencies; (d) our *Stage-aware Skill Graph (SSG)* explicitly represents skill and stage nodes, together with intra-skill flow and cross-skill I/O relations. Stage nodes expose fine-grained retrieval evidence, while skill nodes remain the callable execution units, enabling complete and executable task orchestration.

ecution requirements. Vector-based methods may be distracted by surface-level similarity; existing skill graphs primarily model relations between skills without explicitly representing their internal stages; and hierarchical skill structures provide limited support for cross-branch stage dependencies. These limitations make it difficult to exploit fine-grained stage evidence and connect related stages across skills.

To address these limitations, we propose the **Stage-aware Skill Graph (SSG)**. During graph construction, SSG explicitly models two node levels: the skill level and its internal stage level. In addition to skill nodes that serve as callable units, SSG introduces stage nodes to capture the fine-grained execution steps within a skill. These nodes are connected via typed containment, flow, dependency, and semantic edges, preserving both the internal execution order and cross-skill stage associations. This structure allows retrieval systems to directly utilize evidence from within skills; as long as a query mentions an intermediate stage, the system can locate the skill containing it.

To better leverage the Stage-aware Skill Graph during inference, we propose **stage-aware-retrieval**. This strategy consists of three core steps: (1) *Stage Seed Selection*, which identifies key stages from the task and graph as retrieval seeds; (2) *Stage-aware PageRank*, which runs a propagation algorithm aware of stages, allowing stage evidence to spread through stage-level relations before aggregating at the skill level; and (3) *Stage-aware Query Rewrite*, which rewrites queries to better match the structured fields of stages. Through this process, the system can retrieve complete skill chains relevant to every stage of task execution, ultimately returning only complete skills for execution.

Our main contributions are as follows:

- We introduce a novel Stage-aware Skill Graph (SSG) that represents the skill library at two node levels: skill nodes as callable units and stage nodes capturing the internal execution stages of skills. This enables the retrieval system to utilize intra-skill evidence, allowing queries mentioning intermediate stages to locate the containing skills.

- We propose a *stage-aware-retrieval* strategy that retrieves complete skill chains through stage seed selection, stage-aware PageRank, and stage-aware query rewrite, scoring fine-grained structures without directly returning them.
- We evaluate our method on the ALFWorld (Shridhar et al., 2021), SkillsBench (Li et al., 2026c), and AgentSkillOS (Li et al., 2026a) datasets using three representative models. Experiments demonstrate the effectiveness of our approach: SSG achieves a task success rate of 76.4% on ALFWorld, outperforming the strongest baseline by 8.5%, and achieves up to a 7.8% improvement over the strongest baseline on SkillsBench. Across all three benchmarks and model settings, SSG consistently improves end-to-end task performance, demonstrating the advantage of stage-aware skill retrieval for multi-stage task execution.

2 Related Work

Skill retrieval and routing. Voyager demonstrates procedural reuse through skill libraries (Wang et al., 2023). As libraries grow, SRA-Bench distinguishes retrieval, skill incorporation, and execution, while SkillRet provides large-scale retrieval evaluation and training data (Su et al., 2026; Kang et al., 2026). SkillRouter incorporates skill bodies into a retrieve-and-rerank pipeline (Zheng et al., 2026), while Field-Aware Agent Skill Retrieval separately matches name, description, and body fields before fusing their sparse and dense scores (Goulart et al., 2026). SkillDreamer generates prospective pseudo-skills to align task objectives with procedural descriptions (Liu et al., 2026b). Our proposed Stage-aware Skill Graph (SSG) makes internal execution stages explicit retrieval nodes, enabling local stage matches and their dependencies to jointly inform skill ranking.

Structured skill libraries and dependency-aware retrieval. AgentSkillOS combines capability-tree discovery with DAG orchestration (Li et al., 2026a), while SkillNet supports skill organization and routing at ecosystem scale (Liang et al., 2026b). Graph-of-Skills (GoS) combines hybrid seeding, reverse-aware Personalized PageRank (PPR), and budgeted context construction to retrieve dependency-aware bundles over a typed skill graph (Liu et al., 2026a). PPR also underlies HippoRAG's retrieval over relational memory (Gutiérrez et al., 2024). SkillDAG exposes matches, typed neighbors, and conflicts through an agent-callable interface and updates relations using execution evidence (Bai et al., 2026). GoSkills organizes retrieved skills into role-labeled execution contexts (Zeng et al., 2026). Where GoS and SkillDAG use complete skills as graph nodes, our SSG models intra-skill execution flow and cross-skill dependencies at the stage level, exposing relational evidence hidden by skill-level graphs.

Intra-skill structure and fine-grained evidence. SSL separates scheduling, scene-level structure, and atomic actions, and enriches skill embeddings with structured fields, including scene goals and interface information, for discovery (Liang et al., 2026a). CUA-Skill represents skills through parameterized execution and composition graphs (Chen et al., 2026). Most closely related, SkillRAE builds a community–skill–subunit graph, projects subunit matches through extraction links into skill scores, and compiles selected and rescued evidence into task-specific context (Meng et al., 2026). Building on SSL's scene-level abstraction, our SSG introduces a dedicated stage-subgraph propagation channel over execution flow and cross-skill dependencies. It aggregates the propagated evidence to parent skills and fuses it with a skill-seeded channel over the full graph, allowing stage relations to influence ranking beyond direct local matches. Complete skills remain the retrieval outputs, while stage relations guide their ranking.

3 Method

Existing skill retrieval methods usually construct graphs by treating a skill as an atomic unit. This practice obscures the internal execution stages of skills and often conflates skills that have similar descriptions but fundamentally different execution workflows. To overcome the limitation of hidden internal execution logic, precise retrieval must descend to fine-grained stages and reconstruct the complete task execution chain by matching cross-skill input/output (I/O) dependencies.

Figure 2 illustrates the overall pipeline of the proposed Stage-Aware Skill Graph. The left side shows the offline construction of SSG, while the right side shows how SSG is used for online search. Based

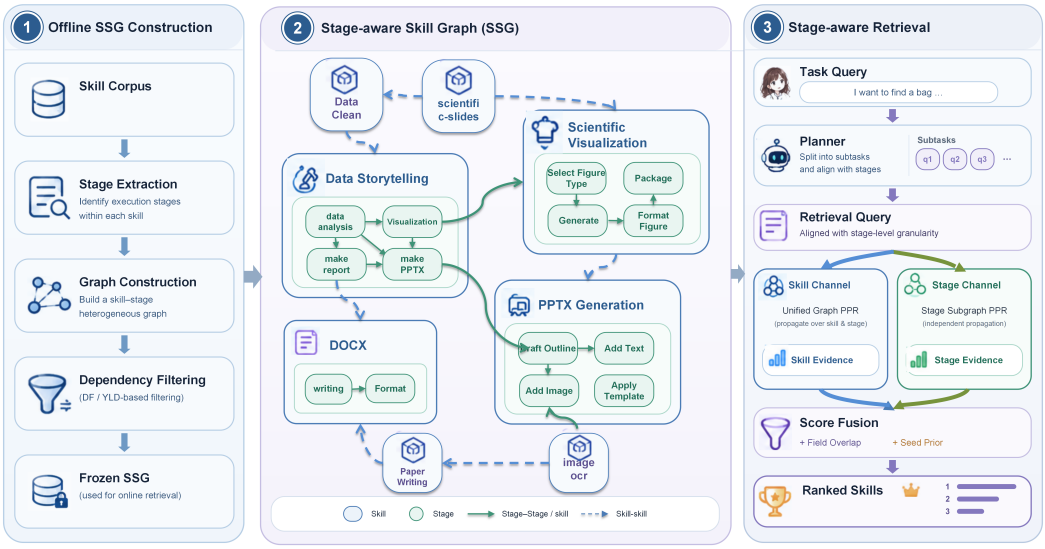

Figure 2: **Overview of the Stage-aware Skill Graph (SSG) framework.** The pipeline consists of three core phases: **(1) Offline SSG Construction**, which extracts stage-level execution structures from the skill corpus, constructs a heterogeneous graph, and prunes weak edges via dependency filtering; **(2) Stage-aware Skill Graph (SSG)**, representing fine-grained skill-skill and stage-stage relationships; and **(3) Stage-aware Retrieval**, where task queries are split into stage-aligned subtasks, processed through dual-channel PPR (stage-aware PPR), and combined via score fusion to produce ranked skills.

on this framework, the remainder of the paper is organized as follows. Section 3.1 briefly introduces the skill retrieval problem and the formal definition of SSG. Section 3.2 details the construction of SSG, including the two node types, the definitions of the associated edges, and their construction methods. Section 3.3 presents the retrieval process on the graph, focusing on the graph diffusion method that integrates multi-level information and the stage-aware query rewriting method.

3.1 Skill Retrieval Problem Definition and SSG Graph Representation

The generic skill retrieval problem aims to accurately match and rank the most relevant skills from a large-scale skill library for a given natural language task query. Let q denote the natural language task query and G the skill library. Let $\mathcal{K}$ be the set of skills in G. The retrieval problem seeks to generate a ranked list of skills:

$$\pi(q) = \underset{k \in \mathcal{K}}{\operatorname{argsort}}\, r(k \mid q),$$

where $r(k \mid q)$ is the final graph-plus-field reranking score defined in Section 3.3. In the SSG, the skill library G is formalized as a graph structure comprising two types of hierarchical nodes:

$$G = (V_{\text{skill}}, V_{\text{stage}}, E),$$

where V_{skill} denotes the set of skill-level nodes, V_{stage} denotes the set of stage-level nodes, and E is the set of edges connecting these nodes. For stage-level nodes in V_{stage}, we define a stage as an execution module with well-defined inputs and outputs, relatively self-contained internal functionality, and the ability to be processed independently of other stages Liang et al. (2026a). The execution process of an entire skill, represented by a skill-level node in V_{skill}, can therefore be regarded as a directed acyclic graph (DAG) composed of stage nodes. However, owing to the complexity of skills, the stage-level nodes in V_{stage} cannot encompass all global information of a skill, such as usage examples and constraints. We therefore retain V_{skill} in the graph to carry such global context, e.g., overall I/O interfaces and main content. Through this two-level node structure, semantic matching can simultaneously exploit global skill information, local target stages, and the required actions, objects, tools, and states, thereby achieving more precise retrieval.

3.2 Stage-Aware Offline Skill Graph Construction

3.2.1 Node and Edge Construction

The graph contains two core types of nodes, which characterize the global semantics and local execution details of skills, respectively. Skill-level nodes V_{skill} carry the global information of skills. Since stage-level nodes cannot encode all details of a complete skill (e.g., usage examples and prohibited rules), each skill-level node retains complete information, including the overall input/output (I/O) interface and the main content. Stage-level nodes V_{stage} carry the local execution information of skills; each node represents a self-contained execution unit with independent inputs and outputs, and is used to model fine-grained execution steps inside a skill.

Based on these two node types, the edge set E in the graph covers three connection categories: connections between skill-level nodes, cross-level connections between skill-level and stage-level nodes, and connections between stage-level nodes. For edges between skill-level nodes, we follow prior work Liu et al. (2026a) and retain four edge types: `dependent`, `workflow`, `alternate`, and `semantic`. We also adopt its candidate-pool filtering strategy based on semantic and I/O similarity: when constructing edges, we first select similar nodes as candidates, thereby significantly reducing the computational complexity of full graph construction.

For cross-level connections between skill-level and stage-level nodes, we introduce a new edge type, `containment`, to model the hierarchical relation that a stage-level node belongs to its parent skill, where $\rho(s)$ denotes the parent skill node of stage node s. To prevent `containment` edges from trapping graph diffusion within a skill's own internal structure, we define them as directed edges. We also retain `dependency` and `semantic` edges for cross-level connections to model cross-level semantic similarity and I/O pipeline structures. To avoid constructing these edges within the same skill, we exclude nodes from the same family (the parent SkillNode plus all sibling StageNodes) during edge construction. For connections between stage-level nodes, we introduce `flow` edges to model the execution order among stage-level nodes within the same skill, thereby modeling the internal execution structure of a skill in the graph. Similarly, we use the same method to model `dependency` and `semantic` edges between stage-level nodes. Appendix B shows the design details of all edge.

3.2.2 Build-Time Dependency Edge Filtering

To reduce redundant dependency edges in large-scale skill graphs, we move dependency-edge filtering from retrieval time to graph construction time. Automatically generated dependency edges often arise from repeated templated I/O fields (e.g., 'session id', 'tool', and 'slug'); although valid under I/O matching, such edges inflate graph storage, loading, and retrieval cost, while experiments show that removing them usually does not reduce retrieval accuracy. We identify non-discriminative hub tokens using two complementary statistics: node document frequency $\mathrm{DF}(t)$, the number of skill/stage nodes whose I/O signature contains token t, and token yield $\mathrm{YLD}(t)$, the number of dependency edges actually driven by t. For a skill pool of size M, the core filtering criterion is

$$\mathrm{Hub}(t) \iff (\mathrm{DF}(t) \geq \tau(M)N) \ \vee \ (\mathrm{YLD}(t) \geq \tau(M)E_{\text{dep}}),$$

where $N = |\mathcal{V}|$ is the number of skill/stage nodes, $E_{\text{dep}} = |\mathcal{E}_{\text{dep}}|$ is the nominal number of dependency edges, and $\tau(M)$ is an adaptive threshold. Exact and substring dependency edges are removed when their driver token satisfies $\mathrm{Hub}(t)$; multi-token Jaccard edges are removed only when every shared token satisfies the DF condition; edges with unparseable evidence are retained by default. This construction-time filtering yields a smaller graph consistent with retrieval-time filtering. Full rules, the two-stage procedure, the schedule of $\tau(M)$, and parameter values are provided in the supplementary material.

3.3 Stage-Aware Online Retrieval

The previous section organized the skill library as a multi-level heterogeneous graph: skill nodes represent complete skills that an agent can directly invoke, stage nodes represent local stages within skills that have independent goals, inputs/outputs, and execution conditions, and edges describe skill containment, stage flow, and cross-skill or cross-stage dependencies. Online retrieval follows this

multi-level structure rather than compressing all nodes into a single candidate set. We retrieve over both skill nodes and stage nodes simultaneously, making full use of the current multi-level graph structure. For skill-node retrieval, following prior work, we select retrieval seeds using semantic similarity and lexical overlap of node keys, then perform graph-diffusion retrieval with reverse-aware relations, and finally add reranking scores Liu et al. (2026a). Unlike prior work, to fully exploit the information of the multi-level graph structure, we propose Stage-aware PPR (S-PPR). Specifically, our skill-node retrieval channel uses $\mathbf{u}_{\text{skill}}$ as the personalization vector and runs reverse-aware PPR Yang et al. (2024) on the unified projection containing both skill and stage nodes. Finally, in the PPR algorithm, we only aggregate the scores of skill nodes as the skill-channel score.

$$\mathbf{p}^{(t+1)} = d\mathbf{u} + (1-d)T^{\mathsf{T}}\mathbf{p}^{(t)}, \qquad \mathbf{p}^{(0)} = \mathbf{u}, \tag{1}$$

We set teleportation probability $d = 0.2$ and graph-following probability $1 - d = 0.8$; thus d is not the PageRank damping/continuation convention. The personalization vector $\mathbf{u}$ is normalized by harmonic seed-rank weights; transition matrices, reverse weights, and dangling-node handling are in the supplement. Skill scores are

$$\mathbf{p}_{\text{skill}} = \text{PPR}(T_G, \mathbf{u}_{\text{skill}}), \qquad p_{\text{skill}}(k) = [\mathbf{p}_{\text{skill}}]_k.$$

For stage-node retrieval, we select their retrieval seeds in the same way as for skill nodes. For the stage-channel graph diffusion, propagation is performed independently on the subgraph consisting only of stage nodes, so as to retrieve the relations among stages in detail. Let $G_{\text{stage}} = (V_{\text{stage}}, E_{\text{stage}})$ denote the stage subgraph projected from the unified graph, where only edges whose endpoints are both stage nodes are retained. Using the stage seed distribution $\mathbf{u}_{\text{stage}}$ as the personalization vector, the stage scores are

$$\mathbf{p}_{\text{stage}} = \text{PPR}(T_{G_{\text{stage}}}, \mathbf{u}_{\text{stage}}), \qquad p_{\text{stage}}(s) = [\mathbf{p}_{\text{stage}}]_s.$$

Stage evidence is then max-aggregated to parent skills. For $\mathcal{S}(k) = \{s \in V_{\text{stage}} : \rho(s) = k\}$,

$$g(k \mid q) = p_{\text{skill}}(k) + \alpha \, \text{Agg}_{s \in \mathcal{S}(k)} \, p_{\text{stage}}(s), \qquad \alpha = 0.5,$$

with empty aggregation $= 0$. Agg uses max aggregation: any highly query-relevant stage under a skill supplies local evidence for its parent. This aggregation is single-layer only, with no recursion to deeper nodes. Separate propagation keeps skill-level support and stage-level execution evidence distinct: skill PPR models overall support in the full graph, while stage-subgraph PPR propagates evidence along stage flow and cross-stage relations without diluting stage quality into the parent via containment edges. Retrieval then performs graph-score candidate selection followed by field-aware reranking (details in the supplement).

3.3.1 Stage-Aware Query Rewriting

For datasets of some complex tasks, we assume that a task can be decomposed into multiple stages. To align the retrieval query with the descriptions of stage nodes, we design stage-aware query rewriting. We provide the agent with additional instructions so that it no longer performs a single holistic retrieval for the original task. Instead, at the beginning of reasoning, a planner first decomposes the task into multiple subtasks arranged in execution order. Each subtask contains a normalized stage label, a local goal, the required precondition state, and the expected output state, thereby forming a state contract that can be passed between adjacent stages. For each subtask, the planner generates two complementary queries in parallel: a normal query preserves concise semantics of the stage and object and mainly matches complete skills; a stage query adds action, object, tool or container, preconditions, and result or verification information, aligning the query with the structured fields of stage nodes.

Each query for each subtask independently triggers the complete stage-aware retrieval process described above, including type-separated seed selection, reverse-aware diffusion, stage-to-skill evidence aggregation, and field-aware reranking. The online rerun only re-executes query-related reasoning; it does not rebuild the skill graph, modify graph edges, or regenerate offline stage annotations. All subtasks share the same frozen graph and retrieval configuration. Under this fixed setting, the system organizes the queries, candidate skills, and their short descriptions in the original order of the subtasks into a retrieval summary and injects it into the agent context. When the summary is length-limited, it prioritizes preserving the complete subtask sequence, preconditions, and output states, and then prunes details of lower-ranked candidates. Thus, stage-aware online retrieval transforms a static matching pass for an overall task into an online rerun for execution stages, and reorganizes scattered local evidence into an executable complete skill chain.

4 EXPERIMENTS

We evaluate different skill retrieval methods across multiple datasets to investigate whether our SSG can improve skill retrieval accuracy and downstream agent performance.

4.1 EXPERIMENTAL SETUP

We evaluate SSG on three datasets: **ALFWorldShridhar et al. (2021)**, **AgentSkillOSLi et al. (2026a)**, and **SkillsBenchLi et al. (2026b)**. **ALFWorld** is an interactive household-task benchmark that evaluates an agent's ability to understand instructions, navigate environments, and manipulate objects. **AgentSkillOS** contains 30 creative tasks across five categories, requiring agents to generate multi-format outputs such as documents, slides, images, and videos, with performance evaluated through pairwise comparison. **SkillsBench** contains 84 tasks across 11 domains and evaluates agents under three settings: without Skills, with curated Skills, and with self-generated Skills.

We compare SSG with representative baselines from three retrieval paradigms: **flat vector-based retrieval**, including Vector Skills and SkillRouter, which retrieve skills through vector similarity; **tree-structured retrieval**, including AgentSkillOS and SkillRAE, which organize skills hierarchically for structured retrieval and selection; and **graph-structured retrieval**, including Graph of Skills (GoS) and SkillDAG, which explicitly model relationships among skills to support structured retrieval and composition. To ensure a consistent comparison, we use the same evaluation benchmarks and task queries for all methods whenever their implementations permit. Detailed implementation settings, including model configurations, retrieval parameters, and reproduction details, are provided in the supplementary material. For skill retrieval, we report Ret@1, Ret@K, and mean reciprocal rank (MRR). Unless otherwise specified, $K = 5$. Ret@1 and Ret@K measures whether the correct skill appears among the top-K retrieved skills. MRR evaluates the reciprocal rank of the correct skill. For downstream evaluation, we report the task performance score R, where a higher value indicates better task execution.

Table 1: Retrieval performance on ALFWorld, AgentSkillOS, and SkillsBench. **Bold** and underlined values denote the best and second-best results in each metric column, respectively, including ties. Results marked with '*' are taken from the results reported in the corresponding original paper. Emb. denotes embedding-based retrieval; RR denotes reranking.

	ALFWorld			AgentSkillOS			SkillsBench		
Method	Ret@1↑	Ret@K↑	MRR↑	Ret@1↑	Ret@K↑	MRR↑	Ret@1↑	Ret@K↑	MRR↑
Vector-based methods									
Vector Skills	0.7	4.5	0.4	56.7	83.3	66.2	3.6*	10.8*	5.8*
SkillRouter (Emb.)	13.6	48.6	26.2	50.0	86.7	64.3	77.6	**93.4**	83.1
SkillRouter (Emb.+RR)	12.1	**74.3**	36.2	56.7	70.0	61.9	**81.6**	88.2	**84.5**
Tree-structured methods									
AgentSkillOS	0.7	50.7	18.5	70.0	83.3	76.7	38.2	44.7	41.0
SkillRAE	9.3	35.6	17.5	40.0	80.0	55.2	69.7	86.8	76.8
Graph-structured methods									
GoS	8.6	9.3	8.9	76.7	86.7	80.3	50.6*	65.5*	57.3*
SkillDAG	20.0	44.8	29.2	43.3	56.1	60.8	66.7*	78.2*	71.3*
Ours	**49.3**	71.4	**57.9**	**80.0**	**93.3**	**83.3**	69.2	86.8	76.8

4.2 MAIN RESULTS

Table 1 reports retrieval results across the three benchmarks. SSG achieves particularly strong performance on structured tasks. On **ALFWorld**, it obtains the best Ret@1 and MRR, reaching 49.3% and 57.9%, respectively, compared with 20.0%/29.2% for SkillDAG and 13.6%/26.2% for SkillRouter with embedding-based retrieval. Its Ret@5 reaches 71.4%, slightly below SkillRouter with retrieval and reranking (74.3%). On **AgentSkillOS**, SSG achieves the best performance on all metrics, with 80.0% Ret@1, 93.3% Ret@5, and 83.3% MRR, outperforming GoS (76.7%, 86.7%, and

Table 2. R scores of different end-to-end methods across various datasets. **Bold** and underlined values indicate the best and second-best available results within each model and benchmark, respectively, including ties. **M2.7** denotes MiniMax-M2.7, and **GPT-5.6** denotes GPT-5.6-Terra.

Model	Method	ALFWorld	AgentSkillOS	SkillsBench
M2.7	Vanilla Skills	47.1*	70.8	17.2*
	Vector Skills	50.7*	72.4	10.4*
	GoS	54.3*	68.0	18.7*
	SkillDAG	67.9	54.9	27.3*
	Ours	**76.4**	**80.9**	**28.0**
GPT-5.2	Vector Skills	60.0	74.9*	27.3
	GoS	63.6	96.2	35.4
	SkillDAG	82.0	94.2	31.4
	AgentSkillOS	13.6	83.5*	17.2*
	SkillRouter	29.3	82.3*	22.0*
	SKILLRAE	27.1	84.6*	29.3*
	Ours	**86.0**	**97.4**	**43.2**
GPT-5.6	GoS	79.3	96.2	40.1
	SkillDAG	85.0	92.3	49.0
	Ours	**85.7**	**96.5**	**51.0**

Table 3. ALFWorld retrieval ablation Study of S-PPR and V_{stage}.

Method	Ret@1	Ret@5	MRR
Baseline	**49.3**	**71.4**	**57.9**
w/o S-PPR	48.9	70.6	57.7
w/o V_{stage}	48.3	69.3	56.6
w/o Rewriting	35.3	46.2	40.0

Table 4. ALFWorld retrieval by library size V_skill. Ours is shown in **bold**.

V_skill	Method	Ret@1↑	Ret@K↑	MRR↑
500	GoS	22.1	64.3	35.5
	SkillDAG	23.7	56.3	36.0
	Ours	**51.3**	**72.9**	**59.5**
1000	GoS	8.6	9.3	8.9
	SkillDAG	20.0	44.8	29.2
	Ours	**49.3**	**71.4**	**57.9**
2000	GoS	0.0	42.9	14.8
	SkillDAG	19.7	43.9	28.6
	Ours	**48.1**	**71.2**	**57.0**

80.3%). On **SkillsBench**, it obtains 69.2% Ret@1, 86.8% Ret@5, and 76.8% MRR, remaining competitive with structured baselines such as SkillRAE. These results show that SSG is effective at retrieving relevant skills under structured task settings, particularly in terms of top-ranked retrieval. However, single-query retrieval scores do not always directly reflect downstream task performance, motivating evaluation of the complete execution process.

Table 2 presents end-to-end task performance under different execution models. SSG achieves the highest task performance across all three execution models and benchmarks. With **MiniMax-M2.7**, it obtains scores of 76.4, 80.9, and 28.0 on ALFWorld, AgentSkillOS, and SkillsBench, improving over SkillDAG by 8.5, 26.0, and 0.7 points, respectively. With **GPT-5.2**, it reaches 86.0, 97.4, and 43.2, compared with 82.0, 94.2, and 31.4 for SkillDAG. With **GPT-5.6**, it achieves 85.7, 96.5, and 51.0, compared with 85.0, 92.3, and 49.0. The gap between retrieval and end-to-end results highlights an important property of multi-stage tasks: a single task may require different skills at different stages, while runtime search and query rewriting can produce multiple stage-specific queries. Thus, useful and complementary retrieval across stages may benefit downstream execution even when single-query retrieval metrics are not consistently the highest.

SSG addresses this by explicitly modeling task stages and stage-level constraints and providing stage-level skill recommendations that are integrated with the task plan. The end-to-end results suggest that this structured organization enables retrieval to translate more effectively into complete task execution, indicating that skill retrieval should be evaluated not only by isolated query-level accuracy but also by its ability to support multi-stage execution.

4.3 Ablation Study

We conduct ablation experiments to examine the contributions of stage-aware search and structural nodes. Let $G = (V_{\text{skill}}, V_{\text{stage}}, E)$ denote the skill structure graph, where V_{skill} and V_{stage} represent skill-level and stage-level nodes, respectively. We denote the complete node set by $\mathcal{V} = V_{\text{skill}} \cup V_{\text{stage}}$, the skill set by $\mathcal{K}$, and the number of skills in the library by $M = |\mathcal{K}|$. For a query q, $\pi(q)$ denotes the ranked skill list produced according to the retrieval score $\text{Score}(q, k)$.

As shown in Table 3, the complete model achieves 49.30% Ret@1, 71.40% Ret@5, and 57.90% MRR. Removing `Stage-aware PPR` reduces these values to 48.88%, 70.63%, and 57.74%, respectively. Although the absolute changes are relatively small, the consistent decrease across all

three metrics indicates that stage-aware propagation contributes to the quality of the ranked skill list.

Removing the stage-level nodes V_{stage} further reduces Ret@1 to 48.33%, Ret@5 to 69.33%, and MRR to 56.57%. Relative to the variant without `Stage-aware PPR`, this corresponds to additional decreases of 0.55, 1.30, and 1.17 percentage points, respectively. This result indicates that stage-level nodes provide useful structural information for organizing and constraining candidate skills during retrieval.

We further investigated the role of task decomposition. The variant `w/o Rewriting`, which completely removes task decomposition, achieves 35.3% Ret@1, 46.2% Ret@5, and 40.0% MRR, substantially lower than the full model. This degradation is markedly larger than that caused by removing other components, indicating that decomposing complex tasks into multiple stages is an important prerequisite for leveraging the stage-aware skill structure. However, the ablation study on AgentSkillOS in Appendix D shows that ablating V_{stage} has an even greater impact. Taken together, both approaches must be combined to ensure SSG's generalization and overall performance across different datasets.

4.4 Skill Library Size Sensitivity

We further evaluate the robustness of SSG to the size of the skill library on ALFWorld. We consider libraries containing 500, 1,000, and 2,000 skills, while keeping the evaluation protocol fixed. Table 4 reports Ret@1, Ret@5, and MRR.

SSG achieves the highest value on all three metrics at every library size. With 500 skills, it obtains 51.3% Ret@1, 72.9% Ret@5, and 59.5% MRR. With 1,000 skills, these values are 49.3%, 71.4%, and 57.9%, respectively. When the library expands to 2,000 skills, the corresponding scores remain at 48.1%, 71.2%, and 57.0%. Thus, although retrieval performance gradually decreases as the number of candidate skills increases, the degradation is relatively limited. In comparison, SkillDAG obtains Ret@1 values of 23.7%, 20.0%, and 19.7% for libraries of 500, 1,000, and 2,000 skills, respectively, while GoS decreases from 22.1% to 8.6% and finally to 0.0%. The corresponding MRR values of SSG are 59.5%, 57.9%, and 57.0%, compared with 36.0%, 29.2%, and 28.6% for SkillDAG and 35.5%, 8.9%, and 14.8% for GoS. These results show that the retrieval performance of SSG remains relatively stable as the skill library expands and that its advantage over the compared graph-structured methods persists under increasingly large candidate pools.

5 Conclusion

We presented SSG, a stage-aware retrieval framework that represents complete skills together with their internal execution stages and typed structural relations. By combining separate skill- and stage-level graph propagation, stage-to-skill evidence aggregation, and stage-aware query decomposition, the framework exposes local execution evidence while returning complete skills for agent use. Across ALFWorld, AgentSkillOS, and SkillsBench, SSG improves both retrieval and downstream execution. With MiniMax-M2.7, it achieves end-to-end scores of 76.4, 80.9, and 28.0, respectively; its Ret@1 reaches 49.3%, 80.0%, and 69.2%. The retrieval advantage also remains stable as the ALFWorld skill library grows to 2,000 entries, where Ret@1 remains 48.1%. These results indicate that stage-level structure is particularly valuable for multi-stage tasks, while the gap between isolated retrieval metrics and end-to-end performance shows that skill retrieval should also be evaluated by its contribution to planning and execution. Future work will investigate query-adaptive propagation and score calibration, more efficient dependency-edge construction, and broader multi-run evaluation under diverse skill libraries, task distributions, and execution models.

Reproducibility statement

All configurations, frozen artifacts, manifests, and reproduction commands are documented in A.1; verified, directional, and provisional measurements are distinguished.

AI USE STATEMENT

Generative models were used for structured scene annotations and agent queries. We validated annotations with automated checks, audited ID mappings, and reviewed generated text and code.

REFERENCES

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.

Tianyi Chen, Yinheng Li, Michael Solodko, Sen Wang, Nan Jiang, Tingyuan Cui, Junheng Hao, Jongwoo Ko, Sara Abdali, Leon Xu, Suzhen Zheng, Hao Fan, Pashmina Cameron, Justin Wagle, and Kazuhito Koishida. CUA-Skill: Develop skills for computer using agent. *arXiv preprint arXiv:2601.21123*, 2026. URL https://arxiv.org/abs/2601.21123v2.

Paimon Goulart, Liang Wu, Kelly Wan, Evangelos E. Papalexakis, and Liangjie Hong. Field-aware agent skill retrieval. *arXiv preprint arXiv:2608.02880*, 2026. URL https://arxiv.org/abs/2608.02880v3.

Bernal Jiménez Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. HippoRAG: Neurobiologically inspired long-term memory for large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Ryangkyung Kang, Hongcheol Cho, and Youngeun Kim. SkillRet: A large-scale benchmark for skill retrieval in LLM agents. *arXiv preprint arXiv:2605.05726*, 2026.

Hao Li, Chunjiang Mu, Jianhao Chen, Siyue Ren, Zhiyao Cui, Yiqun Zhang, Lei Bai, and Shuyue Hu. Organizing, orchestrating, and benchmarking agent skills at ecosystem scale. *arXiv preprint arXiv:2603.02176*, 2026a.

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

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*, 2026c.

Qiliang Liang, Hansi Wang, Zhong Liang, and Yang Liu. From skill text to skill structure: The scheduling-structural-logical representation for agent skills. *arXiv preprint arXiv:2604.24026*, 2026a.

Yuan Liang, Ruobin Zhong, Haoming Xu, Chen Jiang, Yi Zhong, Runnan Fang, Jia-Chen Gu, Shumin Deng, Yunzhi Yao, Mengru Wang, et al. SkillNet: Create, evaluate, and connect AI skills. *arXiv preprint arXiv:2603.04448*, 2026b. URL https://arxiv.org/abs/2603.04448v3.

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.

Shuo Liu, Yutong Yang, Haohao Xiao, Mouxing Yang, and Xi Peng. Imagine before retrieval: Prospective skill retrieval for LLM agents. *arXiv preprint arXiv:2609.01642*, 2026b. URL https://arxiv.org/abs/2609.01642v1.

Xiangcheng Meng, Shu Wang, and Yixiang Fang. SkillRAE: Agent skill-based context compilation for retrieval-augmented execution. *arXiv preprint arXiv:2605.10114*, 2026.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations (ICLR)*, 2024.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations (ICLR)*, 2021.

Weihang Su, Jianming Long, Qingyao Ai, Qiaozhi He, Yichen Tang, Changyue Wang, Yiteng Tu, Yingbo Wang, and Yiqun Liu. Skill retrieval augmentation for agentic AI. *arXiv preprint arXiv:2604.24594*, 2026. URL https://arxiv.org/abs/2604.24594v3.

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.

Mingji Yang, Hanzhi Wang, Zhewei Wei, Sibo Wang, and Ji-Rong Wen. Efficient algorithms for personalized pagerank computation: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 36(9):4582–4602, 2024.

Kun Zeng, Yu Huo, Siyu Zhang, Zi Ye, Yuecheng Zhuo, Haoyue Liu, Yuquan Lu, Junhao Wen, and Xiaoying Tang. Group of skills: Group-structured skill retrieval for agent skill libraries. *arXiv preprint arXiv:2605.06978*, 2026. URL https://arxiv.org/abs/2605.06978v1.

YanZhao Zheng, ZhenTao Zhang, Chao Ma, YuanQiang Yu, JiHuai Zhu, Yong Wu, Tianze Xu, Baohua Dong, Hangcheng Zhu, Ruohui Huang, and Gang Yu. SkillRouter: Skill routing for LLM agents at scale. *arXiv preprint arXiv:2603.22455*, 2026.

APPENDIX

A DETAILED EXPERIMENTAL PROTOCOL

A.1 COMMON PROTOCOL

We use benchmark-specific execution protocols because the three benchmarks expose different agent interfaces. Within each benchmark, all compared retrieval methods share identical task inputs, skill pool, execution-agent family, retry policy, and evaluation code. Unless otherwise stated, every end-to-end experiment is independently run twice, and the reported task score is the arithmetic mean of the two runs.

Unless otherwise specified, graph construction and retrieval use OpenAI's `text-embedding-3-large`. For tasks requiring LLM-based generation or judgment, we align with prior baselines and use `gpt-5-nano`. Stage nodes are generated with `deepseek-v4-flash`. Generated outputs are frozen to mitigate performance fluctuations caused by LLM stochasticity.

For decomposition-enabled runs, the planner is called with temperature 0 and no explicit top-p value; the gateway default is therefore used. For retrieval-metric experiments, planner outputs are typically generated with Minimax2.7, while end-to-end experiments use their corresponding LLMs.

The prompt format is structurally shared across benchmarks: decomposition outputs contain the same ordered-subtask schema and the same `plain`, `scene`, and, where applicable, task-level query fields. The prompts are not identical strings. Few-shot examples, task-domain vocabulary, and suggested decomposition are benchmark-specific, as specified below.

A.2 SHARED RETRIEVAL AND PLANNING CONFIGURATION

Embeddings are L2-normalized before comparison, and retrieval uses cosine similarity, equivalently the dot product of normalized vectors. The GOS API path does not impose a separate tokenizer limit on query or document text; the gateway/model applies its own context limit. For the local encoder path used by the auxiliary open-model evaluation, both the encoder and reranker tokenizer `max_length` are set to 4096 tokens.

The seed candidate pool is the union of the top 20 semantic candidates and top 20 lexical candidates, with duplicate nodes removed. The initial skill node seed budget is 5, and the stage node also uses 5 scene seeds (`seed_top_k_sub=5`). The internal engine default is `top_n=8`; when a benchmark exposes retrieved skills directly to an agent, the final returned budget follows that benchmark protocol: top-5 for AgentSkillOS and the corresponding top-5 search-call budget in the ALFWorld and SkillsBench interfaces. The auxiliary public SkillRouter retrieve–rerank evaluation is separate: its encoder is the released SkillRouter embedding model and its reranker is `pipizhao/SkillRouter-Reranker-0.6B`. No reranker model is used in the graph-retrieval experiments reported here; the graph score and the implemented field-aware score determine the ranking.

For graph construction, deterministic dependency edges require an I/O overlap score of at least 0.6. Cross-edge linking has no separate scalar semantic confidence threshold in the implementation: semantic candidates are obtained from the top-32 overfetched vector/lexical candidate set (configured link top-k is 8 and the overfetch multiplier is 4), restricted to eligible nodes from another skill family, and retained only when the pair-level field evidence gate is satisfied. The value 0.2 associated with semantic edges is a reverse transition coefficient in PPR, not a linking threshold; the forward semantic edge coefficient is $0.4\times$ its candidate score. These construction settings are not tuned separately for the three benchmarks. The PPR damping/teleportation probability is 0.2, with at most 50 iterations and an L1 convergence tolerance of 10^{-6}.

All experiments are run on Linux with Ubuntu 24.04.4 LTS. The primary Python version is 3.12.

A.3 ALFWORLD

We follow the agent design used by prior work (SkillDAGBai et al. (2026)). Our search service is exposed to the agent through the same style of callable interface as the baseline search service.

LLM-generated planning can vary across repeated calls and would otherwise confound retrieval comparisons. We therefore cache and freeze the planning outputs used by the evaluated tasks. All methods or reruns that use the corresponding planning condition consume the same cached decomposition; the graph and retrieval configuration are also held fixed within each paired comparison. Thus, variation in the reported comparison comes from retrieval and downstream execution rather than from regenerating a more favorable plan. Prompts use household action phases.

A.4 AGENTSKILLOS

We use the benchmark's official evaluation code and task workspace layout. The retrieval model processes each official task query before agent execution. We retain the top five retrieved skills and place their skill files in the workspace assigned to that task, making exactly that retrieved set available to the execution agent. Baselines and our method use the same top-five budget and the same workspace-construction procedure.

The AgentSkillOS comparisons pair different model APIs with their corresponding coding-agent frameworks. We preserve this protocol: GPT-5.2 and GPT-5.6-terra are executed with Codex, while MiniMax-M2.7 is executed with Claude Code. This matches the baseline evaluation design instead of forcing every model through a single agent framework that may favor one API family. Prompts use general work phases.

A.5 SKILLSBENCH

For SkillsBench, we align with the evaluation harness used for the previously reported experiments. All compared methods use Codex as the execution agent, independent of the retrieval method. They receive the same task files, runtime constraints, verifier, skill-pool construction, and retrieval budget. Consequently, no method can benefit from a different execution harness or verifier path. Prompts use task and skill-search instructions. During pilot runs, transient network interruptions occasionally terminated an otherwise valid episode. We therefore added automatic network reconnection and retry handling to all evaluated methods, including the baselines.

B FIELD DEFINITIONS FOR SKILL AND STAGE NODES

This supplementary section provides additional implementation details and field definitions that are omitted from the main paper due to space limits.

B.1 SKILL-LEVEL NODE FIELDS

We store each skill as a callable unit with the following fields.

- **Name**: the canonical skill name.
- **Description**: a short natural-language summary.
- **Inputs / Outputs**: normalized I/O interface tokens.
- **Body**: the full skill document content.

B.2 STAGE-LEVEL NODE FIELDS

Each stage is extracted as a semantically coherent execution unit. In addition to its text, we record structured fields for stage-aware retrieval.

- **Goal**: the local objective of this stage.
- **Preconditions**: required environment states or inputs.

Field	Source or construction
`name`, `description`	Frontmatter; otherwise inferred from the Overview, Summary, or Description section, or from the first sentence.
`inputs`, `outputs`	Rule-based values merged with LLM extraction.
`domain_tags`	Rule-based tags merged with the LLM-extracted `tags`, preserving order and removing duplicates.
`tooling`	Rule-based extraction augmented with LLM-extracted `dependencies`.
`example_tasks`	Rule-based extraction.
`script_entrypoints`	Scanned `.py`/`.sh`/`.js`/`.ts` files augmented with LLM-extracted entry points.
`compatibility`, `allowed_tools`	Frontmatter.
`rendered_snippet`	Original skill document together with `allowed_tools` and body content.
`intent_signature`	LLM-extracted description of the skill's intent characteristics.

Table 5: Skill-node fields and their sources.

Skill-node field	Stage-node value
`name`	{`skill_id`}`::`{`stage_id`}
`type`	`stage`
`description`	`stage_goal`
`one_line_capability`	`stage_name`
`inputs`, `outputs` and schema fields	Normalized input and output keys.
`parent_skill_id`	Name of the parent skill.
`metadata_json`	The complete original stage record under `ssl_stage`, together with the ordered normalized step records under `logic_steps`.
Other fields	Empty, including domain tags, tooling, and example tasks.

Table 6: Mapping from skill-node fields to stage-node fields.

- **Effects**: expected state changes or outputs.
- **Operations**: normalized atomic actions (when available).

C Additional Details

C.1 Stage-Aware Offline Skill Graph Construction

Skill-node fields. Skill nodes largely follow the GOS representation, while combining the rule-based fields with structure extracted from the skill document by an LLM. The resulting fields are:

Stage-node fields. Stage nodes reuse the skill-node schema, with stage-specific information stored in `metadata_json`. A stage has the unique storage key {`skill_id`}`::`{`stage_id`}, type `stage`, and its `description` is the stage goal. The field mapping is:

Edge types. The skill–skill edge construction is unchanged from GOS. The additional stage-aware edges are constructed as follows; reverse values below are the coefficients used when adding reverse-aware transitions, not independent edge weights.

C.2 Build-Time Dependency Edge Filtering

Our filtering algorithm is based on two complementary statistics. First, for each I/O token t, we define its node document frequency (DF):

$$\mathrm{DF}(t) = \sum_{v \in \mathcal{V}} \mathbf{1}[t \in \mathrm{SigTok}(v)],$$

Type	Direction	Weight	Reverse	Construction and role
dependency, workflow, semantic, alternative	skill ↔ skill	existing	existing	The original skill–skill edge algorithms are unchanged.
containment	stage → skill	1.0	0.1	Deterministically constructed from SSL `subscenes`; the direction prevents containment from dominating skill-node outgoing transitions.
flow	stage → stage, within one skill	0.3	0.1	Extracted from LLM stage relations and represents the next-stage relation. Step-level next-step rules remain in stage metadata rather than becoming separate edges.
dependency (stage)	stage → stage, across skills	overlap score	1.0	Producer output keys to consumer input keys, using the same deterministic dependency logic as skill nodes and requiring overlap at least 0.6.
semantic (stage)	stage ↔ stage, across skills	$0.4\times$ confidence	0.2	Embedding or lexical candidates are linked when confidence reaches the configured threshold.
dependency/semantic (cross-type)	skill ↔ stage, across families	as above	as above	The stage-level procedure is reused for candidates of the other node type.

Table 7: Stage-aware graph edge definitions.

where $\mathcal{V} = V_{\text{skill}} \cup V_{\text{stage}}$ is the set of all skill- and stage-level nodes in the graph, and $\mathrm{SigTok}(v)$ denotes the set of tokens of the input/output fields of node v. DF measures in how many nodes a token appears; when $\mathrm{DF}(t)$ is high, the token has weak discriminative power and is more likely to produce noisy edges. DF alone is insufficient to identify all noisy edges, because some tokens do not appear frequently as exact strings but can participate in substring matching as shorter strings and drive many edges. Therefore, we further define the token yield:

$$\mathrm{YLD}(t) = \sum_{e \in \mathcal{E}_{\text{dep}}} \mathbf{1}[\mathrm{driver}(e) = t],$$

where $\mathcal{E}_{\text{dep}} \subseteq E$ is the set of dependency edges, and $\mathrm{driver}(e)$ denotes the evidence token that actually drives the generation of dependency edge e. Yield measures how many dependency edges a token actually produces, complementing DF's ability to identify substring collisions.

Given the number of nodes $N = |\mathcal{V}|$, the number of dependency edges $E_{\text{dep}} = |\mathcal{E}_{\text{dep}}|$, and a ratio threshold τ, we define the dual hub predicate

$$\mathrm{Hub}_{\text{dual}}(t) \iff \big(\mathrm{DF}(t) \geq \tau N\big) \vee \big(\mathrm{YLD}(t) \geq \tau E_{\text{dep}}\big).$$

For exact and substring dependency matches, the edge has a single driver token and is removed when that driver satisfies $\mathrm{Hub}_{\text{dual}}$. For a multi-token Jaccard match, no single yield driver is assigned; the edge is removed only when every shared token satisfies $\mathrm{DF}(t) \geq \tau N$. Edges whose evidence cannot be parsed are retained by default. The complete path-dependent rule is given in the supplementary material.

Filtering is performed in two steps during construction. The first step (Layer A') occurs before dependency edges are formally written and uses only the DF clause to skip edges that are already guaranteed to be removed by the full rule. The second step (Layer A) performs the complete path-dependent sweep after graph construction, using the nominal dependency-edge count for the yield

Edge type	Forward weight	Reverse coefficient γ
`dependency`	edge-specific	1.0
`workflow`	edge-specific	0.5
`semantic`	edge-specific	0.2
`alternative`	edge-specific	0.1
`containment`	1.0, stage $\rightarrow$ skill	0.1
`flow`	0.3, earlier $\rightarrow$ later stage	0.1

Table 8: Reverse-aware transition weights.

threshold. The filtering ratio τ may be scheduled according to the skill-pool size as described in the supplementary material. For completeness, the schedule is

$$\tau(M) = \max\left(\tau_{\min},\ \tau_{\max}\sigma\left(\frac{c-M}{s}\right)\right), \qquad \sigma(x) = \frac{1}{1+\exp(-x)}.$$

In small pools, tokens are more likely to cover almost all nodes, i.e., $\mathrm{DF}(t) \approx N$. We control the filtering ratio for small pools via $\tau_{\max}$, avoiding overly aggressive filtering. In addition, to prevent $\tau(M)$ from becoming too low as the pool size increases, we set $\tau_{\min}$ as the minimum threshold. The specific parameter values are given in the supplementary material. Finally, the thresholds in the above hub criterion all use this adaptive ratio, i.e.,

$$\mathrm{Hub}(t) \Longleftrightarrow (\mathrm{DF}(t) \geq \tau(M)N) \vee (\mathrm{YLD}(t) \geq \tau(M)E_{\mathrm{dep}})\,.$$

C.3 Online retrieval and PPR details

C.3.1 Seed personalization

Let the ranked retrieval seeds be indexed by $i = 1, \ldots, m$. When explicit seed weights are unavailable, we use harmonic weights

$$w_i = \frac{1}{i}, \qquad u_i = \frac{w_i}{\sum_{j=1}^{m} w_j}. \tag{2}$$

Duplicate contributions from the same node are summed, and the final vector is normalized again. Thus the personalization vector $\mathbf{u}$ sums to one. Skill and stage seeds are converted into separate personalization vectors. The frozen configuration uses separate retrieval, stage-subgraph PPR, $\alpha = 0.5$, and max aggregation.

C.3.2 Transition matrix and reverse-aware weights

Nodes in the projected graph are assigned a fixed order. For a logical directed edge $e : i \rightarrow j$ with forward weight w_e and type $\tau(e)$, we first construct the unnormalized matrix

$$A_{ij} \mathrel{+}= w_e, \qquad A_{ji} \mathrel{+}= \gamma_{\tau(e)} w_e, \tag{3}$$

where $\gamma_{\tau(e)}$ is the reverse coefficient. We then normalize each source row:

$$T_{ij} = \begin{cases} A_{ij} / \sum_{\ell} A_{i\ell}, & \sum_{\ell} A_{i\ell} > 0, \\ 1, & i = j \text{ and } \sum_{\ell} A_{i\ell} = 0, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

Therefore T is row-stochastic, while PPR uses T^{T}. A node with zero out-degree receives a self-loop; its probability mass is not uniformly redistributed over the graph.

C.3.3 PPR iteration and two-stage ranking

The implemented iteration is

$$\mathbf{p}^{(t+1)} = d\mathbf{u} + (1-d)T^{\mathsf{T}}\mathbf{p}^{(t)}, \qquad d = 0.2, \qquad \mathbf{p}^{(0)} = \mathbf{u}. \tag{5}$$

Feature	Coefficient	Feature	Coefficient
Query task name	0.35	Skill name	1.25
Capability/description	0.90	Domain tags	1.15
Tooling	0.95	Input/output fields	0.75
Example tasks	0.80	Script entry points	0.60

Table 9: Field-aware reranking coefficients.

The iteration runs for at most 50 steps and stops when the L1 difference between consecutive vectors is at most 10^{-6}. The returned vector is normalized to sum to one. Here d is the teleportation probability and $1 - d = 0.8$ is the graph-following probability.

For a skill k, the graph score is

$$g(k \mid q) = p_{\text{skill}}(k) + 0.5 \max_{s \in \mathcal{S}(k)} p_{\text{stage}}(s). \tag{6}$$

The implementation first sorts all node scores by g, retains skill nodes, and selects the requested candidate budget. It then applies field-aware reranking only to those candidates:

$$r(k \mid q) = g(k \mid q) + B(q, k) + \frac{0.2}{\text{rank}_{\text{seed}}(k)} \mathbf{1}[k \in \mathcal{I}_{\text{seed}}(q)]. \tag{7}$$

The field term $B(q, k)$ is an unnormalized sum of overlap terms:

Additional implemented terms are $+1.2$ when the normalized skill name occurs in the query, $+0.9$ times artifact overlap, $+0.08$ when script entry points are present, and $+0.35$ for a domain overlap. The final list is sorted by $(r(k \mid q), g(k \mid q))$ in descending lexicographic order. Consequently, g, B, and the seed term are not calibrated probabilities on a common scale.

C.3.4 Field-aware reranking

The graph score is used to select the candidate set; it is not treated as a calibrated probability and is not directly compared with the field terms below. For the selected candidates, the final reranking score is:

$$r(k \mid q) = g(k \mid q) + B(q, k) + \frac{0.2}{\text{rank}_{\text{seed}}(k)} \mathbb{1}[k \in \mathcal{I}_{\text{seed}}(q)], \tag{8}$$

where $B(q, k)$ is an unnormalized weighted sum of field-overlap features between the query schema and skill fields. It includes the task name, skill name, capability/description, domain, tooling, I/O fields, examples, and script entry points; the exact coefficients are provided in the supplementary material. The seed term is added only when k is a skill seed, and $\text{rank}_{\text{seed}}(k)$ is its one-based seed rank. The final ranked skill list $\pi(q)$ is obtained by sorting the graph-selected candidates by $r(k \mid q)$, with $g(k \mid q)$ as the tie-break score.

C.4 Dependency-edge filtering

For each normalized I/O token t, define document frequency and edge yield as

$$\text{DF}(t) = \sum_{v \in \mathcal{V}} \mathbf{1}[t \in \text{SigTok}(v)], \qquad \text{YLD}(t) = \sum_{e \in \mathcal{E}_{\text{dep}}} \mathbf{1}[\text{driver}(e) = t]. \tag{9}$$

For a single-driver edge, the dual hub predicate is

$$\text{Hub}_{\text{dual}}(t) \iff \text{DF}(t) \geq \tau N \ \vee \ \text{YLD}(t) \geq \tau E_{\text{dep}}. \tag{10}$$

The filtering rule is path-dependent:

$$\text{drop}(e) = \begin{cases} \text{Hub}_{\text{dual}}(t), & \text{exact or substring edge with driver } t, \\ \bigwedge_{t \in W(e)} [\text{DF}(t) \geq \tau N], & \text{multi-token Jaccard edge,} \\ 0, & \text{evidence cannot be parsed.} \end{cases} \tag{11}$$

Query Type	Count	Ret@1	Ret@5
Plain	86	63.95%	81.40%
Scene	86	47.67%	77.91%

Table 10: Query-form diagnostic for AgentSkillOS under the query-level micro-average protocol over 172 queries. Under this setting, plain-text queries outperform scene queries.

For substring matches, the driver is the shorter contained normalized string. For Jaccard matches, yield is not used because no single driver is assigned; an edge is dropped only when every shared token is a DF hub. A non-dependency edge is never removed by this filter.

Construction uses two layers. Layer A' uses only the DF clause to skip edges before materialization. Layer A applies the complete rule after the nominal dependency-edge count is available; its yield denominator includes edges suppressed by A'. In the current experimental setting, the filtering ratio is specified as a pool-size-dependent schedule:

$$\tau(M) = \max\left(\tau_{\min}, \tau_{\max}\sigma\left(\frac{c-M}{s}\right)\right), \qquad \sigma(x) = \frac{1}{1+e^{-x}}. \tag{12}$$

The values used for the current experimental specification are

$$\tau_{\max} = 1.2, \qquad c \approx 448, \qquad s \approx 154,$$

with the filtering value fixed to $\tau(500) = 0.5$ for the $N = 500$ setting. These values define the intended experiment configuration; the corresponding schedule has not yet been synchronized into the released implementation and should therefore be treated as the authoritative paper-side specification. The frozen GOS+SSL recipe uses `dep_edge_filter=none`; the filtering path above therefore documents the optional filtering experiment and does not change the frozen graph or any reported baseline result.

D More Ablation Study

In the main ablation experiments, we reported that rewriting is the primary source of gains. To verify generality, we add two small ablations on AgentSkillOS: the first isolates the role of Query Rewriting, and the second isolates the role of stage nodes (stage/scene nodes). Since these two supplementary ablations use different retrieval protocols, and the second also uses a different embedding model, we do not merge them into the main table and report them only as supplementary diagnostic evidence.

D.1 Supplementary Ablation 1: AgentSkillOS Query Rewriting (Plain-Text Queries vs. Scene Queries)

We evaluate 172 rewritten queries in AgentSkillOS: 86 are plain-text queries and 86 are scene queries. This experiment is used to separately examine the effect of query form variation in Query Rewriting.

Across both query types, the overall scores are: Ret@1 = 55.81% (96/172), Ret@4 = 77.33% (133/172), Ret@5 = 79.65% (137/172), and MRR@10 = 0.6561. Compared with the main ablation, this supplementary ablation gives a negative finding: merely adding scene-oriented query wording does not by itself improve single-query retrieval and even underperforms plain-text queries.

D.2 Supplementary Ablation 2: AgentSkillOS Stage Nodes (Scene Increment Controlled Experiment)

This experiment tests the effect of adding our stageNode on top of the GOS skill subgraph. It also uses the Qwen3-Embedding-4B model, rather than the 3072-dimensional text-embedding configuration used by the current frozen workspace; therefore, its absolute values should not be directly compared with Table 1.

Configuration	Hit@1	MRR@10	R@50
GOS baseline	0.7333	0.8250	0.8917
Hybrid separation + stage-subgraph PPR	0.8333	0.8844	0.8917
Graph PPR with StageNode added	0.8667	0.9028	0.9000

Table 11: Scene increment controlled experiment under the 30-task AgentSkillOS setting. The hybrid graph keeps the frozen skill subgraph unchanged and adds a scene layer.

Without changing the skill subgraph, adding scene-aware propagation improves Hit@1 from 0.7333 to 0.8333, and stage-aware PPR further reaches 0.8667. Compared with the main ablation, this supplementary ablation supports a narrower claim: on this benchmark, stage nodes and their relations can provide useful retrieval evidence, and adding relevant scene nodes can significantly improve retrieval precision. Taken together, the two supplementary ablations show that the value of Query Rewriting and that of stage nodes vary across benchmarks; using both together is what improves generalization and overall capability.

E Case Studies

We provide two paired case studies from the same 140-task ALFWorld set. The episodes are aligned by their ALFWorld task name, rather than by an arbitrary file ordering. In the first comparison, the planner-guided decomposed retriever is compared with the official GoSZeng et al. (2026) result. In the second comparison, it is compared with the result of w/o rewriting. The cases are intended as qualitative evidence about the retrieval–execution interface, rather than as additional aggregate estimates.

E.1 Case study 1: retrieving the state-changing skill

The task at index 0 is "cool some winebottle and put it in diningtable." The decomposed run succeeds in 19 steps. Its planner generates separate subtasks for locating the bottle, cooling it, and placing it. The resulting retrieved set contains `alfworld-object-locator`, `alfworld-object-cooler`, and `alfworld-object-placer`, which correspond directly to the three required phases. The agent then follows the expected chain: it searches receptacles, takes the bottle from the dining table, moves to the refrigerator, cools the bottle, and returns it to the dining table.

The aligned GoS episode fails after 30 steps. Its initial retrieved set is `alfworld-temperature-regulator`, `alfworld-device-operator`, `alfworld-object-state-modifier`, and `alfworld-tool-user`; none is the task-specific cooler, locator, or placer used by the successful episode. Runtime retrieval subsequently returns unrelated skills such as `jackson-security` and `dc-power-flow`. The trajectory consequently spends its budget checking receptacles while explicitly treating the injected skills as irrelevant, and never reaches the required cool-and-place transition.

This example isolates the main failure mode: the difference is visible before execution in the retrieved skill set. Decomposition turns the compound request into phase-specific queries, allowing the state-changing operation ("cool") and the supporting locate/place operations to be retrieved independently. The GoS result instead retrieves broadly related or unrelated skills for the compound request, so the agent lacks an actionable procedure for the required state transition. Across the full paired evaluation, this pattern is consistent with the aggregate result: the decomposed run succeeds on 107/140 tasks (76.4%), compared with 79/140 (56.4%) for GoS; for cool-and-place tasks the corresponding rates are 15/25 (60.0%) and 2/25 (8.0%), respectively.

E.2 Case study 2: decomposition prevents a phase mismatch

The task at index 5 is "put a hot mug in cabinet." The planner-guided run decomposes the instruction into locating the mug, acquiring it, heating it, and placing it in the cabinet. It retrieves `alfworld-object-locator`, `alfworld-object-picker`, `alfworld-heat-object-with-appliance`, and `alfworld-object-placer`; the

	Decomposed planner	GoS
Retrieved skills	locator, cooler, placer	temperature regulator, device operator, state modifier, tool user
Outcome	success, 19 steps	failure, 30 steps
Observed behavior	locate → take → cool → place	repeated search; unrelated runtime skills; no completed cool/place chain

Table 12: Paired ALFWorld case study for cooling and placing a wine bottle (index 0).

	Rewriting Decomposed planner	w/o Rewriting planner
Stages exposed	locate, acquire, heat, place	heat procedure plus generic retrieval
Retrieved skills	locator, picker, heater, placer	heat-appliance skill, generic retriever
Outcome	success, 8 steps	failure, 30 steps
Observed behavior	countertop -¿ take -¿ microwave -¿ cabinet	wrong search/object match followed by recovery actions

Table 13: Paired ALFWorld case study for heating and placing a mug (index 5).

agent completes the task in 8 steps. The recorded trajectory is correspondingly short: it takes the mug from the countertop, opens the microwave, heats the mug, opens a cabinet, and places the hot mug inside.

The w/o rewirting episode for the same task fails at the 30-step limit. Although it retrieves a heating-related skill, its loaded set contains `alfworld-heat-object-with-appliance` and the generic `alfworld-object-retriever`, without the complete locate/pick/place support used by the successful run. The agent searches the wrong receptacles, mistakes a cup for the requested mug, and later issues placement and recovery actions after the object state has become inconsistent. Thus, the contrast is not simply that one agent produced a better final plan: the decomposed query sequence exposes the prerequisites and postconditions of the heating phase, while the earlier run retrieves a procedure for the transformation without a reliable acquisition-and-placement chain.

This case also illustrates why the effect of decomposition is most visible on multi-stage state-changing tasks. The planner provides a separate retrieval opportunity for each phase, and the injected skill brief supplies the agent with an executable order and verification points. In the paired aggregate comparison, the decomposed run improves from 33/140 (23.6%) to 107/140 (76.4%), with the largest task-type gains on heat-and-place (0/16 to 11/16) and two-object placement (4/24 to 21/24). These aggregate values should be read together with the case study: decomposition does not guarantee success when the underlying object search or environment interaction remains difficult, but it substantially reduces failures caused by missing or misaligned phase skills.

F Planner-Guided Decomposition Prompt

The planner prompt is intentionally summarized here rather than reproduced in full. Its purpose is to convert a compound household instruction into an ordered chain of retrieval-oriented subtasks. The prompt is instantiated with the task-specific vocabulary and values at runtime; the notation {`variable`} below marks those instantiations. The same cached planner output is then used by the repeated ALFWorld runs described in Section A.3.

The core prompt has four requirements. First, it asks the model to split the task into {`NUM_SUBTASKS`} ordered subtasks, where the permitted range is represented as {`MIN_SUBTASKS`}–{`MAX_SUBTASKS`}. The subtasks form a scene chain using the applicable phases from {`PHASE_SET`} (typically locate, acquire, transform, place, and inspect). Second, every subtask must include a machine-readable phase tag, local goal, required precondition, and produced postcondition. Third, every subtask generates two retrieval-query styles: one concise `plain` query and one or more structured `scene` queries. Finally, the model must return only the requested JSON object, preserve object names from the input, and keep transformation and placement as separate phases.

The abstracted system instruction is:

```
You are a retrieval planner for an agent backed by a scene-graph skill engine.

1. Decompose {TASK} into {NUM_SUBTASKS} ordered subtasks, with
   {MIN_SUBTASKS} <= {NUM_SUBTASKS} <= {MAX_SUBTASKS}.
   Form a phase chain using {PHASE_SET} as needed.
2. For each subtask, provide:
   - phase: a machine-readable phase tag;
   - goal: the local objective;
   - requires: the precondition state;
   - produces: the expected postcondition state.
3. For each subtask, write:
   - one plain query: concise action/object keywords;
   - {NUM_SCENE_QUERIES} scene query or queries: include the action,
     {OBJECT}, {RECEPTACLE_OR_APPLIANCE}, precondition, and verification.
     Keep each scene query within {MAX_SCENE_WORDS} words and use the
     vocabulary in {SCENE_PATTERN_VOCABULARY}.
4. Preserve object names exactly as written in {TASK}. Never merge a
   transformation phase with a placement phase.

Return only the JSON object.
```

The output contract is:

```
{
  "subtasks": [
    {
      "scene_name": "{SCENE_NAME}",
      "phase": "{PHASE}",
      "goal": "{LOCAL_GOAL}",
      "requires": "{REQUIRED_STATE}",
      "produces": "{PRODUCED_STATE}",
      "queries": [
        {"style": "plain", "q": "{PLAIN_QUERY}"},
        {"style": "scene", "q": "{SCENE_QUERY}"}
      ]
    }
  ]
}
```

For illustration, the prompt's few-shot example is represented abstractly as follows, with both the task and all task-specific entities replaced by variables:

```
Example task: {EXAMPLE_TASK}
Example output:
{
  "subtasks": [
    {
      "scene_name": "{LOCATE_SCENE}", "phase": "LOCATE",
      "goal": "{LOCATE_GOAL}", "requires": "{INITIAL_STATE}",
      "produces": "{LOCATED_STATE}",
      "queries": [
        {"style": "plain", "q": "{LOCATE_QUERY}"},
        {"style": "scene", "q": "{LOCATE_SCENE_QUERY}"}
      ]
    },
    {
      "scene_name": "{TRANSFORM_SCENE}", "phase": "{TRANSFORM_PHASE}",
```

```
      "goal": "{TRANSFORM_GOAL}", "requires": "{HELD_STATE}",
      "produces": "{TRANSFORMED_STATE}",
      "queries": [
        {"style": "plain", "q": "{TRANSFORM_QUERY}"},
        {"style": "scene", "q": "{TRANSFORM_SCENE_QUERY}"}
      ]
    },
    {
      "scene_name": "{PLACE_SCENE}", "phase": "PLACE",
      "goal": "{PLACE_GOAL}", "requires": "{TRANSFORMED_HELD_STATE}",
      "produces": "{PLACED_STATE}",
      "queries": [
        {"style": "plain", "q": "{PLACE_QUERY}"},
        {"style": "scene", "q": "{PLACE_SCENE_QUERY}"}
      ]
    }
  ]
}
```

In the deployed ALFWorld planner, the concrete scene vocabulary includes preparation, action, verification, and locate-family patterns. For example, an instantiated scene query may describe that an object is in hand, that the agent is at {RECEPTACLE}, that a single household action should be performed, and that the resulting state should be verified. These details are retained as retrieval cues rather than as additional instructions to the execution agent. Invalid planner outputs are checked against the schema; a single repair attempt is allowed, after which the deterministic task-template decomposition is used as a fallback.