PROSE: COMPACT AND TRACEABLE SKILL EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents increasingly use skill documents to reuse experience across related tasks, but automatically maintaining these documents remains challenging. A fully autonomous loop must identify lessons that apply beyond a single task, incorporate them without uncontrolled growth, and retain an inspectable basis for each revision. We introduce PROSE (PROcedural Skill Evolution), an experience-driven method for training the skill document of a frozen agent. Inspired by group-relative optimization, PROSE contrasts repeated attempts of the same task for task-conditioned attribution, then uses a batch-level optimizer to consolidate recurring mechanisms across tasks. It turns these lessons into localized updates to a structured skill document, with programmatic evidence and budget checks that bound document growth and keep each change traceable to recorded experience. Starting from an empty document, PROSE improves held-out success by 9.7 to 46.9 points across SpreadsheetBench, AppWorld, and LiveMathematicianBench. It attains the highest mean on every reported metric among the evaluated skill-learning methods. It also produces the most context-efficient skill documents, using 43–88% fewer characters than competing methods. On AppWorld, this also holds when a single open-weight 27B model fills every role. We will release our implementation as open-source code. These results support skill documents as a compact and traceable adaptation layer for frozen agents.

1 INTRODUCTION

LLM agents increasingly face families of related tasks rather than isolated requests. A practical adaptation layer is an *agent skill*: a document of procedures, rules, and facts that the agent loads before acting (Anthropic, 2025; Gautam et al., 2026). Skills are still often written and maintained by hand, but a growing line of work updates them automatically from execution experience (Zhang et al., 2025; Wang et al., 2024; Suzgun et al., 2026; Yang et al., 2026). These methods show that an agent can improve its working knowledge without changing its model weights. Making the loop more autonomous, however, raises two broader questions: what should be learned from experience, and how should that knowledge be incorporated into the skill?

A successful trajectory can contain incidental choices, while a failed attempt may diverge at several points. The first question is which of these differences should become a lesson and drive a skill update. Comparing repeated attempts of the same task holds task difficulty and the current document fixed, making those differences easier to localize (Chen et al., 2026). Such task-conditioned attribution can still produce a lesson specific to one case. Aggregating experience across tasks reveals which lessons recur and are more likely to generalize (Yang et al., 2026; Ni et al., 2026), but pooled outcomes alone mix together tasks of different difficulty. A reliable learning protocol therefore needs both local comparisons that connect behavior to outcomes and cross-task evidence that supports generalization beyond a single case.

Even when a lesson is supported by both within-task comparisons and cross-task evidence, incorporating it into the skill still requires control and traceability. Flexible whole-document editing, especially when repeated over time, can accumulate redundant, conflicting, or narrowly scoped guidance. Because the skill document is loaded before each task, this content consumes context on every invocation and can make useful guidance harder to preserve (Yan et al., 2026; Moll et al., 2026; Hajimiri et al., 2026). Outcome-based validation gates test whether a candidate artifact helps

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.

and can steer later revisions, but a score alone does not record why a particular edit was proposed or accepted (§2). A reliable solution must therefore constrain how the document changes while preserving the evidence behind each change. Existing systems each provide some of these pieces: same-task contrast, cross-task aggregation, bounded document operations, candidate validation, and evidence links. No existing method for a frozen agent connects task-conditioned attribution to verified cross-task generalization and uses both to make localized, evidence-linked changes to the document.

We introduce PROSE (PROcedural Skill Evolution), an experience-driven method for training the skill document of a frozen agent (§3, Fig. 1). Inspired by group-relative policy optimization (Shao et al., 2024), PROSE attempts each task several times under the same document, producing task-conditioned comparisons without applying a gradient to the text. From these comparisons, it identifies candidate lessons together with their supporting trajectory evidence, which host-side code verifies against the recorded trajectories, then consolidates recurring lessons across tasks. It proposes typed operations over a structured skill document, and host-side licensing checks evidence, support, and budgets before any change is applied, so each accepted change stays localized and traceable to the experience that justified it. Validation scores the committed document rather than gating each edit.

We evaluate PROSE on spreadsheet manipulation (SpreadsheetBench (Ma et al., 2024)), code-as-action API tasks (AppWorld (Trivedi et al., 2024)), and tool-free research mathematics (LiveMathematicianBench (He et al., 2026)). Starting from an empty document, PROSE improves held-out success by $+46.9$, $+22.0$ scenario-level, and $+9.7$ points, respectively. It achieves the highest mean on every reported metric among the evaluated skill-learning methods and the most context-efficient skill artifacts, with 43–88% fewer characters. The AppWorld gain persists on unseen apps, and PROSE attains the highest mean when one open-weight 27B model fills every role (§4–4.2).

Our contributions:

- We combine task-conditioned comparisons with batch-level consolidation and admit a lesson only after verified evidence shows it recurring across distinct tasks (§3.2–3.4).
- We introduce a structured skill state with stable entry identifiers, typed operations over it, and host-side evidence licensing, which together bound document growth, check the evidence and support required for each operation, and preserve an inspectable record of every accepted change (§3.1–3.4).
- Across three benchmarks, PROSE achieves the highest means and the most context-efficient skill artifacts; ablations isolate mixed-group contrast and typed operations with evidence licensing, and the result reproduces with a self-hosted 27B model (§4–4.4).

2 Related work

Persistent skill-document optimization. Agents can reuse past interactions as verbal feedback (Shinn et al., 2023), cross-task insights (Zhao et al., 2024), executable code (Wang et al., 2023), or induced workflows (Wang et al., 2024). More recent systems maintain persistent context artifacts: ReasoningBank stores strategies for retrieval (Ouyang et al., 2025), ACE incrementally refines a structured playbook (Zhang et al., 2025), and Trace2Skill consolidates many trajectories into a portable skill directory (Ni et al., 2026). SkillOpt is the closest setting to ours: it treats one external skill document as the trainable state of a frozen agent (Yang et al., 2026).

Evidence for reusable lessons. Methods differ in how they connect an observed behavior to an outcome. Trace2Skill analyzes a broad trajectory collection jointly, while SkillOpt learns from scored rollouts gathered across tasks. Contrastive methods provide a more local signal: Training-Free GRPO distills semantic advantages from grouped rollouts and merges them into an experience library at the batch level (Cai et al., 2025), CORE extracts insights from successful and failed reasoning traces (Nasvytis et al., 2026), and SkillCAT compares same-task success–failure pairs before replaying candidate patches (Chen et al., 2026). Together, these methods provide local contrasts and cross-task aggregation. PROSE uses both and requires verified cross-task recurrence before a lesson enters the document.

Controlling and tracing document updates. Document updates can be controlled at the operation, candidate, or trajectory level. ACE uses structured incremental updates; SkillOpt applies bounded

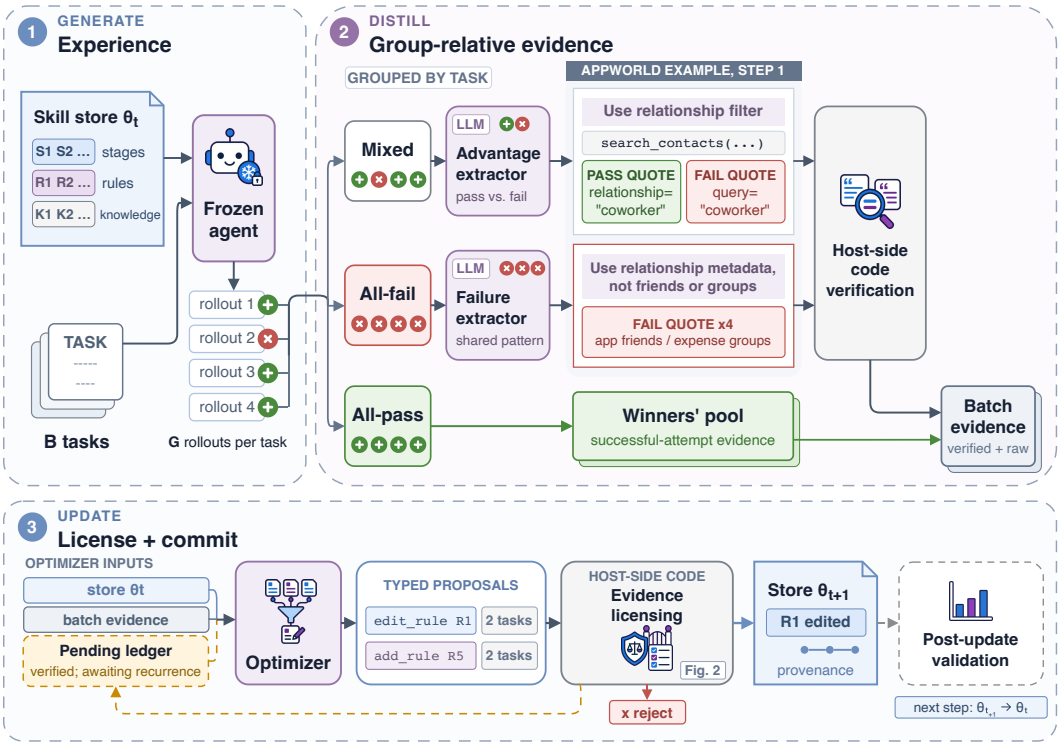


Figure 1: One PROSE training step. A frozen agent attempts each task $G{=}4$ times under the current skill. Mixed groups enter an advantage extractor, all-fail groups a failure extractor, and all-pass groups bypass extraction and retain full trajectories in a winners' pool. Host-side code verifies cited trajectory spans and stamps their recorded outcomes before a batch-level optimizer proposes typed edits. Evidence licensing admits supported edits into the next structured skill state; validation scores each committed version and can revert a harmful update. Colors: blue trained skill state, purple model calls, gray host-side code, amber pending ledger. Cards show examples from Step 1 of the AppWorld run.

add, delete, and replace operations and accepts a candidate that improves held-out performance; and SkillCAT replays candidate patches on their source tasks (Zhang et al., 2025; Yang et al., 2026; Chen et al., 2026). SkillAudit audits paired trajectories with and without a candidate skill and a fixed verifier flags harmful revisions (Gao et al., 2026). MSCE retains evidence links, applicability boundaries, and reliability estimates in its learned skills (Tang et al., 2026). Outcome tests estimate whether a candidate helps, and source links record why. PROSE does both: validation scores each committed version, and licensing ties each typed operation to verified spans.

Learning the curator. Another line trains the model that manages the skill collection. SkillOS pairs a frozen executor with an RL-trained curator and uses grouped task streams plus composite rewards to assign downstream feedback to curation decisions (Ouyang et al., 2026). SkillRise and SAPO likewise optimize a policy for skill evolution, with SAPO co-evolving the policy and skill bank (Yao et al., 2026; Zhang et al., 2026). PROSE studies the complementary fixed-model setting, where all model roles remain unchanged and only the external document evolves.

3 METHOD

PROSE uses a three-module *Generate–Distill–Update* design (Fig. 1). *Generate* attempts each task repeatedly under the same document, which yields task-conditioned comparisons. *Distill* turns those comparisons into verified, quote-backed evidence that can be consolidated across tasks. *Update*

admits lessons into a structured document through typed, evidence-licensed operations, which keeps the document compact and every change traceable. We define the learned state, then each module.

3.1 Problem setup and typed skill state

The *worker*, the frozen agent that loads the skill, is fixed throughout training, with weights, tools, harness, and a task-interface prompt that are identical across conditions (App. A.1). The skill document is the only trained state, and the benchmark's binary grader is the only learning signal.

The state θ, the skill store in Fig. 1, is a structured store with three entry types: an ordered procedure of stages, condition–action rules scoped to a stage or globally, and background knowledge. Each entry has a stable identifier; App. D shows a complete learned store. Rather than rewriting the document wholesale, the optimizer proposes typed operations that add, edit, move, or drop named entries (App. B.2). A fixed renderer ρ converts the store into the document shown to the worker. The typed state does not judge whether a lesson is correct. It makes the target and effect of each proposed change explicit, and stable identifiers let evidence and later revisions attach to the same entry.

Let $\pi(\cdot \mid x, d)$ denote the worker on task x with document d, and let $R(x, \tau) \in \{0, 1\}$ be the reward for trajectory τ. Training maximizes the worker's expected reward over the document state:

$$\max_{\theta} \; J(\theta) \;=\; \mathbb{E}_{x \sim \mathcal{D}} \, \mathbb{E}_{\tau \sim \pi(\cdot \mid x,\, \rho(\theta))}[R(x, \tau)] \,. \tag{1}$$

The objective matches the expected-reward objective used in policy optimization, but the optimized parameter is discrete text rather than model weights. This difference motivates a textual evidence-and-edit procedure in place of a gradient update.

3.2 Generate and distill: three evidence paths

At step t, PROSE samples a batch B_t and attempts each task G times using the same rendered skill document $\rho(\theta_t)$. This grouping is inspired by GRPO's use of repeated rollouts of one prompt to obtain a task-specific baseline (Shao et al., 2024; Yu et al., 2025). With binary rewards, group-relative advantage is informative for a *mixed* group, where some attempts pass and others fail. Because the task and document are fixed within the group, the comparison helps localize behavioral differences without conflating them with task difficulty. PROSE uses this grouping only as an evidence structure.

For a mixed group, the advantage extractor (Fig. 1, panel 2; App. B.1) returns an audit of how existing skill entries were used, concise mechanisms locating where passing and failing behavior diverged, and a stage trace summarizing the procedural path of each rollout. Each mechanism describes the divergence and the behavior on both sides, cites verbatim spans from at least one passing and one failing trajectory, and receives one diagnostic hypothesis: `ignored` when a winner followed applicable guidance that a loser ignored, `missing` when no existing entry would have prevented the failure, `wrong` when a loser followed applicable guidance but still failed, or `procedure_gap` when the procedure lacks or misstates a necessary phase. An optional edit sketch accompanies the mechanism. Host-side code verification (Fig. 1, panel 2) matches every citation to its trajectory and stamps its recorded outcome; unmatched citations are discarded. The optimizer treats diagnoses and sketches as hypotheses; they do not by themselves authorize an edit.

An all-fail group has no within-task contrast, so a separate failure extractor (Fig. 1, panel 2) reads all G failed trajectories jointly. It reports a mechanism as convergent only when verified quotations show the same gap in at least two attempts; otherwise, it returns a small set of divergent mechanisms or no mechanism. It applies the same diagnostic taxonomy and returns fail-only citations, the same per-rollout stage traces, an optional edit or stage-level sketch, and an advisory note identifying the attempt that progressed furthest and what stopped it. The G attempts measure recurrence within one task and therefore count as one task in downstream support.

An all-pass group bypasses extraction. PROSE retains its full trajectories and rollout metadata in the current batch's winners' pool (Fig. 1, panel 2). The optimizer draws on these successful attempts when it needs wording for a rule or a recurring procedure stage; every cited span undergoes the same host-side verification. These citations supply wording for successful behavior; they do not count toward support for a failure mechanism or substitute for a pass/fail contrast.

3.3 UPDATE I: CROSS-TASK GENERALIZATION AND PROPOSALS

Task-conditioned evidence may be specific to one case, so a single batch-level optimizer (Fig. 1, panel 3) receives the current skill state θ_t, verified mechanisms from mixed and all-fail groups, the current batch's winners' pool, and live pending evidence from earlier batches.

The pending ledger (Fig. 1, panel 3) is host-managed cross-batch memory for verified evidence that was insufficient by itself to license an update. It stays outside the worker-visible skill but is available to the optimizer later. Each entry stores a concise lesson, its origin task and step, implicated target and diagnosis, and verified citation references—not full trajectories. An entry stays live for a rolling window of update steps unless a licensed operation consumes it, and newer evidence from the same task for the same target replaces the earlier entry; §4.1 gives the capacity and window.

The optimizer groups mechanisms only when they describe the same behavioral mistake and can be addressed by the same general instruction; otherwise, it keeps them separate. When proposed wording requires successful behavior, the optimizer first uses a successful attempt from the corresponding mixed group and otherwise selects an all-pass trajectory that reached the same decision point.

Using the grouped mechanisms and any relevant pending evidence, the optimizer assigns the final diagnosis and returns a ranked list of typed proposals. **Add** creates an entry, **Edit** changes its content, **Move** changes its placement or the procedure order, and **Drop** removes it. Each proposal specifies the operation, stable target, proposed change, cited trajectory spans, and any pending entries it draws on.

3.4 UPDATE II: HOST-SIDE LICENSING AND APPLICATION

The optimizer chooses the operation, target, proposed change, and supporting evidence. Host-side licensing (Fig. 1, panel 3) verifies these claims and determines whether the proposal may update the skill. Let E denote the evidence-routing rule from §3.2: it applies the advantage extractor to mixed groups, the failure extractor to all-fail groups, and passes the full trajectories of all-pass groups to the winners' pool. Let $\mathcal{O}$ denote the optimizer described above and Λ_t the live pending entries. The state update is

$$\theta_{t+1} \;=\; \theta_t \;\oplus\; \Pi_{\text{lic}}\Big(\,\mathcal{O}\big(\theta_t,\; \bigcup\nolimits_{x\in B_t} E\big(\{(\tau_i^x, R(x,\tau_i^x))\}_{i=1}^{G}\big),\; \Lambda_t\big)\Big), \tag{2}$$

where τ_i^x is attempt i on task x and $R(x,\tau_i^x)$ is the binary reward used in Eq. 1. Here $\mathcal{O}$ produces proposals, Π_{lic} licenses them, and $\oplus$ applies the surviving operations in ranked order. The union over B_t makes the batch the update unit. Figure 2 expands Π_{lic} into three stages.

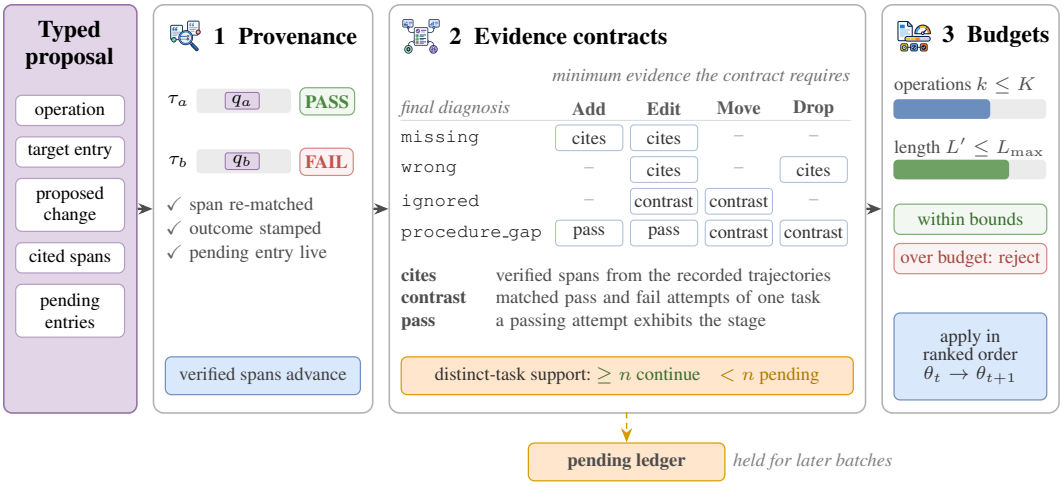


Figure 2: Host-side evidence licensing. Each typed proposal passes provenance verification, the contract selected by its diagnosis and operation, the support floor, and the budgets. Cells name the minimum evidence a contract requires; richer evidence is admissible and a dash marks a disallowed pairing. Evidence below the support floor stays pending; licensed proposals produce θ_{t+1}.

Stage 1 re-matches each cited span to its source trajectory, stamps its recorded outcome, and confirms that any cited pending entry is still live; only verified evidence advances.

Stage 2 applies the evidence contract selected by the final diagnosis and proposed operation. A `missing` diagnosis can support **Add** or **Edit** when required guidance is absent or incomplete. A `wrong` diagnosis can support **Edit** or **Drop** when existing guidance leads behavior in the wrong direction. An `ignored` diagnosis can motivate **Edit** or **Move**, but it cannot introduce new content and requires a matched within-task contrast showing a successful attempt following the guidance and a failed attempt ignoring it. A `procedure_gap` diagnosis invokes the stage-level contract: adding or rewording a stage requires a successful attempt that exhibits the phase, while removing or reordering stages requires matched successful and failed attempts of the same task.

Within these contracts, **Add** requires the same missing lesson across distinct tasks, **Edit** requires evidence that the existing entry should be replaced, and **Drop** requires evidence for removal rather than rewriting or relocation; **Move** and **Drop** may instead cite the usage record accumulated for the target entry across earlier updates (App. B.2). In the `procedure_gap` row of Fig. 2, the four operations act on procedure stages. Host-side code verifies the cited spans, recorded outcomes, required contrasts, and cited pending entries, then counts support over distinct tasks; repeated attempts on one task count once. Verified evidence below the support floor returns to the pending ledger.

In Stage 3, host-side code enforces the per-step operation budget $k \leq K$ and the length cap $L' \leq L_{\max}$. Proposals passing all three stages are applied in ranked order to produce θ_{t+1} (App. B.2; Fig. 3 traces one accepted edit); §4.1 reports the support floor n and the budgets K and $L_{\max}$.

3.5 Post-update validation

After each committed update, PROSE evaluates the new skill on the held-out validation tasks and records its score (Fig. 1, panel 3). Each update therefore enters the version history before its validation score is observed. The score sequence shows whether performance keeps improving, levels off, or declines, and thus when further update steps no longer help and which saved version to retain. We fix the step budget (4, 6, and 8 steps on the three benchmarks) and retain the final committed version; both rules were set before test evaluation (§4.1).

One automatic safeguard protects against large regressions: PROSE reverts the newest version when its score falls more than z=2.5 estimated standard errors below the mean of up to three preceding validation sweeps, which catches unusually harmful updates while allowing ordinary fluctuations.

4 Experiments

4.1 Experimental setup

Benchmarks. We evaluate on three task families with different interfaces. **SpreadsheetBench** (Ma et al., 2024) contains real-world spreadsheet transformations graded by cell-level comparison with a reference file (280 held-out tasks). **AppWorld** (Trivedi et al., 2024) evaluates code-as-action agents in a simulated collection of apps and APIs. We report task goal completion (TGC) and the stricter scenario goal completion (SGC), which requires solving every task variant in a scenario. Test-normal follows the training distribution; every test-challenge task uses an app absent from training and the split is harder on average (App. A.1). **LiveMathematicianBench** (LiveMathBench; He et al., 2026) contains recent research-level multiple-choice mathematics questions without tools (125 held-out questions). All reported metrics are success rates in percent.

Implementation and training configuration. In the main experiments, the worker is GPT-5.6-luna at medium effort in the Claude Code harness (v2.1.152). The advantage and failure extractors are GPT-5.6-sol at medium effort through direct model calls, and the optimizer is DeepSeek v4-pro at high effort in the same harness. Each step draws a batch B_t without replacement from the training split, reshuffled every epoch, and runs G=4 attempts per task. We use $|B_t|$=40, 30, and 35 on SpreadsheetBench, AppWorld, and LiveMathematicianBench, so an epoch spans two, three, and one update step, respectively; on LiveMathematicianBench the batch is the entire training split. The winners' pool is rebuilt each step from that batch's all-pass groups. We evaluate the final committed version after 4, 6, and 8 steps, fixed before test evaluation (App. A.2). The support floor is n=2

distinct tasks, each step admits at most $K=12$ operations, and the rendered skill document is bounded at $L_{\max}=1{,}600$ tokens (about $6{,}400$ characters). The pending ledger retains at most 90 live entries over rolling windows of three, four, and three update steps on the three benchmarks.

Baselines. **No skill** evaluates the frozen worker with only the shared interface contract. **LLM-generated skill** asks the same model used as PROSE's optimizer to produce one skill document in a single call from a high-level benchmark description, without tasks, trajectories, or scores (App. B.3). **Trace2Skill** (Ni et al., 2026) and **SkillOpt** (Yang et al., 2026) are published methods that learn a skill document from the worker's experience. We run both through the same worker, harness, splits, and evaluation protocol as PROSE (App. A.3). All conditions begin without an authored skill.

Small open-weight model stack. To test whether the results depend on frontier models, we repeat the AppWorld experiment with one self-hosted Qwen3.6-27B model in every role: worker, both extractors, and optimizer for PROSE, and the meta-level reasoner for Trace2Skill and SkillOpt. The test set, worker prompt, batch seed, empty start, and licensing thresholds are unchanged (App. C.3).

4.2 Main results

Table 1: Held-out success (%) for one frozen GPT-5.6-luna worker (mean ± sd over 3 trials). Subscripts show paired changes from no skill; bold and underline mark the highest and second-highest column means. †Prompt-guarded run (App. A.3). Paired comparisons are in App. C.1.

	SpreadsheetBench	AppWorld test-normal		AppWorld test-challenge		LiveMathBench
skill source		TGC	SGC	TGC	SGC	
No skill	38.0 ± 1.6	78.0 ± 1.0	63.1 ± 2.7	67.5 ± 1.9	49.6 ± 3.8	33.2 ± 2.7
LLM-generated	$45.9 \pm 0.5_{+8.1}$	$78.4 \pm 1.2_{+0.4}$	$65.5 \pm 3.7_{+2.4}$	$76.4 \pm 2.7_{+8.9}$	$59.7 \pm 5.7_{+10.1}$	$36.4 \pm 3.4_{+3.0}$
Trace2Skill	$41.6 \pm 1.3_{+3.6}$	$80.8 \pm 1.0_{+2.8}$	$73.2 \pm 1.6_{+10.1}$	$70.7 \pm 0.8_{+3.2}$	$51.1 \pm 1.4_{+1.4}$	$36.0 \pm 0.7_{+3.0}$
SkillOpt	$\underline{84.3 \pm 0.0}_{+46.6}$	$\underline{87.9 \pm 0.3}_{+9.9}$	$\underline{81.0 \pm 2.7}_{+17.9}$	$\underline{81.7 \pm 1.2}_{+14.2}$	$\underline{66.2 \pm 1.2}_{+16.6}$	$\underline{38.5 \pm 2.0}^{\dagger}_{+4.9}$
PROSE	$\mathbf{84.7 \pm 1.1}_{+46.9}$	$\mathbf{88.5 \pm 0.3}_{+10.5}$	$\mathbf{85.1 \pm 2.1}_{+22.0}$	$\mathbf{83.3 \pm 0.6}_{+15.8}$	$\mathbf{69.3 \pm 2.2}_{+19.7}$	$\mathbf{43.2 \pm 0.8}_{+9.7}$

PROSE attains the highest mean across the evaluated methods. From an empty skill document, PROSE improves every reported column, from $+9.7$ points on LiveMathematicianBench to $+46.9$ on SpreadsheetBench. SkillOpt, the strongest baseline, trails PROSE by 0.4 to 4.3 points, while PROSE's documents are 1.8–$8.6\times$ smaller (Table 3) and carry a traceable revision history.

Table 2: AppWorld results under the small open-weight model stack (mean ± sd over 3 trials; Qwen3.6-27B in every role, §4.1). Subscripts show paired changes from no skill; bold and underline mark the highest and second-highest means.

	AppWorld test-normal		AppWorld test-challenge	
skill source	TGC	SGC	TGC	SGC
No skill	79.2 ± 2.7	62.5 ± 5.4	69.5 ± 0.5	45.1 ± 1.7
LLM-generated	$84.7 \pm 1.4_{+5.6}$	$72.0 \pm 2.7_{+9.5}$	$77.1 \pm 1.9_{+7.5}$	$56.4 \pm 5.4_{+11.3}$
Trace2Skill	$82.5 \pm 0.3_{+3.4}$	$65.5 \pm 4.1_{+3.0}$	$75.7 \pm 0.1_{+6.2}$	$55.2 \pm 0.8_{+10.1}$
SkillOpt	$\underline{87.5 \pm 2.1}_{+8.3}$	$\underline{75.6 \pm 5.5}_{+13.1}$	$\underline{78.2 \pm 1.6}_{+8.6}$	$\underline{59.7 \pm 2.6}_{+14.6}$
PROSE	$\mathbf{91.9 \pm 0.3}_{+12.7}$	$\mathbf{83.3 \pm 1.0}_{+20.8}$	$\mathbf{79.5 \pm 0.6}_{+9.9}$	$\mathbf{61.6 \pm 2.2}_{+16.6}$

The result holds with a small open-weight model stack. Table 2 repeats the AppWorld comparison with Qwen3.6-27B in every role. On test-normal, PROSE improves the no-skill worker by $+12.7$ TGC and $+20.8$ SGC, close to the frontier-model stack's $+10.5$ and $+22.0$, and again attains the highest mean on both metrics. SkillOpt remains the strongest baseline and trails PROSE by 4.4 TGC and 7.7 SGC points, a larger gap than under the frontier-model stack. On test-challenge, PROSE again attains the highest mean on both metrics. PROSE's document is again the smallest (Table 3). This result holds for Qwen3.6-27B on AppWorld; broader evaluation across open-weight models and benchmarks remains future work.

Table 3: Worker-visible characters of the Table 1 and 2 artifacts; subscripts are the ratio to PROSE.

skill source	SpreadsheetBench	AppWorld	LiveMathBench	AppWorld, open-weight stack
LLM-generated	$9{,}043_{\times 5.6}$	$5{,}949_{\times 2.3}$	$4{,}403_{\times 2.9}$	$5{,}949_{\times 2.1}$
Trace2Skill	$6{,}281_{\times 3.9}$	$13{,}827_{\times 5.4}$	$8{,}768_{\times 5.8}$	$4{,}925_{\times 1.7}$
SkillOpt	$7{,}735_{\times 4.8}$	$4{,}554_{\times 1.8}$	$13{,}028_{\times 8.6}$	$8{,}858_{\times 3.1}$
PROSE (trained)	**1,611**	**2,570**	**1,510**	**2,830**

PROSE learns context-efficient skill documents while retaining traceability. Because the skill document is loaded before each task, its size is recurring context overhead. We measure this footprint using worker-visible characters (Table 3). Across all method–benchmark comparisons, PROSE's final artifacts use 43–88% fewer characters than competing artifacts while PROSE attains higher means in the corresponding result columns. The length bound of §4.1 is a safety limit rather than the source of this compactness: the largest final document uses 44% of it. The artifacts also differ qualitatively: Trace2Skill and SkillOpt retain narrow task-specific rules and examples, such as literal spreadsheet filters, AppWorld task patterns, and corrections for specific mathematical questions. PROSE instead consolidates recurring mechanisms across tasks and requires support from n distinct tasks for quote-backed changes, while move and drop may instead use evidence accumulated for the same entry across earlier updates. This design favors shared procedures over isolated corrections. Compactness does not remove the evidence behind each change: host-side records link every accepted revision to its cited spans, recorded outcomes, and licensing decision (§4.4).

4.3 Ablations

To identify which design choices contribute to PROSE's gains, we compare the full method with two variants on AppWorld under the frontier-model and small open-weight model stacks (§4.1). The variants remove mixed-group contrast or replace typed operations and evidence licensing with free editing. Each variant uses the same rollout batches and training compute as its full-method reference. Each condition is trained once, and Table 4 reports three evaluations of its final document.

Table 4: AppWorld test-normal results for the full method and two ablations under both model stacks (mean ± sd over 3 trials). Subscripts show paired changes from full PROSE within each stack. The two model stacks use different workers and are not compared. Paired comparisons are in App. C.1.

	Frontier-model stack		*Small open-weight model stack*	
Variant	TGC	SGC	TGC	SGC
Free editing	$83.7 \pm 1.2_{-4.8}$	$73.2 \pm 0.0_{-11.9}$	$89.7 \pm 1.4_{-2.2}$	$78.6 \pm 3.6_{-4.9}$
No mixed-group contrast	$86.9 \pm 1.2_{-1.6}$	$79.2 \pm 2.1_{-6.0}$	$87.7 \pm 1.5_{-4.2}$	$76.2 \pm 4.1_{-7.3}$
PROSE (full)	$\mathbf{88.5 \pm 0.3}$	$\mathbf{85.1 \pm 2.1}$	$\mathbf{91.9 \pm 0.3}$	$\mathbf{83.3 \pm 1.0}$

Mixed-group contrast improves scenario completion. The variant without mixed-group contrast discards mixed groups before optimization while retaining the all-fail extractor and cross-task winners' pool, so the optimizer sees no same-task comparison. Under the frontier-model stack, removing this evidence lowers SGC by 6.0 points; the 1.6-point TGC difference is smaller. Under the small open-weight model stack, it lowers TGC by 4.2 points and SGC by 7.3 points. The decrease under both stacks indicates that mixed-group contrast adds information beyond pooled evidence.

Typed operations and evidence licensing outperform free editing. The free-editing variant receives the same host-verified citations as PROSE but replaces typed operations and their licensing with unrestricted rewriting. Under the frontier-model stack, free editing lowers TGC by 4.8 points and SGC by 11.9 points. Under the small open-weight model stack it is also lower on both metrics, and it produces a document roughly twice as large without improving performance.

Performance is stable across the tested group sizes. Varying $G \in \{2, 4, 6\}$ with fixed batches leaves TGC within 1.4 and SGC within 3.6 points of the $G{=}4$ operating point, while raising G from 4 to 6 adds two mixed groups for 50% more training rollouts (Table 7, App. C.2), so we retain $G{=}4$.

4.4 TRACING AN ACCEPTED RULE REVISION

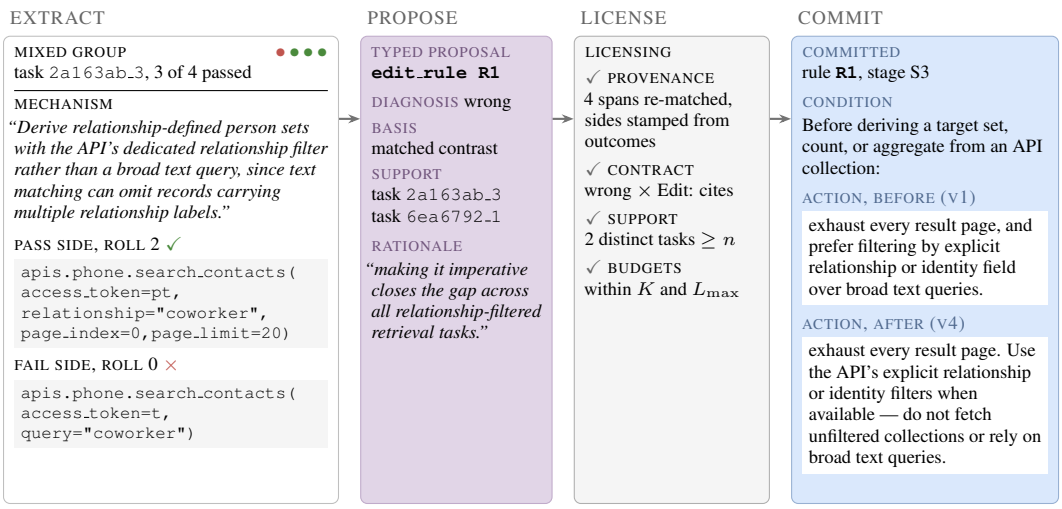


Figure 3: An accepted AppWorld rule revision from evidence to commit. A mixed rollout group yields an evidence-backed candidate, and the batch optimizer combines it with evidence from another task to propose `edit_rule R1`. Host-side licensing rechecks the cited spans, recorded outcomes, and applicable support requirements before committing the updated rule.

Figure 3 traces one accepted revision end to end. A mixed group contrasts a successful relationship-filtered API call with a failed broad text query. The optimizer combines this mechanism with evidence from a second task and proposes a typed revision to R1. Licensing re-matches the citations, confirms their outcomes, and checks support before committing. Because the proposal targets the stable identifier R1, the record ties the supporting trajectories and licensing decision to the exact change, which a whole-document version history does not provide (full skill in App. D).

5 CONCLUSION

Automatically evolving a skill document requires attributing outcomes within tasks, identifying lessons that hold across tasks, and admitting them while bounding growth and recording provenance. PROSE connects the three in one protocol: task-conditioned comparisons yield candidate mechanisms, cross-task consolidation generalizes them, and licensing constrains typed operations.

Across three benchmarks, PROSE improves the no-skill worker by 9.7–46.9 points and attains the highest mean on every reported metric among the evaluated skill-learning methods. Its skill artifacts use 43–88% fewer characters than those produced by competing methods. The AppWorld gain extends to apps absent from training, and PROSE still attains the highest mean with one self-hosted 27B model in every role. Typed entries and recorded evidence also make each accepted revision traceable to the spans and licensing decision behind it. The results show that a skill document can serve as a compact, trainable, and traceable adaptation layer for a frozen agent.

Limitations and outlook. The current method assumes an automatic binary grader and recurring failure mechanisms. Scalar rewards could support within-task ranking, but subjective outcomes would require a calibrated judge (Ye et al., 2026). Host-matched spans establish that the cited evidence exists; they do not verify that the optimizer interpreted it correctly. The distinct-task support floor filters isolated observations but may leave rare, useful changes pending. Grouped rollouts also increase training cost (App. C.4), and each trained condition is a single offline run for one worker and harness, although a second worker reproduces the direction of every gain (App. C.3). Future work should study graded and judge-based feedback, adaptive support requirements, training-run stability, and continual transfer across workers and task streams.

AI USE STATEMENT

LLMs are components of the experiments: they serve as the worker, extractors, and optimizer of PROSE, perform the meta-level reasoning of the Trace2Skill and SkillOpt baselines, and write the LLM-generated skill baseline (§4.1). The authors designed the algorithm and the system and used Claude and ChatGPT to implement them in code, and to draft and edit the manuscript, create figures, and format references. We reviewed all AI-assisted work; AI-written code was tested and its results reproduced by rerunning the recorded configurations. We take responsibility for the final content of this work.

ETHICS STATEMENT

This work uses public benchmarks and simulated environments and involves no human subjects or personal data. The method produces instructions that steer an agent on every subsequent task, so an error or a benchmark-specific shortcut in a learned document propagates to all of them. A learned document should therefore be reviewed before deployment on real systems; the released provenance records tie each accepted revision to its evidence to support that review.

REPRODUCIBILITY STATEMENT

We will release the implementation and artifacts needed to reproduce every reported number: the training and evaluation framework, prompt templates, configurations, split manifests, learned skill documents with their provenance records, and per-task evaluation outputs. §4.1 and App. A describe the benchmarks, splits, model bindings, training schedules, and baseline protocol. App. B gives the prompts and the typed-operation and licensing implementation, App. C the statistical protocol and all paired comparisons, and App. D a complete learned document.

REFERENCES

Anthropic. Equipping agents for the real world with Agent Skills. Anthropic engineering report, 2025. URL https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills. October 16, 2025.

Yuzheng Cai, Siqi Cai, Yuchen Shi, Zihan Xu, Lichao Chen, Yulei Qin, Xiaoyu Tan, Gang Li, Zongyi Li, Haojia Lin, Yong Mao, Ke Li, and Xing Sun. Training-free group relative policy optimization. *arXiv preprint arXiv:2510.08191*, 2025.

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

Haowen Gao, Haoran Chen, Can Wang, Shasha Guo, Liang Pang, Zhaoyang Liu, Huawei Shen, and Xueqi Cheng. SkillAudit: Ground-truth-free skill evolution via paired trajectory auditing. *arXiv preprint arXiv:2606.14239*, 2026.

Srishti Gautam, Arjun Radhakrishna, and Sumit Gulwani. SkillAxe: Sharpening LLM-authored agent skills through evaluation-guided self-refinement. *arXiv preprint arXiv:2606.10546*, 2026.

Sina Hajimiri, Masih Aminbeidokhti, Jose Dolz, Ismail Ben Ayed, Issam H. Laradji, Spandana Gella, and Nicolas Gontier. Are online skill and memory modules always worth their tokens? a budget-constrained study of web agents. *arXiv preprint arXiv:2606.15017*, 2026. EMNLP 2026.

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

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 *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2024. Spotlight.

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. *arXiv preprint arXiv:2605.29668*, 2026. EMNLP 2026.

Linas Nasvytis, Simon Jerome Han, Ben Prystawski, Satchel Grant, Noah D. Goodman, and Judith E. Fan. CORE: Contrastive reflection enables rapid improvements in reasoning. *arXiv preprint arXiv:2605.28742*, 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 preprint arXiv:2603.25158*, 2026.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140*, 2025. ICLR 2026.

Siru Ouyang, Jun Yan, Yanfei Chen, Rujun Han, Zifeng Wang, Bhavana Dalvi Mishra, Rui Meng, Chun-Liang Li, Yizhu Jiao, Kaiwen Zha, Maohao Shen, Vishy Tirumalashetty, George Lee, Jiawei Han, Tomas Pfister, and Chen-Yu Lee. SkillOS: Learning skill curation for self-evolving agents. *arXiv preprint arXiv:2605.06614*, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y.K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Shortcut Research Team. SpreadsheetBench verified: A curated evaluation set. Shortcut (Fundamental Research Labs) blog post, in collaboration with the SpreadsheetBench authors, 2025. URL https://shortcut.ai/blog/posts/spreadsheetbench-verified. December 2, 2025; 400-task human-validated subset of the original 912.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic Cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, Rabat, Morocco, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.eacl-long.333.

Bo Tang, Yang Zhang, Guomian Zhuang, Wenqiang Wei, Gaoyang Zheng, Lindong Xie, Yanchao Tan, Feiyu Xiong, Qingyu Yang, Edward Chung, and Zhiyu Li. From memory to skills: Evidence-grounded co-evolution governance for long-horizon LLM agents. *arXiv preprint arXiv:2607.16621*, 2026.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.850.

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

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

Qianxi Yan, Chunrong Chen, Jiuzhou Zhao, Min Zhang, Yongzhou Xu, and Xiaochuan Xu. SkillEvo: Self-renewing evolution gradients from multi-turn interaction feedback. *arXiv preprint arXiv:2608.13120*, 2026.

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 preprint arXiv:2605.23904*, 2026.

Zhiyuan Yao, Yuxin Chen, Zhengxi Lu, Zishan Xu, Yueqing Sun, Yifu Guo, Yuquan Lu, Zhengzhou Cai, Kangning Zhang, Zhuowen Han, Zi-Han Wang, Ziang Ye, Qi Gu, Xunliang Cai, Weiwen Liu, and Yongliang Shen. SkillRise: Agentic reinforcement learning for cross-task skill evolution. *arXiv preprint arXiv:2607.26784*, 2026.

Tianzhu Ye, Li Dong, Guanheng Chen, He Zhu, Xun Wu, Shaohan Huang, and Furu Wei. LLM-as-a-coach: Experiential learning for non-verifiable tasks. *arXiv preprint arXiv:2607.18110*, 2026.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025. ICLR 2026.

Zhiwei Zhang, Yudi Lin, Nikki Lijing Kuang, Linlin Wu, Xiaomin Li, Songtao Liu, and Fenglong Ma. Co-evolving skill generation and policy optimization. *arXiv preprint arXiv:2606.08755*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024.

A Experimental protocol

A.1 Benchmarks, splits, and worker interface

benchmark	train / val / test	test split	evaluator
SpreadsheetBench	80 / 40 / 280	Verified-400, seed-42 manifest	cell comparison
AppWorld	90 / 57 / 168	test-normal	programmatic tests
AppWorld challenge	– / – / 417	test-challenge, held-out apps	programmatic tests
LiveMathBench	35 / 17 / 125	SkillOpt seed-42 split	exact choice match

For SpreadsheetBench, we use the curated Verified-400 release (Shortcut Research Team, 2025) and adopt SkillOpt's randomized seed-42 manifest verbatim (Yang et al., 2026). AppWorld's challenge split contains every task requiring an Amazon or Gmail API, both absent from training; it is also harder on average than test-normal (47% versus 38% difficulty-3 tasks) (Trivedi et al., 2024). It contains 139 scenarios, compared with 56 in test-normal. For LiveMathematicianBench, we adopt SkillOpt's seed-42 split verbatim.

Each benchmark supplies a fixed task-interface prompt describing the available environment, tools, output location, and grading-facing format. It is byte-identical across conditions and outside every method's editable surface. The no-skill condition receives this interface but no `SKILL.md`; learned behavior is therefore not confounded with instructions required merely to operate the harness. The release includes the exact rendered prompts.

A.2 Training schedules and checkpoints

All PROSE runs start from an empty store, use $G{=}4$, and draw reproducible batches. Infrastructure failures are excluded from evidence, while worker failures remain scored as zero. Each reported checkpoint is fixed before test evaluation.

The main runs use a GPT-5.6-luna worker at medium effort and a DeepSeek v4-pro optimizer at high effort, both through Claude Code 2.1.152. GPT-5.6-sol runs the advantage and failure extractors at medium effort through direct model calls. Worker turn budgets are 45 on SpreadsheetBench and LiveMathematicianBench and 80 on AppWorld, where the environment additionally caps executable interactions at 50. The optimizer runs one session per step with a 60-turn budget.

benchmark	batch	steps
SpreadsheetBench	40	4
AppWorld	30	6
LiveMathBench	35	8

SpreadsheetBench stopped at a budget boundary after step 4. Its validation trace was $37.5 \to 87.5 \to 87.5 \to 67.5 \to 87.5$ ($n{=}40$). The dip followed a harmful weakening of the rule against formula strings. It stayed just under the $z{=}2.5$ revert threshold (about 2.4 standard errors), so no revert fired; later mixed-group evidence reinstated the rule, and the next commit recovered. On AppWorld, step 3 was all-pass even after rerolling; reshuffling exposed new failures and step 4 committed the final two rules. On LiveMathematicianBench, an exploratory two-step extension changed only wording, so v8 remained the prespecified evaluated endpoint.

The AppWorld ablations use the same batches within each model stack. *No mixed-group contrast* discards mixed groups while retaining all-fail extraction and the winners' pool. *Free editing* retains the verified evidence but removes typed entries, typed operations, and licensing. The frontier-model arms run six steps; the small open-weight model arms run five. Thus, comparisons are step-matched within each model stack, not matched by the number of committed versions.

A.3 Baseline protocol

Trace2Skill and SkillOpt run their released code through the same worker, harness, interface prompt, turn budget, splits, and evaluator as PROSE. DeepSeek v4-pro performs each method's meta-level reasoning under the frontier-model stack and Qwen3.6-27B under the small open-weight model stack (App. C.3). All trained conditions start from an empty document and use one evolution seed per benchmark; the reported evaluation cells use three trials.

SkillOpt and PROSE are step-matched: both use four steps on SpreadsheetBench, six on AppWorld, and eight on LiveMathematicianBench. SkillOpt's released validation gate and other hyperparameters are unchanged. Trace2Skill uses its released agentic error analyst on SpreadsheetBench. AppWorld and LiveMathematicianBench lack the offline reference artifact that analyst requires, so those ports use the release's single-pass analyst; consolidation fan-in remains 32.

Some LiveMathematicianBench questions contain a meta-option stating that a stronger result can be proved, and this option is correct in all 85 corpus instances where it appears (54 test items). Without a guard, SkillOpt learned to select it without evaluating the alternatives, reaching 76.3 ± 0.5. We exclude that answer-pattern exploit and report a rerun with an explicit anti-exploit constraint. This is a benchmark–optimizer interaction, not a general assessment of SkillOpt. PROSE instead learned to reconstruct the strongest theorem before comparing every option.

B Prompts and update implementation

B.1 Abbreviated prompt templates

The loop uses three fixed prompt templates. Dynamic briefs insert the current store, rollout files, budgets, and relevant history. The templates below retain the instructions that determine evidence selection and update behavior, while omitting repeated definitions, examples, workspace paths, and the dynamic contents of each brief.

Shared evidence contract

Read the trajectories rather than inferring behavior from the grader message. State each mechanism generally: do not copy task-specific values, identifiers, paths, coordinates, or answers into a description or edit. Diagnose coverage counterfactually: a node covers a mechanism only if following its text as written would have prevented the observed failure. Use exactly one diagnosis— `ignored`, `missing`, `wrong`, or `procedure_gap`—and target stable node ids. Every cited span must identify its task and rollout and quote at most 300 verbatim characters from worker actions, observations, or output; unverified claims do not count. Return raw JSON under the supplied schema and remain within the proposal budget.

Advantage extractor (mixed groups)

You receive several attempts of the same task under one skill, with both passing and failing outcomes. Treat this group as a controlled comparison that holds task difficulty fixed.
1. Audit. Before diagnosing failures, emit one `node_audit` row for every listed rule and knowledge node. Mark whether winners and losers followed or ignored the node, and cite each non-`n/a` judgment from the corresponding outcome side.
2. Contrast. Find the point at which winning and losing behavior diverges. Each mechanism must describe the divergence, winner behavior, and loser behavior, with at least one verified quote from both a passing and a failing rollout.
3. Diagnose and sketch. Label the mechanism against the current store. An ignored node may be moved or strengthened but not duplicated; missing behavior may produce an addition; wrong content may be edited or removed; a procedural gap may suggest adding or moving a stage. Encode a general rule rather than the observed instance.
4. Trace and return. Emit one ordered `stage_trace` per rollout, including skipped, added, reordered, or method deviations. Return the fields `node_audit`, `mechanisms`, and `stage_traces`; each mechanism contains its diagnosis, node, pass/fail citations, and optional typed-operation sketch.

Failure extractor (all-fail groups)

You receive several attempts of the same task and all have failed. There is no winner, so read the attempts jointly rather than treating them as a contrast.

Failure extractor (all-fail groups) (continued)

1. Find convergence. Lead with a failure mechanism shared by the attempts. A convergent mechanism requires verbatim support from at least two distinct rollouts. If the attempts instead flounder in unrelated ways, report the smallest number of distinct mechanisms or one structural `procedure_gap`; do not create one item per rollout merely to appear complete.
2. Diagnose and sketch. Apply the same counterfactual coverage test. Missing behavior may suggest an addition, ignored guidance only a placement or strengthening change, wrong guidance an edit or deletion, and missing procedure structure a quote-backed stage addition or move. It is valid to return no mechanism when only task-specific noise is supported.
3. Record progress and traces. Identify the rollout that progressed furthest and the step that stopped it; this is advisory and licenses no edit by itself. Emit one `stage_trace` per rollout. Return `mechanisms`, `stage_traces`, and `progress`; each mechanism records whether it is convergent or divergent, its diagnosis, implicated node, supporting rollouts and quotes, and an optional add, edit, move, or drop sketch.

Optimizer

You receive the current typed store, verified mixed- and all-fail mechanisms, all-pass trajectories, the pending ledger, and node/task histories. Produce one ranked list of typed operations, or an empty list when no proposal clears the evidence requirements.
1. Cluster. Merge sightings across extractor outputs and pending evidence only when one contract-shaped rule would cover them and following that rule would have changed each cited failure. Support counts distinct tasks, not rollout count.
2. Diagnose. Inspect the cited raw trajectories for every adopted lead and assign the final counterfactual diagnosis yourself; extractor labels are hypotheses. The diagnosis selects the licensing contract.
3. Cure. Prefer same-task winners, then relevant all-pass trajectories, when wording a fix. Winner evidence may supply content without increasing the support count of a failure mechanism. Only recurrent winning phases shape procedure stages.
4. Compose. Make the next skill one coherent document: use at most one operation per node, integrate before adding, scope content where it fires, prune only when the evidence trail supports it, and rank operations by support breadth and corrective value. Put dependent changes in one bundle.
Write `report.json` with `ops`, `observations`, `dropped_leads`, and `rationale`. Each operation records its evidence basis and provenance: verified citations, final diagnoses, matched contrasts, cited pending entries, and adopted leads where applicable. Run the production operation checker, inspect the rendered skill preview, and correct every schema, evidence, licensing, or length error before returning.

The exact versioned templates and JSON schemas, dynamically generated briefs, and ablation-specific variants are release artifacts.

B.2 Typed operations and licensing

The vocabulary has four operation families—add, edit, move, and drop—applied to procedure stages, rules, and knowledge. For stages, editing changes the wording and moving changes the procedure order. Operations apply in order against the running store. Dropping a stage is refused while any node remains scoped to it, so scoped rules are never removed implicitly.

The host screens parse the proposal; re-match quoted spans to their trajectories (ellipsis-aware fuzzy ratio 0.9); stamp the recorded pass/fail outcome; verify the evidence contract; enforce the configured n-task support floor for quote-backed operations; cap each step at 12 operations; and enforce the 1,600-token (about 6,400-character) skill-document length bound. Additional requirements depend on the proposed change. Proposed stage wording needs evidence from a successful attempt. Changes prompted by ignored guidance, and proposals to remove or reorder stages, need matched successful and failed attempts of the same task. Move and drop may instead cite the record accumulated for the target entry across earlier updates. This record includes its prior versions and changes, together with counts of later trajectories that used or ignored it. Content based only on ignored evidence is blocked.

Under-supported evidence enters the pending ledger with its origin task. A later proposal may combine it with current-batch evidence; the host counts the union of distinct origin tasks. Pending evidence never changes the worker-visible document on its own. A new sighting from the same task for the same implicated entry replaces the older live record, while a record cited by an accepted update is consumed. An entry remains live for at least three steps; the system extends this window to one full training epoch plus one step when needed. This gives rolling windows of three, four, and three update steps on SpreadsheetBench, AppWorld, and LiveMathematicianBench, respectively. At most 90 live entries are retained, with the oldest unconsumed entry evicted when the cap is exceeded.

The ledger stores the origin task and step, diagnosis, implicated entry, concise proposed lesson, and verified citation references rather than a pool of full historical trajectories.

B.3 LLM-GENERATED SKILL BASELINE

DeepSeek v4-pro at the optimizer's effort level receives a de-identified public benchmark description covering task format, interface, and grading. It receives no tasks, trajectories, scores, or length constraint. The resulting artifact is mounted exactly like a learned document. The release contains the generation prompt and generated documents.

C STATISTICAL DETAILS AND ADDITIONAL RESULTS

C.1 STATISTICAL PROTOCOL

For three-trial cells, the primary test is Wilcoxon signed-rank on per-task trial means over the infra-clean paired set. AppWorld SGC uses per-scenario means. The reported Δ is the paired-set mean and can differ slightly from subtracting independently aggregated table entries. Single-trial replication cells use exact McNemar tests on task outcomes (or scenario outcomes for SGC).

The paired sets contain 277 SpreadsheetBench tasks (three excluded across conditions), 168 AppWorld test-normal tasks, 417 challenge tasks, and 124 LiveMathematicianBench items. The SkillOpt math comparison contains 123. Thirty-two challenge cells lost to infrastructure were resumed rather than excluded; one AppWorld no-skill trial affected by an API-side error was discarded whole and rerun.

Trial-matched exact McNemar tests agree with the Wilcoxon conclusions for the no-skill, LLM-generated skill, and PROSE conditions. Task-level nondeterminism is 17–18% across repeated identical conditions, which motivates three trials; small single-trial differences often changed sign or significance when repeated. AppWorld grading races occur in roughly 0.5% of cells and can only create false zeros; affected cells were rerun.

Table 5: Complete paired comparisons for Table 1. Each entry reports the paired PROSE difference in percentage points and the two-sided p-value (Δ/p); positive values favor PROSE.

setting	No skill	LLM-generated	Trace2Skill	SkillOpt
SpreadsheetBench	$+46.9/8.9\text{e}{-30}$	$+39.0/1.1\text{e}{-23}$	$+43.3/{<}1\text{e}{-23}$	$+0.4/.754$
AppWorld normal TGC	$+10.5/1.4\text{e}{-6}$	$+10.1/1.2\text{e}{-5}$	$+7.7/2.8\text{e}{-5}$	$+0.6/.403$
AppWorld normal SGC	$+22.0/8.7\text{e}{-5}$	$+19.6/.00040$	$+11.9/.0042$	$+4.2/.102$
AppWorld challenge TGC	$+15.8/1.4\text{e}{-17}$	$+6.9/2.7\text{e}{-6}$	$+12.6/7.0\text{e}{-16}$	$+1.6/.107$
AppWorld challenge SGC	$+19.7/6.7\text{e}{-8}$	$+9.6/.00115$	$+18.2/1.3\text{e}{-6}$	$+3.1/.215$
LiveMathBench	$+9.7/.00392$	$+6.7/.0572$	$+7.2/.0163$	$+4.3/.0844$

Table 6: Paired comparisons for the small open-weight model stack, ablation, and group-size results (Tables 2, 4, and 7). Each entry reports the first condition minus the second in percentage points and the two-sided p-value (Δ/p). TGC uses per-task means; SGC uses per-scenario means.

worker	comparison	TGC Δ/p	SGC Δ/p
GPT-5.6-luna	Free editing − PROSE	$-4.8/.0016$	$-11.9/.0039$
GPT-5.6-luna	No mixed-group contrast − PROSE	$-1.6/.095$	$-6.0/.028$
Qwen3.6-27B	Free editing − PROSE	$-2.2/.088$	$-4.9/.253$
Qwen3.6-27B	No mixed-group contrast − PROSE	$-4.2/.0055$	$-7.3/.049$
Qwen3.6-27B	PROSE − No skill	$+12.7/8.9\text{e}{-9}$	$+20.8/4.3\text{e}{-5}$
Qwen3.6-27B	PROSE − LLM-generated	$+7.1/.00027$	$+11.3/.0076$
Qwen3.6-27B	PROSE − Trace2Skill	$+9.3/4.2\text{e}{-7}$	$+17.9/6.6\text{e}{-5}$
Qwen3.6-27B	PROSE − SkillOpt	$+4.4/.00096$	$+7.7/.026$
Qwen3.6-27B, challenge	PROSE − No skill	$+9.9/7.4\text{e}{-10}$	$+16.6/4.1\text{e}{-6}$
Qwen3.6-27B, challenge	PROSE − LLM-generated	$+2.4/.054$	$+5.3/.076$
Qwen3.6-27B, challenge	PROSE − Trace2Skill	$+3.8/.010$	$+6.5/.025$
Qwen3.6-27B, challenge	PROSE − SkillOpt	$+1.3/.244$	$+1.9/.213$
GPT-5.6-luna	$G{=}2 - G{=}4$	$-1.4/.123$	$-3.6/.307$
GPT-5.6-luna	$G{=}6 - G{=}4$	$-0.4/.769$	$-3.6/.156$
GPT-5.6-luna	$G{=}6 - G{=}2$	$+1.0/.381$	$0.0/.667$

C.2 GROUP-SIZE SENSITIVITY

Table 7 reports the AppWorld sweep over $G \in \{2, 4, 6\}$ discussed in §4.3. Increasing G from 2 to 4 raises the number of mixed groups from 11 to 21; increasing it to 6 yields only two more. No pairwise difference between group sizes is significant (Table 6).

Table 7: Group-size sensitivity on AppWorld test-normal. Each row is one trained document evaluated over three trials; scores are mean ± sd in percent. Training rollouts and mixed groups are totals across six steps. The shaded row is the operating point. Paired comparisons are in App. C.1.

G	training rollouts	mixed groups	TGC	SGC
2	360	11	87.1 ± 1.9	81.6 ± 4.5
4	720	21	88.5 ± 0.3	85.1 ± 2.1
6	1,080	23	88.1 ± 0.6	81.6 ± 2.7

C.3 ADDITIONAL WORKERS

A second frozen worker, DeepSeek v4-flash-0425, repeats the no-skill, LLM-generated, and PROSE conditions with independently learned documents. Table 8 reports one trial per cell for a uniform comparison; SpreadsheetBench also has three trials, whose $44.5 \pm 0.9 \rightarrow 84.3 \pm 2.0$ result agrees in direction and magnitude. Cells are compared only within a worker.

Table 8: Held-out success (%) for a second frozen worker, DeepSeek v4-flash-0425, one trial per cell. Subscripts show changes from no skill; positive changes are green and negative changes are red.

	SpreadsheetBench	AppWorld test-normal		LiveMathBench
skill source		TGC	SGC	
No skill	43.6	78.0	57.1	38.4
LLM-generated	$44.6_{+1.1}$	$80.4_{+2.4}$	$66.1_{+8.9}$	$32.8_{-5.6}$
PROSE	$\mathbf{82.5}_{+38.9}$	$\mathbf{86.9}_{+8.9}$	$\mathbf{75.0}_{+17.9}$	$\mathbf{48.0}_{+9.6}$

The trained condition is higher in every cell. The corresponding p values are $2.2\text{e}{-23}$ (SpreadsheetBench), .0135 and .031 (AppWorld TGC and SGC), and .0501 (LiveMathematicianBench). This replication tests direction across a second worker, not training-run variance.

The small open-weight model stack in §4.2 binds worker, extractor, and optimizer to one self-hosted Qwen3.6-27B endpoint, which also serves as Trace2Skill's analyst and SkillOpt's optimizer. The test ids, rendered worker prompt, batch seed, empty start, and licensing thresholds match the main AppWorld protocol. We report PROSE's five-step checkpoint to match its ablations; SkillOpt follows its six-step AppWorld schedule and Trace2Skill its released pipeline (App. A.3). The LLM-generated document is the Table 1 artifact, held fixed across stacks. The final documents contain 2,830 (PROSE), 4,925 (Trace2Skill), 5,949 (LLM-generated), and 8,858 (SkillOpt) worker-visible characters. The test-challenge cells mount the same frozen documents over 417 tasks and 139 scenarios. Across the twenty three-trial evaluation cells, eight infrastructure failures, none in the baseline or challenge cells, are excluded symmetrically from paired tests. GPU time was not metered.

C.4 TOKEN ACCOUNTING

The reported PROSE training runs recorded 11,317,143 input and 1,534,272 output tokens on SpreadsheetBench, 26,049,332 and 1,744,152 on AppWorld, and 9,633,052 and 2,158,518 on LiveMathematicianBench. These totals include worker rollouts, validation sweeps, extractor calls, and optimizer calls through steps 4, 6, and 8; they exclude test evaluation, baselines, and ablations.

D REPRESENTATIVE LEARNED ARTIFACT

The table identifies the three evaluated versions. The following box reproduces the complete worker-visible `SKILL.md` from the reported AppWorld checkpoint. Version v4 is the active skill after the

six-step schedule because two steps did not commit a new version. It contains five procedure stages and nine rules in 2,570 characters. We preserve its wording, ordering, metadata, and stable identifiers, changing only the Markdown typography for presentation.

benchmark	version	characters
SpreadsheetBench	v4	1,611
AppWorld	v4	2,570
LiveMathBench	v8	1,510

Complete AppWorld v4 worker-visible skill

```
---
name:  appworld-skill
description: Procedures and pitfalls for completing supervisor tasks through the AppWorld app APIs.
---
```

Procedure

1. `[S1]` **Orient** — Inspect the relevant app API docs and locate the data sources the task asks for.
2. `[S2]` **Authenticate** — Get supervisor credentials and log in to every app needed for the task.
3. `[S3]` **Collect** — Fetch all relevant data from every needed endpoint, exhausting every paginated page and preferring explicit relationship or identity filters over broad text queries.
   - `[R1]` Before deriving a target set, count, or aggregate from an API collection: exhaust every result page. Use the API's explicit relationship or identity filters when available — do not fetch unfiltered collections or rely on broad text queries.
   - `[R5]` When a task defines a target set by a personal relationship label: resolve membership from the contact API's explicit relationship metadata. Do not infer it from app friends, expense groups, transaction descriptions, or shared communications.
4. `[S4]` **Build** — Process data programmatically: compute target sets from metadata, calculate aggregates and splits in code, and serialize output with minimal quoting.
   - `[R2]` When serializing delimited text output: quote a field only when it contains the delimiter, a double-quote, or a line break. Emit the header and other fields unchanged.
   - `[R4]` When computing a derived value or target set from fetched records: filter and aggregate programmatically in code. Never assemble counts, sums, or identifier lists manually from printed output.
   - `[R7]` When a task asks to select or activate an existing item: use what already exists — do not create new items, records, or playlists as a substitute
   - `[R8]` When evaluating whether a playlist meets a time requirement: inspect the player's queue-cycle behavior from the API description and current state — one-pass duration is only correct when the queue does not cycle
5. `[S5]` **Deliver** — Verify the result against the original predicate using freshly fetched state, reconcile counts and identifiers, then call `complete_task`.
   - `[R3]` Before calling `complete_task` after a batch mutation: re-fetch every affected collection and re-evaluate the original predicate. Correct all remaining violations, repeating until none remain.
   - `[R6]` When submitting a numeric answer to `complete_task`: pass the value as a number — not as a currency symbol, unit label, or formatted string
   - `[R9]` When a task asks only for a state mutation and not for a value: call `complete_task` with no answer argument

The box contains only the artifact presented to the worker. The typed store, cited trajectories, and licensing history remain host-side provenance records and do not consume worker context. Figure 3 traces the evidence behind rule R1 in this document; the release includes the exact Markdown file and its provenance records.