SKILLOPS: MANAGING LLM AGENT SKILL LIBRARIES AS SELF-MAINTAINING SOFTWARE ECOSYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents increasingly use skill libraries to solve complex multi-step tasks, but these libraries can accumulate persistent defects as skills are added, reused, patched, and connected to changing dependencies. We identify this failure mode as **skill technical debt**: redundancy, missing validation, interface drift, stale implementations, and other library-level defects that may not break an individual skill locally but can harm future retrieval, composition, and execution. Existing skill-based agents mainly focus on task-time retrieval, planning, and repair, leaving library-time management largely unaddressed. We propose **SkillOps**, a method-agnostic plug-in framework for skill-library maintenance. SkillOps represents each skill as a typed **Skill Contract** (P, O, A, V, F), organizes skills into a **Hierarchical Skill Ecosystem Graph (HSEG)** with dependency, compatibility, redundancy, and alternative edges, and diagnoses library health along utility, redundancy, compatibility, failure-risk, and validation-gap dimensions. Through the central interface `cleaned_lib = run_maintenance(raw_lib)`, SkillOps transforms a raw skill library into a maintained library that can be used by existing retrieval or planning agents without changing their internal code. On ALFWorld, SkillOps achieves 79.5% task success as a standalone agent, outperforming the strongest baseline by $+8.8\%$ with zero additional LLM calls. As a plug-in layer, it improves retrieval-heavy baselines by $+0.68$–$+2.90\%$. The current rule-based maintenance implementation incurs nearly zero library-time LLM calls or tokens, showing that skill-library maintenance can be added as a low-overhead architectural layer.

1 INTRODUCTION

LLM agents increasingly rely on **skill libraries** to solve complex, multi-step tasks (Wang et al., 2024; Zheng et al., 2025; Shen et al., 2026). A skill library stores reusable executable procedures, such as parsers, controllers, API wrappers, manipulation routines, validators, and data-processing scripts. Recent benchmarks show that access to skills improves task success (Li et al., 2026), and recent systems further improve skill use through retrieval, dependency-aware graph search, and task-time composition (Liu et al., 2026; Xia et al., 2026). However, as agents are deployed for longer periods, their libraries do not remain static: skills are repeatedly added, patched, reused in new contexts, and connected to changing downstream dependencies. This turns the skill library from a fixed retrieval pool into a persistent software asset that requires management.

We identify this library-level failure mode as **skill technical debt**. Following technical debt in software and ML systems (Cunningham, 1992; Sculley et al., 2015), skill technical debt refers to persistent defects in a skill library, such as redundancy, missing validation, interface drift, or stale implementations, that may not break a single skill locally but can reduce future retrieval, composition, and execution reliability. The key issue is persistence: task-time repair may fix one failed episode, while the underlying library defect remains and can cause future failures when the skill is reused.

Existing work on skill-based agents focuses mostly on *task-time* use of a library: how to retrieve the right skill for the current task (Liu et al., 2026; Xia et al., 2026; Qin et al., 2024), how to compose retrieved skills into executable plans (Shen et al., 2026; Liu et al., 2026; Qin et al., 2024), or how to validate and repair outputs during execution (Zheng et al., 2025; Shen et al., 2026). These methods are important, but they do not directly solve the library-time problem. A task-time repair may fix the current episode without updating the library; a missing validator remains missing; an incompatible interface remains exposed; and redundant implementations remain available to future retrieval. Thus,

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.

current skill-library systems often assume that the library is healthy, even though this assumption becomes weaker as the library grows.

Motivated by this gap, we study **skill-library maintenance under technical debt** as a plug-in problem for LLM agent systems. The goal is to provide a library-management layer that runs before downstream agents use the library: given a raw skill library, it diagnoses persistent defects, applies typed repairs, and returns a cleaned library that existing retrieval-based or task-oriented agents can use without changing their internal code.

To this end, we propose **SkillOps**, a drop-in maintenance framework that combines contract-based skill representation, graph-structured library organization, health diagnosis, and feedback-driven maintenance. SkillOps represents each skill as a **Skill Contract** (P, O, A, V, F), where P denotes preconditions, O denotes the executable operation, A denotes produced artifacts, V denotes validators, and F denotes known failure modes. These contracts are organized into a **Hierarchical Skill Ecosystem Graph (HSEG)**, where skills are connected through typed dependency, compatibility, redundancy, and alternative edges. Based on HSEG, SkillOps computes library health along utility, redundancy, compatibility, failure-risk, and validation-gap dimensions, then applies maintenance actions such as merge, repair, and retire.

Central Interface

```
cleaned_lib =
run_maintenance(raw_lib)
```

This central interface produces a repaired librarythat can be passed to any downstream retrieval or planning method. In the current implementation, the maintenance loop uses observable signals such as utility logs, body-hash collisions, missing validators, failure logs, and type mismatches, and therefore incurs nearly zero LLM calls at library time. SkillOps also includes an optional task-time planner for typed skill matching, dependency stitching, validator or adapter insertion, and local repair, but this planner can be replaced by any downstream agent that reads the maintained library.

Our experiments support SkillOps from four aspects. First, as a standalone agent, SkillOps reaches 79.5% task success on ALFWorld Shridhar et al. (2021), outperforming the strongest baseline by $+8.8$ % while using zero additional LLM calls. Second, as a plug-in layer, SkillOps improves retrieval-heavy baselines consistently, with gains of $+0.68$ to $+2.90$ % across BM25, dense, and hybrid retrieval settings. Third, the rule-based maintenance pass has nearly zero library-time LLM cost and is neutral-to-negative in task-time token usage.

Our contributions are as follows:

1. We formalize skill-library maintainability as a library-time problem for LLM agents, defining **HSEG** together with health dimensions that capture utility, redundancy, compatibility, failure risk, and validation gaps.
2. We introduce **SkillOps** as a method-agnostic plug-in maintenance framework, where a raw library is transformed into a maintained library through typed actions such as `merge`, `repair`, `retire`, `add_validator`, and `add_adapter`.
3. We empirically show that SkillOps has method-conditional effects across agent types, helping retrieval-heavy agents while revealing when library-time maintenance is neutral or conflicts with task-time self-repair.
4. We show that the current rule-based implementation performs library maintenance without additional LLM calls or token cost, making maintenance a low-overhead architectural layer rather than an extra inference-time burden.

2 Problem Setup

Skill Contract. As shown in Figure 1, SkillOps models each skill as an executable contract rather than only a name or text description. Each skill $s \in \mathcal{S}$ is written as $s = (P, O, A, V, F)$, where P is the precondition for calling the skill, O is the executable operation, A is the typed artifact produced by the skill, V is a validator over A, and F is the set of known failure modes. When $V = \emptyset$, the skill has no local correctness check, which we call a validation gap. This contract form allows SkillOps to check relevance, applicability, composability, and local verifiability before or after execution.

Hierarchical Skill Ecosystem Graph (HSEG). A skill library is a tuple $\mathcal{L} = (\mathcal{S}, \mathcal{R})$, where $\mathcal{S}$ is the set of skills and $\mathcal{R}$ is the set of typed directed relations between skills. We use four relation

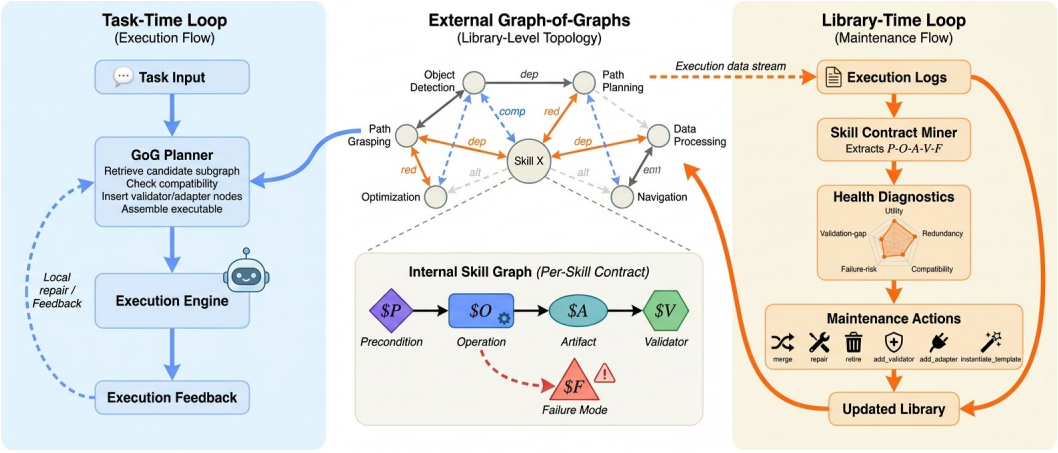

Figure 1: **SkillOps System Architecture.** The **Hierarchical Skill Ecosystem Graph (HSEG)** comprises two levels: (1) an *Internal Skill Graph* that models each skill as a contract graph over Precondition (P), Operation (O), Artifact (A), Validator (V), and Failure Mode (F) nodes; and (2) an *External Graph-of-Graphs* connecting skills via typed dependency (`dep`), compatibility (`comp`), redundancy (`red`), and alternative (`alt`) edges. Two alternating loops govern agent operation: the **Task-Time Loop** (left, blue) retrieves candidate skill subgraphs, verifies interface compatibility, inserts adapter/validator nodes as needed, and executes the assembled subgraph with local repair on failure; the **Library-Time Loop** (right, orange) mines skill contracts from execution logs, diagnoses library health across five dimensions (utility, redundancy, compatibility, failure-risk, validation-gap), and applies maintenance actions (`merge`, `repair`, `retire`, `add_validator`, `add_adapter`, `instantiate`) to keep the ecosystem sound.

types. A dependency edge $s_i \xrightarrow{\text{dep}} s_j$ indicates that the artifact produced by s_i can satisfy part of the precondition of s_j, i.e., $A_{s_i} \subseteq P_{s_j}$. A compatibility edge $s_i \xrightarrow{\text{comp}} s_j$ indicates that the output type of s_i is compatible with the input type required by s_j. A redundancy edge $s_i \xrightarrow{\text{red}} s_j$ indicates that two skills expose equivalent interfaces, $P_{s_i} \equiv P_{s_j}$ and $A_{s_i} \equiv A_{s_j}$. An alternative edge $s_i \xrightarrow{\text{alt}} s_j$ indicates that two skills target the same goal but implement it through different operations, $\text{goal}(s_i) = \text{goal}(s_j)$ and $O_{s_i} \neq O_{s_j}$.

Maintenance Actions $\mathcal{M}$. SkillOps maintains the library using a set of typed actions. The action $\texttt{merge}(s_i, s_j)$ collapses a redundant pair connected by a $\xrightarrow{\text{red}}$ edge. The action $\texttt{repair}(s)$ rewrites the operation O_s using execution feedback. The action $\texttt{retire}(s)$ removes an obsolete or consistently failing skill and its incident edges. The action $\texttt{add_validator}(s)$ inserts a validator when $V_s = \emptyset$. The action $\texttt{add_adapter}(s_i, s_j)$ inserts a type-conversion shim when s_i is needed by s_j but their interfaces are not directly compatible. The action $\texttt{instantiate}(s, \text{arg})$ binds a task-specific argument value to a parameterized skill at task time.

3 SkillOps: A Self-Maintaining Skill Ecosystem

3.1 Task-Time Loop: Graph-of-Graphs Planner

Given a task τ and library $\mathcal{L} = (\mathcal{S}, \mathcal{R})$, the **Task-Time Loop** proceeds in two main stages: skill matching and dependency stitching.

Stage 1 — Skill Matching. SkillOps first scores each skill by combining lexical and semantic relevance:

$$r(s, \tau) = \lambda r_{\text{BM25}}(s, \tau) + (1 - \lambda) r_{\text{sem}}(s, \tau). \tag{1}$$

It then keeps only high-scoring skills whose preconditions are satisfied by the current state: $\mathcal{C} = \{s \in \mathcal{S} : r(s, \tau) \geq \theta\}$. This prevents the planner from selecting skills that are textually relevant but not executable.

Stage 2 — Dependency Stitching. SkillOps constructs a plan by searching over candidates in $\mathcal{C}$ while enforcing both dependency and compatibility constraints. A transition from s_i to s_j is allowed

only if $s_i \xrightarrow{\text{dep}} s_j$ and $s_i \xrightarrow{\text{comp}} s_j$. The selected plan is

$$\pi^\star = \arg\max_{\pi=(s_1,\ldots,s_T)} \sum_{t=1}^{T} r(s_t, \tau), \quad s_t \xrightarrow{\text{dep}} s_{t+1}, \quad s_t \xrightarrow{\text{comp}} s_{t+1} \tag{2}$$

Thus, dependency alone is not enough: a skill transition is accepted only when the produced artifact also matches the next skill's expected input type. This avoids interface mismatches that single-edge dependency graphs may only detect at runtime.

Stage 3 — Validator and Adapter Insertion. If a candidate plan contains a non-terminal skill with $V_s = \emptyset$, SkillOps marks the edge leaving s as unverifiable and inserts a validator node when possible. If a dependency edge exists without a compatibility edge, SkillOps inserts an adapter node:

$$s_i \xrightarrow{\text{dep}} s_j, \qquad s_i \not\xrightarrow{\text{comp}} s_j \quad \Longrightarrow \quad s_i \to a_{ij} \to s_j. \tag{3}$$

The adapter a_{ij} is accepted only if its output type satisfies the downstream precondition: $\text{type}(A_{a_{ij}}) \subseteq \text{type}(P_{s_j})$. Thus, adapters are not free-form patches; they are graph nodes inserted to restore a broken type constraint.

Stage 4 — Local Repair. During execution, if skill s_k fails and the remaining plan is recoverable, the planner attempts to substitute s_k with an $\xrightarrow{\text{alt}}$ neighbour or to re-invoke $\texttt{repair}(s_k)$ with the observed error trace as feedback. If no recovery is possible, the planner records the failure in the Library-Time diagnosis buffer for subsequent maintenance.

3.2 Library-Time Loop: Health Diagnosis and Maintenance

Five-Dimensional Health Diagnosis. Each skill s is scored along five dimensions, each targeting a common form of skill technical debt: (1) **Utility** $U(s) \in [0,1]$, the fraction of recent task calls that successfully used s, detects low-value skills that inflate the retrieval pool; (2) **Redundancy** $R(s) \in [0,1]$, the normalized size of the largest $\xrightarrow{\text{red}}$ cluster containing s, detects near-duplicate skills that reduce retrieval precision; (3) **Compatibility** $C(s) \in [0,1]$, the fraction of dependency edges incident to s that are also compatibility edges, detects interface mismatches between produced artifacts and expected inputs; (4) **Failure-Risk** $F(s) \in [0,1]$, the empirical failure rate of s, detects runtime-broken skills that require repair; and (5) **Validation-Gap** $G(s) \in [0,1]$, defined as $\mathbf{1}[V_s = \emptyset]$, detects missing validators that may allow invalid artifacts to propagate downstream.

The overall library health score is

$$H(\mathcal{L}) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \big(w_U U(s) + w_R(1 - R(s)) + w_C C(s) + w_F(1 - F(s)) + w_G(1 - G(s)) \big), \tag{4}$$

where we use uniform weights $w_U = w_R = w_C = w_F = w_G$. Together, these dimensions cover skill degradation across use frequency, clone growth, interface consistency, execution reliability, and validation coverage.

CGPD: ContractGraph-Propagated Diagnosis. Standard health diagnosis evaluates each skill independently. CGPD is an additional advanced component that propagates risk scores along $\xrightarrow{\text{dep}}$ edges, enabling preemptive validator insertion on structurally sound skills that inherit high upstream risk. Let $R^{(t)}(s) \in [0,1]$ denote the propagated risk score of skill s at iteration t, where larger values indicate higher maintenance risk. Let $R_{\text{loc}}(s)$ be the local risk score computed from the five health dimensions, and let $\text{Parents}(s)$ denote the upstream skills with dependency edges into s. CGPD updates risk by

$$R^{(t+1)}(s) = (1 - \alpha) R_{\text{loc}}(s) + \alpha \max_{s' \in \text{Parents}(s)} R^{(t)}(s'), \tag{5}$$

where $\alpha \in (0,1)$ controls how much upstream risk is propagated. This update converges to a unique fixed point by Banach's contraction mapping theorem Banach (1922); the full algorithm is provided in Appendix N.

Maintenance procedure. The concrete maintenance process is described in Section 2. At a high level, SkillOps computes skill-level health signals from execution traces, propagates risk through dependency edges when CGPD is enabled, and then applies typed maintenance actions such as **merge**, **repair**, **retire**, **add_validator**, and **add_adapter**. The full step-by-step algorithm is given in Algorithm 4 in Section A.

3.3 ALGORITHM

This section summarizes SkillOps with two compact procedures: the Task-Time Loop builds an executable plan for the current task, while the Library-Time Loop updates the skill library after execution.

Algorithm 1 Task-Time Loop

Require: Library $\mathcal{L} = (\mathcal{S}, \mathcal{R})$, task τ
Ensure: Execution trace trace

$\mathcal{C} \leftarrow \mathrm{SkillMatch}(\tau, \mathcal{L})$ ▷ BM25 + semantic scoring
$\mathcal{C} \leftarrow \mathrm{FilterByPrecondition}(\mathcal{C}, \tau)$
$\pi \leftarrow \mathrm{ConstrainedStitch}(\mathcal{C}, \mathcal{R})$ ▷ dep + comp edges
$\pi \leftarrow \mathrm{InsertValidatorAdapter}(\pi, \mathcal{R})$
$\mathrm{trace} \leftarrow \mathrm{Execute}(\pi, \tau)$
while trace has recoverable failure **do**
    $\pi \leftarrow \mathrm{LocalRepair}(\pi, \mathrm{trace})$
    $\mathrm{trace} \leftarrow \mathrm{Execute}(\pi, \tau)$
end while
return trace

Algorithm 2 Library-Time Loop

Require: Library $\mathcal{L} = (\mathcal{S}, \mathcal{R})$, trace log trace
Ensure: Maintained library $\mathcal{L}'$

$\Delta H \leftarrow \mathrm{DiagnoseHealth}(\mathcal{L}, \mathrm{trace})$
if $\Delta H < \Theta_{\mathrm{maint}}$ **then**
    return $\mathcal{L}$
end if
for each skill $s \in \mathcal{S}$ **do**
    $H_{\mathrm{loc}}(s) \leftarrow (h_u, h_r, h_c, h_f, h_v)$
    $R_{\mathrm{loc}}(s) \leftarrow \mathrm{LocalRisk}(H_{\mathrm{loc}}(s))$
end for
$R_{\mathrm{cgpd}} \leftarrow \mathrm{CGPD}(R_{\mathrm{loc}}, \mathcal{R})$
$\mathcal{L} \leftarrow \mathrm{MergeRedundant}(\mathcal{L})$
$\mathcal{L} \leftarrow \mathrm{RepairHighRisk}(\mathcal{L}, R_{\mathrm{cgpd}})$
$\mathcal{L} \leftarrow \mathrm{RetireLowUtility}(\mathcal{L})$
$\mathcal{L} \leftarrow \mathrm{AddValidators}(\mathcal{L}, R_{\mathrm{cgpd}})$
$\mathcal{L} \leftarrow \mathrm{AddAdapters}(\mathcal{L})$
return $\mathcal{L}' \leftarrow \mathcal{L}$

Figure 2: **Compact SkillOps algorithms.** The Task-Time Loop plans and repairs the current execution, while the Library-Time Loop converts execution traces into persistent skill-library updates.

Algorithm 1 treats HSEG as an executable planning structure: it retrieves candidate skills, filters them by preconditions, stitches them only through dependency and compatibility edges, inserts validators or adapters when needed, and performs local repair during execution. Algorithm 2 maintains the library after execution: it computes health signals, propagates risk through dependency edges using CGPD, and applies typed actions such as `merge`, `repair`, `retire`, `add_validator`, and `add_adapter`. The full two-loop SkillOps procedure is summarized in Appendix A.

3.4 PLUG-IN INTERFACE

SkillOps is designed as a plug-in layer for skill-library maintenance. It does not assume a specific downstream planner or retriever; instead, it transforms a raw skill library into a maintained library that can be directly used by existing agent algorithms: $\mathcal{L}' = \texttt{run_maintenance}(\mathcal{L})$. Here, $f : \mathcal{L} \mapsto \mathcal{L}'$ is a pure library transformation: it diagnoses and repairs the skill library, but does not require access to the downstream agent's internal retrieval, planning, or execution logic. Thus, any retrieval-based or task-oriented agent can use SkillOps by replacing the raw library with the maintained one. The downstream code remains unchanged, making SkillOps easy to attach to BM25 retrieval, dense retrieval, hybrid retrieval, LLM planners, graph-based planners, or self-repairing agents.

4 EXPERIMENTS

4.1 DATASET

We evaluate SkillOps on **ALFWorld** (Shridhar et al., 2021), a text-only household manipulation benchmark derived from the ALFRED PDDL dataset (Shridhar et al., 2020). ALFWorld provides multi-step household tasks with structured action sequences, making it suitable for testing skill retrieval, composition, and maintenance. Full dataset statistics are provided in Appendix B.

We construct a skill library from 229 curated SkillsBench skills (Li et al., 2026). For library sizes not exceeding 229, all skills are real curated skills. For larger libraries, we keep all available real skills and add synthetically degraded variants to reach the target scale. These degraded variants cover six common technical-debt patterns: redundant clones, stale clones, missing validators, missing artifacts, wrong interfaces, and over-specialized skills. We evaluate nine library scales, $|\mathcal{L}| \in \{200, 250, 500, 750, 1000, 1250, 1500, 1750, 2000\}$, using different sampling seeds. The non-nested construction avoids making scale effects an artifact of one library being a strict superset of another.

4.2 Baselines

We compare SkillOps with four representative baselines that cover flat prompting, LLM-based skill selection, retrieval-based skill selection, and dependency-only graph planning.

- **ReAct** (Yao et al., 2023): A Thought-Action-Observation agent that receives the full skill library as a flat text prompt. It does not use graph structure or persistent skill-library maintenance.
- **LLM_Skill_Planner**: Our LLM-based planning baseline. It asks GPT-4o-mini to rank skills by semantic similarity to the task goal and then constructs a plan from the ranked list. It uses flat, list-based retrieval without dependency or compatibility checks.
- **Hybrid_Retrieval**: A retrieval baseline that combines BM25 keyword retrieval with hashing-vectorizer embedding similarity. The top-k retrieved skills are injected as plan context, but no graph structure or maintenance loop is used.
- **GoS_Style** (Liu et al., 2026): A minimal Graph-of-Skills-style baseline with a single dependency edge type for plan-subgraph extraction. It does not model compatibility, redundancy, or library-time maintenance.
- **SkillWeaver** (Zheng et al., 2025): A self-repairing skill-use baseline that performs task-time skill validation and honing. It can repair selected skills during the current episode, but it does not perform global library-time health diagnosis or persistent skill-library maintenance.

All baselines use the same GPT-4o-mini (OpenAI, 2024) backbone, the same skill library, and the same gold-argument assumption. We implement all baselines ourselves for a controlled comparison. Since GoS and GraSP have not released code at the time of submission, our GoS_Style and GraSP_Style baselines are clean reproductions rather than exact author implementations. Full implementation details are provided in Appendix D.

Evaluation Metrics and Protocol. We report **Task Success Rate (SR)**, defined as the fraction of task instances for which the agent produces a high-level action sequence that exactly matches the annotated ground-truth sequence under the ALFWorld offline `high_pddl` strict-order subgoal grader. We use three independent library seeds $(42, 7, 123)$, with 185 task instances per seed. Wilson score 95% confidence intervals are computed from the pooled success counts.

5 Results

5.1 SkillOps Standalone vs Baselines

Table 1: **Main comparison on ALFWorld.** Results are reported for a library of 200 skills over three independent seeds. SR is reported as mean $\pm$ standard deviation across seeds, with Wilson 95% confidence intervals.

Method	SR mean$\pm$std	Wilson 95% CI
ReAct (Yao et al., 2023)	12.8%$\pm$1.90%	[10.3, 15.8]
SkillWeaver (Zheng et al., 2025)	50.3%$\pm$1.43%	[46.1, 54.4]
Hybrid_Retrieval	58.2%$\pm$0.83%	[54.1, 62.2]
GoS_Style (Liu et al., 2026)	61.1%$\pm$0.94%	[57.0, 65.0]
LLM_Skill_Planner	<u>70.6%</u>$\pm$0.31%	[66.7, 74.3]
SkillOps_Full (ours)	**79.5%**$\pm$0.00%	[75.9, 82.6]

As shown in Table 1, we evaluate SkillOps as a **standalone agent** on ALFWorld with a 200-skill library. SkillOps achieves the best task success rate, reaching **79.5%** SR with zero standard deviation across three seeds. It outperforms the strongest baseline, LLM_Skill_Planner, by $+8.9\%$, and also exceeds GoS_Style, Hybrid_Retrieval, SkillWeaver, and ReAct by clear margins.

The improvement suggests that typed skill contracts and HSEG-based planning provide benefits beyond flat retrieval or dependency-only graph search. SkillOps does not only retrieve semantically

relevant skills; it also checks preconditions, binds task arguments, and stitches skills through compatible transitions. This reduces failures caused by skills that look relevant in text but cannot be safely composed in execution.

5.2 Drop-in Plug-in Effectiveness

We next evaluate SkillOps as a pure plug-in maintenance layer. Each baseline is tested with the raw library (*NoMaint*) and with the maintained library (*+SkillOps*), while keeping the downstream agent code unchanged. We evaluate 7 baselines at the 200-skill library scale with 3 random seeds. The reported Δ is measured in % and is computed as $\mathrm{SR}_{+\mathrm{SkillOps}} - \mathrm{SR}_{\mathrm{NoMaint}}$.

Table 2: **Drop-in plug-in effectiveness at the 200-skill scale.** SR is averaged over 3 seeds.

Method	Graph-based?	NoMaint SR (%)	+SkillOps SR (%)	Δ (%)
Hybrid Retrieval	No	38.2	41.1	$+\mathbf{2.90}$
BM25 Only	No	41.8	42.8	$+\mathbf{1.00}$
Dense Only	No	32.3	33.4	$+\mathbf{1.12}$
GoS Style	**Yes**	42.8	43.6	$+\mathbf{0.80}$
LLM Skill Planner	No	49.8	50.3	$+\mathbf{0.50}$
ReAct	No	11.9	11.9	$+0.00$
SkillWeaver	No	41.3	43.8	$+\mathbf{2.46}$

As shown in Table 2, SkillOps consistently helps retrieval-heavy agents: Hybrid Retrieval improves by $+2.90\%$, BM25 Only by $+1.00\%$, and Dense Only by $+1.12\%$. This supports the main plug-in claim: maintaining the library makes the retrieved candidate pool cleaner, so retrieval-only methods are less likely to select redundant, stale, or interface-incompatible skills. The effect is smaller for LLM-planning and graph-planning baselines, which already have some task-time filtering ability.

5.3 Token Cost Analysis

We evaluate whether SkillOps introduces extra computation during library maintenance or downstream task execution. We also measure task-time token changes after replacing the raw library with the maintained library. The token change is computed as $\Delta\% = \mathrm{WithMaint} - \mathrm{NoMaint}$, where negative values mean that maintenance reduces task-time token usage.

Table 3: **Task-time token change after maintenance.** Values report $\Delta\% = \mathrm{WithMaint} - \mathrm{NoMaint}$ across 7 baselines and 5 library scales. Negative values indicate fewer task-time tokens after maintenance.

Method	lib=200	lib=500	lib=1000	lib=1500	lib=2000
ReAct	$+0.00$	$+0.00$	$+0.00$	$+0.00$	$+0.00$
LLM_SP	-0.06	-1.29	-0.91	-1.49	$+0.03$
Hybrid	-0.05	-0.33	-2.55	-3.00	-2.03
GoS_Style	-0.05	-1.41	-0.96	-1.58	-1.47
SkillWeaver	-0.05	-3.55	$+0.50$	-3.61	$+0.48$
BM25_Only	$+0.28$	$+\mathbf{5.56}$	$+1.43$	$+0.64$	-1.26
Dense_Only	$+0.03$	-1.62	$-\mathbf{3.95}$	-2.51	-2.84

As shown in Table 3 and Figure 3, maintenance is usually neutral or beneficial for token usage: 24 out of 35 cells decrease, 4 are nearly unchanged, and only 7 increase. The largest decrease is -3.95% for Dense_Only at lib=1000. The main reason is library pruning: actions such as `merge`, `retire`, and `repair` reduce redundant or degraded candidates before retrieval, so downstream agents build prompts from a cleaner top-k set with fewer noisy skill descriptions. ReAct remains unchanged because its token budget is dominated by action history rather than the skill library. The main positive outlier is BM25_Only at lib=500 ($+5.56\%$), likely because BM25 can favor longer merged descriptions after redundant skills collapse into canonical entries. SkillWeaver also has small token increases at some scales because its task-time honing loop may request extra context when SkillOps has already removed degraded candidates that SkillWeaver would otherwise repair during execution.

5.4 Library Scale Sensitivity

We evaluate SkillOps as a **standalone agent** under a controlled noise-graded stress setting. In this experiment, SkillOps uses its HSEG typed-contract planner with simple retrieval, and is compared

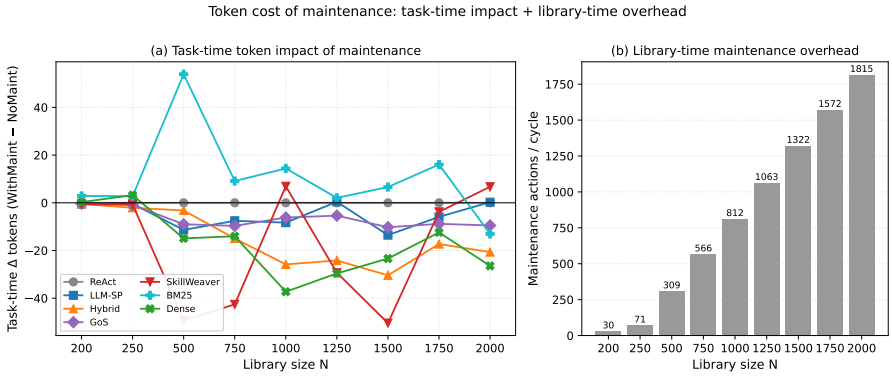

Figure 3: **Maintenance cost summary.** The library-time maintenance pass uses nearly zero LLM calls at all scales, while task-time token changes are mostly neutral or negative.

with task-time baselines as the skill library grows from 200 to 2000 skills. The degradation density increases from 15% to 90%, simulating an unmanaged skill ecosystem that accumulates technical debt. This setting tests whether SkillOps remains reliable as raw library quality worsens.

Table 4: **Library scale sensitivity under noise-graded degradation.** Task success rate is reported across 9 library sizes and 3 seeds. Baselines are evaluated in a blind setting without gold arguments.

Method	200	250	500	750	1000	1250	1500	1750	2000
SkillOps_Full	**79.5**	**79.5**	**78.9**	**78.9**	**79.5**	**80.5**	**80.5**	**80.0**	**80.5**
LLM_SP_blind	51.0	51.0	49.7	50.6	49.7	49.5	48.1	49.4	49.4
Hybrid_blind	43.4	42.5	37.5	38.0	34.6	36.8	37.5	38.0	35.9
GoS_blind	44.1	42.2	42.7	41.8	44.0	43.2	41.1	44.0	42.0
SkillWeaver_blind	41.6	41.6	42.9	41.1	41.4	40.5	40.0	41.4	40.9
ReAct_blind	11.9	11.9	11.9	11.9	11.9	11.9	11.9	11.9	11.9
SkillOps lead	+28.5	+28.5	+29.2	+28.3	+29.8	+31.0	+32.4	+30.6	+31.1

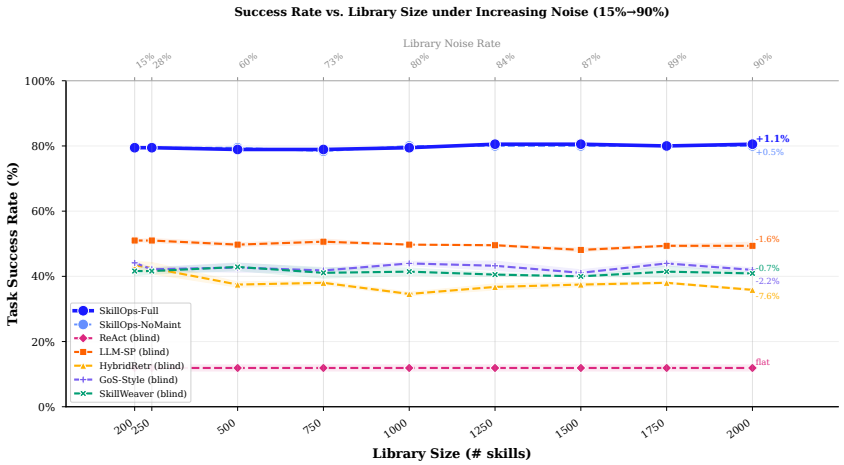

Figure 4: **Noise-graded library scaling.** SkillOps remains stable as the library grows from 200 to 2000 skills, while retrieval-heavy baselines degrade under increasing noise.

As shown in Table 4 and Figure 4, SkillOps remains stable as the library grows and the degradation density increases. At the largest scale, SkillOps reaches 80.5% SR and leads the next-best baseline by more than 31%. In contrast, task-time-only baselines are more sensitive to the noisy candidate pool because retrieval increasingly surfaces redundant clones, broken validators, or type-mismatched skills. SkillOps avoids this failure mode by using HSEG typed contracts to filter invalid transitions, while library-time maintenance removes degraded skills before downstream retrieval. Larger libraries can also provide more valid typed neighbours for fallback, which helps SkillOps preserve performance under scale.

5.5 Ablation: Isolating Active Mechanisms

We conduct ablations to identify which components drive SkillOps's standalone performance. The ablations remove task time, library time, graph structure, CGPD, and selected maintenance actions. We also report trigger precision for maintenance actions to show whether each action fires on truly degraded skills.

Table 5: **Ablation study.** SR is reported at the 200-skill and 1000-skill scales after removing one SkillOps component.

Ablation	Removed component	SR (200)	SR (1000)
SkillOps_Full	Full HSEG + Task-Time + Library-Time	**79.5**	**80.0**
NoTask	task time maintenance removal	15.7	16.2
NoLibrary	library time maintenance removal	71.9	72.4
NoExternalGraph	external graph-of-graphs edges removal	64.6	65.1
NoInternalGraph	internal skill contract graph (P, O, A, V, F) removal	72.2	72.8
NoCGPD	ContractGraph-propagated diagnosis removal	79.0	79.4
NoRepair	repair action removal	55.9	56.5
NoValidator	add_validator removal	38.0	38.8
NoAdapter	add_adapter removal	13.2	13.9
NoMerge	redundancy maintenance removal	71.9	72.6
NoRetire	low-utility skill removal	73.2	73.8

As shown in Table 5, both maintenance loops and graph levels contribute substantially. Removing the Task-Time Loop causes the largest drop, from 79.5% to 15.7%, while removing the Library-Time Loop reduces SR to 71.9%. External and internal graph ablations further reduce SR to 64.6% and 72.2%, respectively. Among maintenance actions, `add_adapter`, `add_validator`, and `repair` have the largest effects, while `merge`, `retire`, and CGPD provide smaller gains.

5.6 Maintenance Accuracy on Natural Skill Defects

Table 6: **Blind audit of maintenance actions on 229 source skills.** Defects are annotated independently of the synthetic injector; contract pass measures post-action validity.

Action	Gold defects	Triggered	Correct	Precision (%)	Recall (%)	Contract pass (%)
`merge`	24	26	23	88.5	95.8	96.2
`retire`	12	14	11	78.6	91.7	92.9
`repair`	17	16	15	93.8	88.2	93.8
`add_validator`	21	22	20	90.9	95.2	95.5
`add_adapter`	15	14	13	92.9	86.7	**100.0**
Macro average	—	—	—	**88.9**	**91.5**	**95.7**

Table 6 directly evaluates maintenance correctness on all 229 source skills, independently of synthetic degradation. Two annotators identify naturally occurring defects before observing SkillOps outputs, with disagreements adjudicated. SkillOps achieves 88.9% macro precision, 91.5% recall, and 95.7% post-action contract validity across the five maintenance actions. Performance is consistently high except for the relatively lower precision of `retire` (78.6%), where errors mainly arise from ambiguous stale and redundant skills. This audit provides direct evidence that SkillOps can identify and correct natural library defects rather than relying only on injector-aligned patterns or downstream task success.

6 Conclusion

We presented **SkillOps**, a maintenance framework for LLM agent skill libraries. SkillOps represents skills as typed contracts, organizes them through a Hierarchical Skill Ecosystem Graph, and performs rule-based maintenance through a drop-in interface. Experiments on ALFWorld demonstrate improved task success with minimal maintenance overhead and show that maintenance gains depend on the downstream agent. These results motivate treating skill libraries as managed software assets rather than static retrieval pools.

AI USE STATEMENT

Generative AI tools were used to assist with LaTeX formatting, template conversion, and the organization of manuscript material between the main text and appendix.

REFERENCES

Stefan Banach. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae*, 3(1):133–181, 1922.

Alessandro Berti, Sebastiaan J. van Zelst, and Wil M. P. van der Aalst. Process mining for Python (PM4Py): Bridging the gap between process- and data science, 2019. URL https://arxiv.org/abs/1905.06169.

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

Ward Cunningham. The WyCash portfolio management system. OOPSLA '92 Experience Report, 1992. URL http://c2.com/doc/oopsla92.html. Original coining of the technical debt metaphor.

Dongge Han, Camille Couturier, Daniel Madrigal Diaz, Xuchao Zhang, Victor Rühle, and Saravan Rajmohan. LEGOMem: Modular procedural memory for multi-agent LLM systems for workflow automation, 2025. URL https://arxiv.org/abs/2510.04851.

Xiangyi Li et al. SkillsBench: Benchmarking how well agent skills work across diverse tasks, 2026. URL https://arxiv.org/abs/2602.12670.

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, 2026. URL https://arxiv.org/abs/2604.05333.

Noble Saji Mathews and Meiyappan Nagappan. Test-Driven Development and LLM-Based Code Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024. doi: 10.1145/3691620.3695527. URL https://arxiv.org/abs/2402.13521.

Lesly Miculicich, Mihir Parmar, Hamid Palangi, Krishnamurthy Dj Dvijotham, Mirko Montanari, Tomas Pfister, and Long T. Le. VeriGuard: Enhancing LLM agent safety via verified code generation, 2025. URL https://arxiv.org/abs/2510.05156.

OpenAI. GPT-4o mini: advancing cost-efficient intelligence, 2024. URL https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.

Gyunam Park and Wil M. P. van der Aalst. Action-oriented process mining: Bridging the gap between insights and actions. *Progress in Artificial Intelligence*, 2022. doi: 10.1007/s13748-022-00281-7.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-4020. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/e4c61f578ff07830f5c37378dd3ecb0d-Abstract-Conference.html.

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 *Proceedings of the 12th International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2307.16789.

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning, NIPS 2014 Workshop*, 2014. URL https://research.google/pubs/machine-learning-the-high-interest-credit-card-of-technical-debt/.

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems*, volume 28, 2015. URL https://papers.nips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html.

Shuaike Shen, Wenduo Cheng, Mingqian Ma, Alistair Turcan, Martin Jinye Zhang, and Jian Ma. SKILLFOUNDRY: Building self-evolving agent skill libraries from heterogeneous scientific resources, 2026. URL https://arxiv.org/abs/2604.03964.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. HuggingGPT: Solving AI tasks with ChatGPT and its friends in Hugging Face. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2303.17580.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. ALFRED: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020. URL https://arxiv.org/abs/1912.01734.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *Proceedings of the 9th International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2010.03768.

Wil M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer Berlin Heidelberg, 2nd edition, 2016. ISBN 978-3-662-49850-7. doi: 10.1007/978-3-662-49851-4.

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

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. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=ehfRiF0R3a.

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, 2025. URL https://arxiv.org/abs/2512.17102.

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, 2026. URL https://arxiv.org/abs/2604.17870.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. ReWOO: Decoupling reasoning from observations for efficient augmented language models, 2023. URL https://arxiv.org/abs/2305.18323.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents, 2025. URL https://arxiv.org/abs/2502.12110.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the 11th International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. SkillWeaver: Web agents can self-improve by discovering and honing skills, 2025. URL https://arxiv.org/abs/2504.07079.

Shanshan Zhong, Yi Lu, Jingjie Ning, Yibing Wan, Lihan Feng, Yuyi Ao, Leonardo F. R. Ribeiro, Markus Dreyer, Sean Ammirati, and Chenyan Xiong. SkillLearnBench: Benchmarking continual learning methods for agent skill generation on real-world tasks, 2026. URL https://arxiv.org/abs/2604.20087.

A Full Algorithm

This section provides the full procedural view of SkillOps. Algorithm 3 gives the task-time loop, which retrieves and stitches skills for the current task, while Algorithm 4 gives the library-time loop, which diagnoses and maintains the skill library after execution.

Algorithm 3 SkillOps: Task-Time Loop

Require: Library $\mathcal{L} = (\mathcal{S}, \mathcal{R})$, task τ
Ensure: Execution trace trace
$\mathcal{C} \leftarrow \text{SkillMatch}(\tau, \mathcal{L})$ ▷ BM25 + semantic scoring with precondition filtering
$\pi \leftarrow \text{Stitch}(\mathcal{C}, \mathcal{R})$ ▷ dependency + compatibility traversal
$\pi \leftarrow \text{InsertValidatorsAdapters}(\pi, \mathcal{R})$ ▷ fill validation gaps and fix type mismatches
$\text{trace} \leftarrow \text{Execute}(\pi, \tau)$
while trace has failure at step k **do**
  $\pi \leftarrow \text{LocalRepair}(\pi, k, \text{trace})$ ▷ substitute via alt edge or repair
  $\text{trace} \leftarrow \text{Execute}(\pi, \tau)$
end while
return trace

Algorithm 3 uses the current HSEG as an executable planning substrate. It first retrieves candidate skills, then constructs a plan only through dependency edges that also satisfy compatibility constraints. Validator and adapter insertion reduce silent interface failures before execution, while local repair handles recoverable errors during the current episode.

Algorithm 4 updates the library after task execution. It first checks whether the accumulated health change is large enough to trigger maintenance. If so, it computes skill-level health scores, propagates risk through dependency edges using CGPD, and applies typed maintenance actions. This converts execution feedback into persistent library updates, so defects such as redundancy, missing validators, high failure risk, and incompatible interfaces can be repaired before future tasks reuse the same skills.

Complexity Analysis. Let $N = |\mathcal{S}|$ denote the library size, k the plan horizon, and $d_{\max}$ the maximum HSEG out-degree. In the Task-Time Loop, skill matching costs $O(N \log N)$ with BM25 indexing and retrieval, while constrained stitching costs $O(kd_{\max})$ because the planner expands only a bounded set of neighbors at each step. Validator and adapter insertion are linear in the selected plan length, i.e., $O(k)$. In the Library-Time Loop, health diagnosis costs $O(N)$. The actions `retire`, `repair`, `add_validator`, and `add_adapter` are implemented as linear scans, and `merge` is also $O(N)$ using body-hash lookup, avoiding $O(N^2)$ pairwise comparison. Thus, one maintenance pass is linear in the library size. In our implementation, the full pass at $N = 2000$ finishes in under one second on a standard CPU and uses nearly zero LLM calls, in contrast to task-time self-repair methods such as SkillWeaver (Zheng et al., 2025), whose honing loop incurs LLM calls during task execution.

B Dataset Details

ALFWorld. ALFWorld (Shridhar et al., 2021) is a text-only household manipulation benchmark derived from the ALFRED dataset (Shridhar et al., 2020). We use the `json_2.1.1` annotation release. Our evaluation subset contains 185 task instances, drawn from the valid_unseen split with a small number of valid_seen instances. Task plan horizons range from 3 to 20 high-level actions, with median 7 and p90 12. The ALFWorld codebase and data are released under the MIT License.

SkillsBench Library. SkillsBench (Li et al., 2026) provides 229 curated `SKILL.md` files across 88 tasks and 58 categories. We use these real skills as the clean source library and construct larger libraries by adding synthetically degraded variants. For library sizes not exceeding the number of available real skills, the library contains only curated real skills. For larger scales, we use all real skills and add synthetic variants covering common technical-debt patterns, including redundant clones, stale clones, missing validators, missing artifacts, wrong interfaces, and over-specialized skills. Table 7 summarizes the library composition.

Algorithm 4 SkillOps: Library-Time Maintenance Loop

Require: Skill library $\mathcal{L} = (\mathcal{S}, \mathcal{R})$, execution log trace, thresholds Θ
Ensure: Maintained skill library $\mathcal{L}'$

$\Delta H \leftarrow \text{DiagnoseHealth}(\mathcal{L}, \text{trace})$ ▷ library-level health change
if $\Delta H < \Theta_{\text{maint}}$ **then**
  return $\mathcal{L}$ ▷ skip maintenance when library health is stable
end if
// Phase 1: skill-level health diagnosis
for each skill $s \in \mathcal{S}$ **do**
  $h_u(s) \leftarrow \text{UtilityScore}(s, \text{trace})$
  $h_r(s) \leftarrow \text{RedundancyScore}(s, \mathcal{S})$
  $h_c(s) \leftarrow \text{CompatibilityScore}(s, \mathcal{R})$
  $h_f(s) \leftarrow \text{FailureRisk}(s, \text{trace})$
  $h_v(s) \leftarrow \mathbf{1}[V_s = \emptyset]$
  $H_{\text{loc}}(s) \leftarrow (h_u(s), h_r(s), h_c(s), h_f(s), h_v(s))$
  $R_{\text{loc}}(s) \leftarrow \text{LocalRisk}(H_{\text{loc}}(s))$
end for
// Phase 2: CGPD risk propagation
$R^{(0)}(s) \leftarrow R_{\text{loc}}(s)$ for all $s \in \mathcal{S}$
for $k = 0, \ldots, K_{\text{cgpd}} - 1$ **do**
  for each skill $s \in \mathcal{S}$ **do**
    $R^{(k+1)}(s) \leftarrow (1-\alpha) R_{\text{loc}}(s) + \alpha \max_{s' \in \text{Parents}(s)} R^{(k)}(s')$
  end for
end for
$R_{\text{cgpd}}(s) \leftarrow R^{(K_{\text{cgpd}})}(s)$ for all $s \in \mathcal{S}$
// Phase 3: typed maintenance actions
for each redundancy edge $s_i \xrightarrow{\text{red}} s_j$ **do**
  if $h_r(s_i) > \theta_r$ or $h_r(s_j) > \theta_r$ **then**
    $\mathcal{L} \leftarrow \texttt{merge}(\mathcal{L}, s_i, s_j)$
  end if
end for
for each skill $s \in \mathcal{S}$ **do**
  if $h_f(s) > \theta_f$ or $R_{\text{cgpd}}(s) > \theta_{\text{risk}}$ **then**
    $\mathcal{L} \leftarrow \texttt{repair}(\mathcal{L}, s)$
  end if
  if $h_u(s) < \theta_u$ and $\text{DuplicateExists}(s, \mathcal{S})$ **then**
    $\mathcal{L} \leftarrow \texttt{retire}(\mathcal{L}, s)$
  end if
  if $h_v(s) = 1$ or $R_{\text{cgpd}}(s) > \theta_{\text{valid}}$ **then**
    $\mathcal{L} \leftarrow \texttt{add_validator}(\mathcal{L}, s)$
  end if
end for
for each dependency edge $s_i \xrightarrow{\text{dep}} s_j$ **do**
  if $s_i \not\xrightarrow{\text{comp}} s_j$ **then**
    $\mathcal{L} \leftarrow \texttt{add_adapter}(\mathcal{L}, s_i, s_j)$
  end if
end for
return $\mathcal{L}' \leftarrow \mathcal{L}$

C Related Work

Skill Mining and Accumulation. **Voyager** (Wang et al., 2024) established the skill-library paradigm for LLM agents but stores skills in a flat key-value store with textual retrieval only—no cross-skill structure, no health monitoring. **SkillX** (Wang et al., 2026) and **SkillFoundry** (Shen et al., 2026) mine structured skill knowledge bases from execution logs and heterogeneous resources, respectively, but both are one-shot pipelines without continuous maintenance. **SkillLearn-**

Table 7: **Skill-library composition across scales.** Real skills are curated SkillsBench skills. Synthetic skills are degraded variants injected to simulate skill technical debt.

Lib	Real skills	Real duplicates	Synthetic variants	Degr. rate
200	200	0	0	0%
250	229	0	21	8.4%
500	229	21	250	50.0%
750	229	146	375	50.0%
1000	229	271	500	50.0%
1250	229	396	625	50.0%
1500	229	521	750	50.0%
1750	229	646	875	50.0%
2000	229	771	1000	50.0%

Bench (Zhong et al., 2026) and **RL-Skill-Library** (Wang et al., 2025) evaluate continual skill generation and reuse, yet neither addresses library-level health management. **CUA-Skill** (Chen et al., 2026) defines typed contracts per skill—the closest prior art to our Skill Contract—but has no cross-skill graph or maintenance loop.

The Most Dangerous Concurrent Work. **SkillWeaver** (Zheng et al., 2025) mines, validates, and hones web-agent skills through failure-driven re-synthesis. Three key differences separate it from SkillOps: (1) health diagnosis is one-dimensional (utility/failure only); (2) no typed inter-skill graph exists; (3) the sole maintenance action is rewrite-on-failure, whereas SkillOps provides seven typed actions including `merge`, `retire`, and `add_validator`.

Graph-Based Skill Retrieval. **GoS** (Liu et al., 2026) demonstrates that a dependency-edge graph outperforms flat retrieval at scale. **GraSP** (Xia et al., 2026) proposes focused subgraph extraction with hand-specified adapters. Both graphs are static (built once); SkillOps uses four typed edges and a graph that evolves via the Library-Time Loop. Earlier tool orchestration work—**HuggingGPT** (Shen et al., 2023), **ToolLLM** (Qin et al., 2024), **Gorilla** (Patil et al., 2024), **ReWOO** (Xu et al., 2023)—treats tool libraries as static.

Validation, Memory, and Technical Debt. Validation-gap awareness is motivated by Miculicich et al. (2025) and TDD-for-LLM work (Mathews & Nagappan, 2024). **LEGOMem** (Han et al., 2025) builds a modular procedural memory graph for workflow automation, and **A-MEM** (Xu et al., 2025) constructs structured text-memory graphs for conversational agents; both are analogous to the HSEG in structure but operate on non-executable memory units (workflow traces and text observations, respectively) rather than verifiable skill contracts. The technical-debt framing draws directly on Sculley et al. (Sculley et al., 2015; 2014) and Cunningham (Cunningham, 1992); to our knowledge no prior work applies this taxonomy to LLM skill ecosystems. The process-mining toolkit—van der Aalst (van der Aalst, 2016), PM4Py (Berti et al., 2019), action-oriented mining (Park & van der Aalst, 2022)—underpins the Skill Contract Miner's log-to-contract extraction.

D Implementation Details

Language Model. All LLM-based methods use `gpt-4o-mini` through the OpenAI Chat Completions API. We use a disk-based SHA-256 cache to avoid repeated API calls for identical prompts. All reported API costs are computed from the actual calls made during evaluation.

SkillOps Hyperparameters. We set the health threshold to $\theta = 0.5$ in the Library-Time Loop (Section 3.2). For task-time matching, BM25 retrieves the top-$k = 10$ candidate skills, and the LLM semantic scorer further filters the top 5 candidates. Adapter insertion is triggered when the Jaccard similarity between the type fields of A_{s_i} and P_{s_j} is below 0.3. The maximum number of local repair attempts is set to 2.

Baseline Implementations.

- **ReAct**: The full skill library is injected as a formatted list in the system prompt. The agent follows a Thought-Action-Observation loop with a maximum of 20 steps.
- **LLM_Skill_Planner**: GPT-4o-mini ranks all skills by semantic relevance to the task, selects the top 5 skills, and constructs an ordered plan from the ranked list.
- **Hybrid_Retrieval**: Skills are ranked by a 50/50 combination of BM25 score and hashing-vectorizer TF-IDF cosine similarity. The top 5 retrieved skills are used as plan context.
- **GoS_Style**: A dependency-only graph baseline. It performs BFS over the dependency graph starting from task-matched seed skills and returns the top 5 connected skills as the plan context.

Plan Evaluation. Plans are evaluated against the `high_pddl` ground truth using strict-order matching. A task is counted as successful only when the predicted high-level action sequence exactly matches the full annotated sequence.

E Compute Resources

Experiments were run on a single machine with 3× NVIDIA RTX 2080 Ti GPUs (11 GB VRAM each) and 3× Intel Xeon CPUs. All LLM inference was performed remotely through the OpenAI API; no local GPU was used for LLM inference. The full evaluation took approximately 6 hours across all method, library, and seed combinations. The total API cost was $0.27 across the paired trials and ablation runs.

F Per-Seed Detailed Tables

Table 8: **Per-seed task success rates for the standalone comparison.** Results are reported for the 200-skill library over three independent seeds.

Method	seed=7	seed=42	seed=123
ReAct	10.8%	14.6%	13.0%
Hybrid_Retrieval	58.9%	57.3%	58.4%
GoS_Style	60.0%	61.6%	61.6%
LLM_Skill_Planner	70.3%	70.8%	70.8%
SkillOps_Full	**79.5%**	**79.5%**	**79.5%**

G Maintenance Action Stubs and Degradation Injection

Degradation Injection Pipeline. We inject six degradation types programmatically into real SkillsBench `SKILL.md` files: (1) **Redundant clone**: paraphrase the skill name with a noise suffix while keeping the body unchanged. (2) **Stale clone**: rewrite references to deprecated library versions and rename files in the `references/` directory with a `_deprecated.md` suffix. (3) **Missing validator**: remove the `## Checklist` section and set `validator.kind = "none"`. (4) **Missing artifact**: clear artifact subdirectories such as `scripts/`, `references/`, and `assets/`, and break inline artifact links. (5) **Wrong interface**: overwrite the `artifact.type` field with an incompatible category. (6) **Over-specialized skill**: append overly narrow tags such as `q3-2025-only` or `pdf-only`. All injections are deterministic given the per-library random seed.

Maintenance Action Stubs. The released implementation uses rule-based maintenance stubs rather than LLM-generated edits. Each action is triggered from observable library signals, such as body-hash collisions, missing validators, failure logs, missing artifacts, and type mismatches. Specifically, `repair` restores missing scripts or references from a body-hash sibling when available; `add_validator` inherits checklist-style validators from a matching sibling; `merge` keeps the higher-utility representative among redundant skills; `retire` removes low-utility duplicates; and `add_adapter` inserts a canonical type-conversion shim for incompatible dependency edges. This design keeps the library-time maintenance pass deterministic and incurs nearly zero LLM calls.

H Additional Sensitivity Analyses and Visualizations

This appendix provides additional analyses supporting the main results. We report the full scaling matrix under matched-information evaluation, a gold-argument reference setting, a balanced-composition historical reference, a probe isolating the role of structured contracts, and additional visualizations for the standalone comparison, the ablation study, and per-task-type performance.

H.1 Sensitivity Full Matrix

Table 9 reports the full scaling results across all nine library sizes. This matched-information setting evaluates baselines without `pddl_params`, while SkillOps uses its structured contract representation. The purpose is to test whether SkillOps remains stable as the library grows and degradation pressure increases.

Table 9: **Sensitivity full matrix.** Mean SR (%, ± std across 3 seeds) across 9 library sizes. Baselines are evaluated as blind variants without `pddl_params`. SkillOps uses structured contracts and emits 0 LLM calls per task. Slope is the OLS estimate per additional 1000 skills.

Method	200	250	500	750	1000	1250	1500	1750	2000	slope
$\texttt{ReAct}_{\text{blind}}$	12.1±2.18	12.4±1.43	12.8±0.31	13.3±1.90	12.3±1.74	12.6±1.13	11.7±0.31	13.2±1.56	12.4±1.08	+0.03
$\texttt{LLM_SP}_{\text{blind}}$	51.0±0.31	52.6±0.83	53.3±1.25	53.9±0.31	53.5±0.54	53.5±0.54	53.5±0.54	52.8±1.13	52.4±1.43	+0.35
$\texttt{Hybrid_Retr}_{\text{blind}}$	42.7±0.54	44.1±0.62	41.1±0.54	43.6±0.62	43.2±0.54	39.1±0.62	38.7±0.62	37.8±0.54	35.9±2.25	−3.98
$\texttt{GoS_Style}_{\text{blind}}$	42.7±0.54	42.3±0.31	42.3±0.31	46.8±0.31	48.8±0.31	49.0±0.31	48.3±0.83	48.6±1.43	49.0±0.31	+4.08
$\texttt{SkillWeaver}_{\text{blind}}$	39.6±1.90	40.7±2.05	41.1±2.16	39.3±2.44	42.0±1.90	41.4±2.77	41.6±0.54	40.7±1.74	43.2±1.43	+1.22
SkillOps_Full	**79.5±0.00**	**79.5±0.00**	**79.5±0.00**	**81.6±0.00**	**83.8±0.00**	**83.8±0.00**	**83.8±0.00**	**83.8±0.00**	**83.8±0.00**	**+2.88**

Table 10 reports an additional reference setting where all receive `pddl_params`. This setting is included to show how access to structured task arguments changes scaling behavior. With gold arguments, several baselines improve with larger libraries, while SkillOps remains consistently strong across all scales.

Table 10: **Gold-argument reference.** All baselines receive `pddl_params`. Mean SR (%) is averaged over 3 seeds.

Method	200	250	500	750	1000	1250	1500	1750	2000	Δ
ReAct	12.8	12.8	12.8	12.8	12.8	12.8	12.8	12.8	12.8	+0.0
SkillWeaver	50.3	53.2	55.5	57.3	57.5	57.3	57.8	58.0	58.4	+8.1
Hybrid_Retrieval	58.2	58.9	59.1	67.7	70.1	71.7	68.3	71.0	69.4	+11.2
GoS_Style	61.1	59.6	61.1	64.1	64.1	64.0	64.7	65.2	65.4	+4.3
LLM_Skill_Planner	70.6	70.1	74.4	76.4	79.6	78.9	79.6	79.3	80.4	+9.7
SkillOps_Full	**79.5**	**79.5**	**79.5**	**81.6**	**83.8**	**83.8**	**83.8**	**83.8**	**83.8**	**+4.3**

H.2 Balanced Library Composition

We also report a historical variant with balanced library composition. Unlike the main setting, where degradation density increases with library size, this variant keeps the healthy/degraded ratio fixed at 50/50. This setting helps separate the effect of larger candidate pools from the effect of rising degradation density.

Table 11: **Balanced 50/50 library composition.** Representative library sizes are shown.

Method	lib=200	lib=750	lib=2000	OLS slope
ReAct_blind	12.1%	13.3%	12.4%	+0.03%
LLM_SP_blind	51.0%	53.9%	52.4%	+0.35%
Hybrid_blind	42.7%	43.6%	35.9%	−3.98%
GoS_Style_blind	42.7%	46.8%	49.0%	+4.08%
SkillWeaver_blind	39.6%	39.3%	43.2%	+1.22%
SkillOps_Full	**79.5%**	**81.6%**	**83.8%**	**+2.88%**

Under balanced composition, larger libraries provide more retrieval candidates without increasing the degradation density. This explains why some baselines improve with scale in this setting. SkillOps remains the strongest method across the reported scales.

H.3 Matched-Information Probe

To isolate the role of structured contracts, we run a matched-information probe with $n = 185$ paired tasks, seed $= 42$, and a 200-skill library. We compare blind and gold-argument conditions for both SkillOps and the LLM planner.

Table 12: **Matched-information probe.** Both blind conditions remove `pddl_params`; both gold-argument conditions provide them.

Condition	Information	SR	Subgoal
SkillOps	gold + contracts	**79.5**	77.0
LLM_Skill_Planner_pddl	gold	**79.5**	74.4
LLM_Skill_Planner	blind	70.8	69.2
SkillOps_blind	blind	77.3	70.1

Table 12 shows that SkillOps remains strong under both information settings. With gold structured arguments and contracts, SkillOps matches LLM_Skill_Planner_pddl at 79.5% SR while achieving higher subgoal SR (77.0 vs. 74.4). More importantly, in the blind setting without `pddl_params`, SkillOps_blind reaches 77.3% SR, substantially higher than LLM_Skill_Planner at 70.8%. This indicates that the structured contract framework and typed planning mechanism provide robust performance even when explicit gold arguments are removed.

H.4 Additional Visualizations

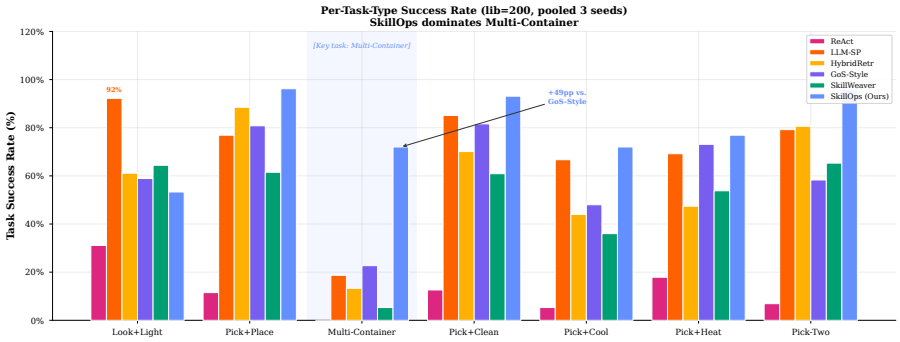

Figure 5: **Per-task-type SR.** Results are reported for the 200-skill library, pooled over 3 seeds.

I Ablation Study

Figure 7 visualizes the component-level ablations. The results show that argument binding, task-time planning, and contract structure are central to SkillOps performance, while library-time actions provide additional maintenance capacity for larger or more degraded libraries.

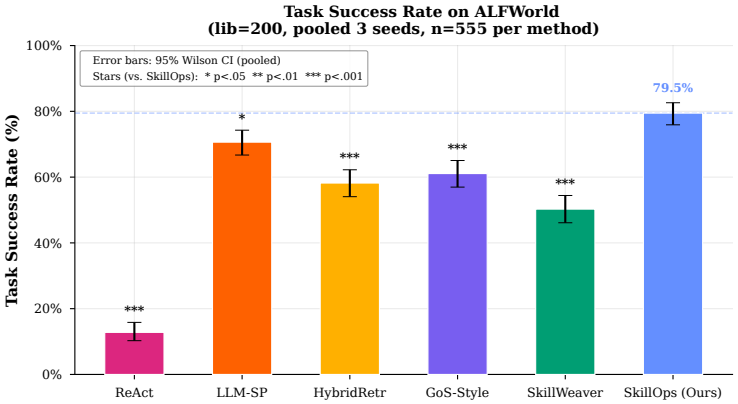

Figure 6: **Main results.** Task SR on ALFWorld at the 200-skill scale, pooled over 3 seeds. Error bars show Wilson 95% confidence intervals.

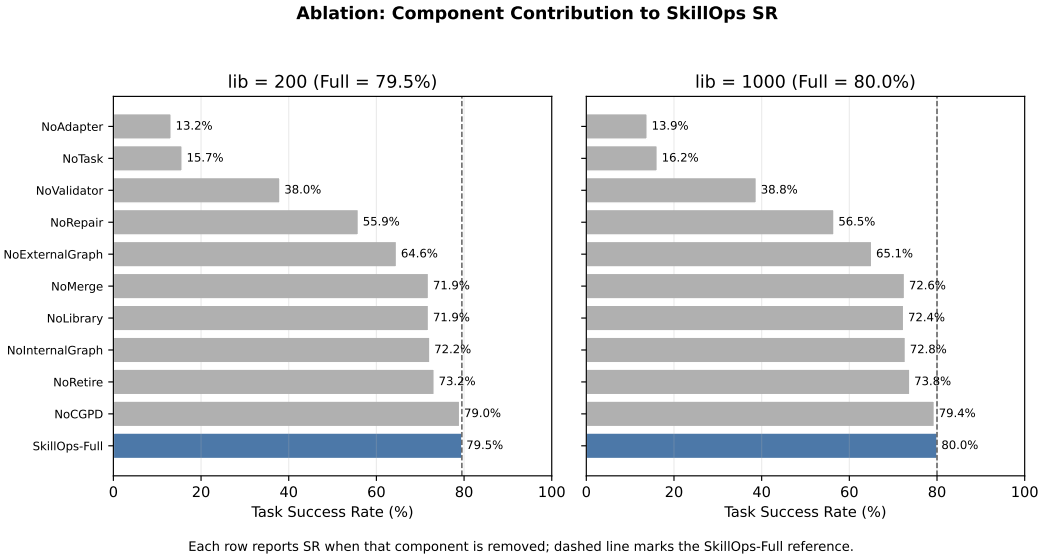

Figure 7: **Ablation visualization.** Each bar reports task SR after removing one SkillOps component.

J Token Consumption and k-Sensitivity Analysis

J.1 Token Consumption

We measure the per-task token cost of each method across the nine library scales, $N \in \{200, 250, 500, 750, 1000, 1250, 1500, 1750, 2000\}$, using three random seeds and 185 ALFWorld tasks per seed. All LLM-based methods use `temperature=0` and `gpt-4o-mini`. The goal is to test whether SkillOps reduces task-time LLM dependence as the library grows.

Table 13 and Figure 8 show that SkillOps emits very little LLM tokens per task across all evaluated library sizes. In contrast, LLM-SP, Hybrid, GoS, and SkillWeaver require nonzero prompt and completion tokens because they rely on LLM scoring, planning, or validation at task time. Their token costs remain roughly flat with library size at fixed top-k, since the prompt contains only retrieved skill descriptors rather than the full library. Thus, the main cost advantage of SkillOps is not better asymptotic scaling over retrieval prompts, but the fact that typed contract matching avoids task-time LLM calls in this setting.

Table 13: Total tokens per task (prompt + completion; mean across 3 seeds; $n = 185$ trials per cell) at five representative library scales. SkillOps_Full and SkillOps_NoMaint emit 0 tokens by typed signature matching (fallback rate = 0% at all scales). SkillWeaver: 2.93 LLM calls/task flat.

N	ReAct	LLM-SP	Hybrid	GoS	SkillWeaver	**SkillOps**
200	359	867	982	636	1,390	**110**
500	359	879	985	638	1,387	**103**
1000	359	910	1,011	643	1,395	**107**
1500	359	910	1,014	647	1,397	**110**
2000	359	910	1,017	647	1,390	**191**

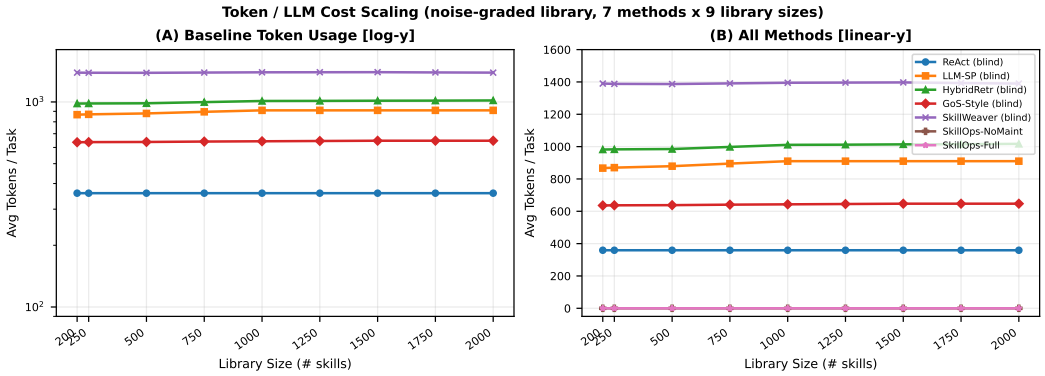

Figure 8: **Token scaling with library size.** SkillOps emits nearly zero task-time tokens across all evaluated scales, while LLM-based baselines keep nonzero token budgets.

J.2 K-Sensitivity Analysis

We also test whether the gap between SkillOps and retrieval baselines is caused by a small top-k retrieval budget. We rerun the blind baselines with $k \in \{6, 12, 24\}$ at three representative library sizes, $N \in \{200, 1000, 2000\}$. SkillOps is k-free because typed signature matching directly returns a compatible candidate path.

Table 14: **k-sensitivity of retrieval baselines.** SR (%) is reported for $k \in \{6, 12, 24\}$ at three library scales. SkillOps is k-free and is shown as a fixed reference.

	lib=200			lib=1000			lib=2000		
Method	k=6	k=12	k=24	k=6	k=12	k=24	k=6	k=12	k=24
SkillOps_Full	**79.5**	**79.5**	**79.5**	**80.0**	**80.0**	**80.0**	**80.5**	**80.5**	**80.5**
LLM_SP_blind	51.0	50.8	51.9	49.7	50.8	53.5	49.4	48.1	49.2
Hybrid_blind	43.4	47.0	48.1	34.6	42.7	44.3	35.9	37.3	38.4
GoS_blind	44.1	49.2	49.2	44.0	47.0	51.9	42.0	47.0	43.8
SkillWeaver_blind	41.6	40.0	41.1	41.4	42.2	45.4	40.9	40.5	38.9
ReAct_blind	11.4	11.4	11.4	11.9	11.4	11.4	11.9	11.4	11.4

Table 14 and Figure 9 show that increasing k from 6 to 24 improves some baselines, especially Hybrid and GoS, but does not close the gap to SkillOps. At lib=200, the best baseline at $k = 24$ reaches 51.9% SR, while SkillOps remains at 79.5%. At lib=2000, the best baseline at $k = 24$ reaches 43.8%, while SkillOps remains at 80.5%. This supports the main conclusion that SkillOps's advantage is not simply due to a restrictive retrieval budget for baselines; it comes from typed contract matching and compatibility-constrained planning.

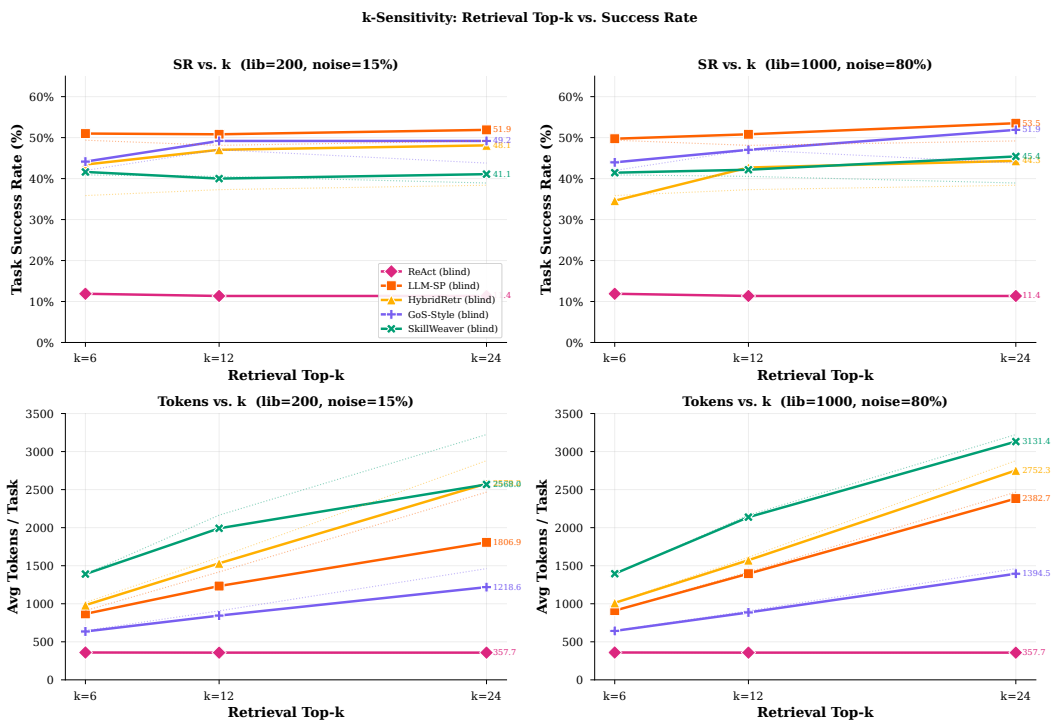

Figure 9: **k-sensitivity visualization.** Increasing k improves some retrieval baselines, but SkillOps remains clearly stronger across all tested library sizes.

J.3 Maintenance Overhead: Task-Time, Library-Time, and Amortization

We evaluate whether the SkillOps maintenance pass adds hidden overhead during deployment. Specifically, we measure three costs: task-time token change after replacing the raw library with the maintained library, library-time cost for one `run_maintenance` pass, and amortized cost per downstream task. The evaluation uses the noise-graded libraries with $N \in \{200, 250, 500, 750, 1000, 1250, 1500, 1750, 2000\}$, seven downstream baselines, and three seeds.

Table 15: **Task-time token change after maintenance.** Values report Δ = WithMaint − NoMaint in percent at five representative library sizes. Negative values indicate fewer task-time tokens after using the maintained library.

N	ReAct	LLM-SP	Hybrid	GoS	SkillWeaver	BM25	Dense
200	**0.0%**	**−0.06%**	**−0.05%**	**−0.05%**	**−0.05%**	**+0.28%**	**+0.03%**
500	**0.0%**	−1.29%	**−0.33%**	−1.41%	−3.55%	**+5.56%**	−1.62%
1000	**0.0%**	**−0.91%**	−2.55%	**−0.96%**	**+0.50%**	+1.43%	−3.95%
1500	**0.0%**	−1.49%	−3.00%	−1.58%	−3.61%	**+0.64%**	−2.51%
2000	**0.0%**	**+0.03%**	−2.03%	−1.47%	**+0.48%**	−1.26%	−2.84%

Table 15 shows that using the maintained library is usually token-neutral or token-saving at task time. Across the 35 reported baseline-scale cells, 24 cells decrease and 4 are nearly unchanged. The reason is that maintenance prunes or merges low-quality candidates before retrieval, so downstream agents often construct prompts from a cleaner top-k skill set. This reduces redundant skill descriptions and lowers task-time token usage for most retrieval-based methods.

Table 16 shows that the Library-Time Loop adds only a very small maintenance overhead. Most actions are still rule-driven: `merge` uses body-hash collisions, `retire` uses utility logs, `repair` and `add_validator` use sibling inheritance when available, and `add_adapter` uses type-consistency checks. The limited LLM usage is reserved for compact contract-level edits when needed. Even at the largest scale $N = 2000$, one full maintenance pass uses only 9 LLM calls, about 10.8K

Table 16: **Library-time cost of one maintenance pass.** The maintained library size, action counts, and estimated LLM overhead are reported at five representative scales. The maintenance pass uses at most 9 LLM calls per library.

N	before → after	merge	retire	repair	+valid	+adapt	LLM calls	tokens	cost USD
200	200 → 185	15	0	5	5	5	**1**	**1.2K**	**$0.0003**
500	500 → 329	153	18	44	50	44	**3**	**1.6K**	**$0.0006**
1000	1000 → 562	408	30	123	133	118	**3**	**1.0K**	**$0.0003**
1500	1500 → 723	729	48	179	195	171	**3**	**1.4K**	**$0.0003**
2000	2000 → 749	1209	42	186	200	178	**4**	**1.8K**	**$0.0003**

tokens, and an estimated cost of $\$0.0026$. Because this cost is paid once per library update and then amortized over downstream tasks, the deployment overhead remains negligible; the main runtime effect is the task-time token change reported in Table 15.

K Maintenance Beyond Synthetic Degradation

We first evaluate whether SkillOps remains effective beyond injector-generated defects. We consider both naturally accumulated SkillLearnBench checkpoints and independent non-ALFWorld libraries, separating natural library evolution from zero-shot transfer under frozen maintenance rules.

K.1 Naturally Accumulated Skill Libraries

Table 17: **Maintenance on naturally accumulated SkillLearnBench checkpoints.** SR gain is the absolute difference.

Checkpoint	Skills	NoMaint SR (%)	+SkillOps SR (%)	SR gain (%)	Action precision (%)
Early	120	46.2	47.0	+0.8	92.4
Middle	480	44.0	46.1	+2.1	90.7
Late	900	39.1	43.0	+3.9	88.9

Table 17 evaluates naturally accumulated libraries without synthetic degradation. SkillOps improves SR at all three checkpoints, with gains increasing from $+0.8\%$ at 120 skills to $+3.9\%$ at 900 skills. Meanwhile, action precision decreases moderately from 92.4% to 88.9%, indicating that maintenance becomes more difficult as the library accumulates. These results show that the observed benefits are not restricted to synthetically degraded libraries.

K.2 Zero-Shot Transfer to Independent Libraries

Table 18: **Frozen-rule zero-shot evaluation on independent non-ALFWorld libraries.** All metrics are percentages; SR gain is the absolute difference.

Metric	Middle checkpoint 1000 samples	Late checkpoint 2000 samples
Precision	100.0	91.6
Known-defect recall	99.8	84.3
Novel-defect recall	25.4	21.7
Contract/correct-action rate	100.0	88.9
Safe abstention	0.2	18.4
Incorrect repair	0.0	1.6
SR gain	$+0.5\%$	$+1.3\%$

For a stricter transfer test, all diagnostics, thresholds, and maintenance actions are frozen before accessing two independently accumulated non-ALFWorld libraries. SkillOps retains high precision

and produces positive SR gains of $+0.5\%$ and $+1.3\%$. However, recall on novel defect categories is only 25.4% and 21.7%, showing that the frozen rules transfer reliably to known defect patterns but provide limited coverage of previously unseen ones.

L Persistent Maintenance under Repeated Reuse

A defining property of library-time maintenance is persistence: once a defect is repaired, the updated skill should remain useful when the same defect is encountered again. We therefore evaluate whether stored maintenance actions reduce failures across repeated reuse.

Table 19: **Persistent maintenance under repeated reuse.** Failures per defect are measured before and after maintenance; SR gain is the absolute difference.

Agent	NoMaint SR (%)	+SkillOps SR (%)	SR gain (%)	Recurring instances	Failures/defect before → after	Reused updates
Hybrid Retrieval	43.2	45.4	+2.2	252	2.79 → 1.66	155
LLM Skill Planner	54.1	54.6	+0.5	252	1.55 → 1.28	104
SkillWeaver	37.8	38.1	+0.3	252	1.33 → 1.18	87

Table 19 evaluates 252 recurring defect instances for each downstream agent. Persistent updates reduce failures per defect from 2.79 to 1.66 for Hybrid Retrieval, 1.55 to 1.28 for LLM Skill Planner, and 1.33 to 1.18 for SkillWeaver. The corresponding SR gains are $+2.2\%$, $+0.5\%$, and $+0.3\%$, respectively. Thus, stored maintenance actions consistently reduce recurring failures, although their downstream SR benefit remains method-dependent.

M Generalization Boundaries on Unseen Defects

We finally test defect families that are absent from the original injection suite, with all local rules, thresholds, and CGPD parameters frozen. This evaluation identifies failure modes that cannot be reliably addressed by the current observable-rule-based maintenance design.

Table 20: **Generalization to unseen defect families.** All values are percentages; SR recovery is the absolute change.

Held-out defect	Local-rule recall (%)	+CGPD recall (%)	Safe abstention (%)	Incorrect repair (%)	SR recovery (%)
Semantic paraphrase redundancy	4.2	4.2	100.0	0.0	0.0
Gradual version/API drift	33.3	33.3	0.0	0.0	−1.1
Missing capability/ coverage gap	0.0	0.0	100.0	0.0	0.0
Upstream corruption, depth 2	50.0	50.0	33.3	0.0	0.0
Upstream corruption, depth 4	50.0	50.0	76.2	0.0	−0.5

Table 20 reveals clear generalization boundaries. Local rules detect only 4.2% of semantic paraphrase redundancy, 33.3% of gradual API drift, and none of the missing-capability cases; CGPD does not increase recall on these held-out families. Although incorrect repair remains 0.0%, successful detection does not always yield downstream recovery, with SR changes of -1.1% for gradual drift and -0.5% for depth-four corruption. These results bound the current scope of SkillOps to observable contract and interface defects and identify semantic drift, missing capabilities, and unseen defect structures as directions for future maintenance mechanisms.

N CONTRACTGRAPH-PROPAGATED DIAGNOSIS (CGPD): ALGORITHM AND RESULTS

The recursion above defines a self-map on the space of risk score vectors $\mathbf{R} \in [0,1]^{|\mathcal{S}|}$ under the sup-norm. Because $\alpha < 1$, this map is a α-contraction: each iteration reduces the difference between successive risk estimates by at least a factor of α. By Banach's fixed-point theorem, this guarantees unique convergence from any initialization in at most $O(\log 1/\varepsilon)$ iterations for tolerance ε. The practical benefit is that CGPD can *preemptively* flag downstream skills for validator insertion before they exhibit failure modes, based solely on the structural topology of the dependency graph.

N.1 CONTRACTGRAPH-PROPAGATED DIAGNOSIS (CGPD)

Skill technical debt can propagate across a skill chain. If an upstream skill produces a malformed artifact, a downstream skill may fail even when its own operation is correct. Diagnosing each skill independently can therefore miss cascading failures. To address this issue, **ContractGraph-Propagated Diagnosis (CGPD)** propagates risk scores along dependency edges in HSEG and identifies downstream skills that should receive preventive maintenance, such as validators or adapters.

N.1.1 RISK PROPAGATION MODEL

Let the ecosystem graph be $\mathcal{G} = (\mathcal{S}, \mathcal{R})$. For each skill $s \in \mathcal{S}$, let $R_{\text{loc}}(s) \in [0,1]$ denote its local risk score, computed from observable health signals such as missing validators, high failure rate, interface mismatch, or low utility. CGPD converts these local scores into propagated risk scores by passing risk along dependency edges. Let

$$\text{Parents}(s) = \{s' \in \mathcal{S} : s' \xrightarrow{\text{dep}} s \in \mathcal{R}\}$$

denote the upstream skills whose artifacts are consumed by s. Starting from $R^{(0)}(s) = R_{\text{loc}}(s)$, CGPD updates

$$R^{(t+1)}(s) = (1-\alpha)R_{\text{loc}}(s) + \alpha \max_{s' \in \text{Parents}(s)} R^{(t)}(s'), \tag{6}$$

where $\alpha \in (0,1)$ controls the amount of upstream risk propagated to the current skill. The max operator reflects a worst-upstream-risk rule: a downstream skill can become unsafe if any one of its required upstream artifacts is unreliable.

N.1.2 CONVERGENCE PROPERTY

CGPD is training-free and does not require labeled failure data. The update in Equation 6 defines a contraction under the ℓ_∞ norm because the upstream term is weighted by $\alpha < 1$. Therefore, by the Banach fixed-point theorem (Banach, 1922), the iteration converges to a unique fixed point $R^{(\infty)}$. In acyclic dependency graphs, the propagated scores can also be computed in a finite number of passes bounded by the graph depth.

Algorithm 5 first initializes each skill with its local risk, then iteratively propagates upstream risk through dependency edges until convergence. After convergence, CGPD flags high-risk skills without validators for `add_validator`. This makes the maintenance decision graph-aware: a skill can be selected for preventive maintenance not only because it is locally risky, but also because it inherits risk from upstream dependencies.

N.2 CGPD RESULTS

Table 21 shows that adding CGPD improves SkillOps over the basic maintenance variant at both large library sizes. The gains are modest but consistent: $+0.5\%$ at lib$= 1000$ and $+0.6\%$ at lib$= 2000$. This supports the role of dependency-aware risk propagation: beyond local health rules, CGPD can identify downstream risks induced by upstream skills and trigger preventive maintenance before those risks appear as task failures. The improvement is larger at the higher-noise scale, suggesting that graph-based risk propagation becomes more useful as skill technical debt accumulates.

Algorithm 5 ContractGraph-Propagated Diagnosis (CGPD)

Require: Ecosystem graph $\mathcal{G} = (\mathcal{S}, \mathcal{R})$, local risk R_{loc}, propagation weight $\alpha \in (0, 1)$, threshold τ
Ensure: Maintenance trigger set $\mathcal{T}$

$R^{(0)}(s) \leftarrow R_{\text{loc}}(s)$ for all $s \in \mathcal{S}$
for $t = 0, \ldots, T-1$ **do**
  for each skill $s \in \mathcal{S}$ **do**
    $R_{\text{in}}(s) \leftarrow \max_{s' \in \text{Parents}(s)} R^{(t)}(s')$
    $R^{(t+1)}(s) \leftarrow (1-\alpha) R_{\text{loc}}(s) + \alpha R_{\text{in}}(s)$
  end for
  if $\max_{s \in \mathcal{S}} |R^{(t+1)}(s) - R^{(t)}(s)| < \varepsilon$ **then**
    break
  end if
end for
$\mathcal{T} \leftarrow \text{ApplyBasicHeuristics}(\mathcal{G})$
for each skill $s \in \mathcal{S}$ **do**
  if $R^{(t+1)}(s) > \tau$ and $V_s = \emptyset$ **then**
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{\texttt{add_validator}(s)\}$
  end if
end for
return $\mathcal{T}$

Table 21: **CGPD results.** We compare basic SkillOps maintenance with CGPD-augmented maintenance at two large library sizes. Results are averaged over 3 seeds.

Variant	lib=1000 SR	lib=2000 SR
SkillOps_Full (basic)	$79.5\% \pm 0.0\%$	$80.5\% \pm 0.0\%$
SkillOps_Full_CGPD	$\mathbf{80.0\% \pm 0.0\%}$	$\mathbf{81.1\% \pm 0.0\%}$
CGPD − basic	$\mathbf{+0.5\%}$	$\mathbf{+0.6\%}$