RETRIEVAL-AUGMENTED SKILL OPTIMIZATION VIA CROSS-HARNESS ADAPTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

An agent skill is a reusable, actionable natural-language artifact that guides an agent to perform a task effectively under a given harness. Recent studies have explored the optimization of agent skills, contributing to a growing collection of publicly available skills spanning diverse tasks, domains, and harnesses. Despite millions of publicly shared skills, existing skill optimization methods largely overlook this accumulated knowledge, instead relying solely on expensive agent rollouts to iteratively refine skills for a target task. To address this, we propose **Retrieval-Augmented Skill Optimization (RASO)**, a framework that leverages an external skill corpus as prior knowledge throughout skill optimization. RASO retrieves relevant knowledge from existing skills and adapts it to the target task and harness via Cross-Harness Adaptation, accounting for mismatches in both domain and harness. RASO comprises two complementary stages: **Retrieval-Augmented Skill Initialization (RASI)** constructs a knowledge-grounded initial skill without requiring agent rollouts, while **Retrieval-Augmented Skill Update (RASU)** iteratively refines the skill by retrieving external knowledge guided by execution feedback. Across four agent benchmarks and two models, extensive experiments show that RASO consistently outperforms baselines without retrieval-augmented skill initialization and updating.

1 INTRODUCTION

Large language models (LLMs) are widely deployed as agents within execution harnesses that define available tools, file access, and scoring procedures (Yao et al., 2023; Yang et al., 2024b). In these settings, performance depends not only on the parametric knowledge of the underlying model but also on the procedural policies governing task execution (Wang et al., 2024; 2025), commonly referred to as agent skills (Anthropic, 2025; Li et al., 2026c). An agent skill is a reusable, actionable natural-language artifact that specifies how an agent should accomplish tasks under a given harness. Unlike policies encoded in model weights, agent skills are expressed as text, making them readily inspectable, auditable, and transferable across models without modification.

To automatically construct effective agent skills, skill optimization has recently attracted growing attention (Wang et al., 2026; Xia et al., 2026; Ding et al., 2026; Chen et al., 2026). Existing methods iteratively refine skills based on agent experience collected through training-task rollouts under a given harness. Through such refinement, millions of skills have been publicly shared (Destefanis et al., 2026), collectively encoding procedural knowledge accumulated across diverse tasks, models, and harnesses. Despite this extensive knowledge, existing methods largely rely on the agent's own experience to optimize each skill (Yang et al., 2026; Alzubi et al., 2026; Ni et al., 2026; Tang et al., 2026), requiring costly rollouts for iterative refinement. As a result, leveraging an external skill corpus as a prior for automatic skill optimization remains largely unexplored.

To this end, we propose **Retrieval-Augmented Skill Optimization (RASO)**, a skill optimization framework that leverages an external skill corpus as prior knowledge. As in Fig. 1, RASO comprises two complementary stages, **Retrieval-Augmented Skill Initialization (RASI)** and **Retrieval-Augmented Skill Update (RASU)**. RASI leverages prior knowledge from an external skill corpus to construct an effective initial skill directly from task and harness descriptions, without using any expensive agent rollouts. Upon observing a failure during task execution, RASU identifies the missing knowledge underlying the failure and retrieves relevant content from the corpus to address it.

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.

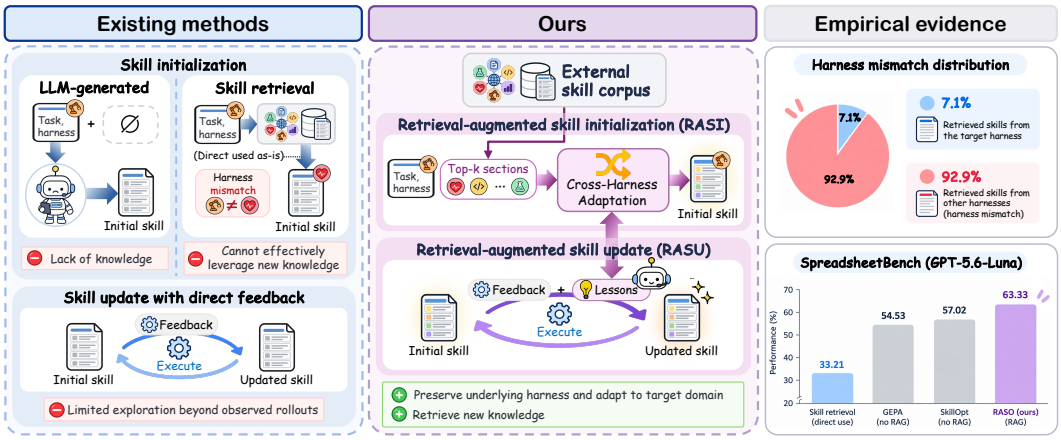

Figure 1: **Comparison of RASO to prior skill optimization.** Existing skill optimization methods (left) either generate an initial skill from the LLM alone or directly reuse retrieved skills, then refine them mainly from execution feedback. However, retrieved skills often come from mismatched harnesses; on SpreadsheetBench 92.9% originate from a different harness (right, top). RASO (middle) addresses this by retrieving relevant external skill knowledge and adapting it to the target domain and execution harness through *Cross-Harness Adaptation*, supporting both skill initialization (RASI) and skill update (RASU). Overall, our RASO results in substantially stronger performance than both direct retrieval and retrieval-free optimization methods.

However, retrieved skills are originally written for specific tasks and harnesses, potentially encoding domain-specific assumptions or referencing tools unavailable in the target environment. To address this domain and harness mismatch, RASO introduces Cross-Harness Adaptation, a shared operation employed by both RASI and RASU. Rather than directly incorporating retrieved content, this operation abstracts away source-domain-specific instruction and re-expresses the underlying procedure in terms of the objects, commands, and units supported by the target harness.

We evaluate RASO through extensive experiments on four agent benchmarks, OfficeQA (Opsahl-Ong et al., 2026), SpreadsheetBench (Ma et al., 2024), ALFWorld (Shridhar et al., 2021), and WebShop (Yao et al., 2022), with two LLMs, Qwen-3.5-9B (Qwen Team, 2026) and GPT-5.6-Luna (OpenAI, 2026). Specifically, RASI synthesizes strong initial skills that outperform both retrieved skills and retrieval-free initialization without requiring agent rollouts, providing a strong initialization for subsequent optimization. Moreover, RASO with RASU consistently outperforms strong skill optimization algorithms, including TextGrad (Yuksekgonul et al., 2025), GEPA (Agrawal et al., 2026), SkillOpt (Yang et al., 2026), and WikiSkill (Tang et al., 2026).

Our contributions are as follows:

- We propose **Retrieval-Augmented Skill Optimization (RASO)**, a framework that leverages an external skill corpus as prior knowledge for both skill initialization and update. Through its two complementary stages, **RASI** and **RASU**, RASO grounds skill construction in retrieved procedural knowledge rather than relying on the optimizer's parametric knowledge.
- We introduce Cross-Harness Adaptation, a shared operation that adapts retrieved procedural knowledge to the vocabulary of the target domain and harness. This enables knowledge transfer across diverse domains and harnesses without requiring domain- or harness-matched skills in the corpus.
- Across four benchmarks and two models, we demonstrate that RASI improves performance without requiring agent rollouts, while RASU further refines the skill by retrieving the missing knowledge guided by execution feedback. Our analysis further shows that these gains are associated with adapting diverse external knowledge rather than relying on particular source documents.

2 Related Works

Agent Skills. Agent skills are reusable textual documents that provide procedural guidance for accomplishing tasks within an execution harness (Anthropic, 2025). Represented as text rather than model parameters, skills are easy to inspect, edit, share, and reuse across models without retraining. Prior work has made this knowledge explicit through executable skill libraries (Wang et al., 2024), reusable workflows (Wang et al., 2025), reflections and insights (Shinn et al., 2023; Zhao et al., 2024), or reasoning and procedural memories (Ouyang et al., 2026; Fang et al., 2026), while large public collections such as GitSkills (Destefanis et al., 2026) now make this knowledge available at scale. However, not every skill is useful: its benefit depends on both the procedural knowledge it contains and how well that knowledge matches the target task and harness. Accordingly, two main approaches have emerged: learning procedural knowledge from the agent's own *execution experience* and reusing knowledge from *external sources*.

Skill Optimization from Execution Experience. A major line of work improves agent skill from its own experience, treating the skill as a textual decision variable optimized with execution feedback on the target task. Methods that optimize prompts or contexts using LLM-generated feedback (Pryzant et al., 2023; Yang et al., 2024a; Chu et al., 2026; Zhang et al., 2026) are directly applicable to skill optimization, most notably TextGrad (Yuksekgonul et al., 2025), which backpropagates textual feedback, and GEPA (Agrawal et al., 2026), which evolves prompts by reflecting on execution traces. Skill-specific optimizers follow the same recipe: SkillOpt (Yang et al., 2026) edits a skill from rollout trajectories and accepts only edits that improve validation performance, WikiSkill (Tang et al., 2026) compiles agent experience into persistent knowledge, and others distill or refine skills from trajectories (Ni et al., 2026; Chen et al., 2026; Moll et al., 2026; Wang et al., 2026; Alzubi et al., 2026; Ding et al., 2026). However, these methods primarily optimize skills from observed execution experience, limiting their exploration of procedural knowledge beyond what can be inferred from the agent's own rollouts. In contrast, our framework supplements execution feedback with procedural knowledge retrieved from an external skill corpus throughout optimization.

Skill Construction from External Knowledge. Procedural knowledge that an agent cannot infer from its own experience often already exists: in shared skills, documentation, and the web, motivating a growing line of work that draws on such external knowledge. Retrieval-based methods, such as SkillRouter (Zheng et al., 2026), select relevant skills from large libraries through improved retrievers (Li et al., 2026a; Miao et al., 2026) or by organizing libraries into graphs and execution structures (Meng et al., 2026; Liu et al., 2026; Fu et al., 2026; Li et al., 2026b; Xia et al., 2026), with dedicated benchmarks for skill retrieval (Kang et al., 2026; Su et al., 2026). Since retrieved skills may refer to different tasks, tools, or actions, other methods adapt external experience to the target interface (Tang et al., 2025) or compile external resources into reusable skills (Pan et al., 2026; Yan et al., 2026). However, these approaches either require repeated retrieval and adaptation for each task instance during test time or use external knowledge only during skill initialization, without further leveraging it as target-task experience accumulates. In contrast, we leverage external knowledge throughout the skill optimization process, using it both to construct the initial skill and to further improve the skill during iterative updates.

3 Problem Formulation

We consider a frozen language model $\mathcal{M}$ acting as an agent through an execution harness h, which defines the available tools, file access, and observation interface. A skill s is a natural-language artifact provided as the agent's context to guide task completion under a given harness. Executing the agent on a task instance x with skill s yields a trajectory $h(\mathcal{M}, x, s)$, which is associated with a reward $r(\cdot) \in [0, 1]$ by a benchmark-specific evaluator. When a reference answer is available, the evaluator compares the agent's final output against it, and otherwise uses the environment's native success criterion. We refer to each agent execution and its corresponding evaluation as a *rollout*. Given disjoint task splits $\mathcal{D}_{\text{train}}$, $\mathcal{D}_{\text{val}}$, and $\mathcal{D}_{\text{test}}$, candidate skills are constructed from rollouts on $\mathcal{D}_{\text{train}}$ and selected based on performance on $\mathcal{D}_{\text{val}}$ as:

$$s^\star = \arg\max_s \mathbb{E}_{x \sim \mathcal{D}_{\text{val}}} \left[r\left(h(\mathcal{M}, x, s) \right) \right]. \tag{1}$$

The selected skill $s^\star$ is then evaluated on the held-out $\mathcal{D}_{\text{test}}$.

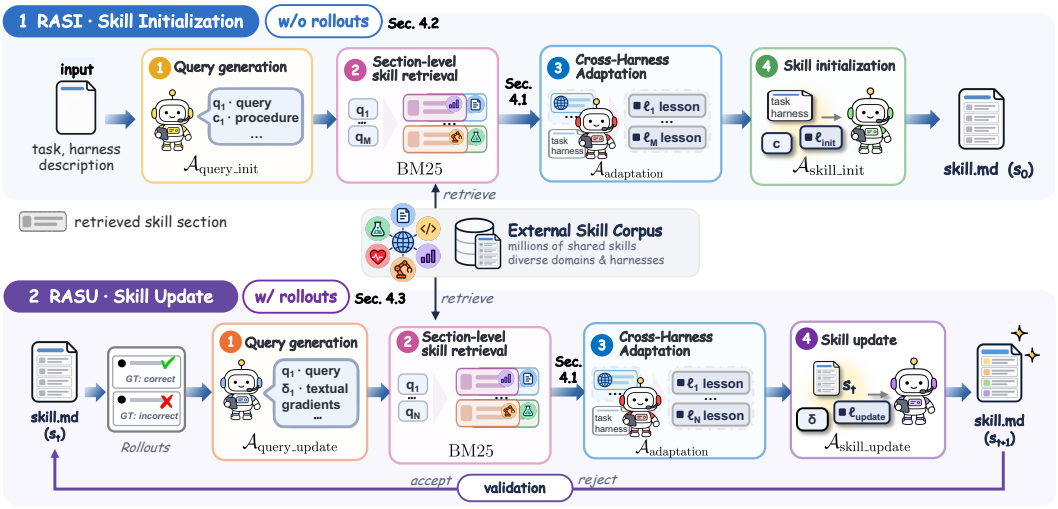

Figure 2: **Overview of RASO.** RASI (top, without rollouts) generates retrieval queries from the task and harness descriptions, retrieves the top-K skill sections per query from an external corpus, and adapts them into grounded lessons via Cross-Harness Adaptation to synthesize the initial skill s_0. RASU (bottom, with rollouts) starts from s_0, executes the current skill, and generates retrieval queries from rollout results. The retrieved knowledge is adapted into grounded lessons via Cross-Harness Adaptation and incorporated into `skill.md` to iteratively refine the skill. Each subsequent iteration performs new rollouts using the committed skill.

Since both $\mathcal{M}$ and h remain fixed throughout optimization, the only decision variable is the natural-language skill s. We include the construction of the initial skill itself in the skill optimization problem, rather than assuming that an initial skill is externally supplied. We therefore define skill optimization to encompass both *skill initialization*, which constructs an initial skill from task and harness descriptions without requiring any rollouts, and *skill update*, which refines the skill using results from training rollouts.

4 RASO: Retrieval-Augmented Skill Optimization

In this section, we introduce **Retrieval-Augmented Skill Optimization (RASO)**, a framework that evolves agent skills through two complementary stages: skill initialization and skill update, illustrated in Figure 2. Both stages share a common knowledge retrieval and adaptation mechanism but differ in the rollout evidence available to guide skill optimization. Section 4.1 describes the shared mechanism, which retrieves procedural knowledge from a large-scale skill corpus (Destefanis et al., 2026) and adapts it to the target task and harness through *cross-harness adaptation*. Section 4.2 introduces **Retrieval-Augmented Skill Initialization (RASI)**, which constructs an initial skill using only the target task, harness description, and retrieved knowledge, without requiring agent rollouts. Section 4.3 introduces **Retrieval-Augmented Skill Update (RASU)**, which leverages execution feedback and retrieved knowledge to iteratively refine the skill.

4.1 Skill Retrieval and Cross-Harness Adaptation

We first perform fine-grained, section-level skill retrieval from an external skill corpus to acquire relevant prior knowledge. We then introduce Cross-Harness Adaptation, a mechanism that bridges the gap between source and target domains and harnesses by transforming retrieved knowledge into actionable guidance tailored to the target task and harness. Skill retrieval and adaptation constitute a shared pipeline used by both RASI (Section 4.2) and RASU (Section 4.3).

Section-level skill retrieval. We first divide skill documents into heading-delimited sections to enable fine-grained retrieval. Since external skill documents are developed for diverse tasks and workflows, retrieving entire documents may introduce irrelevant content and favor documents with similar overall objectives over those containing relevant procedural sections. Section-level retrieval

instead enables us to identify relevant procedural knowledge while improving the signal-to-noise ratio of retrieved content. Given the resulting section-level corpus $\mathcal{C}$, a query-generation agent formulates a query q and retrieves the top-K most relevant sections using BM25 (Robertson & Zaragoza, 2009), *i.e.*, $\mathcal{S}_q = \text{BM25}(q, \mathcal{C}, K)$.

Cross-Harness Adaptation. We employ an adaptation agent, denoted by $\mathcal{A}_{\text{adaptation}}$, to adapt retrieved knowledge to the target task and harness. Let c_i denote a requirement needing external knowledge to be resolved, such as a specific task procedure in RASI or a textual gradient in RASU. Given c_i, the task description T, the harness description H, and the corresponding top-K retrieved sections $\mathcal{S}_{q_i}$, the agent produces a concise, actionable lesson ℓ_i:

$$\ell_i = \mathcal{A}_{\text{adaptation}}\left(c_i, T, H, \mathcal{S}_{q_i}\right). \tag{2}$$

Here, the lesson ℓ_i serves as a refined knowledge snippet that directly guides the agent to handle the requirement c_i within the target task and harness.

To ensure that each lesson addresses the given requirement and remains valid within the target harness, the adaptation process follows three principles: (1) remove domain-specific nouns and omit procedures without counterparts in the target harness, (2) focus exclusively on requirement c_i, excluding unrelated issues, and (3) preserve specific claims about tool or parameter behavior only when corroborated by H, prioritizing correctness within the target harness over potentially inaccurate specificity. Consequently, each lesson is expressed using the objects, commands, and units of the target task and harness.

4.2 Retrieval-Augmented Skill Initialization (RASI)

We introduce **Retrieval-Augmented Skill Initialization (RASI)**, which constructs an initial skill for a target task under a given harness without requiring agent rollouts. Performed once at the beginning of skill optimization, RASI comprises four sequential steps: (1) procedure and query generation, (2) section-level skill retrieval, (3) Cross-Harness Adaptation, and (4) skill initialization.

Procedure and query generation. Given the task description T and harness description H, the query-generation agent $\mathcal{A}_{\text{query_init}}$ generates a set of requirement-query pairs (c_i, q_i):

$$\{(c_i, q_i)\}_{i=1}^{M} = \mathcal{A}_{\text{query_init}}(T, H), \tag{3}$$

where M denotes the number of generated pairs. Here, each c_i represents a specific task procedure or harness constraint (*e.g.,* multi-turn budget management), and q_i is the retrieval query created to search for external skills that address c_i.

Section-level skill retrieval and Cross-Harness Adaptation. Given the generated requirement-query pairs, RASI applies the shared retrieval and adaptation pipeline described in Section 4.1. For each query q_i, BM25 retrieves the top-K relevant sections $\mathcal{S}_{q_i}$ from the corpus $\mathcal{C}$. The adaptation agent $\mathcal{A}_{\text{adaptation}}$ then transforms these sections into a grounded lesson $\ell_i = \mathcal{A}_{\text{adaptation}}(c_i, T, H, \mathcal{S}_{q_i})$. The resulting lessons form $\mathcal{L}_{\text{init}} = \{\ell_1, \ldots, \ell_M\}$, which are used for subsequent skill initialization.

Skill initialization. Finally, a skill-initializer agent $\mathcal{A}_{\text{skill_init}}$ synthesizes the initial skill s_0 from the task description T, harness description H, identified procedures and requirements $\{c_i\}_{i=1}^{M}$, and grounded lessons $\mathcal{L}_{\text{init}}$. The agent integrates each lesson ℓ_i into the execution step corresponding to c_i, yielding:

$$s_0 = \mathcal{A}_{\text{skill_init}}(T, H, \{c_i\}_{i=1}^{M}, \mathcal{L}_{\text{init}}). \tag{4}$$

By incorporating retrieved and adapted knowledge before environment interaction, RASI provides a knowledge-grounded initial skill for subsequent optimization without consuming search rollouts.

4.3 Retrieval-Augmented Skill Update (RASU)

Here, we introduce **Retrieval-Augmented Skill Update (RASU)** that iteratively refines the current skill s_t using execution feedback from agent rollouts. Complementing the rollout-free initialization of RASI, RASU identifies specific failure modes observed in agent trajectories and retrieves relevant external knowledge to address them. Given trajectories generated by the execution agent using s_t, each RASU iteration comprises four sequential steps: (1) textual gradient and query generation, (2) section-level skill retrieval, (3) Cross-Harness Adaptation, and (4) skill update.

Textual gradient and query generation. We first sample a minibatch of tasks from $\mathcal{D}_{\text{train}}$ and execute agent rollouts using the current skill s_t. A gradient-and-query generator agent $\mathcal{A}_{\text{query_update}}$ then analyzes the failed trajectories to identify failure mode. For each failure modes, the agent generates a textual gradient δ_i based on its parametric knowledge and a targeted retrieval query q_i to acquire relevant external knowledge:

$$\{(q_i, \delta_i)\}_{i=1}^{N} = \mathcal{A}_{\text{query_update}}(T, H, s_t, \{\tau_j\}), \tag{5}$$

where N denotes the number of generated query and gradient pairs, and $\{\tau_j\}$ denotes the rollout trajectories.

Section-level skill retrieval and Cross-Harness Adaptation. Given the failure-driven retrieval queries, RASU applies the shared retrieval and adaptation pipeline described in Section 4.1. This process yields a set of grounded lessons $\mathcal{L}_{\text{update}} = \{\ell_1, \ldots, \ell_N\}$ for subsequent skill update.

Skill update. Finally, a skill-updater agent $\mathcal{A}_{\text{skill_update}}$ generates a candidate skill s_{t+1} by integrating the current skill s_t, textual gradients $\{\delta_i\}_{i=1}^{N}$, and trajectory-grounded lessons $\mathcal{L}_{\text{update}}$:

$$s_{t+1} = \mathcal{A}_{\text{skill_update}}(s_t, \{\delta_i\}_{i=1}^{N}, \mathcal{L}_{\text{update}}). \tag{6}$$

The updater refines the current skill s_t to candidate skill s_{t+1} based on the textual gradients and grounded lessons. The candidate skill s_{t+1} is accepted only if it outperforms s_t on the validation set $\mathcal{D}_{\text{val}}$, with s_t retained otherwise. This pipeline is repeated for a fixed number of iterations, progressively refining the skill through execution feedback and retrieved prior knowledge.

5 EXPERIMENT

We evaluate RASO using two target LLMs: GPT-5.6-Luna (OpenAI, 2026) and Qwen-3.5-9B (Qwen Team, 2026). We refer to the model that executes tasks as the target model and the model that generates or updates skill text as the optimizer model. Unless otherwise specified, we use the same model for both roles across all skill optimization methods. For skill retrieval, we use GitSkills (Destefanis et al., 2026), an external skill corpus spanning diverse domains and harnesses. Our evaluation covers four benchmarks with diverse interaction settings: OfficeQA (Opsahl-Ong et al., 2026), SpreadsheetBench (Ma et al., 2024), ALFWorld (Shridhar et al., 2021), and WebShop (Yao et al., 2022). For skill initialization, we compare RASI against three strategies: (1) No Skill, where the agent operates without an initialized skill, (2) SkillRouter (Zheng et al., 2026), which retrieves a relevant skill from the external corpus, and (3) Retrieval-Free Skill Initialization (RFSI), where the target LLM generates an initial skill directly from the task and harness descriptions without access to the external skill corpus. For iterative skill optimization, we compare RASO against TextGrad (Yuksekgonul et al., 2025), GEPA (Agrawal et al., 2026), SkillOpt (Yang et al., 2026), and WikiSkill (Tang et al., 2026). All experiments are conducted with three random seeds, and we report the mean performance over seeds.

5.1 MAIN RESULTS

Skill initialization. We first evaluate the quality of skills produced by different initialization methods, *i.e.*, before any subsequent agent rollout or skill optimization, for both GPT-5.6-Luna and Qwen-3.5-9B in Table 1. Across both models and all four benchmarks, RASI consistently achieves the highest performance. With GPT-5.6-Luna, RASI improves over Retrieval-Free Skill Initialization (RFSI) by +5.63 on OfficeQA (45.74 vs. 40.11), +4.77 on Spreadsheet (49.17 vs. 44.40), +3.24 on ALFWorld (72.64 vs. 69.40), and +1.17 on WebShop (45.06 vs. 43.89). The margin is substantially larger over the No Skill and SkillRouter baselines, particularly on OfficeQA, where both achieve only 11.44. This advantage also holds for the smaller Qwen-3.5-9B backbone, where RASI outperforms RFSI by +5.61 on OfficeQA, +2.74 on Spreadsheet, +5.72 on ALFWorld, and +10.73 on WebShop. SkillRouter, which retrieves external skills without Cross-Harness Adaptation, underperforms even the No Skill baseline on several benchmarks, which suggests direct reuse can introduce irrelevant or mismatched procedural knowledge when the retrieved content is not adapted to the target task and harness. Overall, these results show that combining external knowledge retrieval with LLM-based adaptation, as in RASI, yields a more effective initialization than either retrieval alone or retrieval-free LLM-based skill generation, providing a stronger skill initialization for subsequent optimization.

Table 1: Agent performance with GPT-5.6-Luna (OpenAI, 2026) and Qwen-3.5-9B (Qwen Team, 2026). We separate skill initialization from skill update; initialization-only rows report performance at 0 rollouts. RFSI denotes retrieval-free skill initialization. Results are mean ± standard error over three random seeds.

Model	Method	Benchmark			
		OfficeQA	**Spreadsheet**	**ALFWorld**	**WebShop**
GPT-5.6-Luna	*Skill initialization (without rollout)*				
	No skill	11.44 ± 0.70	32.98 ± 0.12	64.43 ± 1.38	43.56 ± 1.01
	SkillRouter	11.44 ± 0.97	33.21 ± 0.94	55.97 ± 1.14	42.20 ± 0.37
	RFSI	40.11 ± 0.58	44.40 ± 2.74	69.40 ± 1.55	43.89 ± 1.17
	RASI	**45.74 ± 1.40**	**49.17 ± 1.52**	**72.64 ± 1.63**	**45.06 ± 0.40**
	Skill update (with rollout)				
	TextGrad	43.80 ± 1.27	49.64 ± 3.09	70.65 ± 0.66	42.32 ± 3.13
	GEPA	43.99 ± 1.40	54.53 ± 1.46	71.14 ± 1.08	45.54 ± 0.35
	SkillOpt	45.54 ± 2.05	57.02 ± 3.60	72.64 ± 0.90	45.17 ± 0.23
	WikiSkill	41.47 ± 1.18	47.14 ± 3.98	71.14 ± 0.66	44.40 ± 0.64
	RASO	**49.03 ± 1.85**	**63.33 ± 2.69**	**74.13 ± 1.00**	**46.61 ± 0.11**
Qwen-3.5-9B	*Skill initialization (without rollout)*				
	No skill	33.14 ± 1.21	28.81 ± 0.60	33.09 ± 0.66	16.27 ± 0.25
	SkillRouter	34.11 ± 0.51	23.45 ± 0.48	31.34 ± 1.29	9.42 ± 0.47
	RFSI	34.89 ± 2.93	27.74 ± 2.56	42.04 ± 2.87	12.54 ± 2.56
	RASI	**40.50 ± 0.97**	**30.48 ± 0.78**	**47.76 ± 3.53**	**23.27 ± 0.98**
	Skill update (with rollout)				
	TextGrad	36.82 ± 1.59	25.95 ± 0.63	41.04 ± 2.28	10.96 ± 4.44
	GEPA	36.43 ± 1.97	24.05 ± 1.56	43.53 ± 1.99	12.80 ± 2.20
	SkillOpt	37.21 ± 1.21	29.52 ± 2.21	43.03 ± 3.17	13.36 ± 1.70
	WikiSkill	37.40 ± 3.03	29.76 ± 1.80	42.79 ± 2.93	13.43 ± 2.01
	RASO	**42.25 ± 1.52**	**31.55 ± 0.31**	**51.00 ± 2.45**	**24.73 ± 0.41**

Skill Update We compare RASO with existing skill optimization methods in Table 1. Following the conventional skill optimization setup, existing methods start from RFSI, where an initial skill is LLM-generated without rollouts or access to an external corpus, and subsequently refine it using their respective rollout-based optimization procedures. RASO overall outperforms existing skill optimization methods across both backbones. For GPT-5.6-Luna, RASO improves over the strongest competing method by +3.49 on OfficeQA (49.03 vs. 45.54), +6.31 on SpreadsheetBench (63.33 vs. 57.02), +1.49 on ALFWorld (74.13 vs. 72.64), and +1.07 on WebShop (46.61 vs. 45.54). Similarly, with Qwen-3.5-9B, RASO improves over the strongest competing method by +4.85 on OfficeQA (42.25 vs. 37.40), +1.79 on SpreadsheetBench (31.55 vs. 29.76), +7.47 on ALFWorld (51.00 vs. 43.53), and +11.30 on WebShop (24.73 vs. 13.43). These results show that RASO benefits from both retrieval-augmented initialization and subsequent retrieval-augmented skill refinement.

5.2 Analysis

Ablation studies. We conduct an ablation study of the RASO components in Table 2. Relative to the RFSI + RFSU baseline (retrieval-free settings), RASU improves performance from 40.70 to 47.56 (+6.86 points) on OfficeQA and from 51.67 to 61.07 (+9.40 points) on SpreadsheetBench. These gains demonstrate the benefit of incorporating retrieved and adapted external knowledge during iterative skill refinement. Similarly, replacing only the initialization component with RASI improves performance from 40.70 to 45.93 (+5.23 points) on OfficeQA and from 51.67 to 58.45 (+6.78 points) on SpreadsheetBench, demonstrating the benefit of retrieval-augmented initialization under the same subsequent update procedure. Combining both components leads to the strongest performance, reaching 49.03 on OfficeQA and 63.33 on SpreadsheetBench, corresponding to improvements of +8.33 and +11.66 points over the baseline. These results indicate that retrieval-augmented initialization and updating provide complementary gains within RASO.

Table 2: Ablation on the initialization (Init.) and update components with GPT-5.6-Luna.

Method		Benchmark	
Init.	**Update**	**OfficeQA**	**Spreadsheet**
RFSI	RFSU	40.70 ± 1.67	51.67 ± 2.03
RFSI	**RASU**	47.56 ± 1.85	61.07 ± 2.15
RASI	RFSU	45.93 ± 1.58	58.45 ± 1.78
RASI	**RASU**	**49.03 ± 1.85**	**63.33 ± 2.69**

Table 3: Effect of adaptation with GPT-5.6-Luna. RASU used RASI with adaptation as an initial.

		Benchmark	
Method	**Adaptation**	**OfficeQA**	**Spreadsheet**
RASI	×	41.86 ± 1.53	41.43 ± 2.83
RASI	✓	**45.74 ± 1.40**	**49.17 ± 1.52**
RASU	×	46.70 ± 2.05	57.86 ± 2.92
RASU	✓	**49.03 ± 1.85**	**63.33 ± 2.69**

Table 4: Update methods under a fixed RASI initialization with GPT-5.6-Luna. The highlighted row is RASO.

Method		Benchmark	
Init.	**Update**	**OfficeQA**	**Spreadsheet**
RASI	–	45.74 ± 1.40	49.17 ± 1.52
	TextGrad	43.41 ± 1.97	57.97 ± 3.21
	GEPA	45.74 ± 1.72	58.69 ± 0.83
	SkillOpt	46.51 ± 2.10	60.24 ± 1.52
	WikiSkill	47.09 ± 2.42	55.00 ± 4.76
	RASU	**49.03 ± 1.85**	**63.33 ± 2.69**

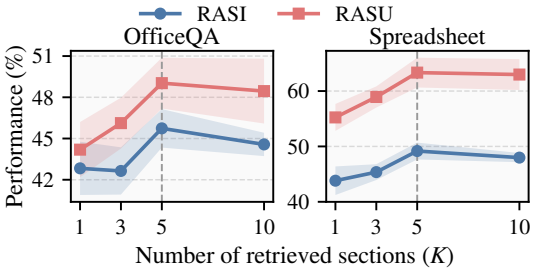

Figure 3: Effect of the number of retrieved sections K with GPT-5.6-Luna. The dashed line marks our choice of $K = 5$, and shading denotes standard error.

Effect of Cross-Harness Adaptation. In Table 3, we examine the effect of Cross-Harness Adaptation in both RASI and RASU. For RASI, applying the adaptation improves performance from 41.86 to 45.74 (+3.88 points) on OfficeQA and from 41.43 to 49.17 (+7.74 points) on SpreadsheetBench. A similar benefit is observed during skill updating. Under the same RASI initialization, incorporating the adaptation into RASU improves performance from 46.70 to 49.03 (+2.33 points) on OfficeQA and from 57.86 to 63.33 (+5.47 points) on SpreadsheetBench. These consistent gains across both stages demonstrate the importance of adapting retrieved knowledge from diverse domains and harnesses to the target task and execution harness for effective skill initialization and iterative refinement.

Comparison with baselines under RASI initialization. Table 4 compares different skill update methods under the same RASI initialization, thereby isolating the effect of the update procedure. RASU achieves the highest performance on both benchmarks, reaching 49.03 on OfficeQA and 63.33 on SpreadsheetBench. Compared with the strongest alternative update method on each benchmark, RASU improves performance by +1.94 points over WikiSkill on OfficeQA (47.09 to 49.03) and by +3.09 points over SkillOpt on SpreadsheetBench (60.24 to 63.33). Relative to RASI initialization without subsequent updates, RASU yields gains of +3.29 and +14.16 points on OfficeQA and SpreadsheetBench, respectively. These results show that, under the same retrieval-augmented initialization, RASU provides more effective iterative skill refinement than existing update methods.

Effect of retrieval size. Figure 3 analyzes the effect of the number of retrieved sections K for both RASI and RASU. Performance generally improves as K increases from 1 to 5, with the highest performance achieved at $K = 5$ on both OfficeQA and SpreadsheetBench. The gains are particularly pronounced for RASU, indicating that access to multiple relevant sections is useful when refining skills based on rollout feedback. Further increasing K to 10 slightly degrades performance for both RASI

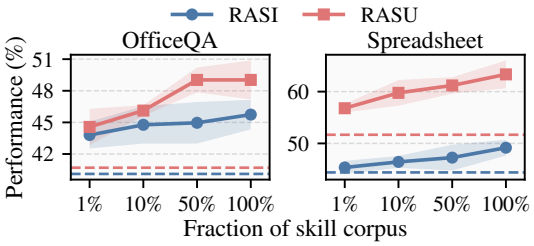

Figure 4: Effect of skill corpus size with GPT-5.6-Luna. Dashed lines mark 0% (no corpus), and shading denotes standard error.

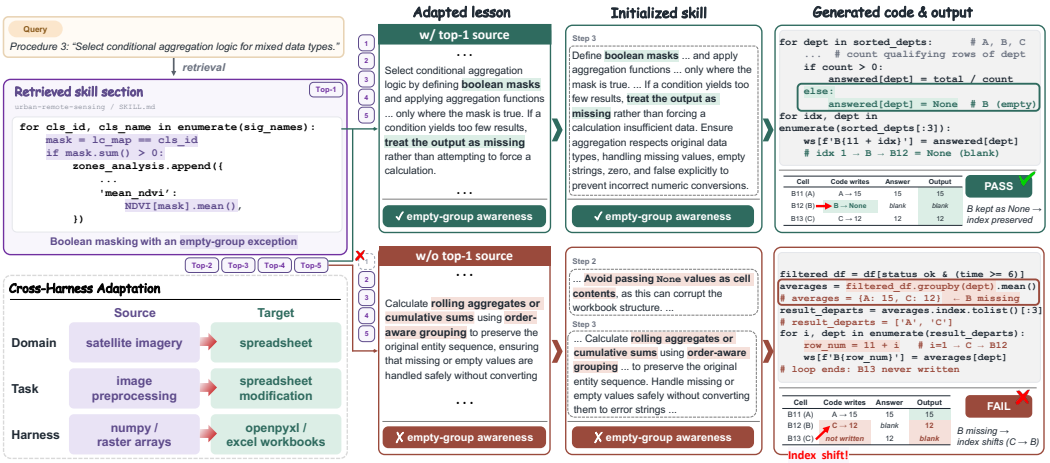

Figure 5: **Qualitative example of Cross-Harness Adaptation.** RASI retrieves a skill from a mismatched domain and harness (satellite imagery with numpy/raster arrays) and adapts its procedure into the target spreadsheet harness. We compare the resulting lessons, initial skills, and generated code *with* (top) and *without* (bottom) the top-1 of the five retrieved sections given as input. Highlights mark the differences between the two runs, which are reflected in the generated code and its output. Without the top-1 skill, the model overlooks the missing group, causing an index shift.

and RASU, suggesting that retrieving additional sections may introduce redundant or less relevant information. Accordingly, we set $K = 5$ for all experiments to balance knowledge coverage against retrieval noise.

Effect of skill corpus size. Figure 4 studies the effect of external skill corpus size for RASI and RASU. Enabling retrieval from only 1% of the corpus already yields a clear performance improvement over no-retrieval counterparts (0%; dashed line), revealing that even a small amount of retrieved external knowledge is beneficial. As the available corpus grows, performance generally continues to improve, with the largest gains observed on SpreadsheetBench. Overall, these results suggest that retrieval itself provides an immediate benefit, while increasing corpus size further improves the opportunity to retrieve useful knowledge.

Qualitative results. Figure 5 presents a SpreadsheetBench task that requires computing the average of qualifying rows for each department. For the given data, department B has no qualifying rows; hence, the expected output for its cell is blank. The top-1 retrieved section comes from an urban remote-sensing skill whose domain and harness differ from the target. Through Cross-Harness Adaptation, RASI rewrites the source's rule of skipping empty groups as "treat the output as missing," and the generated code accordingly leaves B's cell blank. When this section is removed, the generated code groups only the filtered rows, causing C's mean to shift into B's cell. This example suggests that cross-harness adaptation transfers the procedural structure and intent of retrieved sections while adapting them to the target harness format.

6 Conclusion

We propose Retrieval-Augmented Skill Optimization (RASO), which leverages an external skill corpus as prior knowledge to support both skill initialization and updating. Rather than directly reusing retrieved skills, RASO applies Cross-Harness Adaptation to transfer relevant procedural knowledge to the target task and execution harness. RASO combines Retrieval-Augmented Skill Initialization (RASI), which constructs a strong initial skill without agent rollouts, with Retrieval-Augmented Skill Update (RASU), which further refines the skill by retrieving knowledge guided by execution feedback. Across four agent benchmarks and two model backbones, RASO consistently outperforms retrieval-free skill optimization baselines, demonstrating the value of incorporating external skill knowledge at both initialization and update.

Reproducibility Statement

Our skill corpus is the public collection of Destefanis et al. (2026); the retrieval index, the prompts for query generation and cross-harness adaptation, and the validation protocol for accepting an update, along with the evaluation protocols, are given in the Appendix. Every experiment is run three times with different seeds, and we report the standard error over the three runs. All comparisons are made at equal rollout budgets, and model versions and decoding settings are listed in the Appendix.

AI Use Statement

Large language models are the object of study in this work: they serve as the agents being optimized, as the query generator, and as the adapting operator. Beyond this experimental role, we used an LLM assistant for writing assistance: grammar and wording of the manuscript and code editing. All research questions, method design, experiments, and analyses were carried out by the authors, who verified every claim and result reported in the paper and take full responsibility for its content.

Ethics Statement

Our skill corpus is collected from publicly available repositories, and we use it under the terms released by Destefanis et al. (2026); we do not redistribute the underlying skills, and we retain their provenance so that authorship can be traced. Because such skills are written by third parties, they may contain errors, licensing constraints, or sensitive strings; adapted skills inherit this harness.

References

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *ICLR*, 2026.

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Christian Alexander Calvo, Weiyuan Chen, and Tu Vu. EvoSkill: Automated skill discovery for multi-agent systems. In *COLM*, 2026.

Anthropic. Equipping agents for the real world with Agent Skills, October 2025. URL `https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills`.

Kunfeng Chen, Qihuang Zhong, Juhua Liu, and Bo Du. SkillCAT: Contrastive, assessment-augmented and topology-aware skill self-evolution for LLM agents. *arXiv:2606.13317*, 2026.

Jaewon Chu, Jinwoo Seo, Jaewon Cho, Jeehye Na, Yunyang Xiong, Youngdae Kim, and Hyunwoo J. Kim. AgentGrad: Intervention-guided prompt optimization for multi agent systems. In *NeurIPS*, 2026.

Giuseppe Destefanis, Daniel Graziotin, Matteo Vaccargiu, and Marco Ortu. GitSkills: A dataset of agent skills on GitHub. *arXiv:2608.10906*, 2026.

Ruomeng Ding, Wei Cheng, Minglai Shao, and Chen Zhao. SkillGen: Learning domain skills for in-context sequential decision making. In *AAAI*, 2026.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. In *ACL-Findings*, 2026.

Dawei Fu, Cheng Jiang, Sitian Qian, Huainan Wang, and Zhongkai Hao. SE-GoS: Self-evolving graph-of-skills for skill library at scale. *arXiv:2609.08228*, 2026.

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

Fangzhou Li, Pagkratios Tagkopoulos, and Ilias Tagkopoulos. SkillFlow: Scalable and efficient agent skill retrieval system. In *COLM*, 2026a.

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:2603.02176*, 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:2602.12670*, 2026c.

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. In *EMNLP*, 2026.

Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. SpreadsheetBench: Towards challenging real world spreadsheet manipulation. In *NeurIPS*, 2024.

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

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

Johannes Moll, Jean-Philippe Corbeil, Jiazhen Pan, Martin Hadamitzky, Daniel Rueckert, Lisa Adams, and Keno Bressem. GRASP: Gated regression-aware skill proposer for self-improving LLM agents. In *EMNLP*, 2026.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2Skill: Distill trajectory-local lessons into transferable agent skills. *arXiv:2603.25158*, 2026.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition, July 2026. URL `https://openai.com/index/gpt-5-6/`.

Krista Opsahl-Ong, Arnav Singhvi, Jasmine Collins, Ivan Zhou, Cindy Wang, Ashutosh Baheti, Owen Oertell, Jacob Portes, Sam Havens, Erich Elsen, Michael Bendersky, Matei Zaharia, and Xing Chen. OfficeQA Pro: An enterprise benchmark for end-to-end grounded reasoning. *arXiv:2603.08655*, 2026.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *ICLR*, 2026.

Qianjun Pan, Yutao Yang, Junsong Li, Jie Zhou, Kai Chen, Xin Li, Qin Chen, and Liang He. Anything2Skill: Compiling external knowledge into reusable skills for agents. *arXiv:2606.09316*, 2026.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *EMNLP*, 2023.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 2009.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *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:2604.24594*, 2026.

Liyan Tang, Cyrus Rashtchian, Chun-Sung Ferng, Andrew Tomkins, Da-Cheng Juan, and Tu Vu. WikiSkill: Compiling agent experience into persistent knowledge for skill evolution. *arXiv:2608.27454*, 2026.

Xiangru Tang, Tianrui Qin, Tianhao Peng, Ziyang Zhou, Daniel Shao, Tingting Du, Xinming Wei, Peng Xia, Fang Wu, He Zhu, et al. Agent KB: Leveraging cross-domain experience for agentic problem solving. *arXiv:2507.06229*, 2025.

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. *TMLR*, 2024.

Hanyu Wang, Yifan Lan, Bochuan Cao, Lu Lin, and Jinghui Chen. SkillGrad: Optimizing agent skills like gradient descent. *arXiv:2605.27760*, 2026.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *ICML*, 2025.

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

Zhiling Yan, Dingjie Song, Hanrong Zhang, Wei Liang, Yuxuan Zhang, Yutong Dai, Lifang He, Philip S Yu, Ran Xu, Xiang Li, et al. OpenSkill: Open-world self-evolution for llm agents. In *EMNLP*, 2026.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *ICLR*, 2024a.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024b.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive strategy for self-evolving agent skills. *arXiv:2605.23904*, 2026.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. In *NeurIPS*, 2022.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 2025.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *ICLR*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *AAAI*, 2024.

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. In *COLM*, 2026.

A Appendix

A.1 Implementation Details

For all update methods, we run skill optimization for 2 epochs over $\mathcal{D}_{\text{train}}$ with a minibatch size of 40 tasks per iteration. For Qwen-3.5-9B, we use a sampling temperature of 0.0 for agent rollouts and 0.7 for the optimizer model, and serve the model with vLLM on a single NVIDIA RTX A6000 GPU. For GPT-5.6-Luna, we set the reasoning effort to `low` for both the target and optimizer models. Following SkillOpt (Yang et al., 2026), we process the rollout minibatch in chunks when generating textual gradients and retrieval queries (Eq. 5). The trajectories in each minibatch are split into chunks of size 8, and $\mathcal{A}_{\text{query_update}}$ produces gradient–query pairs for each chunk independently. A single LLM call then merges the chunk-level outputs into the final set $\{(q_i, \delta_i)\}_{i=1}^{N}$, consolidating pairs that describe the same failure mode and removing redundant queries. This design keeps each call within the context limit while allowing the gradients to reflect failure patterns across the entire minibatch. Unless otherwise specified, we retrieve the top-$K = 5$ sections per query.

Since the external skill corpus is collected from public repositories, it may contain skills written specifically for our evaluation benchmarks. We therefore construct a per-benchmark blocklist and remove it from the corpus before building the retrieval index. For each benchmark, we search the corpus for keywords that identify it. The benchmark name and the names of the datasets it is built on. Whenever any skill matches, we exclude the entire repository containing it rather than the matched skill alone. We choose repository-level exclusion to be conservative. A repository that targets a benchmark often contains companion skills that encode benchmark-specific procedures without naming the benchmark, and such skills would evade keyword matching.

For each benchmark, we generate a task description T and a harness description H once, before any optimization run, and keep both fixed across all methods, models, and seeds. Both descriptions are produced by a coding agent (codex) that reads the benchmark's harness code and states the task objective and output format (T) and the tools, file access, observation format, interaction budget, and scoring rule exposed by the harness (H). The coding agent was not given access to test instances, and neither description was revised after observing test performance.

To isolate the contribution of retrieval, Retrieval-Free Skill Initialization (RFSI) follows exactly the same pipeline as RASI except for retrieval and adaptation. Specifically, RFSI receives the identical T and H, uses the same procedure and requirement decomposition $\{c_i\}_{i=1}^{M}$, and synthesizes the skill with the same skill-initialization system prompt; the only difference is that the grounded lessons $\mathcal{L}_{\text{init}}$ are omitted. Hence, the gap between RASI and RFSI reflects the retrieved and adapted knowledge rather than the procedure decomposition or the prompt. All skill update baselines start from this RFSI skill, so every update method is compared under the same T, H, and initial skill.

A.2 Baseline Details

Skill initialization. We compare RASI against three initialization strategies, all evaluated at zero rollouts. **No skill** runs the agent without an initialized skill. **SkillRouter** (Zheng et al., 2026) selects skills from a large library with a two-stage retrieve-and-rerank pipeline, in which a bi-encoder retrieves candidate skills and a cross-encoder reranks them using the full skill body. We use it to retrieve a relevant skill from the external skill corpus. **Retrieval-Free Skill Initialization (RFSI)** generates an initial skill with the target LLM directly from the task and harness descriptions, without access to the external skill corpus.

Skill update. We compare RASU against four update methods. **TextGrad** (Yuksekgonul et al., 2025) treats the skill as a text variable and optimizes it by backpropagating natural-language feedback, in which an LLM critiques execution results to produce textual gradients that are then used to revise the skill. **GEPA** (Agrawal et al., 2026) evolves the skill through reflective mutation, in which an LLM reflects on execution traces to propose revisions, and maintains a Pareto front of candidates to preserve diverse strategies during selection. **SkillOpt** (Yang et al., 2026) converts scored rollouts into bounded add, delete, and replace edits on a skill document and accepts an edit only when it improves a held-out validation score. **WikiSkill** (Tang et al., 2026) co-evolves skills with a persistent wiki, in which agent trajectories are consolidated into pages documenting failure modes and successful strategies, and skill updates are proposed by consulting this wiki. Unless otherwise

Table 5: Number of tasks in each split.

Benchmark	Train	Validation	Test
OfficeQA	50	24	172
SpreadsheetBench	80	40	280
ALFWorld	39	18	134
WebShop	80	40	500

specified, we use the same model as both the target and optimizer model for all methods, and none of the baselines accesses the external skill corpus during optimization.

A.3 Benchmark Details

We evaluate on four benchmarks that cover diverse interaction settings, ranging from document-grounded question answering to spreadsheet manipulation, embodied household tasks, and web navigation. For each benchmark, we use disjoint training, validation, and test splits ($\mathcal{D}_{\text{train}}$, $\mathcal{D}_{\text{val}}$, $\mathcal{D}_{\text{test}}$), whose sizes are summarized in Table 5. Following SkillOpt (Yang et al., 2026), we use the same train/validation/test splits for all methods, so update acceptance and skill selection are performed on an identical validation set.

OfficeQA. OfficeQA (Opsahl-Ong et al., 2026) is a benchmark for end-to-end grounded reasoning over enterprise documents. Questions are posed over a corpus of U.S. Treasury Bulletins spanning nearly 100 years and about 89,000 pages, and require the agent to parse documents, retrieve relevant information across both unstructured text and tables, and perform numerical reasoning. Answers are numerical and are scored as correct when they fall within an allowed relative error tolerance. We use all 246 questions and split them into 50, 24, and 172 questions for training, validation, and testing, respectively.

SpreadsheetBench. SpreadsheetBench (Ma et al., 2024) is a benchmark for real-world spreadsheet manipulation, built from questions collected from online Excel forums. Each instruction requires the agent to read and modify spreadsheet files, and is evaluated in an online-judge style, where a solution is considered correct only if it produces the expected result on multiple spreadsheet files serving as test cases. We use 400 tasks, split into 80, 40, and 280 tasks for training, validation, and testing, respectively.

ALFWorld. ALFWorld (Shridhar et al., 2021) is a text-based embodied environment that aligns the ALFRED household tasks with the TextWorld engine. The agent navigates a simulated household and interacts with objects through textual actions to complete goals from six task types: pick and place, examine in light, clean and place, heat and place, cool and place, and pick two and place. We use 39 and 18 games for training and validation, and evaluate on the 134 out-of-distribution evaluation games.

WebShop. WebShop (Yao et al., 2022) is a simulated e-commerce environment containing about 1.18 million real-world products and 12,087 crowd-sourced instructions. Given a natural-language instruction describing a desired product, the agent searches, navigates product pages, and selects options to make a purchase, and is rewarded according to how well the purchased product matches the requested attributes, options, and price. We use 80 and 40 instructions for training and validation, and evaluate on the 500 test instructions.

A.4 Skill Update Methods under Fixed Initialization

To isolate the contribution of the skill update procedure, we compare skill update methods while fixing the initial skill. Table 4 in the main paper reports skill update results of GPT-5.6-Luna initialized with RASI. Tables 6, 7, 8 report the remaining three combinations of initialization and LLM(Qwen-3.5-9B and GPT-5.6-Luna) on OfficeQA and SpreadsheetBench. In all four combinations, RASU

Table 6: Update methods under a fixed RASI initialization with Qwen-3.5-9B. The highlighted row is RASO.

Method		Benchmark	
Init.	**Update**	**OfficeQA**	**Spreadsheet**
RASI	–	40.50 ± 0.97	30.48 ± 0.78
	TextGrad	38.76 ± 1.40	30.48 ± 0.78
	GEPA	38.18 ± 0.70	31.07 ± 0.21
	SkillOpt	40.70 ± 0.89	30.60 ± 0.66
	WikiSkill	40.12 ± 0.34	30.48 ± 0.78
	RASU	**42.25 ± 1.52**	**31.55 ± 0.31**

Table 7: Update methods under a fixed RFSI initialization with Qwen-3.5-9B. The highlighted row is RASU.

Method		Benchmark	
Init.	**Update**	**OfficeQA**	**Spreadsheet**
RFSI	–	34.89 ± 2.93	27.74 ± 2.56
	TextGrad	36.82 ± 1.59	25.95 ± 0.63
	GEPA	36.43 ± 1.97	24.05 ± 1.56
	SkillOpt	37.21 ± 1.21	29.52 ± 2.21
	WikiSkill	37.40 ± 3.03	29.76 ± 1.80
	RASU	**38.48 ± 1.25**	**30.60 ± 2.82**

Table 8: Update methods under a fixed RFSI initialization with GPT-5.6-Luna. The highlighted row is RASU.

Method		Benchmark	
Initialization	**Update**	**OfficeQA**	**Spreadsheet**
RFSI	–	40.11 ± 0.58	44.40 ± 2.74
	TextGrad	43.80 ± 1.27	49.64 ± 3.09
	GEPA	43.99 ± 1.40	54.53 ± 1.46
	SkillOpt	45.54 ± 2.05	57.02 ± 3.60
	WikiSkill	41.47 ± 1.18	47.14 ± 3.98
	RASU	**47.56 ± 1.85**	**61.07 ± 2.15**

achieves the highest performance, showing that retrieval-augmented update improves over retrieval-free updates regardless of the initial skill and the target model.

Under LLM-init (RFSI). With GPT-5.6-Luna (Table 8), RASU improves the initial skill by +7.45 and +16.67 points on OfficeQA and SpreadsheetBench, respectively, and outperforms the strongest competing baseline, SkillOpt, by +2.02 (47.56 vs. 45.54) and +4.05 (61.07 vs. 57.02). With Qwen-3.5-9B (Table 7), RASU improves the initial skill by +3.59 and +2.86 points and outperforms the strongest competing baseline, WikiSkill, by +1.08 (38.48 vs. 37.40) and +0.84 (30.60 vs. 29.76). In contrast, TextGrad and GEPA degrade the initial skill on SpreadsheetBench (25.95 and 24.05 vs. 27.74).

Under RASI. Table 6 shows retrieval-free baselines struggle to improve upon the retrieval-augmented initial skill with Qwen-3.5-9B under RASI. On OfficeQA, TextGrad, GEPA, and WikiSkill fall below the initial skill (38.76, 38.18, and 40.12 vs. 40.50), and SkillOpt improves it by only +0.20. On SpreadsheetBench, the largest gain among them is +0.59 (GEPA). RASU, in contrast, improves the initial skill by +1.75 and +1.07 points, reaching 42.25 and 31.55. TextGrad and WikiSkill return the initial skill unchanged on SpreadsheetBench, since no candidate update was improved over the validation set.

Complementarity of the two stages. The two stages contribute gains that neither achieves alone. With Qwen-3.5-9B on OfficeQA, RASI without any update (40.50) already outperforms LLM-init (RFSI) followed by RASU (38.48), and combining both stages yields the best performance (42.25).

A.5 Optimization Cost

Tables 9 and 10 report the API cost and rollout count of skill optimization with GPT-5.6-Luna. All methods are optimized for the same 2 epochs. The rollout counts differ only because of how each method schedules evaluation. TextGrad, WikiSkill, SkillOpt and RASO update the skill from training rollouts and evaluate the updated skill on the validation set directly. On the other hand, GEPA first re-evaluates each updated skill on a support subset of the training set and proceeds

Table 9: API cost (USD) of skill optimization with GPT-5.6-Luna. The lowest cost in each column is in bold.

Method	OfficeQA	Spreadsheet	ALFWorld	WebShop
TextGrad	1.45	0.97	2.04	41.03
GEPA	1.69	0.58	1.79	18.05
SkillOpt	2.23	2.24	2.75	17.86
WikiSkill	1.06	**0.43**	1.30	11.15
RASO	**0.84**	0.49	**0.79**	**10.28**

Table 10: Number of rollouts used for skill optimization, excluding test-set evaluation.

Method	OfficeQA	Spreadsheet	ALFWorld	WebShop
TextGrad	196	320	114	360
GEPA	272	360	174	370
SkillOpt	236	360	154	400
WikiSkill	196	320	114	360
RASO	196	320	114	360

to validation only when this yields an improvement, which incurs additional rollouts. SkillOpt performs an additional slow, meta update at the end of every epoch, which likewise adds rollouts. RASO therefore uses exactly as many rollouts as TextGrad and WikiSkill, and fewer than GEPA and SkillOpt.

Under this budget, RASO achieves the lowest cost on three of the four benchmarks and remains within \$0.06 of the cheapest method on SpreadsheetBench. Since the budgets are matched, the savings reflect the cost of each rollout and optimization LLM call rather than the number of rollout. At an identical budget, RASO reduces the cost of TextGrad by 42–75%, with the largest gap on WebShop (\$41.03 vs. \$10.28).

A.6 GoS Preprocessing Cost

Data and assumptions. We project the offline LLM-call count of GoS (Liu et al., 2026) for $N = 1.8 \times 10^6$ raw GitSkills documents (Destefanis et al., 2026). Applying the GoS parser to a public sample of 13,000 distinct contents yields 11,292 accepted documents, of which 11,169 require semantic completion.[1] The latter count follows the implementation's checks for missing capability, input/output, or semantic-context fields. We extrapolate these proportions to the parser-accepted subset, excluding input-repair costs, and assume one node per accepted document with collision-free identifiers. Document-level memorization limits semantic completion to one request per document, avoiding repeated requests across chunks.

Call-count derivation. GoS batches the surviving relation candidates for each focus node into one validation request; a node with no surviving candidates issues none. Let q be the number of validation requests per node in one pass. GoS Table 17 reports $1{,}050/2{,}000 = 0.525$ and $192/200 = 0.96$ for two builds with different metadata completeness and concurrency (Liu et al., 2026). We use these as sensitivity settings rather than confidence bounds or a validated scaling law. For one completion pass, one relation pass, and no retries,

$$\begin{aligned} C_{\text{sem}} &= N\frac{11{,}169}{13{,}000} \approx 1.55\text{M}, \\ C_{\text{rel}} &= N\frac{11{,}292}{13{,}000}q \approx 0.82\text{–}1.50\text{M}, \\ C_{\text{total}} &= C_{\text{sem}} + C_{\text{rel}} \approx 2.37\text{–}3.05\text{M}. \end{aligned} \tag{7}$$

The first term counts documents requiring completion; the second counts validation requests over accepted nodes.

[1] https://github.com/giuseppedestefanis/gitskills-sample

Computational cost. We estimate GPU cost using Qwen2.5-7B-Instruct (BF16) on one A100 80GB with vLLM 0.6.3 and batch size one. A linear fit to the published benchmark gives $t_{\text{ref}} \approx 13.16$ seconds for a reference request of 4,761 input and 1,039 output tokens.[2] Applying this workload to both stages gives the serial estimate

$$H_{\text{ref}} = \frac{C_{\text{total}} t_{\text{ref}}}{3{,}600} \approx 8.7\text{–}11.1\text{k A100-hours.}$$

A.7 Task Descriptions

We reproduce the task descriptions T used by the optimizer in RASI and RASU. For each benchmark, T is fixed throughout optimization and reproduced below verbatim apart from formatting.

OfficeQA.

Task description T

Answer one question per episode, grounded in historical U.S. Treasury Bulletin documents. The question asks for one or a small fixed set of numeric, percentage or short-text values derived from figures in specific bulletin pages. The question's own wording states the quantity wanted and the arithmetic or statistical procedure required to produce it, and states the format the answer must be returned in. Some questions name that procedure by an established statistical or financial term – a transformation, a measure of risk or dispersion, a smoothing or moving-window method – rather than spelling out the arithmetic step by step, and the correct answer depends on that named procedure's exact, standard definition.

SpreadsheetBench.

Task description T

Write one self-contained Python script that reads the workbook at the given `INPUT_PATH` and, following a natural-language instruction, writes a workbook to the given `OUTPUT_PATH` whose cells in one specified answer range hold the correct final values. The instruction is written the way a person asks for help on a spreadsheet forum rather than as a formal spec, and asks for things like conditional lookups across sheets (the kind a spreadsheet does with VLOOKUP, INDEX/MATCH, or COUNTIFS), conditional sums with several criteria, date- and time-based logic, consolidating data spread across sheets or rows, deleting or extracting rows matching a condition, or extracting and transforming free-text substrings.

The output is a transformed copy of the episode's existing input workbook, not a replacement workbook assembled from scratch. Solving the requested calculation and preserving the workbook artifact are separate obligations: the script must keep the input workbook's sheets and existing content while changing the requested answer range. General spreadsheet or dataframe techniques are useful only when they obey that input/output contract.

ALFWorld.

Task description T

A household instruction is given as one sentence in the first observation, in the form "Your task is to: ...". It names an object type and a destination receptacle, using the simulator's own names and numbering. Some instructions require an intermediate operation on the object first – heating it, cooling it, or cleaning it at the appropriate appliance – and some require two objects of the same kind. The opening observation lists the receptacles in the room by name; it does not say what is inside them or where the wanted object is.

The work is an online, partially observed search-and-execution problem under an action budget. At each turn the agent must update which locations are still unvisited and which have complete observations, choose the next admissible action by the information or task progress it can buy, and preserve enough

[2] The reference lengths are GoS Table 17's aggregate token counts divided by its 1,050 validator requests, not measured per-stage averages. The timing fit is $t(L, O) = 0.000175483L + 0.0118651O$: `https://qwen.readthedocs.io/en/v2.5/benchmark/speed_benchmark.html`.

Task description T

steps for every manipulation still required after the object is found. This has the structure of budgeted active search and receding-horizon planning: exploration order, value of information, and remaining cost-to-go must be planned together rather than treating search, processing, and placement as independent phases.

WebShop.

Task description T

A shopping instruction on a text-only e-commerce site names, in one sentence, a product to buy and the attributes wanted of it. The attributes are of two sorts and both are scored: properties of the product itself, and options set on the product page before buying, such as size, colour, count or scent. The instruction usually also states a price ceiling.

The site has a search page, a results page listing products with an identifier, a title and a price, and a product page carrying the description, the available options and the buy control.

A.8 Harness Descriptions

We reproduce the harness descriptions H used by the optimizer in RASI and RASU. These descriptions specify the target execution interface and constraints used to ground Cross-Harness Adaptation. For each benchmark, H is fixed throughout optimization and reproduced below verbatim apart from formatting.

OfficeQA.

Harness description H

- Tools, callable zero or more times per round: `glob(pattern)` over the document corpus; `read(path, start, limit)`, a line-limited read returning lines `start` through `start+limit-1` (default 80 lines) and truncated to 4000 characters; `grep(pattern, path)`, a case-insensitive substring match returning at most 20 matching lines with their line numbers.
- Given automatically at the start of every episode: the question text, the bulletin page(s) the answer is drawn from pre-rendered as plain text, up to 20 candidate file paths, and the original citation strings.
- The corpus is roughly 700 text files, one per bulletin issue. Page boundaries appear as `## Page <n>`. Tables are rendered as markdown tables.
- At most 24 model rounds per episode. Each round the agent either calls tools or emits a final answer. A round that does neither ends the episode immediately as a failure, and reaching round 24 without an answer ends it the same way.
- No one is present. The tools answer, and nothing else does: there is no one to ask for clarification and no way to wait for more information than a tool call returns.
- The final answer is returned as `<answer>...</answer>` and scored by exact match against a normalized form of the gold answer.

SpreadsheetBench.

Harness description H

WORKBOOK AND HARNESS

The agent receives an informal spreadsheet question and writes one self-contained Python script. `INPUT_PATH` and `OUTPUT_PATH` are supplied by the harness and refer to an existing workbook and its transformed copy. The answer range named in the question is the intended edit surface; sheets, formulas, formatting, and content outside that surface belong to the input artifact and normally survive unchanged.

Harness description *H*

The preview is a small window into the workbook rather than a schema. It gives sheet names, dimensions, and example rows, but the real file can contain more rows, repeated regions, merged cells, and header labels in different positions. Forum-style requests in this family are usually solved by recovering a source-to-target mapping matrix from those labels: a predicate/source field, a return field, an output schema, and a write anchor are not necessarily the same column. Fixed preview letters are therefore only clues.

COMMON WORKBOOK BEHAVIOR

The workbooks often mix repeated or multi-row headers with blank spacer cells and records whose entity continues over several rows. Spreadsheet code that handles this family is commonly described with `header=None`, selective fill-forward of header labels, and order-aware grouping such as `groupby(sort=False)` or `maintain_order`; a `PARTITION BY` running state is another way people describe cumulative logic that must not reorder entities. Other questions contain comments or identifiers with punctuation, dashes, or empty pieces, so delimiter-aware text parsing is only appropriate when the instruction and the observed values support it.

The cell values themselves may be missing, empty strings, zero, false, numeric, dates, or text with different case. Refer to helpers such as `safe_cell_value` when comparing these values without turning a missing cell into the string `"None"`. Whether a delimiter is part of a value, whether case is significant, and whether zero/false differs from missing are decisions about this workbook rather than blanket conversion rules.

COORDINATES AND STRUCTURAL CHANGES

Excel and `openpyxl` worksheet coordinates are one-based. A utility such as `column_index_from_string` belongs at the boundary between column letters and the worksheet API; a zero-based dataframe index is a separate representation. Checking row offsets and letters beyond `Z` before writing avoids a subtle shift in the answer range.

Clearing a cell is different from changing workbook structure. When a question actually removes rows or columns, examples use `delete_rows` or `delete_cols` in reverse index order so later targets do not move. Rebuilding a guessed rectangle, compacting a sheet, or changing `max_row`/`max_column` does not preserve the existing artifact and can leave stale content behind.

FORMULAS AND CHECKING

Existing formulas are workbook artifacts, while a requested answer may be a literal value. The normal formula-preserving workbook view should be used for the transformation, and only the requested cells should change unless the question explicitly asks for a broader edit. A checker may open workbooks with `data_only=True` to compare cached results; that describes scoring, not a generic way to read and save the input. Correctness therefore requires checking the target cells and nearby boundaries as well as unrelated formulas and formatting, not merely observing that an output file exists.

OUTPUT CONSUMER CONTRACT

After the script exits, the harness reopens `OUTPUT_PATH` and reads the named answer cells from the serialized workbook. There is no interactive spreadsheet session or post-hoc repair step. The write path must distinguish source formulas, computed answer values, and the representation consumed by the checker, and should verify the reopened target cells before finishing.

FORMULA AND VALUE REPRESENTATION

In related spreadsheet work, formula expressions, cell references, cached values, and literal values are distinct terms. Workbook-handling discussions may describe these ideas alongside `openpyxl` and its `data_only` option; which distinction matters depends on the requested cells and the artifact.

EXECUTION LIMITS

The script has a 120-second wall-clock limit and at most 30 attempts. A runtime error consumes an attempt, while an incomplete or missing output is a failed episode. The standard library, `openpyxl`, and `pandas` are available, but no particular library, rounding rule, streaming mode, or formula is mandatory unless the question and workbook require it.

ALFWorld.

Harness description *H*

PROBLEM-TYPE LABELS

The task includes concerns commonly named:

- partially observable search as a belief-state update problem
- frontier exploration over a visited-set / coverage ledger
- precondition/effect tracking for admissible actions and state transitions
- resource-bounded planning with cost-to-go and receding-horizon replanning
- multi-object continuation and completion feasibility under a finite action budget

These are problem-type labels, not command templates or a prescribed action sequence. The current admissible-action list remains the source of exact commands.

ACTION CONTRACT

- The agent emits exactly one text command per turn inside an `<action>...</action>` block, preceded by a `<think>...</think>` block. There is no code execution, file access, or tool-calling interface.
- Every turn supplies the task sentence, the number of steps already taken, recent observation/action pairs, the current observation, and the complete admissible-action list for the current state. Choose a command from that list; an invented or currently unavailable command cannot advance the task and still spends the turn.
- The exact syntax for taking, heating, cooling, cleaning, and placing is the syntax shown in the current admissible-action list. There is no general instruction to replace those commands with a generic `use` command or to invent an unlock action. Copy the applicable admissible command exactly.
- Actions change the state and therefore can change the next observation and admissible-action list. Replan from the new state after every action rather than committing to a fixed command sequence whose later actions may no longer be admissible.

SEARCH STATE AND INFORMATION VALUE

- The opening observation names every receptacle in the room by name and number. This is the complete, fixed candidate set of places where an object can be; contents are initially hidden.
- An observation of a reached and, where necessary, opened receptacle reports all of its contents, not a sample. Its contents will not later reveal an object omitted from that observation. Record each such receptacle and its contents in a persistent visited ledger.
- A settled receptacle offers no new location information when revisited, but the revisit still costs a step. An unvisited admissible location can reduce uncertainty. A settled location can still be revisited for irreversible task progress, such as retrieving a remembered object or placing it at the named destination; it just must not be revisited as if its contents were unknown. Exploration should choose among unvisited candidates and should never restart from the beginning or discard the ledger after moving, processing an object, or placing the first object in a two-object task.
- The environment supplies no probability table for hidden object locations. The agent must choose an exploration order from the task's object and receptacle names, observations already obtained, and the available admissible actions, then broaden to other unvisited locations after likely locations have been exhausted.

BUDGETED SEARCH AND COMPLETION

- An episode has at most 50 steps. Every response costs one step, including an invalid command, redundant revisit, empty response, or other action that changes nothing. The prompt reports steps already taken, so the remaining budget is computable even though it is not printed directly.

Harness description *H*

- The score is binary: only the simulator's `won` flag at episode end counts. Unused steps have no value of their own, and partial progress without the full required placement has the same score as no progress.
- Finding an object is not completion. The remaining budget must still cover the admissible actions needed to acquire it, perform every required heat, cool, or clean operation, reach the named destination, and place it. A task requiring two objects is not complete after placing only the first; the visited ledger and a feasible route to the second must be retained.
- A two-object task requires both instances to reach the same named destination, but it does not require holding both simultaneously. The first may be processed and placed before acquiring the second. The destination is not required to be empty by the task contract; do not spend an action merely to prove emptiness when an admissible placement action is already available.
- At each decision, compare an action's new information or irreversible task progress with its one-step cost and the remaining cost-to-go. Reserve enough steps for the shortest currently feasible completion sequence. Once every required object has been found, additional exploration has no value unless another required object is still missing; use the budget on processing and placement instead.
- Required processing must not be skipped while a complete solution remains feasible. If completion has become impossible, stopping does not preserve value: unused steps and partial completion both score zero, and every response still consumes a step.
- There is no person to clarify ambiguity and no way to wait, pause, or save a step. Every choice must be made and acted on in the current turn.
- Consequently, a policy or skill must never instruct the agent to "stop acting," wait, emit nothing, or terminate early. Even when its current estimate says guaranteed completion is impossible, the agent must emit one currently admissible action that maximizes the remaining chance of full completion.
- A completion route is feasible when the remaining budget is greater than or equal to its estimated action cost; do not require an extra unused step by testing for a strict inequality.
- "Settled" means that a receptacle's contents are fully known. If a required object was observed there, returning to take that remembered object is task progress. If it was not observed there, do not search that settled receptacle again for the missing object; choose among unvisited candidates.

WebShop.

Harness description *H*

- The agent emits one action per turn, drawn from a two-command grammar: `search[keywords]`, accepted only when the current page has a search bar, and `click[value]`, where the value must be one of the values listed on the VALID ACTIONS line for the current page.
- Each turn's message contains the current page text, a step counter reading `[Step n/50 - k action(s) remaining including this one]`, and the VALID ACTIONS line. The counter is the only accounting there is: it states how many of the fifty are used and how many remain against that maximum, and nothing warns that the remaining quota is running low or that what has been spent so far bought anything.
- The action is matched at the END of the response: the parser takes the last `search[...]` or `click[...]` and ignores everything before it. The response is capped at 64 tokens – a second budget, spent per response rather than per episode, and never reported.
- The episode lasts 50 steps, and ends when the agent buys or when the fiftieth step is spent. Searching, clicking into a listing, going back, setting an option and buying all cost one step out of that same 50. A step not spent is not carried anywhere: the counter is not an input to the reward, and its value at the end of the episode is not read.
- Results are returned ten to a page and the catalogue holds more than one page of them, and further pages and further searches remain available while steps remain. The search is a fixed index: the same keywords return the same results in the same order every time.

Harness description H

- The conversation is never trimmed: every previous turn's page text and every action already taken stay in the prompt for the rest of the episode. Nothing has to be written down to be remembered – the searches already issued and the listings already opened are visible as they were. Combined with the fixed index, that means a repeat of either returns what is already in view. Nothing in the page text states whether a candidate is a good enough match, so the choice between searching on and buying what is in hand – an exploration versus exploitation choice – is settled by the agent and by nothing in the environment.
- The reward is computed at the moment of purchase, against the goal for this episode, as

$$\frac{(\text{attributes matched}) + (\text{options matched}) + (1 \text{ if the price is under the ceiling else } 0)}{(\text{attributes asked for}) + (\text{options asked for}) + 1},$$

  multiplied by a factor for the kind of item bought. That factor is 1.0 whenever the item came from the same query, or shares at least two of the goal's category segments, or its title shares more than a fifth of the goal's nouns; below that it falls to 0.5, 0.1 or 0.0.
- The score that is recorded is the fraction of episodes whose reward is exactly 1.0. Reaching 1.0 requires every attribute asked for to hold of the item, every option asked for to have been set on it, the price to be under the ceiling, and the kind factor to be 1.0 – all four at once.
- A purchase that misses any one of them is recorded as zero.
- The four are not symmetric in what they cost to pursue. Setting an option the goal asked for adds to the numerator and never subtracts, so a listed option that matches costs one step and cannot lower the reward. The kind factor is decided by the title of whatever is bought, and the price by its listed price, both readable before buying. Whether the item's properties are the ones asked for is settled after the episode ends, against the description text, and is never reported during it. No page displays it and no action queries it, so the amount known about it is the same at step 5 as at step 49 – spending further steps changes what is known about the other three, and leaves that one where it was.
- An episode that ends without a purchase scores zero.
- No one is present. Nothing reads what the agent writes except the action parser, and nothing answers it: there is no one to ask and no way to wait. Every response is a step, including one that changes nothing.

A.9 System Prompts

We reproduce the optimizer-side system prompts used in RASI and RASU. Runtime inputs such as TASK, ENVIRONMENT, CURRENT_SKILL, RECORDS, and retrieved SECTIONS are instantiated with the corresponding experimental artifacts. In RASU, chunk-level rollout diagnosis and minibatch consolidation are implemented as two prompts; together, they instantiate the textual-gradient and query generation stage described in Section 4.3 and Appendix A.1. The prompt bodies are reproduced below verbatim apart from formatting; subsection titles follow the terminology of the main paper.

A.9.1 RASI

Requirement and Query Generation.

RASI **System prompt**

```
You will be given a TASK an agent repeats and the ENVIRONMENT it runs
in.

Produce two lists.

"procedure" -- the steps the work actually takes, in order, 6 to 8 of
them. For each:

- "name": what the step does here, a short verb phrase.
```

RASI **System prompt**

```
- "transfer_goal": the missing general mechanism, edge-case
discipline, or
decision rule that an external technical document could contribute to
this
step. State what kind of knowledge is sought, not the answer to the
TASK.
- "query": words to find how other people carry out this same step.

"budget" -- what is finite here and how the work has to be run because
of it, 2 to 4 items. Every episode has to produce its answer before
something runs out, and what runs out differs. For each:

- "what": the finite thing, in this environment's terms, and what
spends it.
- "runout": what happens to the outcome when it is gone.
- "query": words to find how other people work under that same limit.

About every query. The catalogue is matched by plain word overlap
against document text, so a query only works if it is a NAME that
other people's documents also contain:

- the name of a tool, library, call, file format or interface this
involves -- when the work genuinely runs on one; or
- the established name of the technique or mechanism, as engineering
practice calls it -- when there is one.
Pair the name with the plain words a document about it would use in
its own description. A query that is a paraphrase of our situation,
with neither kind of name in it, matches nothing: it has to be a name.

Coverage matters as much as chronological workflow. Where the TASK and
ENVIRONMENT contain a corresponding boundary, the procedure list must
include:

- the highest-consequence ambiguous choice: selection, ranking,
stopping,
disambiguation, or choosing among plausible interpretations;
- the representation boundary where an internally correct result can
become
a wrongly scored output because it is serialized, written, parsed,
cached,
addressed, or committed in the wrong representation; and
- preservation of accumulated state or an existing artifact across
actions,
especially when an edit, retry, revisit, or structural change can
destroy
or duplicate prior work.
Do not force a category when the TASK/ENVIRONMENT has no such
counterpart.
Do not put a guessed fix into "name". Describe the neutral decision or
boundary
and use "transfer_goal" and "query" to seek established external
mechanisms.

For the ambiguous-choice concern, prefer a named rule-based,
constrained, or
multi-criteria decision mechanism when the environment supplies no
learned
values or outcome feedback. A broad field label such as "reinforcement
learning", "decision theory", or "prompt engineering" alone is not a
usable
mechanism. For the representation-boundary concern, seek knowledge
about both
```

RASI **System prompt**

```
the producer representation and the consumer/checker representation:
querying
only how the checker reads an output misses failures introduced while
writing,
serializing, materializing, or committing it.

Do not name the benchmark, the dataset, or the harness. 6 to 14 words
per query, no underscores, no sentence.

Return ONLY:
{"procedure": [{"name": "...", "transfer_goal": "...", "query":
"..."}], "budget": [{"what": "...", "runout": "...", "query": "..."}]}
```

Cross-Harness Adaptation.

RASI **System prompt**

```
You are given a TASK, the ENVIRONMENT it runs in, one CONCERN of the
work, and SECTIONS of documents in which other people deal with that
same concern in their own settings.

Write what those sections say about it, in the terms of this TASK and
this ENVIRONMENT.

- Do not judge whether the sections are relevant or whether their
setting resembles this one. Translate.
- Use only this task's and this environment's own objects, commands
and units. No noun from a section's own domain may survive.
- Where a section's procedure has no counterpart here, leave it out
rather than inventing one.
- Write about THIS CONCERN ONLY. The work has other parts and they are
being written about separately; do not restate them, do not walk the
reader from the beginning of the task to the end, and do not describe
what happens after this concern is dealt with. If the sections say
nothing about this concern beyond what the CONCERN line already
states, say only what they do add, and say less rather than padding.
- Write it as instructions for this concern here. 3 to 6 sentences,
prose, no headings.
- Do not state that a specific function, method, parameter, mode, or
command behaves a particular way unless that exact behavior is already
established in the ENVIRONMENT text above, or a section states it
about a tool this environment's own procedure already names. A section
about a same-named parameter or tool used for a different purpose than
the one at hand (a flag documented for reading, when this concern is
about writing) is not evidence for a claim about this purpose -- leave
that claim out rather than extending the section's fact to a setting
it did not describe.
- If, after applying the rule above, you are not certain a specific
technical detail is correct, describe the goal or constraint in plain
terms instead of naming an unverified function, parameter, or mode. A
shorter, correct answer is better than a longer, specific-sounding one
that risks being wrong.

Return only that text.
```

Skill Initialization.

RASI — **System prompt**

```
You are writing a skill document for an agent that carries out this
TASK, repeatedly, in this ENVIRONMENT.

You are given: the steps the work takes and what is known about
carrying out each; and the finite things this work is done under, what
happens when each runs out, and what is known about working within
them.

Write the skill. It is a document the agent reads before acting, not a
program and not a transcript: never emit code as the skill itself. Lay
out the procedure step by step in this environment's own terms, and
carry the budget knowledge into the steps where it bites rather than
leaving it in a section of its own.

For each finite thing, the skill must state what running out actually
costs, in the outcome's own terms, at the step where the agent would
be deciding whether to spend more of it. An agent that knows only the
limit and not the cost of reaching it cannot weigh going on against
settling for what it has, and that weighing is the decision the limit
exists to force. Where the material says an incomplete result scores
something and an abandoned one scores nothing, that comparison belongs
in the step where the agent would otherwise keep going.

Keep the knowledge you were given; add nothing that is not in it; do
not mention documents or sources.

Return only the skill document.
```

A.9.2 RASU

Chunk-Level Rollout Diagnosis.

RASU — **System prompt**

```
You analyze execution records to identify reusable causal corrections.
Apply exactly the same policy to every task domain.

You receive an immutable TASK, immutable HARNESS / ENVIRONMENT,
CURRENT_SKILL, and a small mixed batch of execution RECORDS. Each
record contains its input, generated output or trajectory, and
evaluator feedback. Do not modify the task or harness. Do not write a
skill, patch, lesson, or search query in this step.

Analyze every record, including correct or positive-reward records.
For an incorrect record, begin at the evaluator-visible final result
and trace backward through the generated output or trajectory until
you find the earliest decision that made the failure likely. The cause
must identify something that could change a future action or output;
statements such as "the answer was wrong", "the wrong item was
chosen", "the task was not completed", or "the budget was exhausted"
merely restate the outcome and are not causes.
```

RASU **System prompt**

```
When an expected answer is supplied, first compare it with the
generated final output. Determine whether the decisive divergence is
representation, precision, interpretation, source selection,
calculation, tool semantics, or another observable mechanism, then
trace backward to the step that created it. When no expected answer is
supplied, compare the final state and actions with the task
requirements, evaluator feedback, observed reward, and successful or
higher-reward records. Identify whether the decisive divergence is
entity identity, missing domain knowledge, state tracking, action
validity, search policy, operation semantics, budget allocation,
completion policy, or another observable mechanism.

Do not stop at a generic process criticism. If repeated search failed,
identify what reusable knowledge or control rule could reduce the
search space. If a tool-produced artifact was scored unexpectedly,
identify the evaluator-visible representation or tool behavior
responsible. If an imperfect completed action receives positive reward
while non-completion receives none, preserve completion as a fallback
rather than converting preferences into unconditional rejection gates.

For correct or positive-reward records, identify only behavior that
should be preserved when correcting nearby failures. Do not infer that
every incidental action caused success.

All conclusions must be grounded in the supplied record, task,
harness, or current skill. Do not invent hidden state or memorize
record-specific answers, identifiers, filenames, products,
coordinates, dates, thresholds, or numeric constants as reusable
policy. A proposed repair must be executable through the existing
interface.

Return JSON only with exactly this schema:
{
"record_analyses": [
{
"record_id": "exact supplied record id",
"outcome": "Correct" or "Incorrect",
"final_evaluator_state": "compact statement of the final output/state
and feedback",
"backward_trace": ["last relevant event", "preceding causal event",
"earliest decisive divergence"],
"decisive_cause": "earliest actionable causal mechanism, or null for a
correct record",
"counterfactual_change": "the concrete changed decision/output that
would address the cause, or null",
"external_knowledge_need": "what reusable external procedure, tool
fact, domain knowledge, or control policy could help, or null",
"preserve_constraint": "successful or positive-reward behavior that a
later update must preserve, or null"
}
]
}

Include every supplied record exactly once. Do not output markdown
fences or additional keys.
```

Minibatch Textual-Gradient and Query Merge.

RASU — **System prompt**

```
You consolidate compact execution diagnoses into reusable
textual-gradient and retrieval-query patterns. Apply exactly the same
policy to every task domain.

You receive an immutable TASK, immutable HARNESS / ENVIRONMENT,
CURRENT_SKILL, and analyses covering one complete training minibatch.
Do not modify the task or harness. The final groups will guide
retrieval and a later skill update, so preserve causal precision and
successful behavior.

Group incorrect records only when the same generalized correction can
change the decisive error in every member. Surface similarity, shared
final failure, or use of the same broad tool is insufficient. A group
must contain at least two distinct incorrect record_ids. If an
incorrect record has no second record with the same remedy, place it
in excluded_singletons instead of weakening another group.

A record may belong to multiple groups only when it independently
exhibits multiple distinct causal mechanisms. Merge groups whose
corrections become equivalent after record-specific details are
removed. Do not create formula, field, product, location, or
task-instance catalogs.

Separate hard contracts from soft preferences. A hard interface or
evaluator violation may justify rejection, but failure to satisfy one
preference, ranking factor, or uncertain property must be represented
as a penalty or ranking signal when a completed imperfect action
receives positive credit. Always preserve an executable completion
fallback and state when it takes precedence over further exploration
or rejection.

If CURRENT_SKILL already contains a generic control rule and the
failures show that it was followed but was insufficient, do not
restate that control rule as the retrieval need. Identify the missing
reusable mechanism: a domain prior, vocabulary or identity
distinction, affordance, tool/library behavior, representation
convention, or decision criterion. Search for that missing knowledge
while keeping the query domain-neutral in wording.

For search and manipulation tasks, distinguish a control policy from
missing world knowledge. If the current skill already says how to
avoid redundant search but the records show the agent still lacks
information to choose promising candidates, search for reusable
entity-to-location, entity-to-affordance, terminology, or
state-transition knowledge rather than another generic search policy.
For any property, price, dimension, or preference, call it a hard
constraint only when TASK, HARNESS, or evaluator feedback establishes
that violating it makes the action invalid or receives no credit;
otherwise represent it as a ranking factor and preserve the completion
fallback.

Before writing each group:

1. State the observable failure signature shared by all members.
2. State the earliest causal mechanism, not the final symptom.
3. State the counterfactual behavior that would change the generated
action, artifact, or final output.
```

RASU **System prompt**

```
4. Preserve constraints demonstrated by correct or positive-reward
records. In particular, do not turn a preference into an unconditional
rejection or stopping rule when bounded completion receives positive
reward.
5. Decide what external knowledge could make the correction more
concrete. Always produce one retrieval query for the reusable
mechanism; a later retargeting stage may return an empty lesson when
the corpus adds nothing.

The textual_gradient must be a compact reusable instruction with an
observable trigger, a concrete action or decision, and a verification,
precedence, or fallback. It must be executable through the existing
interface. It may strengthen an existing instruction when the
diagnoses show that the current wording was ignored, but it must not
casually reverse valid behavior in CURRENT_SKILL.

The retrieval_need describes knowledge that a useful external section
would contain, not a restatement of the failure. The query must be a
concise search-engine query using vocabulary likely to occur in
procedural skill documents: name the underlying mechanism,
tool/library behavior, discipline, and desired procedure, with useful
synonyms where appropriate. Do not include record IDs, sampled
answers, task-instance constants, or prose such as "how do I solve
this record".

TEXTUAL GRADIENT AND QUERY HAVE DIFFERENT JOBS

- First formulate the textual_gradient: the reusable behavior change
supported by the failed records.
- Then independently inspect CURRENT_SKILL, the supporting diagnoses,
TASK, and HARNESS to identify knowledge that is missing despite that
behavior change. State it in knowledge_gap.
- Do not form the query by shortening, paraphrasing, or
keyword-extracting the textual_gradient. The query must target an
external section that could add a mechanism, factual prior,
affordance, terminology distinction, tool behavior, representation
contract, decision criterion, verification method, or fallback that
CURRENT_SKILL does not already provide.
- If no deep external fact is needed, query for a concrete
verification or execution procedure rather than restating the desired
behavior.
- Classify the affected condition as hard, soft, conditional, or
unknown from the supplied evidence. A requested property is not
automatically hard. Hard means the task, interface, or feedback
establishes invalidity or zero credit; otherwise preserve graded
ranking and timely completion.

Return JSON only with exactly this schema:
{
"groups": [
{
"id": "g1",
"record_ids": ["at least two exact incorrect record ids"],
"failure_signature": "shared observable signature",
"root_cause": "shared earliest causal mechanism",
"counterfactual_change": "how future behavior/output changes",
"knowledge_gap": "external knowledge missing beyond the behavioral
correction",
"hardness": "hard" or "soft" or "conditional" or "unknown",
"hardness_evidence": "evidence supporting that classification",
"textual_gradient": "trigger -> action ->
verification/precedence/fallback",
```

RASU — **System prompt**

```
"success_constraints": ["behavior from correct or positive-reward
records that must remain valid"],
"current_skill_gap": "what is absent, ambiguous, weak, or
operationally ineffective in CURRENT_SKILL",
"retrieval_need": "knowledge a useful external section should
contain",
"query": "one corpus-oriented search query"
}
],
"excluded_singletons": [
{
"record_id": "incorrect record with no matching remedy",
"decisive_cause": "compact cause",
"counterfactual_change": "compact repair",
"reason_excluded": "no second record supports the same generalized
correction"
}
]
}

Every incorrect record must appear in at least one group or in
excluded_singletons. Before returning JSON, compare the union of all
group record_ids and excluded_singletons against the supplied
incorrect IDs; put any unsupported ID into excluded_singletons instead
of omitting it. Correct records must not appear in record_ids. Use
empty arrays when appropriate. Do not output markdown fences or
additional keys.
```

Cross-Harness Adaptation.

RASU — **System prompt**

```
You are given a TASK, the ENVIRONMENT it runs in, one CONCERN of the
work, and SECTIONS of documents in which other people deal with that
same concern in their own settings.

Write what those sections say about it, in the terms of this TASK and
this ENVIRONMENT.

- Do not judge whether the sections are relevant or whether their
setting resembles this one. Translate.
- Use only this task's and this environment's own objects, commands
and units. No noun from a section's own domain may survive.
- Where a section's procedure has no counterpart here, leave it out
rather than inventing one.
- Write about THIS CONCERN ONLY. The work has other parts and they are
being written about separately; do not restate them, do not walk the
reader from the beginning of the task to the end, and do not describe
what happens after this concern is dealt with. If the sections say
nothing about this concern beyond what the CONCERN line already
states, say only what they do add, and say less rather than padding.
- Write it as instructions for this concern here. 3 to 6 sentences,
prose, no headings.
- Do not state that a specific function, method, parameter, mode, or
command behaves a particular way unless that exact behavior is already
established in the ENVIRONMENT text above, or a section states it
about a tool this environment's own procedure already names. A section
about a same-named parameter or tool used for a different purpose than
the one at hand is not evidence for a claim about this purpose; leave
that claim out.
```

RASU **System prompt**

```
- If, after applying the rule above, you are not certain a specific
technical detail is correct, describe the goal or constraint in plain
terms instead of naming an unverified function, parameter, or mode. A
shorter, correct answer is better than a longer, specific-sounding one
that risks being wrong.

You also receive CURRENT_SKILL, SUPPORT_EVIDENCE, SUCCESS_CONSTRAINTS,
and a HARDNESS classification. Use them to keep the translation
faithful to the observed target behavior:

- The textual gradient and hardness classification are hypotheses. The
task, environment, feedback, rewards, and supporting diagnoses take
precedence.
- Retrieved knowledge is useful in two cases: it supplies a missing
mechanism, or it independently supports a mechanism that CURRENT_SKILL
already mentions but the failed records did not execute reliably.
- Never discard a mechanism merely because CURRENT_SKILL already
contains it. Repeated failure to follow an existing rule means its
current wording is not sufficiently salient, operational, or
enforceable. Translate the retrieved support into a compact
reinforcement lesson.
- For such reinforcement, state an observable trigger, the mandatory
action, and an immediate pre-commit verification. Make the rule
decisive enough to override the shortcut seen in SUPPORT_EVIDENCE,
without deleting compatible successful behavior.
- Do not pad the lesson by restating unrelated parts of CURRENT_SKILL.
Reinforce only the rule implicated by this concern and its
verification step.
- Never turn a soft, graded, or unknown factor into an unconditional
rejection, stopping, or no-completion rule. Express it as ranking or
normal-path guidance and preserve a timely executable completion
fallback.
- Use an unconditional gate only when the supplied evidence
establishes a hard interface or zero-credit contract.
- Preserve successful and positive-reward behavior in
SUCCESS_CONSTRAINTS. When normal behavior and fallback differ, state
their observable condition and precedence.
- Return NO_ACTIONABLE_LESSON only when the retrieved sections neither
add nor corroborate any mechanism relevant to the observed concern.
Duplication with CURRENT_SKILL is not a reason to return it.

Return only the lesson text or NO_ACTIONABLE_LESSON.
```

Skill Update.

RASU **System prompt**

```
You update a reusable assistant skill from execution evidence. Use
exactly the same policy for every task domain.
```

RASU **System prompt**

```
You receive CURRENT_SKILL, RECORDS, and ADDITIONAL_EVIDENCE.
CURRENT_SKILL is the authoritative editing base. RECORDS contain
executions and evaluator feedback; each entry carries its record_id
and outcome, and those ids are the ones to cite as support.
ADDITIONAL_EVIDENCE contains, per evidence group, a textual gradient
(a record-grounded proposed correction) and a lesson: what other
people's documents say about the same concern, translated into this
task's terms. A lesson may overlap with the gradient or add a
mechanism; an empty lesson means the documents added nothing, and the
gradient alone is still full-strength evidence. Return local edit
operations, not a rewritten skill; a deterministic program will apply
them to CURRENT_SKILL.

EVIDENCE PRIORITY

- Preserve the immutable task interface, evaluator-visible output
contract, and behavior supported by successful or positive-reward
records.
- Evaluator feedback and observed rewards are direct evidence. If they
conclusively contradict a factual scoring, completion, or
representation claim in CURRENT_SKILL, correct the smallest
conflicting span rather than preserving the false claim.
- A textual gradient is a record-grounded proposed correction. A
retrieved lesson is optional supporting knowledge. Use either only
when compatible with the records and the available interface.
- Do not derive unrelated new behavior from RECORDS. Do not memorize
sampled answers, identifiers, paths, coordinates, products, or other
instance-specific constants.

RECURRENT-INSTRUCTION POLICY

- When a textual gradient or lesson says the same thing as text
already present in CURRENT_SKILL, that is not a duplicate to discard.
The agent was expected to follow that text and did not: the cited
failed records show that the existing statement was ignored because it
was conditional, weakly prioritized, buried, ambiguous, or lacked an
executable trigger or check.
- Treat every such repetition as an adherence failure. Do not append a
second copy, and do not emit no edit merely because the rule already
exists. Replace the smallest existing span with one canonical rule
written so that the agent can actually follow it: a mandatory
pre-action or pre-output step at the point where the violation
occurred, with an observable trigger, the required decision, its
precedence over competing instructions, and an evaluator-visible
check.
- A strengthened replacement must retain every still-valid scope item,
example category, budget, exception, output requirement, and factual
statement about the evaluator, tools, or environment from the old
text. Strengthening may add a trigger or check; it must not silently
narrow the rule by dropping valid enumerated cases or facts.

CONFLICT POLICY

- Never leave incompatible unconditional instructions for the same
observable situation.
- If CURRENT_SKILL is directly disproved by evaluator feedback or
observed reward, replace the smallest false span with the
evidence-supported rule.
```

RASU **System prompt**

```
- When correcting a disproved claim, scan all of CURRENT_SKILL for
other statements that assert the same claim and correct every
contradictory occurrence. Do not leave an earlier or later sentence
that reverses the new policy. List every such other span in that
replacement's same_claim_spans and emit a replacement for each of
them.
- If two behaviors are valid under different observable conditions,
replace the relevant existing span with explicit branches and
precedence: state when each behavior applies and which fallback wins.
- Preserve bounded completion and terminal-action behavior. Do not
turn a ranking preference, partial-credit signal, or best-effort check
into an all-or-nothing rejection rule unless the observed rewards
establish that violation makes completion worthless.
- When positive reward is observed for an imperfect completed action,
preserve that completion fallback and distinguish it from the normal
full-success policy.
- Distinguish an unsupported invented fallback from an output
representation demonstrated by the task artifact. Preserve
demonstrated representations such as blank, null, zero, false, or an
existing error marker; prohibit a fallback only when neither the task
nor the artifact supports it.
- If a conflict cannot be resolved from the records and supplied
evidence, retain CURRENT_SKILL and omit the proposed edit.

EDIT POLICY

- Prefer a replacement when strengthening an existing but ignored
rule, resolving a conflict, or correcting a false claim. Quote the
smallest exact contiguous old_text that occurs once. The new_text must
be one canonical rule and retain every still-valid part of old_text.
- Use an insertion only for genuinely missing, nonconflicting
behavior. Place it beside the most relevant existing instruction using
a short exact unique anchor copied from CURRENT_SKILL.
- Place a rule at the earliest section whose scope matches it. A rule
that applies before every action, every write, or every final response
must not be buried under a later specialized subsection.
- Every edit must be compact and actionable: observable trigger,
mandatory action or decision, and only the essential invariant,
precedence, fallback, or final check.
- Merge evidence that prescribes the same behavior. Do not emit one
edit per evidence item, duplicate safeguards, rationale, tutorials,
evidence summaries, or example catalogs.
- A list of per-procedure, per-formula, per-field, or per-task recipes
derived from individual records is a catalog and is prohibited even
when phrased as mandatory checks; express the shared mechanism once,
at the level of the trigger and decision common to those records.
- After drafting all edits, verify that no behavior appears in more
than one edit or in both an edit and an untouched span. If a
replacement's new_text already states a behavior, do not also insert
it elsewhere; when two evidence items prescribe the same behavior,
express it once and cite both record sets. The edited document must
state each rule exactly once.
- Do not rewrite unrelated sections. Keep headings, ordering,
terminology, budgets, allowed actions, and output syntax unchanged
unless the supplied evidence directly requires a local correction.
- Do not split a sentence, list introduction, or numbered step with an
insertion. Anchor on a complete sentence or heading and keep the
resulting document grammatically continuous.
- Anchors must remain unchanged by replacements. Replacements must not
overlap. Every edit requires supporting record_ids.

UNIVERSAL CONDITION, PRESERVATION, AND ACTIONABILITY RULES
```

RASU **System prompt**

```
- Apply this same policy regardless of task domain. Do not infer
domain-specific behavior that is unsupported by CURRENT_SKILL,
RECORDS, or ADDITIONAL_EVIDENCE.
- Treat textual gradients and lessons as proposed corrections, not as
proof. They may supply missing behavior, but they cannot override a
valid existing instruction unless the cited execution evidence
conclusively establishes the contradiction.
- Distinguish (a) a hard interface or evaluator contract, (b) a normal
procedure, and (c) a preference or fallback. Never turn a preference,
ranking heuristic, or fallback into an unconditional rejection gate.
- Preserve successful and positive-reward behavior. If an imperfect
completed attempt receives positive credit, preserve a bounded
completion fallback: use the normal policy when possible, otherwise
take the best supported executable terminal action early enough to
finish. Do not invent a fallback when the records provide no support
for it.
- When normal behavior and fallback behavior differ, state an
observable condition and precedence. Never leave two unconditional
rules for the same situation.
- Preserve evaluator-visible representations and types demonstrated by
the task artifacts. Do not silently coerce values, blanks, nulls,
dates, strings, formulas, actions, or final answers unless the
evidence establishes that conversion.
- Every new or strengthened instruction must contain an observable
trigger, one concrete action or decision, and the essential invariant,
verification, or fallback that determines what happens next. Replace
vague reminders such as "verify", "be careful", "consider", or "use
the correct value" with the object and procedure to execute.
- Put a rule at the earliest scope where its trigger is visible. A
rule that applies before every action, write, search, or final
response must not be buried in a later specialized subsection.
- Merge overlapping evidence into one canonical conditional rule. Do
not duplicate a behavior in multiple edits or leave it both in a
replacement and in an untouched span.
- Before returning the patch, verify that every changed span is
necessary, supported by cited record_ids, and strictly local. Preserve
all unrelated headings, procedures, exceptions, budgets, allowed
actions, and output syntax byte-for-byte.

EXACT PATCH ANCHOR RULE

- Every replacement old_text and insertion anchor must be copied
verbatim as one exact contiguous substring of CURRENT_SKILL, including
capitalization, punctuation, Markdown markers, and whitespace.
- Prefer the smallest complete sentence, list item, or heading that
occurs exactly once. Never reconstruct, normalize, truncate, or guess
an anchor.
- If no exact unique local span can safely express an edit, omit that
edit instead of producing an unappliable patch.

OBSERVED-REWARD OVERRIDE

- Hardness words in a textual gradient or lesson are hypotheses, not
evaluator evidence. Re-derive constraint hardness from RECORDS before
writing each edit.
- A completed terminal action with positive soft reward proves that
its imperfections do not make completion worthless. A requirement
violated by such a record may remain mandatory for a perfect result,
but it must not become an unconditional rejection, no-purchase,
no-submission, or keep-searching gate.
```

RASU **System prompt**

```
- In that case, write one conditional rule with explicit precedence:
normally optimize or verify the requirement; if no fully compliant
result is found before the bounded completion threshold, execute the
strongest compatible terminal fallback in time. Never sacrifice
positive completed credit while pursuing perfection.
- A requirement may prohibit completion only when the records or
immutable interface establish that violating it makes the terminal
action invalid or yields zero total credit, not merely loss of one
reward component.

ROBUST EXACT-ANCHOR OUTPUT

- Do not use a multi-line Markdown list or a paragraph containing such
a list as replacement old_text or as an insertion anchor. Blank lines
and list formatting make that span unsafe.
- To strengthen behavior around a list, replace one exact unique lead
sentence, one exact list item, or insert beside an exact unique
heading or complete sentence. Copy it byte-for-byte from
CURRENT_SKILL.

MANDATORY REWARD-CONTRADICTION PASS

- Before drafting edits, inspect all of CURRENT_SKILL for absolute
scoring or completion claims, including claims that a missing
requirement makes the total score zero or that a terminal action may
occur only after every preference is satisfied.
- Compare every such claim with every completed RECORD that received
positive reward. If a positive-reward record violates any requirement
named by an all-or-nothing claim, the claim is conclusively false as a
statement about total reward or whether completion is worthwhile.
- Resolve that contradiction with a replacement of the smallest exact
existing span. An insertion beside the false claim is insufficient,
and preserving the false wording is prohibited. Cite the
positive-reward record_ids in the replacement.
- The canonical replacement must distinguish the normal perfect-result
policy from the bounded terminal fallback: optimize all requirements
normally, but when a fully compliant result is unavailable before the
completion threshold, execute the strongest compatible terminal action
in time to retain positive credit.
- Do not introduce `only if`, `never complete`, `do not buy`, `do not
submit`, or an equivalent terminal gate for a requirement that
positive completed records show is graded. Such a requirement may
remain a strict condition for a perfect result, but not for terminal
completion itself.

CANONICAL LOCAL RULE PASS

- After drafting replacements, inspect their immediately adjacent
untouched sentences. If an adjacent sentence would restate the same
trigger, action, or verification, replace the smallest exact
contiguous prose span that can express one canonical rule instead of
leaving semantic duplication.
- Preserve every compatible fact, exception, representation, and
fallback from that span. Do not absorb a Markdown list into old_text.

DOCUMENT-STRUCTURE PRESERVATION

- When replacement old_text is a lead sentence whose next nonblank
line in CURRENT_SKILL is a Markdown bullet or numbered list, the
replacement new_text must remain a grammatical introduction to that
untouched list and must end with a colon. Never leave an orphaned
list.
```

RASU **System prompt**

```
- Do not move or copy a behavior into an earlier section while leaving
an existing statement of that behavior elsewhere in CURRENT_SKILL.
Strengthen the existing canonical statement at its current location;
use a separate edit for a genuinely different global trigger.
- Before returning JSON, check every replacement boundary against the
unchanged text immediately before and after it. The resulting prose,
headings, and lists must remain grammatically continuous, and each
behavioral rule must occur in one canonical location only.

GRADED-EVIDENCE PRESERVATION

- Do not omit an evidence group merely because its proposed hardness
is contradicted by positive reward. If the group identifies a reusable
trigger, check, ranking factor, or verification mechanism, retain that
mechanism at the evidence-supported strength.
- For a graded requirement, write it as the normal path plus a bounded
fallback: verify and prefer a compliant terminal result while
sufficient budget remains; if none is available before the completion
threshold, take the strongest compatible terminal action in time. The
fallback must not claim that the unmet factor was satisfied.
- Omit the group only when it contains no reusable actionable
mechanism after correcting its hardness, or when all of its behavior
is already fully represented by another edit in this same patch. In
the latter case, name that edit in the disposition reason.
- same_claim_spans is only for another span asserting the same false
factual claim. Do not list a true statement about failure to take any
terminal action merely because both statements mention zero reward.

TERMINAL FALLBACK SEMANTICS

- A fallback for a graded requirement must remain executable even when
that requirement cannot be satisfied. Do not describe only a compliant
fallback, because that merely repeats the normal policy.
- State the precedence explicitly: seek and verify a compliant result
during normal exploration; when the bounded completion threshold is
reached without one, take the strongest otherwise-compatible terminal
action in time, while truthfully retaining that the graded factor is
unmet.
- Never require indefinite search or no terminal action solely because
a graded price, attribute, option, representation, or preference
remains unsatisfied.

VALID-FACT PRESERVATION

- Preserve every evaluator fact, scoring heuristic, allowed
alternative, example category, and compatibility rule in CURRENT_SKILL
unless RECORDS conclusively disprove that exact fact. A different
retrieved mechanism is not evidence that an existing fact is false.
- When strengthening one clause of a multi-clause span, retain all
unrelated valid clauses byte-faithfully in new_text. Do not drop a
ranking or scoring fact merely because the new rule has higher
precedence.
- Every dropped_from_old_text entry must be a JSON object with exactly
`text` and `reason`; never emit bare strings or a standalone reason
field inside that array.

Return JSON only with exactly this schema:
{
"evidence_dispositions": [
{
"group_id": "evidence group id",
"disposition": "replace" or "insert" or "omit",
```

RASU **System prompt**

```
"reason": "why this evidence was applied this way or omitted"
}
],
"insertions": [
{
"anchor": "exact unique text copied from CURRENT_SKILL",
"position": "before" or "after",
"instruction": "compact actionable text",
"record_ids": ["supporting record id"]
}
],
"replacements": [
{
"old_text": "smallest exact conflicting or under-enforced span copied
from CURRENT_SKILL",
"new_text": "single canonical replacement rule",
"dropped_from_old_text": [
{
"text": "sentence or clause of old_text that new_text does not
retain",
"reason": "why it is no longer valid"
}
],
"same_claim_spans": ["exact other span of CURRENT_SKILL asserting the
claim this replacement corrects; each must also be replaced"],
"record_ids": ["supporting record id"],
"reason": "specific contradiction or adherence failure established by
the records"
}
]
}

Every evidence item must appear in evidence_dispositions.
dropped_from_old_text must be an empty list when new_text retains
everything in old_text. same_claim_spans must be an empty list unless
the replacement corrects a factual claim that other spans also assert.
Use empty arrays when no safe edit exists. Do not output markdown
fences, a complete skill, analysis, or additional keys.
```

A.10 Initial Skills

We reproduce the initial skills s_0 generated by RASI for each benchmark and target model. These skills are constructed before any task rollout and serve as the starting point for subsequent skill optimization. Since the same model is used as the target and optimizer model unless otherwise specified, we report the resulting skills separately for GPT-5.6-Luna and Qwen-3.5-9B. The skill bodies are reproduced below verbatim apart from formatting.

A.10.1 GPT-5.6-Luna

OfficeQA.

GPT-5.6-Luna **Initial skill s_0**

```
# Skill: Answer Historical Treasury Bulletin Questions

## 1. Locate the cited bulletin pages

1. Read the question, its citation strings, page numbers, requested
quantity, calculation method, rounding, and required output format.
```

GPT-5.6-Luna **Initial skill s_0**

```
2. Inspect the supplied candidate file paths first. Select the issue
matching the citation, then use `read` to locate the cited `## Page
<n>` boundary.
3. Confirm the page number, table or figure title, labels, units, and
issue details before extracting values.
4. If the candidates do not contain the required issue or page, search
the corpus with `glob` and targeted `grep` searches using citation
text, issue identifiers, distinctive table wording, and subject terms.
Then read the matching page directly.
5. If the 20 supplied candidate paths are insufficient, broaden the
corpus search rather than assuming the correct issue is absent.
Retrieve only the document content needed for the answer.

Plan these calls so the answer is submitted within the 24-round
budget. Every tool call and the final response consumes a round;
reaching round 24 without submitting an answer fails the episode, so
avoid redundant searches and leave a round for the final answer.

## 2. Read the relevant tables and figures

1. Read the cited page and the surrounding lines containing the
requested table or figure.
2. Use the page markers to ensure that values are taken from the
specified page, not an adjacent page.
3. Carefully preserve row labels, column headings, units, signs,
dates, and distinctions between related figures.
4. Treat rendered tables as Markdown tables. For scanned or
image-based material, interpret the extracted text cautiously and
verify the alignment of labels and values.
5. Extract only the figures needed for the calculation.

Each `read` response is limited to 4000 characters. If content is
truncated, make narrower reads around the relevant table or use `grep`
to locate its heading or distinctive rows, then reread the needed
range. If the figures remain unavailable, do not invent or infer them
from unrelated text; retain only what was actually found and continue
searching the cited issue or page.

## 3. Identify and verify source values

1. Record each source value with its label, unit, date or observation
position, and page location.
2. Ensure every value used comes from the specified bulletin page or
the matching page found in the corpus.
3. Check that the values correspond to the question's requested series
and not a similarly named series, subtotal, alternative maturity, or
different unit.
4. If a required value is missing from the supplied page text, search
the corpus for the cited issue and page, then extract it from the
matching page.
5. Use only values appearing in the bulletin material; do not
substitute outside data.

A `grep` search returns at most 20 matching lines. If more matches
exist, the omitted lines are not evidence that the needed value is
absent. Narrow the search by page number, table title, date, or
distinctive wording, or inspect the relevant file with targeted `read`
calls.

## 4. Apply the requested calculation exactly
```

GPT-5.6-Luna **Initial skill s_0**

```
1. Follow the question's stated arithmetic or the exact standard
definition of any named statistical or financial procedure.
2. For moving-window calculations, use the stated window length and
calculate the average or other operation for each required set of
observations.
3. For percentages, ratios, growth rates, transformations, risk
measures, and dispersion measures, use the conventional definition of
the named measure, including its specified denominator, scale, and
treatment of observations.
4. Keep units consistent and preserve signs and ordering.
5. Apply only the rounding, precision, tolerance, and formatting
requested by the question.
6. Recheck the calculation against the extracted labels and values
before answering.

Do not spend additional rounds on searches once the required values
and method are sufficiently established unless a missing or
conflicting figure prevents a reliable result. Continuing until the
24-round limit causes the episode to fail without an answer; an
incomplete result submitted before exhaustion can still be scored,
whereas an abandoned episode scores nothing.

## 5. Format and submit the answer

1. Return the result exactly in the question's specified format.
2. For integer or explicitly exact outputs, use the exact requested
representation.
3. For numeric results requiring precision, preserve the required
decimal places or rounding.
4. Return only the requested value or values inside the required
wrapper:

`<answer>...</answer>`

5. Do not add explanations, units, citations, punctuation, or
alternative representations unless the question explicitly requests
them.
6. Submit before the 24-round budget is exhausted.
```

SpreadsheetBench.

GPT-5.6-Luna **Initial skill s_0**

```
# Skill: Transform an Existing Workbook into a Verified Answer
Workbook

## 1. Parse the request and identify the edit surface

Read the natural-language instruction and determine:

- The requested operation: lookup, conditional sum, date/time logic,
consolidation, filtering, deletion, extraction, text transformation,
or cumulative/grouped calculation.
- The source sheets and fields.
- The criteria or predicate fields.
- The returned or computed fields.
- The target answer schema.
- The exact answer-range anchor and dimensions.

Treat any preview as incomplete. Inspect the actual workbook for:
```

GPT-5.6-Luna **Initial skill s_0**

```
- Sheet names and dimensions.
- Header rows, repeated or multi-row headers, blank spacer cells, and
merged ranges.
- Representative values, formulas, dates, times, text, and missing
cells.
- Records whose entity continues across multiple rows.
- Labels that do not align with preview column letters.
- Columns beyond `Z`, checking coordinate conversions explicitly.

Load the workbook with:

```python
openpyxl.load_workbook(INPUT_PATH, data_only=False)
```

Use a separate `data_only=True` view only for inspecting cached
formula results when necessary. Do not save the values-only view.

Preserve the input workbook as the basis of the output. Unless the
instruction explicitly requires a structural change, only the
requested answer range may be modified.

2. Recover the source-to-target mapping

Build the mapping from workbook labels and observed layout rather than
assuming fixed preview coordinates. Identify:

- The source predicate or criteria field.
- The source return field or fields.
- The output columns and their order.
- The target starting cell and target extent.
- Any relationship between source rows and target rows.
- Whether headers occupy one row or multiple rows.
- Whether labels need selective fill-forward.
- Whether records continue over multiple rows.

Keep values semantically distinct:

- Missing cells are not automatically the text `"None"`.
- Empty strings, zero, and `False` may have different meanings.
- Dates and times must be compared as dates and times when
appropriate.
- Text case and delimiters matter when the instruction or observed
values makes them relevant.
- Do not split or normalize free text without evidence that the
requested operation requires it.

Use one-based Excel coordinates at the worksheet boundary. If using
zero-based dataframe indices internally, convert carefully and verify
the conversion before writing.

3. Define calculation semantics before writing

Translate the instruction into explicit Python-side logic. Decide from
the request and workbook values:

- Which rows qualify.
- How multiple criteria combine.
- How missing and empty values behave.
- Whether case-sensitive comparison is required.
- Whether zero and `False` count as present values.
- How dates, times, and ranges are compared.
- Whether repeated entities should be grouped.
```

GPT-5.6-Luna **Initial skill s_0**

```
- Whether row order must be preserved.
- Whether delimiters belong to extracted text.
- Whether the output should contain literal values or explicitly
requested formulas.

Do not apply blanket conversions that erase distinctions among
missing, empty, zero, false, numeric, date, and text values.

## 4. Compute in source order

Preserve the workbook's existing row order unless the instruction
explicitly specifies another ordering.

For cumulative or entity-based logic:

- Maintain state in encounter order.
- Use order-preserving grouping such as `groupby(sort=False)` if
grouping is needed.
- Do not reorder entities merely because a grouping operation permits
it.
- If the instruction explicitly requires sorting, sort by every
specified key and direction before computing.

Compute only the requested answer values. Avoid unnecessary
full-workbook transformations or repeated scans when a bounded
calculation is sufficient.

## 5. Apply only authorized edits

Edit the existing workbook rather than creating a replacement
workbook.

Normally:

- Change only the named answer cells.
- Preserve all sheets and existing content.
- Preserve formulas, formatting, comments, merged ranges, dimensions,
and unrelated cells.
- Do not rebuild a guessed rectangle.
- Do not compact a sheet or alter `max_row` or `max_column` as a
substitute for editing.

If the instruction explicitly requires deleting rows or columns:

- Identify the actual target indices.
- Delete them in reverse index order so later targets do not shift
incorrectly.
- Preserve all remaining workbook content.

Use verified worksheet coordinates, especially for columns beyond `Z`
and for answer ranges whose row offsets are not obvious.

## 6. Choose the serialized answer representation

Use the formula-preserving workbook as the transformation source.

Write literal computed values when the checker needs serialized final
results and no recalculation engine is available. Do not save a
workbook opened with `data_only=True`, because that can replace
formulas with cached constants.
```

GPT-5.6-Luna **Initial skill s_0**

```
If the instruction explicitly requires formulas, write formulas only
when their representation is appropriate and ensure recalculation is
available. The workbook library does not calculate formulas; an
unevaluated formula may leave the checker seeing a blank or stale
cached result.

The output must contain the intended values in the exact specified
answer cells after serialization, not merely in memory.

## 7. Save and verify the output

Save the transformed workbook to `OUTPUT_PATH`.

Reopen the saved file and verify:

- Every requested answer cell.
- The first and last rows and columns of the answer range.
- Nearby boundary cells.
- Important unrelated formulas and content.
- That sheets, formatting, and other workbook structure remain
present.
- That the serialized representation contains the intended values
rather than stale, blank, or misaddressed content.

A separate `data_only=True` reopen may be used to inspect cached
formula results, but cached values may be absent if the workbook was
not recalculated. Do not treat the existence of the output file as
proof of success.

## 8. Work within the execution budgets

The entire script has 120 seconds for loading, inspection,
computation, saving, and verification. If this limit is reached, the
process is terminated before a usable output can be produced or
validated. Use a direct bounded workflow:

1. Load the existing workbook once for transformation.
2. Inspect only the structure needed to resolve the request.
3. Compute only the requested values.
4. Update only the answer range or explicitly required structural
area.
5. Save once to `OUTPUT_PATH`.
6. Reopen once for targeted verification.

Avoid unnecessary full-workbook passes, repeated loads, expensive
formatting operations, and replacement-workbook construction.

There are at most 30 execution attempts. Each runtime error or retry
consumes an attempt. Before running, check coordinates, imports,
workbook paths, range dimensions, and save logic. If a run fails, make
a targeted correction based on the failure rather than repeating the
same error. Once the attempt budget is exhausted, no further
correction is possible; a missing or invalid output fails the task.

The input workbook's existing artifact is also finite: destructive
rebuilding or broad edits consume its accumulated structure. If that
structure is lost, duplicated, or replaced, the transformed copy is
invalid. Preserve the original workbook and make only the authorized
changes.
```

GPT-5.6-Luna **Initial skill s_0**

```
Finally, the serialized output representation is finite: incorrect
coordinates, formulas, or unwritten values cause the harness to read
wrong, stale, or unevaluated answers even when the in-memory
calculation was correct. Always save, reopen, and verify the exact
cells the harness will consume.
```

ALFWorld.

GPT-5.6-Luna **Initial skill s_0**

```
# Skill: Budgeted Household Object Search and Placement

## 1. Parse the instruction

Read the first observation's sentence beginning **"Your task is
to:"**. Extract:

- the required object type;
- the exact named destination receptacle;
- any required heating, cooling, or cleaning operation;
- whether one object or two instances of the same kind are required.

Preserve the simulator's exact names and numbering. Completion
requires every extracted requirement to be satisfied.

## 2. Build the coverage ledger

Treat every receptacle named in the opening observation as a possible
object location. Contents are initially unknown.

Create and maintain a ledger for every receptacle:

- **unvisited**: contents unknown;
- **settled**: fully observed, with all reported contents recorded.

An observation of a reached and, where necessary, opened receptacle
reveals all its contents. Record every object and its receptacle
immediately. Do not discard this ledger after moving, processing an
object, or placing the first object of a two-object task.

A settled receptacle provides no new search information. Do not
revisit it to look for an object that was not observed there. Revisit
it only to retrieve a remembered object or to perform required
placement.

## 3. Track the budget before every action

The episode allows at most 50 emitted commands. Every response
consumes one step, including an invalid command, redundant revisit,
empty response, or action that changes nothing.

At each turn, calculate the remaining steps from the reported number
already taken. Before spending a step, compare its information or task
progress with the shortest currently feasible completion route.
Reserve enough steps for all remaining:

- retrieval;
- movement;
- heating, cooling, or cleaning;
- destination access;
- placement;
```

GPT-5.6-Luna **Initial skill s_0**

```
- and, when required, the complete continuation for the second object.

A route is feasible when its estimated cost is less than or equal to
the remaining budget. Do not require an extra spare step.

If the budget runs out before all required objects are fully processed
and placed, the episode ends without a winning placement. Finding an
object or completing only part of the task scores the same as failure,
so do not spend further exploration once it threatens a feasible
complete route.

If completion has already become impossible, do not stop, wait, emit
nothing, or terminate early. Continue issuing one currently admissible
action that maximizes the remaining chance of full completion.

## 4. Choose the next search target

Among unvisited receptacles, choose an admissible location using:

- likelihood suggested by the task and observed evidence;
- information gained by inspecting it;
- travel and manipulation cost afterward;
- the remaining budget and cost of full completion.

There is no probability table for hidden locations. Begin with
locations that appear most promising from the task names and current
observations, then broaden to other unvisited receptacles when likely
candidates are exhausted.

Do not restart the search from the beginning. Once a receptacle is
settled, exclude it from exploration unless it contains a remembered
required object or is the named destination.

Before exploring, verify that the remaining budget can still cover the
shortest feasible completion route. Exploration is worthwhile only
when its information value does not eliminate the possibility of
completing every required placement.

## 5. Inspect and update

Use exactly one currently admissible command to reach or open the
selected unvisited receptacle. After the action:

1. read the new observation;
2. record the receptacle as settled;
3. record every reported object and its location;
4. update which required objects remain missing;
5. update the admissible actions and remaining budget.

Do not assume an unreported object is hidden in that receptacle. Its
complete observation establishes that it is absent there.

## 6. Retrieve and process required objects

When a required object is known, use the exact currently admissible
taking command. Do not invent a generic command or an unlock action.

If heating, cooling, or cleaning is required, perform the applicable
admissible operation before placement. Processing is part of
completion and must not be skipped while a complete route remains
feasible.
```

GPT-5.6-Luna **Initial skill s_0**

```
After every action, replan because object state and admissible actions
may change. Keep enough remaining capacity to finish the current
object and, for a two-object task, to search for, retrieve, process,
and place the second instance.

Two required objects do not need to be held simultaneously. The first
may be processed and placed before acquiring the second.

## 7. Replan after every command

Each turn provides the task, step count, recent history, current
observation, and complete admissible-action list. Use the new state
rather than following a fixed sequence.

At every decision:

1. update the ledger;
2. update object states and completed placements;
3. identify remaining required work;
4. estimate the shortest feasible completion cost;
5. compare candidate actions by information or irreversible progress
per step;
6. choose one command from the current admissible-action list.

An invented or unavailable command cannot advance the task and still
consumes a step. Copy the applicable admissible syntax exactly.

Once every required object has been found, stop exploring. Use the
remaining budget for required processing and placement.

## 8. Place every required object

Before placing, verify that:

- the object is at the named destination;
- every required heat, cool, or clean operation is complete;
- the placement command is currently admissible;
- enough steps remain for all outstanding requirements.

The destination need not be empty. Do not spend a step proving
emptiness when an admissible placement action is already available.

For a two-object task, placing the first object is not completion.
Preserve the settled ledger, continue to the second required instance,
process it as required, and place it in the same named destination.

The task is successful only when every required object has reached the
named destination and the simulator reports the winning state.
```

WebShop.

GPT-5.6-Luna **Initial skill s_0**

```
# Shopping Agent Skill

## 1. Parse the instruction

Identify:

- The requested product type and distinctive terms.
- Requested product attributes that must be true of the item itself.
```

GPT-5.6-Luna **Initial skill s_0**

```
- Requested selectable options, such as size, colour, count, or scent.
- The price ceiling.

Remember that a perfect score requires all requested attributes, all
requested options, an under-ceiling price, and a suitable item
kind/title match.

## 2. Search efficiently

Use a focused query containing the product type and distinctive
requested characteristics:

`search[keywords]`

A search costs one step from the shared 50-step episode quota. The
same query always returns the same results, so do not repeat a search
whose results are already visible. If the results do not expose a
promising candidate, use another focused query while enough steps
remain to inspect and buy.

The episode ends at step 50. If that happens before a purchase, the
result scores zero. Reserve enough steps for opening a candidate,
setting every required option, and buying it.

## 3. Inspect result listings

Read the visible listings for identifiers, titles, and prices. Results
contain ten products per page; use a valid next-page action when
additional results are worth the steps.

Prioritize listings whose:

- Title clearly matches the requested product.
- Price is below the ceiling.
- Title provides a strong kind match, since the kind factor is based
on the bought title and can reduce the reward.

Opening a listing, changing pages, returning, and other clicks each
consume one step. Do not spend steps repeating listings or searches
already visible. Further exploration may improve the observable title,
price, or options, but may leave insufficient steps to complete the
purchase.

## 4. Open the strongest candidate

Select a promising listing using its valid action:

`click[value]`

On the product page, inspect the description, available options, and
buy control. The description does not verify whether the requested
product properties actually match. No page or action can reveal that
during the episode, and further steps cannot improve this knowledge.
Assess the observable evidence, then decide whether continued search
is worth risking the remaining steps.

## 5. Set every requested option

Treat selectable values separately from product properties. For every
requested option, choose the matching available value on the product
page.
```

GPT-5.6-Luna **Initial skill s_0**

```
Each option change costs one step, but a matching option adds to the
reward and cannot reduce it. Use only values listed as valid actions
on the current page. Ensure all required options are set before
buying.

## 6. Decide whether to continue or buy

Continue searching only when the likely improvement in title, price,
or available options justifies the steps and still leaves enough
budget to open, configure, and buy a candidate.

Do not continue searching to learn whether the item's inherent
properties match: that remains unknowable until after the episode. An
incomplete purchase can receive partial reward during evaluation, but
an episode that reaches the step limit without purchasing scores zero,
so settle for the best adequately supported candidate when the
remaining budget becomes tight.

## 7. Buy before the limit

Before buying, verify:

- The title matches the requested product kind.
- The listed price is under the ceiling.
- Every requested selectable option has been set.
- The current page's buy control is a valid action.

Issue the buy action before step 50. A purchase that misses any
required condition is recorded as zero for the episode's exact-1.0
score, while no purchase also scores zero.

## 8. Format every response safely

Emit exactly one valid command and place it at the end of the
response:

- `search[keywords]`
- `click[value]`

The parser uses only the last such command. Keep the entire response
within 64 tokens; if the response is truncated, the intended action
may be ignored or malformed. Do not add explanatory text. Ensure the
command value exactly matches the current page's valid action.

Use the step counter as the hard accounting source. The counter
includes the current action and covers searches, clicks, option
changes, and purchase together. Unused steps have no value after the
episode and cannot be carried forward.
```

A.10.2 QWEN-3.5-9B

OfficeQA.

Qwen-3.5-9B **Initial skill s_0**

```
Skill: Extract and Calculate Treasury Bulletin Data

1. **Locate the Correct Page**
```

Qwen-3.5-9B **Initial skill s_0**

```
   Begin by identifying the specific bulletin issue and page
   containing the required data. Use `glob` to find files matching the
   issue date or filename pattern indicated in the question. Once the
   correct file path is identified within the provided candidate list,
   invoke `grep` to search for the specific page header (e.g., `##
   Page <n>`) within that document. If the data appears in a table,
   use `read` with a starting line number and limit to fetch the
   surrounding context containing the numeric values. Before
   proceeding to calculation, verify that the retrieved lines actually
   include the required figures. If the `read` operation returns fewer
   lines than requested or cuts off text at 4000 characters, the page
   data is incomplete; do not proceed with arithmetic on missing rows
   as the result will be incorrect. Instead, adjust the `read` offset
   or target a smaller data range that fits within the tool's
   capacity.

2. **Extract Numerical Values**
   Extract the relevant table rows and numerical values from the text
   using the tools. Apply `grep` with a pattern matching table headers
   to isolate relevant sections, then use `read` to fetch the full
   table rows containing the numeric values needed for the
   calculation. Ensure all retrieved numbers are clean and free of
   formatting artifacts before substituting them into the procedure
   defined in the question. Restrict all file searches exclusively to
   the list of up to 20 candidate paths provided at the start of the
   episode; do not attempt to access bulletin issues or paths not
   explicitly listed.

3. **Identify the Required Procedure**
   Identify the specific statistical term or transformation named in
   the question (e.g., standard deviation, semi-variance, smoothing).
   Recognize the term according to its standard definition in finance
   and public treasury data. Do not assume the existence of
   specialized statistical functions unless explicitly supported by
   standard financial conventions applicable to the bulletin data.
   Ensure the identified term aligns with the arithmetic steps needed
   to derive the requested numeric, percentage, or short-text value.

4. **Execute the Calculation**
   Perform the arithmetic or statistical calculation on the extracted
   values exactly as defined by the question's wording. If the task
   specifies a named procedure, apply its standard rules precisely,
   including any conditional logic (such as selecting only values
   below a mean). For calculations involving ratios, compute the
   numerator and denominator separately before dividing, handling
   cases where the denominator might be zero.

5. **Format the Result**
   Format the final result according to the specified output
   constraints. If the question requires a percentage, explicitly
   multiply any proportion by 100. Round the outcome to the specific
   number of decimal places required by the question (e.g., two places
   for currency, one for percentages) using standard mathematical
   rounding rather than truncating. Ensure the output string displays
   exactly the requested number of decimal digits.

6. **Finalize and Emit**
```

Qwen-3.5-9B **Initial skill s_0**

```
Emit the final answer within `<answer>...</answer>` tags
immediately after processing. Ensure the answer matches the
required format exactly. If you have not produced a valid answer
before reaching the 24-round threshold, the episode fails
automatically; therefore, structure tool calls to gather all
required data in the fewest possible steps.
```

SpreadsheetBench.

Qwen-3.5-9B **Initial skill s_0**

```
Skill: Transform Workbook Answer Range

Objective
Write one self-contained Python script that reads a workbook from
`INPUT_PATH`, applies a natural-language instruction to calculate
specific values, and writes the result to an answer range in a new
workbook at `OUTPUT_PATH`. The output must be a transformed copy of
the input, preserving all existing sheets, formulas, formatting, and
content outside the specified answer range.

Constraints and Costs
- Wall-clock limit: 120 seconds. If you exceed this, the script
terminates prematurely and the episode fails. Use `iter_rows` for
large sheets or `read_only=True` to process data efficiently.
Structural changes like row deletion must use reverse indexing to
avoid coordinate shifts that could cause timeouts.
- Execution attempts: 30. Any runtime error consumes one attempt. If
all attempts are exhausted, the episode fails. Ensure code is robust;
do not rely on external tools.
- Memory limit: Loading the entire workbook into a pandas DataFrame
may cause an OutOfMemoryError if the file is large. If you anticipate
memory constraints, stream data using `iter_rows` or chunked reads
rather than loading the full sheet.

Procedure

1. Parse Informal Instruction to Target Range
   - Read the natural-language instruction to identify the
   source-to-target mapping. Determine which sheet contains the data
   and which cells constitute the "answer range" where final values
   must be written.
   - Do not assume fixed column letters or header positions based on a
   small preview. Infer the mapping by observing how multi-row
   headers, blank spacers, and labels align with data records.
   - Identify the predicate (source condition/field) and the return
   field (target value) for the requested logic. Note that these may
   differ from the output column.
   - Define the answer range coordinates (one-based: row, column)
   explicitly before loading the workbook.

2. Load Workbook Preserving Structural Integrity
   - Use `openpyxl.load_workbook` to load the workbook from
   `INPUT_PATH`. Open in read/write mode to preserve existing formulas
   and formatting.
   - Do not create a new workbook; work with the loaded instance.
   - Avoid using `pandas` for the initial structural load to prevent
   loss of merged cells, blank spacers, or non-standard header
   positions.
```

Qwen-3.5-9B **Initial skill s_0**

```
   - If the workbook is massive, consider `read_only=True` to save
   memory, but be aware this may limit access to certain formatting
   properties needed for complex transformations.
   - Verify that the active sheet reference is correct and that the
   identified answer range coordinates exist within the sheet
   dimensions.

3. Resolve Multi-Row Headers and Entity Grouping
   - Treat the first visible row of the sheet as the potential header
   source, but do not assume a single flat row. Inspect the preview
   and actual data to infer how multi-row headers or blank spacers
   align with records.
   - Use the observed alignment to map source labels to target fields.
   Do not rely on fixed letters if the preview suggests variability.
   - If the instruction requires deleting or extracting rows, plan to
   use reverse index order for deletion to prevent shifting
   coordinates mid-execution.
   - Verify that your logic respects the existing structure, ensuring
   that entity continuity over multiple rows is handled correctly.

4. Disambiguate Missing Values Versus Zero
   - Treat missing values and zero as distinct states. Check if a cell
   object is `None` versus if its value equals `0`.
   - When reading cells, use `data_only=True` only if comparing cached
   results to determine if a cell is missing or uncalculated. For
   logic implementation, rely on raw cell values.
   - Ensure conditional sums or lookups explicitly guard against
   treating zero as missing. Do not apply blanket conversion rules;
   decide based on the specific workbook context.

5. Select Appropriate Aggregation Function
   - Match the natural-language requirement to a standard operation
   (e.g., sum, average, count, conditional sum).
   - Choose the aggregation method that correctly processes the
   specific columns and criteria without reordering the data. If the
   instruction implies preserving order, use `groupby(sort=False)` or
   order-aware grouping.
   - If conditional logic is involved, evaluate criteria before
   performing aggregation.
   - Ensure the chosen function handles missing or zero values
   appropriately according to the data context.

6. Write Calculated Values to Answer Range
   - Calculate the final values for the identified answer range using
   the logic derived in previous steps.
   - Use one-based coordinates for rows and columns when writing to
   the workbook.
   - Write computed literals directly into the target cells. Do not
   write formulas unless the instruction explicitly requires it; the
   normal expectation is to write final values.
   - Verify that the written values are literals, not formula strings
   or `None` objects.
   - Ensure that no unrelated structures (merged cells, headers
   outside the range, other sheets) are altered.

7. Verify Output Against Checker Contract
   - Before finalizing, verify the target cells by loading the output
   workbook (or the modified instance) with `data_only=True` to ensure
   the checker receives computed results rather than formula strings.
   - Iterate through the specific answer range and confirm each cell
   holds the correct final value as a literal.
```

Qwen-3.5-9B **Initial skill s_0**

```
   - Ensure the distinction is clear: source formulas remain intact
   outside the answer range, while the answer range contains the
   calculated literals.
   - Save the workbook to `OUTPUT_PATH` only after verification
   passes.

8. Handle Runtime Limits and Error Recovery
   - Monitor execution time; if approaching the 120-second limit,
   simplify logic or use streaming to finish.
   - If a runtime error occurs, treat the episode as failed for that
   attempt. Do not attempt to bypass harness constraints.
   - Ensure the script exits cleanly, leaving a valid file at
   `OUTPUT_PATH` if possible, or failing gracefully if an error is
   unavoidable.

Budget Application
- **Wall-clock (120s):** Apply this constraint when deciding whether
to process rows in memory or stream them. If the dataset is large,
switch to `iter_rows` immediately to avoid premature termination. When
planning structural changes, calculate reverse indexing costs to
ensure they fit within the time limit.
- **Attempts (30):** Apply this constraint when evaluating the
robustness of your code. If a step might fail (e.g., coordinate
mismatch), ensure you have a plan to correct it without consuming all
attempts. Verify the output before saving to avoid wasting attempts on
invalid files.
- **Memory:** Apply this constraint when deciding on data loading
strategies. If the preview suggests a large number of rows, avoid
loading the entire sheet into a DataFrame; instead, iterate row-by-row
to prevent OutOfMemoryError and script crash.
```

ALFWorld.

Qwen-3.5-9B **Initial skill s_0**

```
Skill: Household Object Search, Processing, and Placement

Objective
Execute the task defined in the opening observation ("Your task is to:
...") by locating the specified object(s), performing any required
intermediate processing (heating, cooling, cleaning), and placing the
result(s) into the named destination receptacle. The task is complete
only when the destination contains the required object(s). Partial
progress or failure to process/place yields a score of zero.

Constraints and Costs
- Budget: Exactly 50 steps per episode. Every action, including
invalid commands or redundant moves, consumes one step.
- Failure Condition: If the episode ends with fewer than 50 steps used
and the destination is not correctly filled, the score is zero. If
completion becomes impossible due to insufficient steps, continuing to
act is the only way to preserve any chance of success; stopping
guarantees failure.
- Two-Object Tasks: Both instances must reach the destination. They do
not need to be held simultaneously. The first may be processed and
placed before acquiring the second.
- Destination State: The destination does not need to be empty. If a
placement action is admissible, do not waste steps verifying
emptiness.
```

Qwen-3.5-9B **Initial skill s_0**

```
- Settled Locations: A receptacle is "settled" once its contents are
fully observed. Revisiting a settled location yields no new location
information and costs a step unless retrieving a previously observed
object or performing a necessary placement.

Procedure

1. Initialize Ledger and Calculate Remaining Budget
   - Maintain a persistent ledger recording every receptacle visited,
   its name, and its observed contents.
   - Calculate the remaining budget by subtracting the steps already
   taken (reported in the prompt) from the episode limit of 50.
   - Determine the minimum steps required to complete the task from
   the current state: estimated cost to find missing objects + cost of
   required processing (heat/cool/clean) + cost to reach destination +
   cost to place.
   - Decision Point: If the remaining budget is less than the
   estimated cost-to-go, the task is currently infeasible. Do not
   stop. Select the admissible action that maximizes the probability
   of future feasibility (e.g., finding a missing object or starting a
   processing chain).

2. Filter Admissible Actions for Exploration vs. Execution
   - Review the current admissible-action list.
   - Classify actions into:
     - Exploration: Moving to an unvisited receptacle.
     - Retrieval: Moving to a settled receptacle containing a required
     object.
     - Processing: Heating, cooling, or cleaning an object.
     - Placement: Moving an object to the named destination.
     - Invalid/Redundant: Actions that do not advance the state or
     search.
   - Rule: Never move to a settled receptacle for the purpose of
   searching for new objects. Such moves waste the step budget without
   reducing uncertainty.

3. Select Next Action Based on Budget and Task State
   - If required objects are missing and unvisited receptacles exist:
     - Select an unvisited receptacle that is reachable within the
     remaining budget while preserving enough steps for all subsequent
     processing and placement.
     - If no unvisited receptacle allows a feasible completion path,
     select an action that reduces uncertainty or initiates a
     necessary process, even if it seems suboptimal, because stopping
     guarantees failure.
   - If required objects are found but unprocessed:
     - Select the exact processing command from the admissible list.
     Copy the syntax exactly; do not invent generic commands.
     - Ensure the remaining budget covers this step plus the estimated
     cost to place the object.
   - If objects are processed and ready:
     - Select the placement command to move the object to the named
     destination.
     - Do not verify the destination is empty if a placement action is
     available.
   - If objects are placed but a second object is still required
   (two-object task):
     - Resume search for the second object using the updated ledger.
     Retain the knowledge of where the first object was placed.

4. Update Ledger and Replan
   - After every action, observe the result.
```

Qwen-3.5-9B **Initial skill s_0**

```
   - If the action reveals new contents in a receptacle, update the
   ledger immediately to mark it as settled.
   - Re-evaluate the remaining budget (50 - steps taken) and the
   cost-to-go.
   - If the state changes (e.g., an object is now available for
   processing), adjust the completion plan accordingly.
   - Continue emitting one admissible action per turn. Never emit an
   empty response or terminate early, as this consumes a step toward
   the limit without progress.

5. Termination Logic (Implicit)
   - The agent acts until the episode ends (step 50 reached) or the
   task is completed.
   - If the task is completed before step 50, the remaining steps are
   wasted but the score is 1.
   - If step 50 is reached and the task is incomplete, the score is 0.
   - There is no "give up" state; the agent must always emit a valid
   admissible action until the episode limit is hit.
```

WebShop.

Qwen-3.5-9B **Initial skill s_0**

```
Skill: Purchase Matching Product

OBJECTIVE
Execute a purchase that achieves a final score of 1.0. This requires
four simultaneous conditions:
1. All inherent product properties match the goal (verified
post-episode).
2. All selectable options (size, color, etc.) set on the page match
the goal.
3. The listed price is strictly under the stated price ceiling.
4. The reward factor is 1.0 (derived from title similarity to the
goal's category segments and nouns).
Failure to meet any single condition results in a recorded score of
0.0.

PROCEDURAL STEPS

STEP 1: Extract Constraints and Form Query
Parse the instruction to isolate the target product name, required
inherent properties, required options (e.g., size, color), and the
price ceiling. Construct a search query that emphasizes core keywords
and category segments to maximize the probability of achieving the
title similarity factor required for the 1.0 reward multiplier.

STEP 2: Execute Initial Search
If the current page displays a search bar, issue `search[keywords]`.
If the search bar is absent, wait for a page with one. This action
costs one step. The environment returns ten results in a fixed order;
do not attempt to filter or sort. Note that repeating this search
yields identical results.

STEP 3: Evaluate Results Page (Price and Title)
Inspect the results list for identifiers, titles, and prices.
- **Price Check:** Discard any candidate where the listed price meets
or exceeds the price ceiling. A match here is impossible.
- **Title Check:** Assess if the title shares significant nouns or
category segments with the goal to target the 1.0 reward factor.
```

Qwen-3.5-9B **Initial skill s_0**

```
- **Decision:** If a candidate passes the price check and offers a
promising title, proceed to inspect it. If no candidate meets these
criteria on the current page, consider paginating or refining the
search if steps remain.

STEP 4: Inspect Product Details (Click Listing)
Select the most promising candidate from STEP 3 and issue
`click[value]` for that specific listing. This action costs one step.
The parser executes only the final command in the response; ensure no
other action follows. This navigates to the product page containing
the description, options, and buy control.

STEP 5: Set Required Options
On the product page, inspect the VALID ACTIONS line for available
option values.
- **Action:** If a required option (e.g., specific size or color) is
listed, issue `click[value]` for that exact value. This costs one
step.
- **Constraint:** Do not attempt to verify inherent properties (e.g.,
scent, burn time) as no action exists to query them; these are
evaluated only at the end.
- **Warning:** Setting an option that does not match the goal adds a
step but cannot lower the reward; however, failing to set a required
option guarantees a zero score. Do not set an option unless it is
explicitly required by the instruction.

STEP 6: Final Purchase Decision
Before issuing `click[Buy]`, verify the following visible constraints:
1. The item was found via the initial search (or shares sufficient
title similarity for the factor).
2. The listed price is strictly under the ceiling.
3. All required options have been set via previous `click` actions.
4. Steps remain sufficient to complete the buy action.

**Cost-Benefit Weighing:**
- **Exploration Cost:** Further `click` actions (to different listings
or to set more options) consume steps from the 50-step budget. Wasting
steps here risks exhausting the budget before a purchase can be made,
resulting in a zero score.
- **Option Cost:** Setting an option costs one step. If the option
matches the goal, it contributes positively to the numerator. If no
option matches, this step is still spent, but the reward remains
capped by other missing factors.
- **Buying Cost:** Issuing `click[Buy]` costs one step. If any
condition (price, options, or inherent properties) fails, the episode
scores 0.0 regardless of steps remaining.
- **Abandonment Cost:** If the step counter reaches 50 without a
purchase, the episode ends with a zero score.

Execute `click[Buy]` only if all visible criteria are met and a
purchase is imminent. If criteria are not met and steps remain, issue
a new `search[keywords]` or navigate to the next page to find a better
match. If the step budget is critically low, prioritize the
highest-probability match visible on the current page over further
exploration.

BUDGET CONSTRAINTS

FINITE: 50 Total Steps
- **Cost:** Every `search`, `click` (listing, option, or buy), and
every turn where no action occurs consumes one step.
```

Qwen-3.5-9B **Initial skill s_0**

```
- **Failure Mode:** Reaching 50 steps without a completed `click[Buy]`
terminates the episode with a score of 0.0.
- **Decision Point:** At every step, weigh the certainty of meeting
the purchase criteria against the risk of running out of steps. An
incomplete search scores 0.0; an abandoned search scores 0.0. The
agent must decide whether to spend the remaining steps to verify a
match or accept the risk of stopping early.

FINITE: 64 Tokens per Response
- **Cost:** The environment truncates the response if it exceeds 64
tokens.
- **Failure Mode:** Truncation prevents the parser from seeing the
action command, effectively wasting the current step and yielding no
progress.
- **Decision Point:** The agent must ensure the final command string
is concise and under the limit to avoid losing a step.

FINITE: 1 Step per Interaction
- **Cost:** Clicking a listing, setting an option, or buying costs
exactly one step.
- **Failure Mode:** Clicking a listing with a price over the ceiling
or one with insufficient title similarity wastes a step toward a
guaranteed 0.0 score. Setting an option that does not exist or is not
required wastes a step.
- **Decision Point:** Only click a listing if the price is acceptable
and the title looks promising. Only set an option if it is explicitly
required by the goal.
```

A.11 Optimized Skills

We reproduce the final optimized skills $s^\star$ produced by RASO for each benchmark and target model. Starting from the corresponding initial skill s_0, these skills are obtained after iterative RASU updates and validation-based acceptance. The skill bodies are reproduced below verbatim apart from formatting.

A.11.1 GPT-5.6-Luna

OfficeQA.

GPT-5.6-Luna **Optimized skill $s^\star$**

```
# Skill: Answer Historical U.S. Treasury Bulletin Questions

1. Parse the question before searching. Identify:
    - The cited bulletin issue, date, and page number(s).
    - The exact figures required.
    - The requested arithmetic or named statistical/financial
    procedure.
    - The required output format and number of values.
2. Start with the supplied candidate file paths. Preserve the
citation's issue, date, and page information while selecting source
data. Locate each cited page by its `## Page <n>` marker.
3. Read only the relevant page text and tables. Use small,
page-bounded `read` calls. Preserve table rows, columns, units, and
distinctions when extracting figures.
4. If the supplied page text is insufficient, use targeted `grep`
searches for distinctive table labels, wording, dates, page
identifiers, or filenames. Then use `read` around the matching line
numbers to recover the relevant context. Do not rely on omitted
matches or truncated output.
```

GPT-5.6-Luna **Optimized skill $s^\star$**

```
5. Treat a `grep` result as incomplete if its 20-line match allowance
is exhausted or the output does not show the needed source. Narrow the
pattern or read the relevant line range directly. Avoid broad searches
that spend matches on irrelevant lines.
6. Treat a `read` result as incomplete if it approaches the
4000-character response limit or cuts off a table or passage. Re-read
narrower line ranges, retrieving only the rows and columns needed for
the calculation. Do not calculate from a truncated table.
7. If the correct issue is not among the candidates, search the corpus
using specific combinations of the bulletin date, page number, title,
table labels, and distinctive wording. Avoid generic topic searches.
Read only the strongest matches and verify them against the cited
page.
8. Extract only figures that directly support the requested result.
Check that every value comes from the correct cited issue and page,
and preserve units and category distinctions.

When repeated dates, years, or series appear, or when a table has
flattened or multi-level headers, read a narrow range around the cited
page and reconstruct the complete header-to-column mapping before
selecting any value. Enumerate every requested date in order, attach
each row to its displayed year or month, series, column, and unit, and
verify the count and identity of all observations before comparing
them. When selecting a maximum or first crossing from a multi-year
table or chart, align the exact specified series values by their
displayed year labels before comparing them. Preserve fiscal-year
conventions and the requested inequality or maximum criterion; before
committing, verify the selected year against both adjacent years, and
for a crossing confirm it is the first qualifying year. Carry the
fully identified selected date unchanged into any second-table lookup
and recheck its row and column there. If complete header
reconstruction remains impossible near the round limit, use the
best-supported completed value rather than infer an unverified
alignment or fail to answer.

1. Apply the exact procedure stated by the question. For a named
statistical or financial term, use its standard definition. Include
all required transformations, ratios, percentages, moving averages,
dispersion or risk calculations, windows, and rounding rules. Do not
substitute an informal interpretation.
```

GPT-5.6-Luna **Optimized skill $s^\star$**

```
For any fixed monthly statistic or risk estimate, enumerate every
required month and confirm the count matches the stated observation
total before calculating. Verify the selected bulletin row and column,
distinguish the requested series from adjacent series, preserve the
stated units, apply the specified population convention, repair any
count, identity, or unit mismatch before calculating, and round only
the final result. When a calculation uses a fixed inclusive time
window or overlapping bulletin tables, enumerate every labeled
observation exactly once, reconcile overlapping vintages by selecting
the appropriate finalized and internally consistent series, and verify
each observation's date label, series identity, units, count, and
source vintage before calculating; do not silently mix inconsistent
values. When the question names a geometric mean, verify the complete
inclusive input set and use only the specified source measure; for
levels, average logarithms or take the product root, and for
percentage growth rates, convert each rate to a multiplicative factor
before averaging logarithms and converting back to the requested
units. Before final rounding, recheck the complete input list and
count to ensure that no observation was omitted, duplicated,
misclassified, arithmetically averaged, rounded early, or taken from
an unresolved vintage. If a source remains unresolved near the round
limit, use the best-supported verified subset as a completion fallback
rather than silently mixing values or failing to answer, while
preserving the required answer-only format.

1. After every tool round, assess whether the evidence is sufficient
to compute the answer. Stop retrieving as soon as it is sufficient.
There are at most 24 model rounds, including tool calls and the final
response; reaching round 24 without answering fails the episode. If
the evidence remains incomplete as the limit approaches, return the
best-supported result rather than spending rounds on repeated
searches, because reaching the limit produces no answer.
2. Return the result exactly in the requested format as:
`<answer>...</answer>`
3. Include no explanation, headings, labels, units, alternatives, or
extra whitespace unless the question explicitly requires them.
Exact-match scoring means the answer must contain only the requested
value or fixed set of values in the specified format. After
calculation, discard intermediate values and unrequested unit markers
or symbols, then emit only the requested value(s) inside the required
`<answer>...</answer>` wrapper and any explicitly requested brackets
or separators. Before submitting, verify that the final string
contains no extra text, percent signs, units, punctuation, or
intermediate values beyond what the requested format permits.
```

SpreadsheetBench.

GPT-5.6-Luna **Optimized skill $s^\star$**

```
# Skill: Transform an Existing Workbook into a Verified Answer
Workbook

## 1. Parse the request and identify the edit surface

Read the natural-language instruction and determine:

- The requested operation: lookup, conditional sum, date/time logic,
consolidation, filtering, deletion, extraction, text transformation,
or cumulative/grouped calculation.
- The source sheets and fields.
```

GPT-5.6-Luna **Optimized skill $s^\star$**

```
- The criteria or predicate fields.
- The returned or computed fields.
- The target answer schema.
- The exact answer-range anchor and dimensions.

Treat any preview as incomplete. Inspect the actual workbook for:

- Sheet names and dimensions.
- Header rows, repeated or multi-row headers, blank spacer cells, and
merged ranges.
- Representative values, formulas, dates, times, text, and missing
cells.
- Records whose entity continues across multiple rows.
- Labels that do not align with preview column letters.
- Columns beyond `Z`, checking coordinate conversions explicitly.

Load the workbook with:

```python
openpyxl.load_workbook(INPUT_PATH, data_only=False)
```

Use a separate `data_only=True` view only for inspecting cached
formula results when necessary. Do not save the values-only view.

Preserve the input workbook as the basis of the output. Unless the
instruction explicitly requires a structural change, only the
requested answer range may be modified.

2. Recover the source-to-target mapping

Before any bulk write, mandatory mapping resolution must use workbook
labels, header structure, and region boundaries rather than assumed
preview coordinates or coincidental numeric values: explicitly
identify each predicate or selector field, source record region,
return field, target anchor and extent, selector-to-source
relationship, repeated or multi-row header structure, selective
fill-forward rules, and entity boundaries; validate representative
first, last, selector, nonempty/nonzero, and boundary targets and
confirm that the mapped source records and values match the intended
records. Before any bulk write, treat any disagreement between labels,
candidate regions, selector-to-source relationships, source-to-target
offsets, or representative
first/last/selector/nonempty/nonzero/boundary records as a blocking
mapping failure: correct the mapping, then explicitly recheck those
representative and boundary mappings and confirm their source values
and target keys agree; do not write until the checks pass. Collect
every relevant header occurrence and candidate table region, resolve
source fields, selector fields, output labels, answer anchors, and
source-to-target offsets independently, distinguish source values from
labels and result values, and validate at least one known
source-to-output mapping. If that mapping or its offsets do not align,
correct the mapping before writing. Identify:

- The source predicate or criteria field.
- The source return field or fields.
- The output columns and their order.
- The target starting cell and target extent.
- Any relationship between source rows and target rows.
- Whether headers occupy one row or multiple rows.
- Whether labels need selective fill-forward.
- Whether records continue over multiple rows.
```

GPT-5.6-Luna **Optimized skill $s^\star$**

```
Keep values semantically distinct:

- Missing cells are not automatically the text `"None"`.
- Empty strings, zero, and `False` may have different meanings.
- Dates and times must be compared as dates and times when
appropriate.
- Text case and delimiters matter when the instruction or observed
values makes them relevant.
- Do not split or normalize free text without evidence that the
requested operation requires it.

Use one-based Excel coordinates at the worksheet boundary. If using
zero-based dataframe indices internally, convert carefully and verify
the conversion before writing.

## 3. Define calculation semantics before writing

Before calculating each target row or cell, translate the instruction
into explicit Python-side logic and resolve every criterion, selector,
entity boundary, and target key from the workbook. Bind each
calculation to the preexisting target row or cell identified by its
resolved key; combine supplied criteria conjunctively, apply each
criterion to its own source field with the required date, user, sign,
and comparison rules, and treat an omitted criterion as unconstrained
rather than as null or as a generated combination. Verify
representative first, last, selector, boundary, and omitted-criterion
targets before bulk writing. Decide from the request and workbook
values:

- Which rows qualify.
- How multiple criteria combine.
- How missing and empty values behave.
- Whether case-sensitive comparison is required.
- Whether zero and `False` count as present values.
- How dates, times, and ranges are compared.
- Whether repeated entities should be grouped.
- Whether row order must be preserved.
- Whether delimiters belong to extracted text.
- Whether the output should contain literal values or explicitly
requested formulas.

Do not apply blanket conversions that erase distinctions among
missing, empty, zero, false, numeric, date, and text values.

## 4. Compute in source order

Preserve the workbook's existing row order unless the instruction
explicitly specifies another ordering.

For cumulative or entity-based logic:

- Maintain state in encounter order.
- Use order-preserving grouping such as `groupby(sort=False)` if
grouping is needed.
- Do not reorder entities merely because a grouping operation permits
it.
- If the instruction explicitly requires sorting, sort by every
specified key and direction before computing.
```

GPT-5.6-Luna **Optimized skill $s^\star$**

```
Compute only the requested answer values. Avoid unnecessary
full-workbook transformations or repeated scans when a bounded
calculation is sufficient.

## 5. Apply only authorized edits

Edit the existing workbook rather than creating a replacement
workbook.

Normally:

- Change only the named answer cells.
- Preserve all sheets and existing content.
- Preserve formulas, formatting, comments, merged ranges, dimensions,
and unrelated cells.
- Do not rebuild a guessed rectangle.
- Do not compact a sheet or alter `max_row` or `max_column` as a
substitute for editing.

If the instruction explicitly requires deleting rows or columns:

- Identify the actual target indices.
- Delete them in reverse index order so later targets do not shift
incorrectly.
- Preserve all remaining workbook content.

Use verified worksheet coordinates, especially for columns beyond `Z`
and for answer ranges whose row offsets are not obvious.

## 6. Choose the serialized answer representation

Use the formula-preserving workbook as the transformation source.
```

GPT-5.6-Luna **Optimized skill $s^\star$**

```
Immediately before writing each requested answer cell, determine the
evaluator-visible representation from the task contract and artifact;
if final values are consumed and no calculation engine has produced
verified recalculated caches, calculate the logic in Python and write
a literal value only to that authorized cell, never formula text,
while preserving formulas elsewhere. Record the expected literal value
and, after saving, reopen the output and compare every requested
target cell plus the first and last rows and columns of the answer
range against those expected values. If the evaluator consumes final
values and no calculation engine with verified recalculated cached
results is available, calculate the requested logic in Python and
write a literal value only to that authorized answer cell; never use
formula text or stale/missing cached results as a substitute. Compare
representative computed values with their intended target cells before
saving, and after saving reopen the output and compare every requested
answer cell, including range boundaries, with the expected literal
values. Use a formula only when it is explicitly required and
recalculated cached results are available and verified, while
preserving the formula-bearing workbook for the transformation source.
When final values are consumed and no calculation engine is available,
compute the answer in Python and write literal values only to the
authorized answer cells; do not rely on calculation flags or formula
text. Preserve the formula-bearing transformation workbook and never
save a workbook opened with `data_only=True`, because that can replace
formulas with cached constants. If the instruction explicitly requires
formulas, write them only when their representation is appropriate and
recalculated cached results are actually available and verified. After
saving, reopen the output and assert the exact expected values in the
answer range and its first and last rows and columns; formula-string
survival alone is not sufficient.

The output must contain the intended values in the exact specified
answer cells after serialization, not merely in memory.

## 7. Save and verify the output

Save the transformed workbook to `OUTPUT_PATH`.

Reopen the saved file and verify:

- Every requested answer cell.
- The first and last rows and columns of the answer range.
- Nearby boundary cells.
- Important unrelated formulas and content.
- That sheets, formatting, and other workbook structure remain
present.
- That the serialized representation contains the intended values
rather than stale, blank, or misaddressed content.

A separate `data_only=True` reopen may be used to inspect cached
formula results, but cached values may be absent if the workbook was
not recalculated. Do not treat the existence of the output file as
proof of success.

## 8. Work within the execution budgets

The entire script has 120 seconds for loading, inspection,
computation, saving, and verification. If this limit is reached, the
process is terminated before a usable output can be produced or
validated. Use a direct bounded workflow:

1. Load the existing workbook once for transformation.
```

GPT-5.6-Luna **Optimized skill $s^\star$**

```
2. Inspect only the structure needed to resolve the request.
3. Compute only the requested values.
4. Update only the answer range or explicitly required structural
area.
5. Save once to `OUTPUT_PATH`.
6. Reopen once for targeted verification.

Avoid unnecessary full-workbook passes, repeated loads, expensive
formatting operations, and replacement-workbook construction.

There are at most 30 execution attempts. Each runtime error or retry
consumes an attempt. Before running, check coordinates, imports,
workbook paths, range dimensions, and save logic. If a run fails, make
a targeted correction based on the failure rather than repeating the
same error. Once the attempt budget is exhausted, no further
correction is possible; a missing or invalid output fails the task.

The input workbook's existing artifact is also finite: destructive
rebuilding or broad edits consume its accumulated structure. If that
structure is lost, duplicated, or replaced, the transformed copy is
invalid. Preserve the original workbook and make only the authorized
changes.

Finally, the serialized output representation is finite: incorrect
coordinates, formulas, or unwritten values cause the harness to read
wrong, stale, or unevaluated answers even when the in-memory
calculation was correct. Always save, reopen, and verify the exact
cells the harness will consume.
```

ALFWorld.

GPT-5.6-Luna **Optimized skill $s^\star$**

```
# Skill: Budgeted Household Search and Placement

## 1. Parse the task

Read the first observation's sentence beginning **"Your task is
to:"**. Extract:

- the required object type;
- the named destination receptacle;
- any required heating, cooling, or cleaning;
- whether one or two instances are required.

Preserve the simulator's exact object and receptacle names and
numbers. The opening observation lists the complete set of possible
receptacles, but their contents are initially unknown.

Completion requires every required instance to undergo its specified
processing, if any, and be placed in the named destination. A
two-object task is incomplete until both objects are there.

## 2. Initialize and maintain the search ledger

Create a persistent ledger containing every named receptacle. Mark
each as either:

- **unvisited**, when its contents are unknown; or
- **settled**, once it has been reached and its complete contents
observed.
```

GPT-5.6-Luna **Optimized skill $s^\star$**

```
Record all contents of every settled receptacle. An observation of a
receptacle is complete: an omitted object will not later appear there.

After every action, update:

- settled and unvisited receptacles;
- remembered object locations;
- object processing states;
- objects already placed;
- the number of steps used and remaining;
- the currently admissible actions.

Never discard or rebuild the ledger after moving, processing, or
placing an object.

## 3. Choose the next action under the budget

Each response consumes one step, including an invalid, redundant,
empty, or ineffective response. The episode has at most 50 steps. If
the budget runs out before all required objects are processed and
placed, the episode ends without a winning placement; partial progress
scores the same as failure.

Before every action, consult the persistent ledger and current
admissible-action list, reject any settled receptacle as an
information-gathering target, and reserve the shortest currently
feasible route to completion; proceed with the chosen search or
acquisition action only when that route cost is less than or equal to
the remaining steps, including:

- searching remaining unvisited receptacles;
- returning to retrieve remembered objects;
- taking objects;
- heating, cooling, or cleaning;
- traveling to the destination;
- placing every required object.

A feasible route requires remaining steps **greater than or equal to**
its estimated cost. Do not require an extra spare step.

Prefer an admissible action that provides useful new information or
irreversible task progress. Avoid redundant revisits and actions that
do not advance completion. Once every required object is known, stop
exploring and spend the remaining budget on processing and placement.

If completion has become impossible, do not stop, wait, emit nothing,
or terminate early. Partial completion still scores zero, and unused
steps have no value. Continue issuing a currently admissible action
that gives the best remaining chance of full completion.

## 4. Explore receptacles

Select an unvisited admissible receptacle, prioritizing it using:

- the required object and destination names;
- observations already obtained;
- available admissible actions;
- the remaining cost-to-go.
```

GPT-5.6-Luna **Optimized skill $s^\star$**

```
The environment provides no probability table for object locations, so
broaden the search to other unvisited receptacles after likely
locations are exhausted.

When a receptacle is reached and, where necessary, opened, record its
complete contents and mark it settled. After a receptacle's complete
contents are observed, permanently mark it settled and reject it for
every information-gathering action; revisit it only for necessary task
progress, such as retrieving a remembered object or placing one, while
preserving its recorded contents. After every action, verify the
ledger, object states, admissible commands, and remaining
completion-route cost before selecting the next command.

Reserve enough steps for all required actions still needed after the
search. Exploration or redundant actions can consume the route's
remaining action budget and make full completion infeasible; if that
occurs, continue with admissible actions that maximize the chance of
completion rather than stopping.

## 5. Acquire and process an object

Before taking an object, confirm that:

- its receptacle has been observed;
- the object is recorded there;
- the taking action is currently admissible;
- enough budget remains for all required work afterward.

Use exactly the taking command shown in the current admissible-action
list. After taking the object, update its state and the admissible
actions.

If heating, cooling, or cleaning is required, perform the
corresponding operation only when the object and appropriate appliance
satisfy the current admissibility conditions. Use the exact command
listed; do not substitute a generic command or invent an unlock
action. Confirm the resulting state after the action and replan.

Do not skip required processing while a complete solution remains
feasible. Every required object must be processed before placement.

## 6. Route to the destination and place

After each action, replan from the new observation and
admissible-action list. Do not commit to a fixed sequence whose later
commands may no longer be admissible.

Once an object is ready, route it to the named destination and use the
exact currently admissible placement command. The destination does not
need to be empty; do not spend a step proving emptiness when placement
is already admissible.

Before spending a step on exploration or manipulation, compare it with
the remaining cost of completing all required placements. If the
remaining actions needed for search, processing, retrieval, travel,
and placement exceed the remaining steps, full completion is no longer
guaranteed; continue with the admissible action that best preserves
the chance of winning.

## 7. Handle two-object tasks
```

GPT-5.6-Luna **Optimized skill $s^\star$**

```
A two-object task requires both instances to reach the same
destination, but they need not be held simultaneously.

After placing the first object:

1. retain the complete visited ledger;
2. retain the location and state of the second required object, if
known;
3. search only unvisited admissible receptacles if the second object
is not yet found;
4. do not rescan settled receptacles as unknown;
5. reserve enough steps to retrieve, process, route, and place the
second object;
6. replan after every action.

Place the second object in the same named destination. The task is won
only when both required objects have been placed before the 50-step
episode budget is exhausted.
```

WebShop.

GPT-5.6-Luna **Optimized skill $s^\star$**

```
# Shopping Agent Skill

## 1. Parse the instruction

Identify:

- The requested product type and distinctive terms.
- Requested product attributes that must be true of the item itself.
- Requested selectable options, such as size, colour, count, or scent.
- The price ceiling.

Remember that a perfect score requires all requested attributes, all
requested options, an under-ceiling price, and a suitable item
kind/title match.

## 2. Search efficiently

Use a focused query containing the product type and distinctive
requested characteristics:

`search[keywords]`

A search costs one step from the shared 50-step episode quota.
Maintain a ledger of issued queries, opened listings, titles, prices,
available controls, matched fields, rejected or unknown evidence, and
the minimum steps needed to open, configure, and buy. The same query
always returns the same results, so do not repeat a search whose
results are already visible or reopen a known candidate without new
visible evidence. If the results do not expose a promising candidate,
use another focused query while enough steps remain to inspect and
buy.

The episode ends at step 50. If that happens before a purchase, the
result scores zero. Reserve enough steps for opening a candidate,
setting every required option, and buying it.

## 3. Inspect result listings
```

GPT-5.6-Luna **Optimized skill $s^\star$**

```
Read the visible listings for identifiers, titles, and prices. Results
contain ten products per page; use a valid next-page action when
additional results are worth the steps.

When a candidate title is accessory-like, adjacent-category, generic,
or lacks explicit requested-kind evidence, record that identity
weakness and, while the completion reserve remains intact, compare
distinct candidates that could improve verified evidence; rank
explicit requested product-kind identity first, then verified
requested attributes and exact selectable options, then price below
the ceiling, and retain disqualifying accessory or unrelated-category
evidence when comparing candidates. Prioritize listings whose:

- Title clearly matches the requested product.
- Price is below the ceiling.
- Title provides a strong kind match, since the kind factor is based
on the bought title and can reduce the reward.

Opening a listing, changing pages, returning, and other clicks each
consume one step. Do not spend steps repeating listings or searches
already visible. Further exploration may improve the observable title,
price, or options, but may leave insufficient steps to complete the
purchase.

## 4. Open the strongest candidate

Select a promising listing using its valid action:

`click[value]`

On the product page, inspect the description, available options, and
buy control. The description does not verify whether the requested
product properties actually match. No page or action can reveal that
during the episode, and further steps cannot improve this knowledge.
Assess the observable evidence, then decide whether continued search
is worth risking the remaining steps.

## 5. Set every requested option

Treat each selectable value as a separate per-candidate requirement,
distinct from product properties, and record it as exact, unavailable,
unknown, or merely related; map it only to an exact matching value
shown as a valid control on the current product page, counting a
combined control only when it visibly covers every requested value on
that same candidate, never treating a generic, approximate, or merely
related value as equivalent or combining evidence across candidates.
Click every available exact matching value, then re-read the current
page to confirm each requested value remains selected before buying;
if an exact value is unavailable or unverified, keep searching
distinct candidates while the completion reserve permits, otherwise
preserve the strongest available matches for the terminal purchase.

Each option change costs one step, but a matching option adds to the
reward and cannot reduce it. Use only values listed as valid actions
on the current page. Ensure all required options are set before
buying.

## 6. Decide whether to continue or buy
```

GPT-5.6-Luna **Optimized skill $s^\star$**

```
Maintain the ledger as persistent state across searches, navigation,
candidate changes, and subgoals: after each search or inspection, mark
queries and listings as inspected, update the best under-budget
incumbent from visible title, price, intrinsic evidence, and exact
selected controls, and continue only if a distinct query or candidate
can improve the ranked evidence while enough actions remain to open,
configure, re-check, and buy. Do not repeat an inspected query or
listing without new visible evidence, and treat missing
intrinsic-property evidence as unknown rather than satisfied or as a
reason to reject every candidate. When the completion reserve is
reached, or no distinct candidate can improve the incumbent without
consuming that reserve, stop exploring, configure every available
exact matching option on the strongest incumbent, re-check identity,
price, selected controls, any visible intrinsic evidence, and the
valid buy action, then purchase before expiry; this terminal purchase
takes precedence over further exploration because a completed
partial-credit purchase is preferable to episode expiry, which scores
zero.

## 7. Buy before the limit

Before buying, verify:

- The title matches the requested product kind.
- The listed price is under the ceiling.
- Every requested selectable option has been set.
- The current page's buy control is a valid action.

Issue the buy action before step 50. A purchase that misses any
required condition is recorded as zero for the episode's exact-1.0
score, while no purchase also scores zero.

## 8. Format every response safely

Emit exactly one valid command and place it at the end of the
response:

- `search[keywords]`
- `click[value]`

The parser uses only the last such command. Keep the entire response
within 64 tokens; if the response is truncated, the intended action
may be ignored or malformed. Do not add explanatory text. Ensure the
command value exactly matches the current page's valid action.

Use the step counter as the hard accounting source. The counter
includes the current action and covers searches, clicks, option
changes, and purchase together. Unused steps have no value after the
episode and cannot be carried forward.
```

A.11.2 QWEN-3.5-9B

OfficeQA.

Qwen-3.5-9B **Optimized skill $s^\star$**

```
Skill: Extract and Calculate Treasury Bulletin Metrics

1. Locate and Filter Candidate Documents
```

Qwen-3.5-9B **Optimized skill $s^\star$**

```
    - Use `glob` with a recursive pattern (e.g., `*/*`) to list all
    files in the corpus. Refine the pattern using keywords from the
    question if the initial list is too large to manage within the
    round budget.
    - Cross-reference the returned paths against the episode context
    to identify the specific bulletin issues containing the answer.
    - **Budget Awareness:** This step must be completed quickly. If
    too many files are returned, the search strategy is inefficient
    and risks exhausting the 24-round limit before data extraction
    begins. Do not spend excessive rounds iterating on glob patterns
    if the initial broad search suffices to find the target directory
    structure.
2. Locate Specific Data within Files
    - Use `grep` with a case-insensitive pattern to find lines
    containing specific terms, row identifiers, or table markers
    (e.g., `## Page <n>`) within the candidate files.
    - **Budget Awareness:** `grep` returns a maximum of 20 matching
    lines. If the search yields more than 20 lines, the excess matches
    are discarded. You cannot retrieve these additional matches. If
    the target data is likely buried beyond the first 20 matches, you
    must refine your search term to reduce noise and ensure the
    specific row or page header appears within the visible results. If
    results are empty, verify the file path or broaden the term
    slightly, but do not assume data exists if the tool returns
    nothing.
    - Note line numbers provided in the output to determine precise
    reading positions.
3. Extract Raw Data Rows
    - Use `read` starting from the line numbers identified in the
    previous step to retrieve the table content.
    - **Budget Awareness:** Each `read` call has a hard limit of 4000
    characters. If the table or required data spans more than this,
    the tool will truncate the output, potentially missing the final
    rows needed for calculation.
    - If a single `read` call does not return sufficient lines to
    capture the full table or all necessary values, do not retry the
    same broad read. Instead, use the line numbers to calculate a
    tighter start position and a smaller limit, ensuring the retrieved
    text exactly covers the required data range without hitting the
    truncation limit.
    - Ensure the extracted text includes complete rows from the
    markdown table structure, preserving vertical bar delimiters until
    the full dataset for the calculation is verified.
4. Parse and Isolate Values
    - Manually parse the retrieved text to extract the specific
    numeric or short-text values required by the question.
    - Verify that the extracted values correspond exactly to the
    column headers and row content identified in the bulletin pages.
    Before calculating any statistical measure, explicitly verify the
    exact text of the row label and column header matches the
    question's description to ensure the correct data series is
    selected. Cross-check extracted values against the context of the
    table to ensure logical consistency and distinguish between
    similarly named line items, such as 'Total reserves' versus 'Total
    reserves assets' or different coin categories.
```

Qwen-3.5-9B **Optimized skill $s^\star$**

```
    - **Budget Awareness:** Do not apply any transformations (e.g.,
    averaging, summing) at this stage unless the question explicitly
    asks for a subset of data before the statistical procedure. The
    goal here is strictly accurate extraction. If the question
    requires a specific statistical measure (e.g., standard
    deviation), ensure all necessary values for that measure are
    present in the extracted set. Missing a single data point due to
    truncation or a bad grep pattern will make the subsequent
    calculation impossible.
5. Apply Statistical Transformation
    - Convert the extracted numeric strings into a list of numbers.
    Treat any null or zero-representing text as zero.
    - **Strict Definition:** If the question names a procedure (e.g.,
    "population variance," "standard deviation," "moving average"),
    execute the arithmetic according to that standard definition.
    - For variance and standard deviation: Calculate the mean of the
    values. Compute the squared deviation of each value from the mean.
    Sum these squared deviations and divide by the total count of
    values (population formula). Take the square root of the result
    for standard deviation.
    - **Budget Awareness:** This step consumes no tool rounds, but it
    consumes the episode's success. If the data set is incomplete due
    to earlier budget constraints (truncation or grep limits), the
    calculation will be mathematically invalid or yield a wrong result
    that fails exact matching. Do not proceed to formatting until the
    arithmetic is complete.
6. Format and Emit Final Answer
    - Format the calculated result as a string using a standard
    decimal point. Apply rounding rules specified in the question only
    at the final output stage. Retain full floating-point precision
    for all intermediate calculations, including sums of logs, roots,
    and iterative growth computations. Round only the final result to
    the required precision. If the final rounded result does not match
    the ground truth, re-calculate using maximum available precision
    without any intermediate rounding.
    - Do not include thousands separators or currency symbols unless
    explicitly requested.
    - Emit the final answer using the tag `<answer>...</answer>`. This
    is a mandatory final step that must be executed immediately after
    any numerical calculation is complete. If the `grep` tool returns
    matching lines, immediately call `read` at the provided line
    numbers to extract the full table data, ensuring you do not skip
    the extraction step after a search.
    - **Budget Awareness:** This is the only action that prevents
    episode failure due to the round limit. If you have not extracted
    and calculated the correct value, or if you are unsure of the
    procedure's standard definition, do not emit a guess. Emitting a
    wrong answer scores nothing, and the episode ends. Ensure the
    format matches the normalized gold answer exactly (e.g., correct
    number of decimal places) before emitting. If the current round is
    the 24th and you have not yet emitted an answer, you must do so
    now; doing nothing causes immediate failure.
```

SpreadsheetBench.

Qwen-3.5-9B **Optimized skill $s^\star$**

```
Skill: Transform Workbook Based on Informal Instruction

1. Parse the Request
```

Qwen-3.5-9B **Optimized skill $s^\star$**

```
Translate the informal natural-language instruction into specific
technical requirements. Map terms like "lookup" or "sum" to available
`openpyxl` functions. Treat the input workbook as a fixed artifact;
recover column mappings from the provided preview labels without
assuming fixed positions. Use one-based indexing for all worksheet
coordinates when defining the answer range. Apply value-comparison
logic carefully to handle missing cells, empty strings, and case
differences, avoiding generic regex unless the instruction explicitly
requires text pattern matching within a cell. When performing any
comparison or arithmetic on cell values, convert the value to a
string, strip whitespace, attempt to parse it as a float, and default
to 0.0 if the conversion fails or the value is empty/null. Use this
coerced numeric value for all logic to ensure strings like '0' match
numeric 0 and empty strings are treated as 0.0. Then, verify that the
script preserves all existing sheets, formulas, and formatting outside
the target cells before writing the final output to the specified
path. Ensure the final script runs within the execution limits and
produces a valid workbook that the harness can immediately read and
validate.
2. Load and Verify Structure
Load the workbook at `INPUT_PATH` using `load_workbook` from
`openpyxl` to inspect available sheets and structure. Immediately
retrieve the list of merged cell ranges via `ws.merged_cells.ranges`
for each sheet to understand the layout before any data processing.
Verify that the sheet names and dimensions observed in the preview
match the actual loaded workbook, accounting for potential blank
spacer cells or multi-row headers. Ensure that no modifications are
made to cells within merged regions except for the top-left cell of
each range to preserve the existing artifact. Compare the loaded sheet
count and names against the expected preview to confirm the source
structure is intact before proceeding.
3. Map Source to Target Columns
Recover the source-to-target mapping by identifying a predicate column
and a return field from the workbook's headers, treating column labels
as clues rather than strict keys. Apply case-insensitive, fuzzy
matching between the source headers and the target field names,
prioritizing semantic similarity and data pattern validation over
exact string equality. Validate the selected columns by checking
sample values to ensure the source data matches the expected format
for the target field, such as verifying numeric patterns for ID
fields. Confirm the final mapping with the user if the automatic
matching produces ambiguous results or leaves fields unmapped. Use the
confirmed mapping to populate the answer range in the output workbook
while leaving all other sheets and content unchanged.

To recover column mappings, scan the header row of the relevant sheet
for terms like 'ITEM', 'BR', or others mentioned in the instruction.
Identify the column index for each term using case-insensitive
matching. Verify the match by checking sample values in the identified
column to ensure they match the expected data type (e.g., numeric for
IDs). Use the verified column indices for all subsequent lookups or
aggregations.

1. Select Calculation Method
```

Qwen-3.5-9B **Optimized skill $s^\star$**

```
Identify the logic type in the natural-language instruction, such as
conditional aggregation, grouping, or multi-criteria summarization. If
the task involves grouping by a specific attribute, apply a grouping
operation that respects the original row order, ensuring the
calculation logic matches the predicate and return fields implied by
the instruction. To identify these fields, scan the header row of the
relevant sheet for terms like 'ITEM', 'BR', or others mentioned in the
instruction, identifying the column index for each term using
case-insensitive matching. Validate the match by checking sample
values in the identified column to ensure they match the expected data
type (e.g., numeric for IDs). Use the verified column indices for all
subsequent lookups or aggregations. For aggregate calculations like
sums or counts across these groups, use a function that allows
defining multiple aggregation rules for different output columns
within the same operation. When the logic depends on sorted data to
ensure performance or correctness, sort the relevant data by the
grouping key before applying the aggregation, then reset the index to
maintain the original row sequence for the final write. When writing
sorted or filtered data back to a target range, use a sequential
counter variable initialized to the target start row; for each item in
the processed list, write to `target_start + counter` and increment.
Avoid creating new columns that exceed the requested answer range, as
the script must only modify the specified cells while preserving the
rest of the workbook's structure.
2. Execute Transformation
Do not merge or join data between sheets, as combining rows from
multiple sources discards unmatched records and violates the
requirement to preserve the input workbook's existing content.
Instead, calculate the requested values within the specified answer
range using the existing data structure, ensuring that any logic
relies only on the current sheet's data or explicit lookups rather
than altering the sheet's row count. When writing results, verify that
the target cells receive the correct literal values without
accidentally overwriting existing formulas or shifting the workbook's
structural boundaries. If the instruction implies filtering or
deleting rows, execute removals in reverse index order to prevent
offset errors, but prioritize preserving the original artifact unless
the specific edit surface requires a structural change. Always confirm
that the final output reads as a transformed copy where only the
designated answer cells differ from the input, leaving all other
sheets, columns, and untargeted rows intact.
3. Write to Target Cells
Use one-based row and column coordinates when addressing cells, as the
worksheet API treats these indices as matching Excel's native
numbering. Convert user-provided column letters to zero-based indices
only if the underlying library requires it, then increment the result
before writing to align with the one-based rule. When specifying the
target answer range, ensure that both the starting and ending
coordinates are explicitly one-based to avoid off-by-one errors in the
final values. Verify that any calculated literals or formulas assigned
to these specific cells do not inadvertently rely on zero-based
assumptions from external tools. Always treat the provided
`INPUT_PATH` and `OUTPUT_PATH` as the source and destination
artifacts, writing directly to the latter without creating
intermediate files. Confirm that the written values appear correctly
in the opened workbook before considering the calculation complete.
4. Validate Output
```

Qwen-3.5-9B **Optimized skill $s^\star$**

```
Use `load_workbook` with `data_only=True` to re-read the target cells
in the output workbook and compare them against the expected literal
values. This flag ensures the validation reads the cached results
written by your script rather than the formula strings that might
still exist in the cells. Do not save the workbook after loading it
with this flag, as that action would permanently replace the formulas
with their calculated values. Verify the specific target cells contain
the correct final values before the script exits to satisfy the
harness contract.

Budget Constraints

Runtime Wall-Clock Time
Limit: 120 seconds.
Cost of Exhaustion: The episode fails immediately with a timeout error
preventing any output generation.
Decision Point: Keep the entire script within the 120-second
wall-clock limit. If a calculation or workbook transformation requires
more time, avoid using interactive execution methods that have
stricter limits and instead run the logic as a background subprocess
to prevent the harness from terminating the run. Monitor execution
duration during development to ensure the script completes before the
deadline. Remember that a runtime error or missing output file also
counts against the attempt limit, so prioritize completing the
transformation within the fixed time window.

Number of Attempts
Limit: 30 attempts.
Cost of Exhaustion: The agent stops execution after the 30th failure,
resulting in a failed episode despite potential correctness.
Decision Point: Runtime errors consume one attempt from this pool. Any
calculation, sheet modification, or cell update that raises an
exception reduces the remaining count. If a transient failure occurs
during the workbook transformation, implement a retry strategy with
backoff to recover the state before exhausting the limit. When the
30th attempt is consumed by a runtime error, the script must terminate
immediately without attempting further recovery or partial operations.
The final output file must not be left in an incomplete state, and the
agent should verify that the script exited cleanly after the last
attempt to avoid hanging processes.

Memory Usage
Limit: Available system RAM (process termination threshold).
Cost of Exhaustion: The process is killed by the OS or harness due to
memory overflow, losing all accumulated state.
Decision Point: If the workbook size exceeds available memory and the
process risks termination, verify that `openpyxl` is present as a core
dependency before proceeding, since it handles the workbook directly.
To manage memory, avoid loading the entire file into a dataframe
unless the dataset is small, as `pandas` is only optional and requires
extra memory for DataFrame conversions. Instead of converting to a
larger format like SQLite, process the file incrementally or use
`openpyxl` utilities to read and write specific ranges without holding
the full artifact in memory. Ensure that any intermediate data
structures do not duplicate the workbook content, keeping the working
set within the 120-second execution limit and available RAM.
```

ALFWorld.

Qwen-3.5-9B **Optimized skill $s^\star$**

```
Skill: Household Object Search, Processing, and Placement

Objective
Execute the task defined in the opening observation ("Your task is to:
...") by locating the specified object(s), performing any required
intermediate processing (heating, cooling, cleaning), and placing the
result(s) into the named destination receptacle. The task is complete
only when the destination contains the required object(s). Partial
progress or failure to process/place yields a score of zero.

Constraints and Costs

- Budget: Exactly 50 steps per episode. Every action, including
invalid commands or redundant moves, consumes one step.
- Failure Condition: If the episode ends with fewer than 50 steps used
and the destination is not correctly filled, the score is zero. If
completion becomes impossible due to insufficient steps, continuing to
act is the only way to preserve any chance of success; stopping
guarantees failure.
- Two-Object Tasks: Both instances must reach the destination. They do
not need to be held simultaneously. The first may be processed and
placed before acquiring the second.
- Destination State: The destination does not need to be empty. If a
placement action is admissible, do not waste steps verifying
emptiness.
- Settled Locations: A receptacle is "settled" once its contents are
fully observed. Revisiting a settled location yields no new location
information and costs a step unless retrieving a previously observed
object or performing a necessary placement. Upon arriving at an
observed or known closed receptacle, you must execute the open command
immediately to reveal contents; failure to do so wastes steps and may
cause mission failure.

Procedure

1. Initialize Ledger and Calculate Remaining Budget
    - Maintain a persistent ledger recording every receptacle visited,
    its name, and its observed contents.
    - Calculate the remaining budget by subtracting the steps already
    taken (reported in the prompt) from the episode limit of 50.
    - Determine the minimum steps required to complete the task from
    the current state: estimated cost to find missing objects + cost
    of required processing (heat/cool/clean) + cost to reach
    destination + cost to place.
    - Decision Point: If the remaining budget is less than the
    estimated cost-to-go, the task is currently infeasible. Do not
    stop. Select the admissible action that maximizes the probability
    of future feasibility (e.g., finding a missing object or starting
    a processing chain).
2. Filter Admissible Actions for Exploration vs. Execution
    - Review the current admissible-action list.
    - Classify actions into:
        - Exploration: Moving to an unvisited receptacle.
        - Retrieval: Moving to a settled receptacle containing a
        required object.
        - Processing: Heating, cooling, or cleaning an object.
        - Placement: Moving an object to the named destination.
        - Invalid/Redundant: Actions that do not advance the state or
        search.
```

Qwen-3.5-9B **Optimized skill $s^\star$**

```
    - Rule: Never move to a settled receptacle for the purpose of
    searching for new objects. Such moves waste the step budget
    without reducing uncertainty.
3. Select Next Action Based on Budget and Task State
    - If required objects are missing and unvisited receptacles exist:
        - Select an unvisited receptacle that is reachable within the
        remaining budget while preserving enough steps for all
        subsequent processing and placement.
        - If no unvisited receptacle allows a feasible completion
        path, select an action that reduces uncertainty or initiates a
        necessary process, even if it seems suboptimal, because
        stopping guarantees failure.
    - If required objects are found but unprocessed:
        - Select the exact processing command from the admissible
        list. Copy the syntax exactly; do not invent generic commands.
        - Ensure the remaining budget covers this step plus the
        estimated cost to place the object.
    - If objects are processed and ready:
        - Select the placement command to move the object to the named
        destination.
        - Do not verify the destination is empty if a placement action
        is available.
    - If objects are placed but a second object is still required
    (two-object task):
        - Resume search for the second object using the updated
        ledger. Retain the knowledge of where the first object was
        placed.
4. Update Ledger and Replan
    - After every action, observe the result.
    - If the action reveals new contents in a receptacle, update the
    ledger immediately to mark it as settled.
    - Re-evaluate the remaining budget (50 - steps taken) and the
    cost-to-go.
    - If the state changes (e.g., an object is now available for
    processing), adjust the completion plan accordingly.
    - Continue emitting one admissible action per turn. Never emit an
    empty response or terminate early, as this consumes a step toward
    the limit without progress.
5. Termination Logic (Implicit)
    - The agent acts until the episode ends (step 50 reached) or the
    task is completed.
    - If the task is completed before step 50, the remaining steps are
    wasted but the score is 1.
    - If step 50 is reached and the task is incomplete, the score is
    0.
    - There is no "give up" state; the agent must always emit a valid
    admissible action until the episode limit is hit.
```

WebShop.

Qwen-3.5-9B **Optimized skill $s^\star$**

```
Skill: Purchase Matching Product

OBJECTIVE
Achieve a reward score of exactly 1.0 by purchasing a product that
satisfies all user-specified criteria. This requires four simultaneous
conditions:

1. The product's inherent properties must match the goal attributes.
```

Qwen-3.5-9B **Optimized skill $s^\star$**

```
2. All selectable options (size, colour, count, scent) must be set to
their requested values.
3. The product's listed price must be strictly under the stated price
ceiling.
4. The "kind factor" must be 1.0, determined by the relation between
the product title and the goal's category segments/nouns.

If any single condition fails, the recorded score for that episode is
0.0. Bounded Exploration Policy: If 3 or fewer steps remain, execute
`click[Buy]` immediately on the current valid candidate (price <
ceiling, exact options matched) or the lowest-price candidate from the
last viewed results if the current one is invalid or no valid
candidate exists, to maximize soft reward. If more than 3 steps remain
and a candidate with all options exactly matched is not found, inspect
one alternative; if that fails or steps drop to 3, trigger the
fallback. If no candidate with exact option match is found within the
step budget, purchase the candidate with the lowest price that has the
closest visible option match (or no options if necessary) to secure
partial reward. Ensure `click[Buy]` is executed before step 50.

STEP 1: Parse Instruction and Define Criteria
Extract the target product name, the list of required inherent
attributes, the list of required selectable options, and the hard
price ceiling from the single-sentence instruction.

- Decision Point: Establish the exact keyword string for the initial
search.
- Cost Awareness: Every action consumes one of the 50 available steps.
Missing the ceiling price or failing to set an option later guarantees
a score of 0.0, regardless of how many steps remain.

STEP 2: Search and Filter Initial Results
Execute `search[keywords]` using the defined keyword string.

- Upon receiving the results page (listing 10 items with ID, title,
price), evaluate each item immediately against known constraints.
- Filter Criteria:
    - The title must plausibly contain the required attributes (though
    final verification is post-purchase, the title is the only proxy
    available).
    - The listed price must be strictly under the price ceiling. If
    the price is at or above the ceiling, discard the item
    immediately; purchasing it yields a score of 0.0.
- Action: Use `click[value]` on the ID of a listing that meets the
price and title criteria to view the product page. Do not click items
that fail the price check.

STEP 3: Set Required Options and Pre-Purchase Verification
On the product page, inspect the `VALID ACTIONS` line for available
option values (e.g., size, colour).

- Decision Point: Select the specific option value that matches the
user's goal for each required attribute.
- Mandatory Verification: Before executing `click[Buy]`, perform two
strict checks: 1) Scan `VALID ACTIONS` for the exact goal option
string; if missing, abort purchase for this candidate. 2) Re-read the
product page price; if it exceeds the price ceiling, abort purchase
and search for another candidate.
- Action: If an option is present and matches the goal exactly,
execute `click[value]` to select it.
```

Qwen-3.5-9B **Optimized skill $s^\star$**

```
- Execution Constraint: Do not click "Buy" until every required option
has been successfully selected AND both the pre-purchase verification
checks (exact option match and price ceiling) pass. A purchase with
unselected options, incorrect options, or price > ceiling yields a
score of 0.0.

STEP 4: Verify Price and Title Before Purchase
Before initiating the final purchase, re-verify the current product
page details.

- Check Price: Confirm the listed price is still under the ceiling.
- Check Title: Confirm the title matches the product name and
attributes identified in Step 1.
- Decision Point: Assess if the "kind factor" potential exists. The
factor is 1.0 if the item comes from the same query, shares two+
category segments, or the title shares >1/5 of the goal's nouns. While
the exact calculation happens post-purchase, ensure the title is
sufficiently relevant to avoid a factor drop to 0.0, 0.1, or 0.5.
- Cost Awareness: Spending steps to search for a better match is
futile if the current item fails these checks, as the episode ends
with 0.0 upon purchase or step exhaustion. Do not waste steps on items
failing these visible checks.

STEP 5: Execute Purchase
If the product satisfies the visible price constraint, the option
constraints, and the title plausibility, and if the required options
are all set:

- Action: Execute `click[value]` on the "Buy" control found in the
`VALID ACTIONS` line.
- Outcome Logic: This action consumes one final step. If the purchase
is successful, the reward is calculated based on the four criteria. If
any criterion was missed (price too high, option missing, title
mismatch, or kind factor failure), the episode records a score of 0.0.
- Failure Mode: If the "Buy" control is not in `VALID ACTIONS` or if
the agent is unsure of the match, do not buy. Continuing to search is
only viable if steps remain (50 total). If steps are exhausted, the
episode ends with 0.0.

CONSTRAINTS AND LIMITS

- Step Budget: 50 actions total. Each action (search, click listing,
click option, click buy) costs 1 step. Reaching 0 steps remaining
before purchase results in a 0.0 score.
- Response Budget: 64 tokens per response. The parser extracts only
the final `search[...]` or `click[...]` command. Any text before this
command is ignored. Ensure the command is at the end of the response
to prevent the step from being wasted.
- Fixed Index: Repeating a search or clicking a previously viewed
listing returns the exact same state. Do not repeat actions that yield
no new information.
- Hidden Properties: The inherent product attributes are never
displayed on the page. They are only evaluated against the goal after
the purchase. You must proceed with the best available evidence (title
and price) to maximize the chance of a perfect match.
```