SKILLREFINER: OFFLINE SKILL REFINEMENT FROM HISTORICAL AGENT TRACES

Anonymous authors
Paper under double-blind review

ABSTRACT

Skills capture procedural knowledge that helps agents solve complex tasks. Existing skill-refinement methods improve these instructions by generating new rollouts with candidate skills, which can be costly or impractical when environments are not replayable or supervision arrives only after deployment. We introduce SKILLREFINER, an offline method that refines deployed agent skills from historical execution traces and observed outcomes. SKILLREFINER summarizes long trajectories, clusters recurring behaviors and failure modes, and derives targeted edits from each cluster. Successful clusters reinforce effective behavior, while failure clusters produce guardrails and procedural corrections that are checked against supporting evidence. We evaluate SKILLREFINER on spreadsheet manipulation, mathematical reasoning, and production pull-request review. SkillRefiner improves over both the LLM and human authored skills across all settings while using 1.4–42× fewer refinement tokens than competing refinement methods. [1]

1 INTRODUCTION

Large language model (LLM)-based agents increasingly rely on *skills*: reusable instructions that encode procedures, tool-use policies, and domain-specific guidance for solving tasks. Skills substantially shape how an agent approaches a task, and their usage is standardized (Agent Skills Contributors, 2025) across most agentic coding tools, so weaknesses in a skill can lead to systematic errors that recur for many users across many executions. Improving the skill is therefore a promising way to improve agent behavior across tasks.

Recent work has focused on automating skill improvement (Yang et al., 2026; Agrawal et al., 2026; Wang et al., 2026). Various existing approaches follow a common optimization loop: propose a candidate skill, execute the agent on new tasks, evaluate the resulting behavior, and use this feedback to generate the next candidate. Methods such as GEPA (Agrawal et al., 2026), SkillOPT (Yang et al., 2026), SkillGrad (Wang et al., 2026), and SkillEvolver (Zhang et al., 2026a) can all be viewed as variants of this paradigm. By repeatedly testing the agent with candidate modified skills, replay-based methods can directly measure whether a proposed change improves downstream performance.

This paradigm assumes that tasks can be replayed and that their outcomes can be evaluated quickly. Unfortunately, these assumptions do not hold when the task requires a long-horizon feedback signal such as human interaction or real-world deployments. To evaluate an agent-generated PR review, for example, requires understanding how the agent's feedback impacts a human decision maker, and that often does not come until the PR has been merged and the code evaluated in production. Even when replay is available, repeatedly generating executions solely for skill optimization is costly and may not reflect the conditions under which the skill is actually used.

Our key observation is that deployed agents already accumulate a rich source of supervision in the form of historical execution traces and outcomes. Historical traces provide information about the strategies attempted, tools used, and successful or failed behaviors. In certain settings, they also provide signals–such as human feedback–that only become available after the original execution of the agent. The existence of historical data suggests a different approach to skill improvement: instead of replaying tasks for optimization, one can refine a skill in an *offline* manner from existing executions.

[1] The code for SkillRefiner is available here: `https://figshare.com/s/9d786fe4255a7ebaa861`

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.

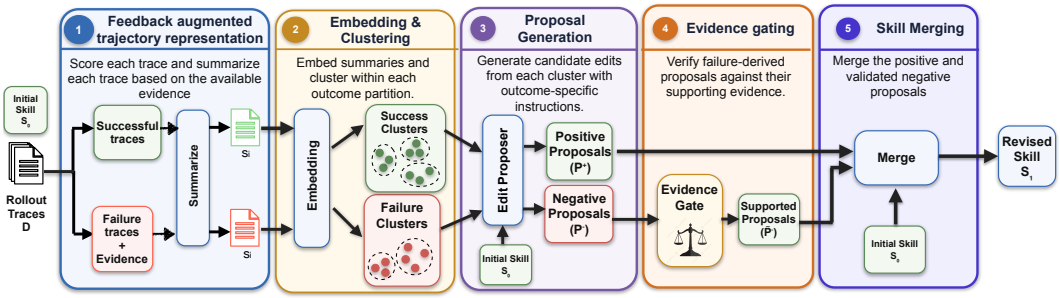

Figure 1: Overview of the SKILLREFINER pipeline. Historical executions are partitioned by outcome, summarized, and clustered into recurring successful and failed behaviors. Cluster-level refinements are then proposed, with failure-derived refinements verified against their supporting evidence before being merged into the revised skill.

To address these challenges, we introduce SKILLREFINER, an offline framework for improving skills from historical execution traces and their observed outcomes. SKILLREFINER first summarizes long traces into compact representations of task context, reasoning strategy, performed actions, and evaluator feedback. The resulting summaries are then partitioned by outcome and clustered into semantically related groups. Clustering allows SKILLREFINER to identify recurring features and contexts across traces that are associated with success or failure. In addition, clustering helps distinguish recurring patterns from isolated events that might inject noise into the skill.

We evaluate SKILLREFINER on a diverse set of tasks, including spreadsheet manipulation, mathematical reasoning, and pull-request review generation. Across these settings, SKILLREFINER improves over the initial skill and is competitive with or outperforms existing skill refinement methods. The gains are especially pronounced for weaker agent models, where improved procedural guidance can significantly improve agent behavior. Since SKILLREFINER operates entirely on previously collected executions, it avoids the cost of repeatedly generating task replays; on spreadsheet tasks, for example, it achieves accuracy comparable to GEPA (Agrawal et al., 2026) while using an order of magnitude fewer refinement tokens.

In summary, this paper makes the following contributions:

- We formulate *offline skill refinement from historical execution traces* as an alternative to rollout-based optimization. This setting is motivated by deployed tasks where replay is expensive or impossible and where useful supervision may arrive only after downstream human actions.
- We introduce SKILLREFINER, which uses clustering to identify recurring patterns in successful and failed executions, and uses these patterns to reinforce effective behaviors, while deriving targeted repairs for recurring failures.
- We evaluate SKILLREFINER in three different domains: spreadsheet manipulation, mathematical reasoning, and pull-request review.

  Across eight evaluation settings, SKILLREFINER improves the initial skill in every setting and matches or improves over the strongest refinement baselines while operating at a substantially better performance–cost tradeoff, using 1.4–13× fewer refinement tokens than Trace2Skill and 2.4–42× fewer than GEPA.

2 METHOD

2.1 SKILLREFINER OVERVIEW

SKILLREFINER takes as input an initial natural-language skill S_0, together with a collection of historical agent execution traces and their observed outcomes, and produces a revised skill S_1. Importantly, the method operates entirely over previously collected executions and does not generate new agent executions. Figure 1 gives an overview of the pipeline, which proceeds in five stages.

Feedback-augmented trajectory representation. SKILLREFINER starts by constructing a representation of each execution that incorporates both the agent's behavior and, for failed executions, any additional feedback available from the task evaluator (grader). For example, a spreadsheet evaluator may report which output cells differ from the expected result, while a mathematical reasoning benchmark may expose the discrepancy between the predicted and reference answers. Given a trace augmented with such evaluator feedback, SKILLREFINER then summarizes it into a compact description of its task context, reasoning strategy, and salient actions such as tool use.

Embedding and clustering. Next, SKILLREFINER maps each trace summary to a vector representation using an embedding model and clusters these representations into semantically related groups. We cluster successful and failed executions separately because the two partitions serve different roles in refinement: successful clusters identify recurring behaviors worth reinforcing, whereas failed clusters identify recurring failure modes that may require a repair. We use these clusters, rather than individual traces, as the unit of refinement so that a proposed change reflects a recurring pattern rather than an isolated execution.

Proposal generation. The next stage of SKILLREFINER extracts a candidate skill edit from each cluster. For successful clusters, it identifies recurring behaviors that should be reinforced or generalized. For instance, if successful traces repeatedly verify spreadsheet edits by reloading the output and checking the modified range, SKILLREFINER can promote this behavior into an explicit verification rule. For failed clusters, it identifies the recurring failure mode and proposes a targeted correction, such as adding a verification step, restricting when a tool-use strategy should be applied, or making a missing condition explicit.

Evidence gating. For failure-derived proposals, SKILLREFINER employs an LLM judge to assess if the proposed suggestions are derived from the summaries. Given a proposal and its failed cluster, the step checks whether the targeted behavior recurs across the cluster and whether the available evidence implicates that behavior. Because candidate generation is open-ended, it may produce plausible repairs that are not supported by observed failures, so the goal of evidence gating is to filter out such unsupported proposals.

Skill merging. Finally, SKILLREFINER merges the resulting proposals with the initial skill S_0. The merge step incorporates positive proposals that reinforce successful behaviors and supported negative proposals that restrict or correct behaviors associated with failure, while preserving the remaining guidance in S_0.

2.2 FEEDBACK-AUGMENTED TRAJECTORY REPRESENTATION

We represent execution traces as a dataset $\mathcal{D} = \{(x_i, y_i, g_i)\}_{i=1}^{N}$, where x_i denotes a trace, $y_i \in \{0, 1\}$ its observed outcome, and g_i feedback from the task evaluator. We begin by partitioning the dataset according to the observed outcome:

$$\mathcal{D}^+ = \{x_i : y_i = 1\}, \qquad \mathcal{D}^- = \{x_i : y_i = 0\}$$

We separate successful and failed executions so that we can learn which behaviors are associated with success and which are associated with failure. For failed executions, the binary outcome $y_i = 0$ may be too coarse-grained to indicate what went wrong. When evaluator feedback is available, we attach it to the failed trace before summarization, yielding an evidence-augmented trace $\widetilde{x}_i$. We then use an LLM to summarize each execution into a compact textual representation s_i of its task context, strategy, salient actions, and outcome:

$$s_i = \begin{cases} \textsc{Summarize}(x_i), & y_i = 1, \\ \textsc{Summarize}(\widetilde{x}_i), & y_i = 0. \end{cases}$$

For failed executions, the resulting summary therefore reflects both the agent's behavior and the available evidence about the observed failure.

2.3 OUTCOME-AWARE BEHAVIORAL CLUSTERING

We next group executions that exhibit semantically similar behavior. The goal is to identify recurring patterns within successful and failed executions, rather than derive a refinement from either outcome

partition as a whole. Let $e_i = \mathrm{Embed}(s_i) \in \mathbb{R}^d$ denote the embedding of execution summary s_i, and define the index sets

$$I^+ = \{i : y_i = 1\}, \qquad I^- = \{i : y_i = 0\}.$$

We cluster the embeddings in the two outcome partitions independently:

$$\mathcal{C}^+ = \{C_1^+, \ldots, C_{m_+}^+\} = \mathrm{Cluster}\big(\{e_i : i \in I^+\}\big),$$
$$\mathcal{C}^- = \{C_1^-, \ldots, C_{m_-}^-\} = \mathrm{Cluster}\big(\{e_i : i \in I^-\}\big)$$

where each $C_j^+ \subseteq I^+$ and $C_k^- \subseteq I^-$ indexes a group of executions with semantically related summaries. Singleton clusters are treated as noise and do not independently induce a refinement.

In our implementation, we reduce the embedding dimension using UMAP (McInnes et al., 2020) and then apply HDBSCAN (Campello et al., 2013) to minimize the choice of clustering parameters.

2.4 Cluster-Level Candidate Generation

We next use an LLM to generate one candidate skill edit from each cluster. The LLM interprets the success and failure clusters differently.

For a successful cluster C_j^+, the summaries provide direct evidence of a behavior that repeatedly occurs in successful executions. Given these summaries and the current skill S_0, the LLM proposes an edit that reinforces or generalizes this behavior:

$$p_j^+ = \mathrm{Prop}^+\big(S_0, \{s_i : i \in C_j^+\}\big).$$

For a failed cluster C_k^-, the summaries instead identify a recurring failure pattern. They provide evidence about what is going wrong and the trace-level hypotheses about preventing said failures. We use the pretrained LLM as a source of prior knowledge: given the current skill S_0 and the observed failure pattern, it proposes a candidate correction that it predicts will avoid the failure:

$$p_k^- = \mathrm{Prop}^-\big(S_0, \{s_i : i \in C_k^-\}\big).$$

Thus, for failed clusters, the historical traces localize the problem, while the proposed remedy is supplied by the LLM's pretrained knowledge. The resulting edit is therefore a hypothesis about how the skill should change, rather than a correction implied by just the historical evidence.

2.5 Evidence Gating

A repair proposed from a failed cluster is only a hypothesis about how the skill should change. Because proposal generation is open-ended, the model may suggest edits whose rationale is only weakly supported by the observed executions. We therefore apply a secondary *evidence gate* that checks each proposed edit against the evidence from the cluster from which it was derived. This generate-then-verify procedure separates the open-ended task of proposing a repair from the narrower task of determining whether that repair is grounded in the observed execution evidence.

Concretely, after generating a proposal p_k^- from a failed cluster C_k^-, we provide a separate LLM with the fixed proposal together with the execution summaries. The gating LLM is prompted to assess whether the behavior targeted by the proposal is supported by the observed failures: namely, whether the behavior is present across the relevant executions and whether the available evidence connects it to the failures. The gating LLM does not generate or revise the proposal, it only judges the evidential support for the proposed rationale. We retain p_k^- only if it passes this support check.

2.6 Skill Merging

The final stage combines the proposals from all clusters into a revised skill. Let $\mathcal{P}^+$ denote the proposals derived from successful clusters, and let $\widetilde{\mathcal{P}}^-$ denote the failure-derived proposals retained by the evidence gate. We use an LLM to merge these proposals with the initial skill:

$$S_1 = \mathrm{Merge}\Big(S_0, \mathcal{P}^+, \widetilde{\mathcal{P}}^-\Big)$$

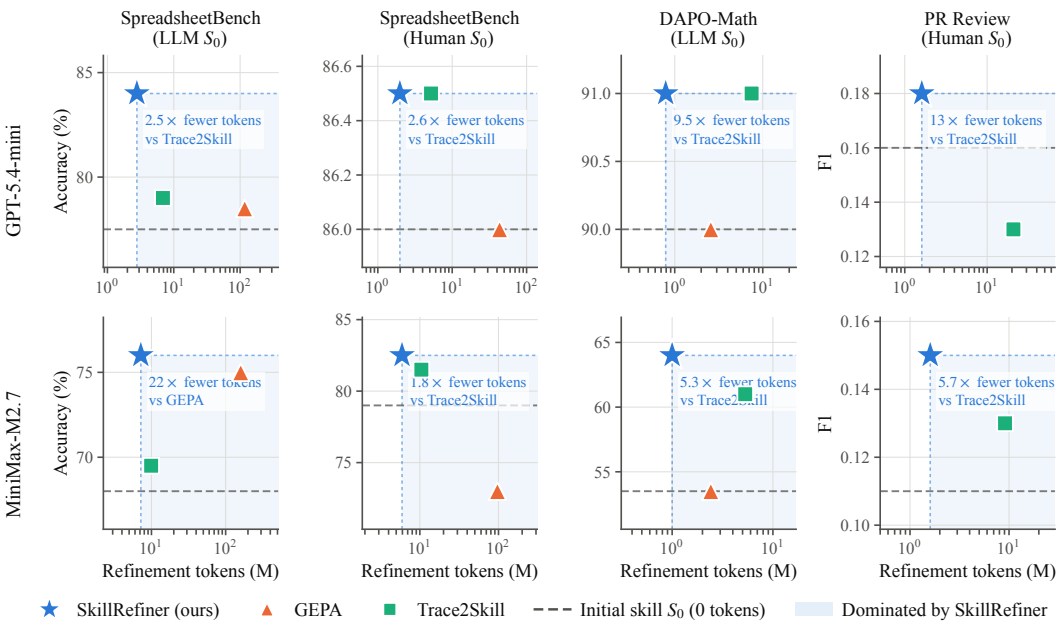

Figure 2: Comparison of SKILLREFINER against GEPA and TRACE2SKILL on SpreadsheetBench, DAPO-Math, and PR Review using GPT-5.4-mini (top) and MiniMax-M2.7 (bottom). The x-axis shows refinement tokens (millions; log scale), while the y-axis reports task accuracy or F1. Dashed horizontal lines denote the performance of the initial skill S_0 with no refinement, and shaded regions indicate performance–token configurations dominated by SKILLREFINER.

The merge step incorporates compatible proposals while preserving guidance in the existing skill that is not targeted by any refinement. It also resolves conflicts between positive and negative proposals. In particular, if a failure-derived proposal restricts a behavior that is also supported by successful executions, the LLM is instructed to qualify the restriction to the contexts in which the failure was observed rather than remove the behavior unconditionally.

3 EXPERIMENTAL SETUP

We evaluate SKILLREFINER across three domains that differ in task structure and in available supervision: spreadsheet manipulation, mathematical reasoning, and pull-request review. The first two settings provide deterministic benchmark feedback, while pull-request review provides delayed supervision derived from subsequent developer actions. This lets us test the method both when outcomes are immediately verifiable and when useful feedback emerges only after deployment.

For both Trace2Skill and SKILLREFINER, refinement uses only a fixed corpus of historical execution traces and their observed outcomes. GEPA generates replay executions during refinement. The final performance is measured on a disjoint held-out evaluation set.

3.1 BASELINES

We measure baseline performance using the initial skill S_0, which captures agent performance before refinement. We additionally compare against TRACE2SKILL (Ni et al., 2026), the closest prior method for learning skills offline from execution traces. For settings in which tasks can be replayed under a revised skill, we also compare against GEPA (Agrawal et al., 2026), which iteratively proposes and evaluates revisions to a textual artifact.

GEPA differs from the offline baselines in that it performs iterative search over candidate revisions, requiring additional agent executions to evaluate each candidate. We evaluate GEPA under a fixed search budget of five candidate revisions. This budget allows GEPA to improve over the seed while keeping the comparison computationally tractable. We report both final task performance and total

refinement cost, so that GEPA is compared as a point on the performance–cost curve rather than solely by its final accuracy.

For each refinement budget, all methods receive the same pool of N training examples. SKILLREFINER and TRACE2SKILL use this pool directly for offline refinement. GEPA requires separate training and internal validation subsets for reflection and candidate selection. Following the recommended GEPA configuration for datasets of this size, we split the same N-example pool equally between these roles. The GEPA validation subset is used during optimization and is distinct from the held-out evaluation set used to report final performance.

3.2 TASKS AND EVALUATION

For each setting, we employ the same LLM for the baselines and SKILLREFINER. For our agent, we use the OpenHands (Wang et al., 2025a) agent harness. We choose two models to evaluate SKILLREFINER with different baseline capabilities: GPT-5.4-mini (OpenAI, 2026) provides a relatively strong agent, while MiniMax-M2.7 (Chen et al., 2026) provides a weaker setting in which there is greater room for skill refinement to affect behavior. Additionally, SKILLREFINER uses Qwen3-Embedding-4B (Zhang et al., 2025) to embed trajectory summaries for clustering. We provide more details about clustering parameters for SKILLREFINER in Appendix A.2

SpreadsheetBench. SPREADSHEETBENCH (Ma et al., 2024) evaluates agents on realistic spreadsheet-manipulation tasks. Given an existing workbook and a natural-language instruction, the agent must modify the spreadsheet and produce a final workbook satisfying the requested transformation. For example, the agent can edit values, insert formulas, or restructure rows and columns. This benchmark uses an executable evaluator that compares the resulting workbook against the expected output, allowing correctness to be assessed automatically. For failed training executions, the evaluator reports cell-level discrepancies between the produced and expected workbooks.

DAPO-Math. Our second setting uses competition-level mathematical reasoning problems from DAPO-MATH (Yu et al., 2025). The dataset contains problems spanning areas such as algebra, geometry, number theory, and combinatorics. For each task, the agent receives a mathematical problem, produces a multi-step reasoning trajectory, and submits a final answer. Unlike SPREADSHEETBENCH, the agent does not interact with an external artifact. Correctness is determined automatically by comparing the submitted answer against the ground-truth using a rule-based evaluator. DAPO-Math provides a complementary test of SKILLREFINER because the available failure signal is substantially coarser than in SpreadsheetBench. For a failed execution, we know the agent's predicted answer and the reference answer, but the evaluator does not identify which intermediate reasoning step caused the error.

Pull-request review. Our third setting evaluates code-review agents on merged pull requests from production deployments. Given a pull request, the agent inspects the proposed changes and produces review suggestions. Unlike SpreadsheetBench and DAPO-Math, the quality of a review is not observable when it is written. The quality is revealed only by the developer's subsequent revisions before merge, typically days to weeks later. We use these revisions as delayed supervision. For each pull request, we retain the agent generated review together with the final merged revision, and treat agreement between these revisions and the agent's suggestions as evidence of a useful suggestion. For evaluation, we replay each method's skill on held-out historical pull requests, whose outcomes are already known, and compare the resulting suggestions against the issues the developer addressed. We compare against Trace2Skill but not GEPA in this setting. We consider 639 historical pull requests for refinement and 145 for evaluation (Table 3).

Evaluation metrics. For SpreadsheetBench and DAPO-Math, we use the benchmarks' official evaluators and report accuracy. For pull-request review, evaluation requires determining whether an agent suggestion refers to the same underlying issue as one inferred from the developer's subsequent edits. Exact string or code-location matching is insufficient because the same issue may be described differently or fixed through a different implementation. We therefore use GPT-5.4-mini as a fixed LLM judge. Given the two descriptions and the relevant code context, the judge determines whether they refer to the same issue. We use these matches to compute F1, which we report as the evaluation metric for pull-request review.

Table 1: Task performance on held-out evaluation sets. SpreadsheetBench and DAPO-Math report accuracy (%), while PR Review reports F1. Sub- and super-scripts $_*$ and † indicate significant differences in score from Trace2Skill and GEPA, respectively, as determined by paired bootstrap ($p < 0.05$, $N = 10,000$).

Task	Init.	Model	Initial	GEPA	T2S	SKILLREFINER
SpreadsheetBench	LLM	GPT-5.4-mini	77.5	78.5	79.0	$84.0^{\dagger}_{*}$
	LLM	MiniMax-M2.7	68.0	75.0	69.5	76.0_{*}
	Human	GPT-5.4-mini	86.0	86.0	86.5	86.5
	Human	MiniMax-M2.7	79.0	73.0	81.5	$84.5^{\dagger}$
DAPO-Math	LLM	GPT-5.4-mini	90.0	90.0	91.0	91.0
	LLM	MiniMax-M2.7	53.5	53.5	61.0	$64.0^{\dagger}$
PR Review	Human	GPT-5.4-mini	0.16	–	0.13	0.18_{*}
	Human	MiniMax-M2.7	0.11	–	0.13	0.15

Training and evaluation data. The amount of refinement data varies across settings. For SPREADSHEETBENCH we use the same subset reported by Trace2Skill (Ni et al., 2026) to enable direct comparison (200 for learning and 200 for evaluation). For DAPO-Math we use 400 samples for learning, and 200 for evaluation.

Initial skills. We evaluate two sources of initial skills: human-authored skills written by domain experts and LLM-generated skills produced from a high-level task description. SPREADSHEETBENCH includes both types of initialization. DAPO-Math uses an LM-generated initial skill because no human-authored skill was available. For pull-request review, we refine the human-authored skill used in the production deployment. All refinement methods start from the same initial skill S_0.

4 RESULTS

Main results. Our primary question is whether SKILLREFINER can improve task performance while reducing the refinement cost associated with existing approaches. Figure 2 summarizes this tradeoff across all evaluation settings by plotting held-out task performance against the number of tokens used during refinement; points toward the upper left are preferable. We report detailed token costs for each technique in Table 6.

Across all eight settings, SKILLREFINER lies on the performance–cost frontier among the evaluated configurations. It achieves the best or tied-best task performance in every setting while using substantially fewer refinement tokens than competing refinement methods. Compared with Trace2Skill, SKILLREFINER uses 1.4–13$\times$ fewer refinement tokens; where rollout-based optimization is applicable, it uses 2.4–42$\times$ fewer tokens than GEPA.

Importantly, this efficiency does not come from sacrificing task performance. For example, on SpreadsheetBench with an LLM-authored initial skill and GPT-5.4-mini, SKILLREFINER improves accuracy to 84.0%, compared with 79.0% for Trace2Skill and 78.5% for GEPA, while using $2.5\times$ fewer refinement tokens than Trace2Skill. **Across settings, SKILLREFINER either beats or matches the strongest competing method at a lower refinement cost.**

Task-level performance. Table 1 reports the performance axis separately. SKILLREFINER improves over the initial skill in all eight settings, for both LLM- and human-authored initial skills. It achieves the highest performance in six settings and ties the highest performance in the remaining two. The improvements are generally larger for MiniMax-M2.7, suggesting that explicit procedural refinement provides greater benefit when the underlying agent has more room for improvement. We additionally report the refined skill lengths in Appendix A.3.

When replay is infeasible. PR review highlights an additional advantage of offline refinement. GEPA cannot be applied in this setting because candidate skills cannot be evaluated through immediate replay with an automated feedback signal. SKILLREFINER, in contrast, can refine directly from historical outcomes and improves F1 from 0.16 to 0.18 with GPT-5.4-mini and from 0.11 to

Table 2: Ablations across all evaluation settings. Each entry reports the change in performance relative to the full SKILLREFINER pipeline; more negative is worse.

Task	Init.	Model	w/o Summ.	w/o Clusters	w/o C^+	w/o C^-	w/o Gating
SpreadsheetBench	LLM	GPT-5.4-mini	-5.5	-3.5	-1.5	-4.5	-5.5
	LLM	MiniMax-M2.7	-5.0	-11.5	0.0	-8.0	-5.0
	Human	GPT-5.4-mini	-2.0	-3.5	-0.5	-3.0	-6.0
	Human	MiniMax-M2.7	-4.5	-7.0	-1.0	-8.0	-6.5
DAPO-Math	LLM	GPT-5.4-mini	-2.0	-0.5	0.0	-2.0	-2.0
	LLM	MiniMax-M2.7	-7.0	-3.5	-1.5	-9.0	-3.0
PR Review (F1 Δ)	Human	GPT-5.4-mini	-0.07	-0.01	0.0	-0.04	-0.01
	Human	MiniMax-M2.7	-0.06	-0.03	-0.05	-0.02	-0.01

0.15 with MiniMax-M2.7. Offline refinement therefore provides both a favorable cost–performance tradeoff when replay is available and a viable refinement mechanism when it is not.

4.1 ABLATION STUDIES

We ablate five components of SKILLREFINER to assess their contributions to downstream performance, presenting the results in Table 2. Removing trajectory summarization, clustering, failure-derived refinements, or evidence-gating reduces performance across all models and domains. The contribution of success-derived refinements is smaller and varies across settings.

Trajectory summarization improves refinement. Removing trajectory summarization reduces accuracy by 2.0–7.0 percentage points on SpreadsheetBench and DAPO-Math, and F1 by 0.06–0.07 on PR review. On PR review, this is the largest reduction among the evaluated ablations for both models. These results support summarization as a useful part of the refinement pipeline, although this experiment does not distinguish the benefits of the *form* of the compact representation from those of the reduced context processed during refinement.

Clustering is particularly useful on SpreadsheetBench. Deriving refinements from individual executions rather than clusters reduces performance in every setting. On SpreadsheetBench, the reduction is 3.5 percentage points for GPT-5.4-mini with either initial skill source, and 7.0–11.5 points for MiniMax-M2.7. Clustering also improves performance on DAPO-Math and PR review, with respective reductions of 0.5–3.5 accuracy points and 0.01–0.03 F1 when it is removed.

Failure-derived proposals usually contribute more. Removing failure-derived proposals (C^-) causes a larger performance drop than removing success-derived proposals (C^+) in seven of eight settings. Across the accuracy-based tasks, removing C^- reduces performance by 2.0–9.0 percentage points, while removing C^+ reduces it by at most 1.5 points. Failure-derived proposals account for most of the observed benefit from proposal generation in these settings, while successful executions provide useful signals in some cases.

Evidence gating improves failure-derived refinement. Evidence gating applies a second-pass check to filter failure-derived proposals. Removing this reduces performance in every setting: by 5.0–6.5 percentage points on SpreadsheetBench, 2.0–3.0 points on DAPO-Math, and 0.01 F1 on PR review. Together with the C^- ablation, these scores support both generating proposals from failures and filtering those proposals before incorporating them into the skill.

4.2 QUALITATIVE ANALYSIS

We next analyze what the refinements learned by SKILLREFINER contribute to the skills they improve.

Finding a common cause across different failures. In SpreadsheetBench, several failures that look unrelated at the task level turn out to have the same underlying cause. Failures involving lookup, date filtering, row consolidation, and conditional aggregation all arise because the agent verifies

that the intended formula is present, but does not ensure that the workbook has been recalculated before checking the resulting cell value. SKILLREFINER groups these cases and learns a single rule requiring formulas to be evaluated before their resulting values are checked. This replaces several task-specific fixes with one correction that captures the common source of the failures.

Learning when not to act. We see a different pattern in some of the refinements from the PR review domain. Here, the agent often raises plausible concerns about dependency changes, lockfiles, regression tests, or PR readiness, but many of these concerns are not supported by the developer's subsequent edits. SKILLREFINER therefore learns to require concrete evidence of breakage or a policy violation before raising such concerns. Here, the refinement changes when the agent should comment rather than changing how the check itself is performed.

Restricting overly broad guidance. The refinements also reveal cases where the initial skill gives advice that is useful in general but too broad. For example, in SpreadsheetBench, the initial skill encourages the agent to prefer formulas, but this leads to brittle formula chains for tasks involving FIFO logic, occurrence tracking, and complex conditional aggregation. SKILLREFINER learns to qualify this rule, allowing the agent to use Python when the computation is difficult to express reliably as a formula.

Table 5 shows further SpreadsheetBench clusters and the edits derived from the identified clusters.

5 RELATED WORK

Optimization through task replay. A common approach to prompt and skill improvement involves proposing a revision, executing it on a task set, and using the resulting scores to guide further revisions. In this space, ProTeGi (Pryzant et al., 2023) uses natural-language critiques to improve prompts, while OPRO (Yang et al., 2024) treats an LLM as an optimizer over prompt candidates. DSPy (Khattab et al., 2024) and MIPRO (Opsahl-Ong et al., 2024) extend this idea to multi-stage LM programs and search jointly over instructions and demonstrations. TextGrad (Yuksekgonul et al., 2024) propagates textual feedback through a computation graph, whereas GEPA (Agrawal et al., 2026) uses reflective evolutionary search over candidate texts. More recent work applies similar ideas directly to agent skills. SkillOPT (Yang et al., 2026), SkillGrad (Wang et al., 2026), SkillRevise (Liu et al., 2026), CoEvoSkills (Zhang et al., 2026b), and SkillEvolver (Zhang et al., 2026a) all improve skills using feedback obtained from agent executions. SkillWeaver (Zheng et al., 2025) similarly develops skills through continued interaction with a live web environment. In contrast to these approaches, SKILLREFINER learns entirely from historical executions and does not require obtaining additional rollouts for improvement.

Learning from prior agent experience. Another line of work improves agents by reusing information from previous executions. Some methods use prior attempts to generate feedback for subsequent ones, as in Reflexion (Shinn et al., 2023) and Self-Refine (Madaan et al., 2023). Others extract reusable experience and retrieve it at inference time (Zhao et al., 2024; Suzgun et al., 2026; Tang et al., 2025). A related set of approaches turns experience into more persistent artifacts, such as manuals, workflows, or executable skills (Chen et al., 2024; Wang et al., 2025b; 2024; Li et al., 2026). In contrast, SKILLREFINER instead focuses on refining a single existing skill from a fixed corpus of past executions. Rather than retrieving prior experience at inference time or constructing a separate skill library, it compiles recurring patterns directly into the skill itself.

Learning skills from prior executions. Trace2Skill (Ni et al., 2026) is the closest prior work to our setting. It extracts lessons from individual trajectories and then aggregates them into a global skill. In contrast, SKILLREFINER first groups related executions by outcome and proposes refinements from recurring patterns across each group. For failed executions, it also incorporates evaluator feedback and filters proposed edits against the evidence in the corresponding cluster.

6 CONCLUSION

We introduced SKILLREFINER, an offline method for improving agent skills from historical executions. Rather than learning from individual traces, SKILLREFINER clusters related successful and

failed executions and uses these recurring patterns to propose targeted edits to the skill. Across spreadsheet manipulation, mathematical reasoning, and production pull-request review, SKILLREFINER improves over the initial skill in all eight settings while using substantially fewer refinement tokens than competing methods. These results show that recurring patterns in historical executions can be used to refine agent skills without repeatedly generating new rollouts.

AI USE STATEMENT

In this work we used AI tools to assist in writing code, finding related research and editing drafts of the paper, mainly towards refining the writing. The first two authors have reviewed the related work and code artifacts generated by AI tools. We take responsibility for the SKILLREFINER work and any artifacts that were generated using AI assistance.

REFERENCES

Agent Skills Contributors. Agent Skills Specification: An Open Standard for Extending AI Agent Capabilities, 2025. URL https://agentskills.io. Accessed: 2026-09-25.

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 *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RQm2KQTM5r.

Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. Density-based clustering based on hierarchical density estimates. In Jian Pei, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu (eds.), *Advances in Knowledge Discovery and Data Mining*, pp. 160–172, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37456-2.

Aili Chen, Aonian Li, Baichuan Zhou, Bangwei Gong, Binyang Jiang, Boji Dan, Changhao Zhang, Changqing Yu, Chao Wang, Cheng Ma, Cheng Zhong, Cheng Zhu, Chengjun Xiao, Chengyi Yang, Chengyu Du, Chenyang Zhang, Chi Zhang, Chuangyi Huang, Chunhao Zhang, Chunhui Du, Chunyu Zhao, Congchao Guo, Da Chen, Deming Ding, Dianjun Sun, Dong Li, Dongyu Zhang, Enhui Yang, Fei Yu, Guang Zheng, Guodong Zheng, Guohong Li, Haichao Zhu, Haigang Zhou, Haimo Zhang, Han Ding, Hao Zhang, Haohai Sun, Haolin Lyu, Haonan Lu, Haoyu Wang, Huajie Shi, Huiyang Li, Jiacheng Chen, Jian Zhang, Jiaqi Zhuang, Jiaren Cai, Jiaxin Pan, Jiayao Li, Jiayuan Song, Jichuan Zhang, Jie Wang, Jihao Gu, Jin Zhu, Jingwei Dong, Jingyang Li, Jingyu Zhang, Jingze Zhuang, Jinhao Tian, Jinli Liu, Jinyi Hu, Jun Tao, Jun Zhang, Junbin Ruan, Junhao Xu, Junjie Yan, Junteng Liu, Junxian He, Kang Xu, Ke Ji, Ke Yang, Kecheng Xiao, Keyu Duan, Keyu Li, Le Han, Letian Ruan, Li Yuan, Lianfei Yu, Liheng Feng, Lijie Mo, Lin Li, Linge Du, Lingye Bao, Lingyu Yang, Lingyuan Zhou, Loki, Lu Chen, Lunbin Zeng, Ming Li, Ming Zhong, Mingliang Tao, Mingyuan Chi, Mujie Lin, Nan Hu, Ningxin Chen, Peiyin Zhu, Peng Gao, Pengcheng Gao, Pengfei Li, Penglin Li, Pengyu Zhao, Qibin Ren, Qibing Ren, Qidi Xu, Qihan Ren, Qile Li, Qin Wang, Quanliang Chen, Qunhong Zeng, Rong Tian, Rongxin Guo, Rui Dong, Ruitao Leng, Ruize Zhang, Shanqi Liu, Shaoxiang Chen, Shaoyu Chen, Sheng Jia, Shun Yao, Shuoran Zhao, Shuqi Yu, Sichen Li, Sicheng Pan, Songquan Zhu, Tengfei Li, Tian Xie, Tiancheng Qin, Tianle Li, Tianrun Liang, Wei Liu, Weiqi Xu, Weitao Li, Weixiang Chen, Weiyu Cheng, Weiyu Zhang, Wenhu Chen, Wenqian Zhao, Xiancai Chen, Xiangjun Song, Xiangyuan Wang, Xianzhen Luo, Xiao Luo, Xiao Su, Xiaobo Li, Xiaodong Han, Xiaojie Wu, Xihao Song, Xingyi Han, Xinyu Guan, Xuan Lu, Xun Zou, Xunhao Lai, Xutong Li, Xuyang Shen, Yan Gong, Yan Ma, Yang Jiao, Yang Wang, Yang Xu, Yangsen Wang, Ye Tang, Yicheng Chen, Yihang Wang, Yinran Qiu, Yiqi Shi, Yiting Guo, Yiwen Huang, Yixuan Wang, Yongyi Hu, Yu Gao, Yu Zhang, Yuan Li, Yuanxiang Ying, Yuanzhen Zhang, Yubo Wang, Yuchen Song, Yufeng Yang, Yuhang Meng, Yuhang Miao, Yuhao Li, Yujie Liu, Yulin Hu, Yunan Huang, Yunji Li, Yunyi Huang, Yusen Zhang, Yusu Hong, Yutao Xie, Yutong Zhang, Yuwen Liao, Yuxuan Shi, Yuze Wenren, Zebin Li, Zehan Li, Zejian Luo, Zeyu Jin, Zeyuan Sun, Zhanpeng Zhou, Zhaochen Su, Zhendong Li, Zhengmao Zhu, Zhengyuan Peng, Zhenhua Fan, Zhi Zhang, Zhichao Xu, Zhiheng Lv, Zhikang Xu, Zhitao He, Zhiwei He, Zhongyuan Li, Zibo Gao, Zijia Wu, Zijian Song, Zijian

Zhou, Zijun Sun, Zishan Huang, Ziying Chen, and Ziyue Ge. The minimax-m2 series: Mini activations unleashing max real-world intelligence, 2026. URL https://arxiv.org/abs/2605.26494.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. Automanual: Generating instruction manuals by LLM agents via interactive environmental learning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=Pwl9n4zlf5.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=sY5N0zY5Od.

Yixuan Li, Mingshu Cai, Ziyang Xiao, Wanyuan Wang, Yanchen Deng, and Bo An. Mind-skill: Quality-guaranteed skill generation via multi-agent induction and deduction, 2026. URL https://arxiv.org/abs/2605.08670.

Yuxuan Liu, Zhaochen Su, Lingyun Xie, Yuhao Zhang, Qing Zong, Jiahe Guo, Zhongwei Xie, Yiyan Ji, Yauwai Yim, Hongyu Luo, Xiyu Ren, Ruan Chenyu, Haoran Li, and Yangqiu Song. Skillrevise: Improving llm-authored agent skills via trace-conditioned skill revision, 2026. URL https://arxiv.org/abs/2606.01139.

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 *The Thirty-eighth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=KYxzmRLF6i.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=S37hOerQLB.

Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2020. URL https://arxiv.org/abs/1802.03426.

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, 2026. URL https://arxiv.org/abs/2603.25158.

Ollama. Ollama, 2023. URL https://ollama.com.

OpenAI. Gpt-5.4 thinking system card, 2026. URL https://deploymentsafety.openai.com/gpt-5-4-thinking. Accessed for GPT-5.4 mini model specifications.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7957–7968, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.494. URL https://aclanthology.org/2023.emnlp-main.494/.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=vAElhFcKW6.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In Vera Demberg, Kentaro Inui, and Lluís Marquez (eds.), *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7080–7106, Rabat, Morocco, March 2026. Association for Computational Linguistics. ISBN 979-8-89176-380-7. doi: 10.18653/v1/2026.eacl-long.333. URL https://aclanthology.org/2026.eacl-long.333/.

Xiangru Tang, Tianrui Qin, Tianhao Peng, Ziyang Zhou, Daniel Shao, Tingting Du, Xinming Wei, Peng Xia, Fang Wu, He Zhu, Ge Zhang, Jiaheng Liu, Xingyao Wang, Sirui Hong, Chenglin Wu, Hao Cheng, Chi Wang, and Wangchunshu Zhou. Agent kb: Leveraging cross-domain experience for agentic problem solving, 2025. URL https://arxiv.org/abs/2507.06229.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=ehfRiF0R3a.

Hanyu Wang, Yifan Lan, Bochuan Cao, Lu Lin, and Jinghui Chen. Skillgrad: Optimizing agent skills like gradient descent, 2026. URL https://arxiv.org/abs/2605.27760.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=OJd3ayDDoF.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Forty-second International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=NTAhi2JEEE.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers, 2024. URL https://arxiv.org/abs/2309.03409.

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

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru 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, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=2a36EMSSTp.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text, 2024. URL https://arxiv.org/abs/2406.07496.

Genrui Zhang, Erle Zhu, Jinfeng Zhou, Caiyan Jia, and Hongning Wang. Skillevolver: Skill learning as a meta-skill, 2026a. URL https://arxiv.org/abs/2605.10500.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, Xue Liu, Xiaoxiao Li, and Philip S. Yu. Coevoskills: Self-evolving agent skills via co-evolutionary verification, 2026b. URL https://arxiv.org/abs/2604.01687.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'24/IAAI'24/EAAI'24. AAAI Press, 2024. ISBN 978-1-57735-887-9. doi: 10.1609/aaai.v38i17.29936. URL https://doi.org/10.1609/aaai.v38i17.29936.

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

Table 3: Dataset Statistics about the Pull Request Review Setting.

Split	Samples (PRs)	Reflected Suggestions	Avg. Reflected/PR
Learning	639	763	1.19
Eval	145	121	0.83

Table 4: Change in skill length (lines) relative to the initial skill S_0. Parentheses report the resulting skill length.

Task	Init.	Model	S_0	GEPA	Trace2Skill	SkillRefiner
SpreadsheetBench	LLM	GPT-5.4-mini	265	−228 (37)	+132 (397)	+3 (268)
	LLM	MiniMax M-2.7	416	−126 (290)	+101 (517)	−18 (398)
	Human	GPT-5.4-mini	288	0 (288)	+183 (471)	−23 (265)
	Human	MiniMax M-2.7	288	+55 (343)	+375 (663)	+249 (537)
DAPO-Math	LLM	GPT-5.4-mini	132	0 (132)	+318 (450)	+30 (162)
	LLM	MiniMax M-2.7	127	+56 (183)	+30 (157)	+3 (130)
PR Review	Human	GPT-5.4-mini	200	–	+165 (365)	−7 (193)
	Human	MiniMax M-2.7	200	–	+82 (282)	+43 (243)

A Appendix

A.1 PR Review Dataset

We provide details of the skill learning and evaluation datasets. Note, reflected suggestions are the suggestions that were incorporated into the final merged state of the PR. The average reflected suggestions per PR is 1.19 for learning and 0.83 for evaluation. This number is attributed to PRs that are modular and authored by experienced engineers. The task of the agent is to carefully review the PR and codebase and then propose targeted suggestions. We want our agent to refrain from producing redundant suggestions that an expert reviewer would discard. We observe that both gpt-5.4-mini and MiniMax m-2.7 generate redundant suggestions, while missing true positive suggestions. We label a PR as positive when atleast 50% of the agent's suggestions match a ground-truth suggestion. PRs with no suggestions are labelled positive when the agent does not produce suggestion, otherwise they are labelled negative.

A.2 Parameters for clustering

Trace summaries are embedded with qwen3-embedding:4b (4B parameters, served locally via Ollama (Ollama, 2023)). Embeddings are L2-normalized, reduced with UMAP, and clustered with HDBSCAN. UMAP uses cosine distance with n_components=20, n_neighbors=15, and min_dist=0.1. HDBSCAN then operates on the reduced space with Euclidean distance, min_cluster_size=2, min_samples=1, and excess-of-mass cluster selection. The positive and negative partitions are embedded and clustered independently. We use the same fixed clustering hyperparameters for both positive and negative partitions.

A.3 Edits made during skill refinement

Table 4 reports the change in skill length produced by each refinement method relative to the initial skill S_0. Trace2Skill consistently increases the skill length across all eight settings, suggesting that it primarily refines skills by appending additional instructions. In contrast, GEPA can substantially increase or decrease the skill length. In two evaluated settings, GEPA does not identify an improved candidate over the initial skill and therefore retains the base seed. Finally, SkillRefiner produces smaller changes in skill length than Trace2Skill across all eight settings, resulting in a shorter final skill in each case.

Table 5: Representative trace clusters discovered by cluster-level refinement on SpreadsheetBench (MiniMax-M-2.7 rollouts). Each cluster groups traces sharing a common behavioral pattern; the refiner emits one targeted skill edit per cluster rather than a single global summary. Positive clusters (REINFORCE) come from successful traces and strengthen a behavior already in the skill; negative clusters (SOFTEN) come from failed traces and soften or remove a conflicting instruction. N = number of traces in the cluster (of $\sim$200 train rollouts)

Polarity	Cluster type	N	Learned cluster-level refinement	What it addresses
REINFORCE	Dynamic formula construction (SUMPRODUCT / SUMIF / COUNTIF)	21	Codify `SUMPRODUCT`-based multi-criteria patterns (case-exact matching, expanding cumulative ranges, rank-by-count) as a "Proven Formula Patterns" subsection, always paired with a zero-error `recalc.py` check.	Generalizes formula idioms that repeatedly worked across many traces into an explicit, reusable template, instead of leaving them implicit in individual successful rollouts.
	Safe structural edits (row/column deletion & lookups)	17	Add explicit rules: store cell values before mutating, delete rows/columns in reverse index order, and re-run `recalc.py` immediately after any formula-touching edit.	Prevents a specific, recurring index-shift bug class (deleting top-down instead of bottom-up) that otherwise silently corrupts adjacent rows.
	Adaptive fallback under terminal/heredoc failure	12	Add a "Fallback Script Creation Strategies" subsection: if heredoc corrupts multi-line scripts, switch to `printf`, inline `python -c`, or have Python write its own script file.	Captures a recovery tactic successful traces converged on after their first heredoc attempt failed, so future runs skip the failed attempt entirely.
	Post-edit verification protocol	7	Add a "Verification Pattern" note: after saving, reload the output file and print the exact edited range to confirm values landed correctly at the specified output path.	Reinforces a lightweight check that distinguished traces which caught silent write/format errors before submission from traces that did not.
SOFTEN	Heredoc-based script creation (terminal corruption)	22	Insert an explicit terminal-limitation warning: do *not* use `cat « 'EOF'` heredocs for multi-line Python; use `printf`/`echo` line-by-line or have Python write the file.	The single largest failure cluster: shell heredoc syntax silently mangles quotes/newlines, costing 3–10+ failed attempts per trace before agents discovered a workaround on their own.
	Formula-first bias on complex logic	11	Soften the "always prefer formulas" directive: for FIFO/occurrence-tracking/multi-condition aggregation logic, compute in Python and write the result, rather than forcing a brittle formula chain.	The skill's blanket formula-first instruction was actively harmful on tasks whose logic formulas express poorly, producing repeated `#REF!`/`#VALUE!` cycles.
	Reactive (detect-then-fix) error handling	7	Replace "recalculate first, patch errors after" guidance with a proactive rule: wrap lookups/division/conditional formulas in `IFERROR` *before* the first `recalc.py` pass.	Removes a documented anti-pattern (fix errors only after they surface) that wasted a full recalculation cycle on predictable edge cases (empty cells, missing matches, text-as-number).

Table 6: Token usage across benchmarks, initialization settings, models, and methods.

Benchmark	Init	Model	SkillRefiner	Trace2Skill	GEPA
SpreadsheetBench	LLM	GPT-5.4-mini	2.78M	6.87M	117.22M
SpreadsheetBench	LLM	MiniMax M-2.7	7.19M	9.90M	159.87M
SpreadsheetBench	human	GPT-5.4-mini	1.98M	5.19M	43.34M
SpreadsheetBench	human	MiniMax M-2.7	5.93M	10.45M	97.00M
DAPO-Math	LLM	GPT-5.4-mini	0.79M	7.48M	2.57M
DAPO-Math	LLM	MiniMax M-2.7	1.00M	5.31M	2.42M
PR Review	human	GPT-5.4-mini	1.61M	20.84M	–
PR Review	human	MiniMax M-2.7	1.60M	9.11M	–

A.4 TOKEN USAGE

Table 6 outlines the amount of tokens consumed by each technique. We do not consider the initial cost towards collecting the rollouts for both SKILLREFINER and Trace2Skill since these are traces that have already been computed from previous tasks. We observe that SKILLREFINER consumes 1.4-13x fewer refinement tokens compared to Trace2Skill and 2.4-42x fewer tokens compared to GEPA.

A.5 PROMPTS USED FOR SKILLREFINER

A.5.1 FEEDBACK AUGMENTED TRAJECTORY SUMMARIZATION PROMPT

Feedback Augmented Trajectory Summarization Prompt used for SkillRefiner (polarity=positive)

```
Below is a cleaned execution trace of an AI agent run.
Each section is one agent turn or tool result.

--- TRACE START ---
{trace_content}
--- TRACE END ---

{header} Reconstruct the behavioral recipe that produced success.

Analyze this trace and answer:
1. What did the agent do? List the key tool calls (name + what it was
trying to accomplish) and what each returned.
2. Which decisions, orderings, or checks were instrumental to the
successful outcome?
3. What is the behavioral recipe? State the reusable rules that made
this run succeed.

Reply as JSON (no markdown wrapper):
{
"summary":
    "<3-5 sentence account of what the agent did and why it succeeded>",
"observations":
    [
        "<codifiable behavior worth reinforcing>",
        "..."
    ]
}

observations must be codifiable behaviors rules you could write into
a skill so the pattern repeats, e.g. "agent verified the diff against
the files manifest before commenting" not "agent was careful".
```

Summarization Prompt used for SkillRefiner (polarity=negative)

```
Below is a cleaned execution trace of an AI agent run. Each section
is one agent turn or tool result. Internal model reasoning has been
stripped you are seeing only tool calls, tool results, and agent outputs.

--- TRACE START ---
{trace_content}
--- TRACE END ---

{header} Diagnose the failure chain.

Analyze this trace and answer:
1. What did the agent do? List the key tool calls (name + what it was
trying to accomplish) and what each returned.
2. Diagnose the failure chain: what was the earliest decision that led
the run astray, and what would a correct trace have done there?
3. What instruction or guardrail would have prevented this failure?

Reply as JSON (no markdown wrapper):
{"summary": "<3-5 sentence account of what the agent did and where
```

```
it went wrong>", "observations": ["<failure mode phrased as
a preventable rule>", "..."]}

observations must be failure modes phrased as preventable rules,
e.g. "agent skipped reading the diff before commenting require a
diff read first" not "agent made mistakes".
```

A.5.2 Proposal Generation Prompts

Proposal Generation Prompt used for SkillRefiner (polarity=positive)

```
Current skill content:
```markdown
{skill_content}
```

The following {n} agent trace summaries form a semantic cluster (cluster {
    cluster_id}). All traces in this cluster come from runs that PASSED their
    evaluation -- they represent validated, successful behavior.

{summaries_block}

Based on this cluster of successful traces:
1. Identify the common theme across these traces.
2. Identify the specific behaviors that drove success and should be REINFORCED in
     the skill -- patterns to codify so they repeat reliably.
3. Propose one concrete, targeted edit to the skill that codifies the winning
    pattern.

Do NOT propose removals or warnings -- these traces succeeded; your job is to
    lock in what worked.

Reply as JSON (no markdown wrapper):
{"theme": "<what these traces have in common>", "reinforce": ["<behavior to
    strengthen>", "..."], "suggested_edit": "<one concrete minimal edit that
    codifies the winning pattern>"}
```
```

Proposal Generation Prompt used for SkillRefiner (polarity=negative)

```
Current skill content:
```markdown
{skill_content}
```

The following {n} agent trace summaries form a semantic cluster (cluster {
    cluster_id}). All traces in this cluster come from runs that FAILED their
    evaluation -- they represent a recurring failure mode.

{summaries_block}

Based on this cluster of failing traces:
1. Identify the common theme across these traces.
2. Identify the specific behaviors that caused failure and should be SOFTENED or
    removed -- instructions that misled the agent, missing guardrails, or
    friction that wasted effort.
```
```

```
3. Propose one concrete, targeted edit to the skill -- a guardrail to add or a
    misleading instruction to remove -- that would prevent this failure mode
    from repeating.

Do NOT invent praise -- these traces failed; your job is to stop the failure from
     repeating.

Reply as JSON (no markdown wrapper):
{"theme": "<what these traces have in common>", "soften": ["<behavior to remove
    or de-emphasise>", "..."], "suggested_edit": "<one concrete minimal edit
    that prevents this failure mode>"}
```

A.5.3 Evidence Gating Prompt

Evidence Gating prompt used for negative clusters

```
You are auditing a proposed skill edit for cluster {cluster_id}. The edit was
    inferred from a cluster of FAILING agent traces and is meant to prevent a
    recurring failure mode. Your job is to decide whether the proposed lesson is
     actually SUPPORTED by the trace evidence below -- not merely plausible.

Proposed theme:
  {proposal.theme}

Proposed behaviors to soften/remove:
{soften}

Proposed edit:
  {proposal.suggested_edit}

Trace evidence (the failing traces this was inferred from):

{summaries_block}

A proposal is SUPPORTED only if the traces show concrete evidence that the named
    behavior actually occurred AND plausibly caused the failures. Reject
    proposals that generalize beyond the evidence, blame a cause the traces do
    not show, or restate a generic best-practice not grounded in these traces.

Reply as JSON (no markdown wrapper):
{"supported": true/false, "reason": "<one sentence citing the evidence>"}
```

A.5.4 Skill Merging Prompt

Merge Prompt used for SkillRefiner

````
Current skill content:
```markdown
{skill_content}
```

The following {len(proposals)} cluster(s) of agent behavior were identified and
    analyzed across {total_traces} traces:

{cluster_1}
{cluster_2}
...
````

```
Clusters marked polarity=positive come from PASSING traces - validated behaviors
    to preserve and strengthen. Clusters marked polarity=negative come from
    FAILING traces - failure modes that ACTUALLY occurred, and must be weighed
    as seriously as the positive signal (a failure is at least as strong
    evidence as a success pattern of equal cluster_size). Validated behaviors
    are the default, NOT sacrosanct. Conflict rule: when a negative cluster
    shows that a behavior a positive cluster endorses is being over-applied,
    over-trusted, or misused, do NOT keep the positive text loud and unchanged -
     qualify, scope, or condition that behavior AT ITS SOURCE so the caveat sits
     together with the behavior it limits, editing the existing text down rather
     than appending a distant caveat. Delete a validated behavior outright only
    when the failure evidence directly contradicts it; otherwise narrow it to
    the cases where it still holds.

Reply as JSON (no markdown wrapper):
{"proposed_content": "<full revised skill, placeholders intact>", "rationale":
    "<1-3 sentence explanation of what changed and why>", "key_observations":
    ["<recurring behavior that drove a change>", "..."], "confidence": <self-
    assessed confidence, 0.0 to 1.0>}
```

A.6 LLM-GENERATED PROMPTS

A.6.1 SPREADSHEETBENCH

LLM-Generated skill for SpreadsheetBench (GPT-5.4-mini)

```
---
name: xlsx
description: Use this skill when creating, editing, analyzing, or validating
    spreadsheet files in .xlsx, .xlsm, .csv, or .tsv format. It is especially
    useful for targeted workbook edits, formula updates, formatting changes,
    data cleanup, and pre-delivery verification using openpyxl.
---

# xlsx

Use this skill to work with spreadsheet files safely and predictably using `
    openpyxl` for Excel-compatible workbooks. Prefer small, targeted changes
    over broad rewrites. Preserve existing workbook structure, formulas,
    formatting, and metadata unless the task explicitly requires otherwise.

## Core principles

- Make the minimum necessary change.
- Inspect first, edit second, verify last.
- Preserve:
  - sheet names and order
  - cell values, formulas, and number formats
  - merged cells, filters, freeze panes, conditional formatting, data validation,
     charts, and images when possible
  - workbook properties and macros when handling `.xlsm`
- Be careful with formulas:
  - `openpyxl` reads and writes formulas, but does not calculate them.
  - Editing precedent cells may leave cached formula results stale until Excel
    recalculates the workbook.
- Prefer explicit, traceable edits:
  - change only the intended cells/ranges
  - avoid rewriting entire sheets unless necessary
  - keep a record of what changed and why
```

```
## File type guidance

### `.xlsx`
Standard workbook format. Fully supported for reading and writing with `openpyxl
    `.

### `.xlsm`
Macro-enabled workbook. `openpyxl` can preserve VBA if loaded with `keep_vba=True
    `. Do not expect to modify macro code; preserve it unless the task requires
    otherwise.

### `.csv`
Flat text data. No workbook features such as formulas, formatting, or multiple
    sheets. Treat as a single table.

### `.tsv`
Flat text data separated by tabs. Same limitations as CSV.

## Inspecting a workbook before editing

Before making changes, understand the workbook structure and content.

### Identify the file type and goals
- Confirm whether the file is `.xlsx`, `.xlsm`, `.csv`, or `.tsv`.
- Clarify the requested outcome:
  - data edit
  - formula update
  - formatting change
  - cleanup
  - extraction/analysis
  - conversion between tabular formats

### Load safely
Use `openpyxl.load_workbook()` for Excel workbooks.

Recommended options:
- `data_only=False` when you need formulas
- `keep_vba=True` for `.xlsm`
- `read_only=True` for large read-only inspections
- `keep_links=True` when preserving external links matters

### Inspect workbook structure
Check:
- sheet names and order
- active sheet
- dimensions and populated ranges
- hidden sheets or rows/columns
- merged cells
- freeze panes
- filters and tables
- named ranges
- data validation
- conditional formatting
- charts, images, comments, hyperlinks
- workbook calculation settings if relevant

### Inspect the data
For each relevant sheet:
- identify header rows
- determine whether there are multiple tables or sections
- locate formulas, constants, blanks, and errors
- check for mixed data types in important columns
```

```
- note number formats, date styles, and text wrapping
- detect leading/trailing whitespace and inconsistent labels

### Decide the safest edit strategy
Choose one of these:
- direct cell edits for isolated changes
- row/column insertion or deletion with reference updates as needed
- range-level copy/paste for repetitive transformations
- table-aware updates when working inside structured tables
- export/import only when the workbook is simple and workbook features are not
    important

## Making targeted edits

### General editing rules
- Change only the cells that need to change.
- Preserve formulas unless the task asks to replace them.
- If shifting cells, consider the impact on:
  - formulas
  - merged cells
  - charts
  - named ranges
  - data validation
  - conditional formatting
  - tables
- When replacing values, keep formatting intact unless otherwise specified.
- When clearing cells, decide whether to clear:
  - value only
  - formula only
  - formatting
  - comments/hyperlinks/validation
  - entire row or column content

### Editing cell values
- Assign values directly for literals.
- Use `datetime`, `date`, or numeric types as appropriate.
- Preserve text values that look like numbers if the original column is textual.
- Do not accidentally convert identifiers such as ZIP codes, account numbers, or
    IDs into numbers.

### Editing formulas
- Update formulas only where necessary.
- Use relative/absolute references carefully.
- If inserting or deleting rows/columns, check that formulas still point to the
    correct ranges.
- When copying formulas, make sure references adjust the way you intend.
- For complex workbooks, inspect precedent and dependent cells manually after
    structural changes.

### Inserting and deleting rows or columns
- Use workbook operations that shift cells when possible.
- Verify that related formulas, formatting, and merged cells remain sensible.
- Check that table ranges expand or shrink correctly.
- Re-evaluate any hard-coded references in formulas, charts, and validations.

### Formatting changes
Prefer narrow formatting edits:
- font
- fill
- border
- alignment
- number format
- row height / column width
```

```
Keep existing styles unless the task requires a new style. When applying styles
    across many cells, reuse style objects where practical.

### Working with ranges and tables
- Identify tables and their boundaries before editing.
- If modifying data inside a table, preserve header structure and table formulas.
- If adding rows beneath a table, ensure the table range extends if needed.
- Avoid breaking structured references.

### Working with comments, notes, and hyperlinks
- Preserve comments unless the request concerns them.
- If replacing a cell's content, verify whether you should also preserve its
    hyperlink or note.
- If copying cells, remember hyperlinks and comments may need special handling.

## Handling formulas and recalculation

`openpyxl` does not calculate formulas. It can preserve formulas, but not compute
     their results.

### When editing formula-dependent workbooks
- If you change input cells used by formulas, expect cached results to become
    outdated until recalculation occurs in Excel or another calculation engine.
- If the workbook must open with updated displayed results, ensure it is set to
    recalculate.

### Recalculation strategy
If appropriate:
- set workbook calculation properties to force recalculation on open
- preserve formulas as formulas
- avoid replacing formulas with hard-coded results unless requested

Typical approaches:
- set full recalculation on load/open in Excel
- mark workbook calculation as dirty if supported by the environment
- document that formulas will recalculate when opened in Excel

### Formula verification
After edits:
- inspect formula cells for syntax preservation
- ensure formulas still reference the intended cells/ranges
- check for obvious broken references like `#REF!`
- confirm array/spill-related behavior if the workbook uses modern Excel formulas
    , noting that `openpyxl` support may be limited

### If you need values instead of formulas
- Use a workbook opened with `data_only=True` to inspect cached results.
- Remember that cached values may be stale if the workbook has not been
    recalculated recently.
- Do not save over a workbook in a way that destroys formulas unless the task
    explicitly requests formula removal.

## CSV and TSV handling

For `.csv` and `.tsv` files:
- treat them as plain tabular text
- preserve delimiter, quoting, and line endings when possible
- be careful with embedded commas, tabs, quotes, and newlines
- preserve leading zeros and exact text values where relevant
- avoid unintended type coercion when round-tripping data

Because CSV/TSV lack workbook features:
```

```
- formulas are not preserved as formulas
- formatting, merged cells, tables, and multiple sheets do not exist
- analysis should focus on rows, columns, headers, and data quality

## Verification before finishing

Always verify the result before completing the task.

### Structural verification
Confirm:
- expected sheets exist and are in the right order
- no unintended sheets were altered or deleted
- dimensions/ranges are correct
- inserted/deleted rows or columns affected the intended area only
- merged cells, tables, filters, and validations still make sense

### Content verification
Check:
- edited cells contain the intended values
- unaffected cells remain unchanged where expected
- formulas still appear correct
- formatting changes applied only where requested
- key headers, labels, and totals remain consistent

### Formula and calculation verification
- Review formula cells for broken references.
- If possible, inspect cached results or recalc behavior.
- Ensure the workbook is set up so Excel will recalculate when needed.

### File integrity verification
- Save the workbook and reopen it if needed to confirm it loads successfully.
- For `.xlsm`, ensure VBA is preserved when required.
- For CSV/TSV, confirm delimiter and quoting are correct and the file is still
    parseable.

### Deliverable check
Before finishing, confirm:
- the file is in the requested format
- the requested edits were completed
- no unintended damage was introduced
- any limitations or recalculation requirements are clearly understood

## Recommended workflow

1. Inspect the workbook structure and data.
2. Determine the smallest safe change.
3. Edit only the relevant cells/ranges.
4. Preserve formulas and workbook features whenever possible.
5. Save with appropriate file settings.
6. Reopen or otherwise verify the result.
7. Confirm recalculation expectations for formula-driven files.

## Practical cautions

- Do not assume the first row is always a header.
- Do not assume all sheets are tabular.
- Do not overwrite existing styles accidentally when only changing values.
- Do not convert formulas to values unless explicitly requested.
- Do not break macros in `.xlsm` by saving without VBA preservation.
- Do not trust cached formula values if the workbook may be stale.
- Do not use broad replace operations without checking their impact.

## Good defaults
```

```
- Use `openpyxl` for Excel workbook edits.
- Preserve workbook features by default.
- Load with `keep_vba=True` for macro-enabled files.
- Inspect formulas before and after structural changes.
- Verify the workbook opens cleanly and the requested changes are present.
```

LLM-Generated skill for SpreadsheetBench (MiniMax-M2.7)

````
---
name: xlsx
description: >
  Skill for creating, editing, and analyzing Excel spreadsheets (`.xlsx`, `.xlsm
    `, `.csv`, `.tsv`)
  using the `openpyxl` library. Use this skill when you need to programmatically
    read,
  modify, or generate workbook files, update formulas, preserve styling, add or
    remove
  sheets, handle named ranges, data validation, or convert tabular data from CSV/
    TSV
  into a proper Excel workbook. This skill covers inspection, targeted edits,
    formula
  handling, and verification before finalizing the file.
---

# XLSX Skill

## When to Use This Skill

- You need to **open an existing Excel file** and change specific cells, rows,
    columns, or sheet properties.
- You are **generating a new workbook** from scratch (reports, templates, data
    exports).
- You want to **add, modify, or audit formulas** and ensure they evaluate
    correctly.
- You need to **preserve or apply styling** (fonts, fills, number formats, merged
     cells).
- You are **importing CSV/TSV data** into a native Excel format or exporting an
    Excel sheet to CSV/TSV.
- You need to **verify the result** of edits before delivering the file.

## 1. Inspect a Workbook Before Editing

### 1.1 Load the Workbook

```python
from openpyxl import load_workbook

# Open for read/write (default). Use read_only=True for large files.
wb = load_workbook('path/to/file.xlsx')

# For macro-enabled files, keep the VBA project intact:
# wb = load_workbook('path/to/file.xlsm', keep_vba=True)
```

### 1.2 Explore the Structure

| Task | Code |
|------|------|
| List all sheet names | `wb.sheetnames` |
````

````
| Select a sheet | `ws = wb['Sheet1']` |
| Get the active sheet | `ws = wb.active` |
| Sheet dimensions (rows x cols) | `ws.dimensions`, `ws.max_row`, `ws.max_column`
    |
| Iterate rows (values only) | `for row in ws.iter_rows(min_row=1, max_row=10,
   values_only=True): print(row)` |
| Iterate columns | `for col in ws.iter_cols(min_col=1, max_col=5): ...` |

### 1.3 Check Named Ranges & Defined Names

```python
for name in wb.defined_names.definedName:
    print(name.name, name.attr_text)   # name and its reference, e.g., '
    SalesTotal' -> "'Sheet1'!$B$2:$B$100"
```

### 1.4 Inspect Cell Values, Types, and Formulas

```python
cell = ws['A1']
print('value         :', cell.value)        # formula string if a formula,
    otherwise the cached/constant value
print('data_type     :', cell.data_type)   # 'n' numeric, 's' string, 'f' formula
    , 'b' boolean, etc.
print('number_format :', cell.number_format)  # e.g., '0.00%', 'MM/DD/YYYY'
```

- **Formulas**: `cell.value` returns the formula text (e.g., `"=SUM(B1:B10)"`).
- **Cached values**: Opening with `data_only=True` returns the last calculated
    value (may be stale). Use the default mode for editing.

### 1.5 Review Styles, Merged Cells, Data Validation, and Protection

```python
# Font, fill, border, alignment
print('font      :', cell.font)
print('fill      :', cell.fill)
print('border    :', cell.border)
print('alignment :', cell.alignment)

# Merged cell ranges
print('merged cells:', list(ws.merged_cells.ranges))

# Data validation objects
for dv in ws.data_validations.dataValidation:
    print(dv)

# Sheet protection
print('protection:', ws.protection.sheet, ws.protection.password)
```

### 1.6 Understand File-Type Capabilities

| Extension | openpyxl Support | Remarks |
|-----------|------------------|---------|
| `.xlsx`   | Full read/write | Default modern Excel format. |
| `.xlsm`   | Read, write (preserve macros) | Use `keep_vba=True`; macro code
    cannot be edited. |
| `.csv` / `.tsv` | Read via Python `csv` or load as single-sheet workbook | No
    native styles; after conversion you can add styling. |

## 2. Making Targeted Edits
````

````
### 2.1 General Workflow

1. **Load** the workbook (or create a new `Workbook()`).
2. **Navigate** to the target sheet(s).
3. **Modify** cells, rows/columns, formulas, styles, or workbook structure.
4. **Save** with `wb.save('output.xlsx')`.

### 2.2 Editing Cell Values

```python
ws['C3'] = 'New Text'               # string
ws['D4'] = 42                       # integer
ws['E5'] = 3.14                     # float
ws['F6'] = datetime(2025, 12, 31)  # date (datetime object)
```

- **Dates** must be `datetime` objects; openpyxl stores them as Excel serial
    dates automatically.
- **Booleans**: `True` / `False`.

### 2.3 Inserting / Deleting Rows & Columns

```python
# Insert a blank row above row 5
ws.insert_rows(5)

# Delete two columns starting at column B
ws.delete_cols(2, amount=2)

# Add a new sheet before the active sheet
new_ws = wb.create_sheet('Data', 0)
```

### 2.4 Updating Formulas

- **Write a formula** (start with `=`):

```python
ws['G1'] = '=SUM(A1:F1)'

# Example: apply a formula across a range
for row in ws.iter_rows(min_row=2, max_row=10, min_col=1, max_col=6):
    for cell in row:
        cell.value = f'={cell.coordinate}*2'   # simple transformation
```

- **Absolute / relative references**: Use `$` for locking (`$A$1`, `A$1`, `$A1`)
    .
- **Multi-sheet references**: `=Sheet2!A1+Sheet3!B2`. Quote sheet names with
    spaces: `'Jan 2025'!A1`.

### 2.5 Preserving and Applying Styles

```python
from openpyxl.styles import Font, PatternFill, Alignment

header_font   = Font(bold=True, color="FFFFFF")
header_fill   = PatternFill(start_color="4472C4", end_color="4472C4", fill_type="
    solid")
header_align  = Alignment(horizontal="center", vertical="center")

for col in range(1, ws.max_column + 1):
    cell = ws.cell(row=1, column=col)
````
````

````
    cell.font = header_font
    cell.fill = header_fill
    cell.alignment = header_align
    cell.number_format = '0.00'      # optional explicit number format
```

- **Copy a style** from another cell: `target_cell._style = source_cell._style`.

### 2.6 Handling Merged Cells

- **Unmerge** before editing a cell inside a merged range, then **re-merge**
    after.

```python
ws.unmerge_cells('A1:C1')   # split
ws['B1'] = 'New Value'
ws.merge_cells('A1:C1')    # re-merge
```

### 2.7 Column Width / Row Height

```python
ws.column_dimensions['A'].width = 20
ws.row_dimensions[1].height = 30
```

### 2.8 Adding or Renaming Sheets

```python
ws = wb.create_sheet('Summary')
ws.title = 'Summary 2025'
```

### 2.9 Working with CSV / TSV Files

- **Read** using `pandas` or the built-in `csv` module, then write to an `
    openpyxl` workbook:

```python
import pandas as pd
from openpyxl import Workbook

df = pd.read_csv('data.csv')               # delimiter detection handled
    automatically
wb = Workbook()
ws = wb.active
for row in df.itertuples(index=False):
    ws.append(list(row))
wb.save('data.xlsx')
```

- **Write back to CSV**:

```python
df.to_csv('output.csv', index=False)
```

### 2.10 Sheet Protection (Optional)

```python
ws.protection.sheet = True
ws.protection.password = 'secret'
```
````
````

````
> **Note**: openpyxl only records protection settings; the protection is enforced
     when the file is opened in Excel.

## 3. Handling Formulas and Recalculation

### 3.1 Writing Formulas

- Always store the **formula string** (beginning with `=`). Example:

```python
ws['B10'] = '=SUM(B2:B9)'
```

### 3.2 Relative vs. Absolute References

- Relative references adjust automatically when copying/pasting.
- Use `$` to lock rows/columns when you need fixed references (`$A$1`, `A$1`, `
    $A1`).

### 3.3 Multi-Sheet / External References

- Prefix sheet names with quotes if they contain spaces: `'Jan 2025'!A1`.
- External file links (`=[Book2.xlsx]Sheet1!$A$1`) are possible but risky; Excel
    will prompt to update links when opened.

### 3.4 Array Formulas (CSE)

- Openpyxl can store array formulas using the `{=...}` notation (curly braces are
     display only in Excel):

```python
ws['D1'] = '{=MAX(IF(A:A>0,A:A))}'   # array formula entered as a string
```

### 3.5 Cached Values vs. Formula Mode

| Mode | `cell.value` returns | Typical use |
|------|----------------------|-------------|
| Default (`data_only=False`) | Formula text (e.g., `"=SUM(B2:B9)"`) | Editing
    formulas |
| `data_only=True` | Last cached numeric result (may be stale) | Reading pre-
    calculated values only |

- **Recalculation options**:
  1. **Excel**: Open the file; Excel recalculates automatically if auto-calc is
     enabled.
  2. **Automated**: Use external engines like `xlcalculator` or `pycel` to
     compute results without Excel.
  3. **Manual caching**: After writing a formula, you may optionally store a pre-
     computed value as a comment for documentation, but the formula itself
     remains the source of truth.

```python
from openpyxl.comments import Comment
ws['B10'].comment = Comment('Pre-computed: 150', 'Agent')
```

### 3.6 Updating Named Ranges

If you add or delete rows that affect a defined name, update its reference:

```python
````
````

````
if 'MyRange' in wb.defined_names:
    dn = wb.defined_names['MyRange']
    dn.attr_text = "'Sheet1'!$A$1:$A$10"
```

## 4. Verifying the Result Before Finishing

### 4.1 Save and Reload

```python
output_path = 'edited_workbook.xlsx'
wb.save(output_path)

# Reload in a fresh context
wb_verify = load_workbook(output_path)
ws_verify = wb_verify.active
```

### 4.2 Basic Assertions (Cell Values)

```python
assert ws_verify['C3'].value == 'New Text', "C3 mismatch"
assert ws_verify['D4'].value == 42, "D4 mismatch"
```

### 4.3 Check Formulas

```python
assert ws_verify['G1'].data_type == 'f', "G1 should be a formula"
assert ws_verify['G1'].value == '=SUM(A1:F1)', "Formula text mismatch"
```

### 4.4 Validate Styles

```python
assert ws_verify['A1'].font.bold == True, "Header should be bold"
assert ws_verify['A1'].fill.fgColor.rgb == 'FF4472C4', "Header fill color
    mismatch"
```

### 4.5 Structural Checks

- **Sheet names**:
  ```python
  assert 'Summary 2025' in wb_verify.sheetnames
  ```
- **Dimensions**:
  ```python
  assert ws_verify.max_row == expected_rows
  assert ws_verify.max_column == expected_cols
  ```
- **Merged cells**:
  ```python
  assert 'A1:C1' in [str(r) for r in ws_verify.merged_cells.ranges]
  ```

### 4.6 Data Consistency for CSV/TSV Conversion

- Compare row and column counts between the source CSV and the generated workbook
    .
- Verify delimiter detection and quoting consistency.

### 4.7 Optional Visual Preview
````
````

````
If you have `pandas` and a display environment:

```python
import pandas as pd
df_preview = pd.read_excel(output_path, sheet_name='Summary 2025')
print(df_preview.head())
```

### 4.8 Final Checklist

- [ ] File saved without exceptions.
- [ ] All target cells contain expected values or formulas.
- [ ] Formulas reference correct ranges; absolute/relative references are as
    intended.
- [ ] Named ranges, data validations, and protection settings are preserved (if
    required).
- [ ] Styles (fonts, fills, number formats) appear correct.
- [ ] No unintended merged cells remain.
- [ ] CSV/TSV conversions maintain order and delimiter consistency.
- [ ] Macro-enabled files (`.xlsm`) still open without corruption.

## 5. Best Practices & Tips

| Practice | Reason |
|----------|--------|
| **Always keep a backup** of the original workbook before overwriting. |
    Prevents data loss if the script has bugs. |
| **Use a context manager or explicit `wb.close()`** (not required by openpyxl
    but good practice). | Guarantees file handles are released. |
| **Prefer formulas over hard-coded numbers** for derived metrics. | Makes the
    spreadsheet self-updating. |
| **Set `number_format` explicitly** when a specific display is needed (e.g.,
    `0.00%`). | Ensures consistency across Excel versions. |
| **Minimize loading large files for read-only tasks** (`read_only=True`). |
    Improves performance and memory usage. |
| **When inserting/deleting rows, refresh any cached references** (e.g., named
    ranges). | Prevents broken references after structural changes. |
| **Test formula evaluation with a lightweight engine** (`xlcalculator`) before
    final delivery. | Confirms that formulas produce expected results without
    opening Excel. |
| **Document any manual steps** required after script execution (e.g., enabling
    macros). | Helps the end-user understand what the script cannot automate. |
| **Use `datetime` objects for dates**; avoid string dates. | Excel treats them
    as serial numbers for calculations. |
| **Avoid editing locked/protected cells** unless protection is removed. |
    Prevents `AttributeError` and maintains intended security. |

## 6. Common Pitfalls & Fixes

| Pitfall | Symptom | Fix |
|---------|---------|-----|
| Editing in `data_only=True` mode | Changes not saved because mode is read-only
    for values. | Re-open with `data_only=False` (the default). |
| Merged cell editing | `KeyError` or unexpected overwriting. | Unmerge before
    writing, then re-merge if needed. |
| Forgot to call `wb.save()` | No file output. | Add `wb.save(path)` after all
    edits. |
| Incorrect sheet name (case sensitivity) | `KeyError`. | Use exact case as shown
     in `wb.sheetnames`. |
| Date stored as string | Formula `DATEVALUE` errors. | Convert Python `datetime`
     to Excel serial via openpyxl; use `datetime` objects. |
````

```
| Large file performance | Slow load/save. | Use `read_only=True` for reading;
    consider `write_only=True` for massive writes. |
| Unsupported file format (e.g., `.xls`) | `InvalidFileException`. | Convert `.
    xls` to `.xlsx` using `pandas` or `pyexcel`. |

## 7. Quick Reference Commands

```python
# Load workbook
wb = load_workbook('file.xlsx')

# Create new workbook
wb = Workbook()

# Access sheet
ws = wb['Sheet1']

# Save
wb.save('output.xlsx')

# Inspect a cell
cell = ws['A1']
print(cell.value, cell.data_type)

# Write a formula
ws['B1'] = '=SUM(A1:A10)'

# Apply style
from openpyxl.styles import Font
ws['A1'].font = Font(bold=True)

# Insert row
ws.insert_rows(3)

# Delete column
ws.delete_cols(5)

# Merge cells
ws.merge_cells('D4:E4')

# Unmerge
ws.unmerge_cells('D4:E4')
```

```

A.6.2 DAPO

LLM-Generated skill for DAPO (GPT-5.4-mini)

```
---
name: dapo-math
description: Use this skill for competition-style math word problems that require
     careful parsing, multi-step reasoning, algebraic setup, geometry, counting,
     probability, number theory, or a verified final numeric or symbolic answer.
---

# dapo-math
```

```
This skill helps solve competition-style math word problems with clear,
    disciplined reasoning and a final answer that has been checked for
    consistency.

## Core approach

1. **Read the problem carefully**
   - Identify exactly what is being asked.
   - Note all given quantities, constraints, and hidden assumptions.
   - Distinguish between what is stated and what must be inferred.
   - Restate the problem in your own words if helpful.

2. **Organize the information**
   - List variables, definitions, and relevant relationships.
   - If the problem has multiple parts, break it into subproblems.
   - Track units, domains, and conditions such as integers, positivity, parity,
     or bounds.

3. **Choose a solution strategy**
   - Look for a natural method based on the problem type:
     - **Algebraic setup** for equations, systems, and relationships.
     - **Casework** when conditions split naturally.
     - **Geometric reasoning** for shapes, angles, areas, and similarity.
     - **Counting arguments** for combinatorics.
     - **Probability rules** for random processes and expected value.
     - **Number theory tools** for divisibility, modular arithmetic, primes, gcd/
     lcm, and Diophantine equations.
     - **Invariants, symmetries, or extremal arguments** when the problem
     suggests them.
   - Prefer the simplest valid method that fully addresses the question.

4. **Show working clearly**
   - Write out the logic step by step.
   - Justify key transformations and conclusions.
   - Avoid skipping major inferential steps.
   - If using a formula or theorem, ensure it is applicable and explain why.
   - Keep calculations organized and readable.

5. **Check the result**
   - Verify that the answer satisfies the original question.
   - Substitute back where possible.
   - Check units, sign, magnitude, and domain restrictions.
   - Make sure no extraneous solutions remain.
   - For counting or probability problems, ensure the count/sample space is
     correct.
   - For geometry, confirm consistency with angle sums, side lengths, and
     plausibility.
   - For number theory, confirm divisibility and congruence conditions.

6. **Finish with a precise final answer**
   - State the answer clearly and concisely.
   - If the problem asks for a set, list all elements.
   - If there are multiple valid forms, provide the one most appropriate to the
     question.
   - If approximation is requested, include the correct level of precision.
   - If exactness is required, avoid unnecessary decimal approximations.

## Problem-solving habits

- **Define variables early.**
  Use names that match the meaning of the quantities.

- **Watch for hidden constraints.**
```

```
  Problems may require integers, positive values, distinct objects, non-
    overlapping intervals, or geometric feasibility.

- **Use structure before computation.**
  Look for simplifications such as symmetry, factoring, telescoping, cancellation
    , or invariants.

- **Break complex expressions into parts.**
  Compute intermediate results cleanly instead of trying to do everything at once
    .

- **Prefer exact reasoning.**
  Work symbolically when possible; approximate only when the problem calls for it
    .

- **Be cautious with "obvious" conclusions.**
  Confirm them with algebra or logic.

## Common strategy patterns

### Algebra
- Translate the statement into equations.
- Solve systematically.
- Check for extraneous roots or impossible values.
- If expressions are messy, substitute helpful variables or use identities.

### Geometry
- Draw an accurate mental or rough diagram.
- Mark equal angles, equal lengths, parallel lines, and cyclic relationships.
- Use congruence, similarity, coordinates, vectors, trigonometry, or area
    relations as appropriate.
- Verify that the proposed configuration is geometrically consistent.

### Counting
- Decide whether order matters.
- Determine whether objects are distinct or identical.
- Use systematic casework, complements, bijections, inclusion-exclusion, or
    recursion if helpful.
- Avoid double counting.

### Probability
- Define the sample space clearly.
- Use symmetry when valid.
- Compute favorable outcomes over total outcomes, or use conditional probability
    carefully.
- For expected value, linearly decompose the quantity into simpler indicator
    variables when useful.

### Number theory
- Consider divisibility, parity, modular arithmetic, prime factorization, gcd/lcm
    , and digit constraints.
- Use congruences to narrow possibilities.
- Check whether solutions must be integers and whether all integer conditions are
     met.

## Quality standards

A good solution under this skill should:
- Be logically complete.
- Be easy to follow.
- Use the most appropriate mathematical tools.
- End with a verified answer.
- Avoid unsupported leaps, vague statements, or unexamined assumptions.
```

```
## Final answer format

- Provide the final result explicitly.
- If there is a single answer, state it directly.
- If there are multiple answers, enumerate them clearly.
- If the problem asks for an equation, expression, or proof conclusion, present
    it in a clean final form.
- If relevant, include a short verification sentence confirming consistency with
    the problem.

## Mindset

Treat each problem as a precise puzzle:
- understand it,
- structure it,
- solve it,
- check it,
- and then answer it cleanly.
```

LLM-Generated skill for DAPO (MiniMax-M2.7)

```
---
name: dapo-math
description: A step-by-step skill for solving competition-style math word
    problems. Use this skill whenever an agent must interpret a non-routine
    problem, formulate a plan, carry out the solution, and verify the final
    answer.
---

## Overview
The goal of `dapo-math` is to guide an agent through a disciplined workflow that
    turns a word problem into a clear, correct answer. The workflow is divided
    into four core phases--**parse**, **plan**, **execute**, and **verify**--
    with supporting best practices that improve reliability and readability.

---

## Phase 1 - Parse the Problem

1. **Read carefully**
   - Read the entire statement at least twice.
   - Note the exact wording of the question (e.g., "how many", "what is the sum",
     "find the value of").

2. **Identify the target**
   - Determine what you are asked to find (the unknown quantity).
   - Write a concise statement of the goal in your own words.

3. **Extract given data**
   - List all numerical values, relationships, and constraints explicitly.
   - Assign symbolic names (variables) to unknowns as soon as they appear.

4. **Check units**
   - Verify that all quantities share consistent units (or note necessary
     conversions).
   - Highlight any implied units (e.g., "per hour", "per meter").

5. **Spot hidden information**
```

```
   - Look for implied constraints (e.g., "integer solutions", "positive", "non-
     zero").
   - Detect any special conditions (e.g., "the triangle is right-angled").

6. **Re-express in mathematical form**
   - Translate the narrative into equations, inequalities, or combinatorial
     expressions.
   - Use diagrams or tables when they clarify the situation (e.g., Venn diagrams,
     geometry sketches).

---

## Phase 2 - Choose a Solution Strategy

1. **Classify the problem type**
   - Algebraic (equations, systems, polynomials)
   - Geometric (area, volume, angle chasing)
   - Combinatorial (counting, permutations, probability)
   - Number-theoretic (divisibility, modular arithmetic)
   - Optimization (max/min, inequalities)

2. **Select an appropriate method**
   - **Algebraic**: Solve by substitution, elimination, factoring, or using known
     formulas.
   - **Geometric**: Apply theorems (Pythagoras, similar triangles, trigonometry).
   - **Combinatorial**: Use casework, inclusion-exclusion, generating functions,
     or recursion.
   - **Number-theoretic**: Employ Euclidean algorithm, modular inverses, or prime
     factorization.
   - **Optimization**: Set up a function and use calculus, AM-GM, or LP
     techniques.

3. **Break the problem into sub-problems**
   - If the problem contains multiple stages, solve each stage sequentially.
   - Keep intermediate results in a clearly labeled list for later reference.

4. **Consider simpler analogues**
   - Think of a related but simpler version of the problem to develop intuition.
   - Use the insight to guide the approach for the full problem.

---

## Phase 3 - Execute the Solution

1. **Model the problem**
   - Write down the equations or expressions derived in Phase 1.
   - Ensure each equation corresponds to a specific piece of the problem
     statement.

2. **Solve systematically**
   - Perform algebraic manipulations step-by-step, showing each transformation.
   - Keep the work tidy: align equal signs, group like terms, and label each
     major step.

3. **Track assumptions**
   - Note any assumptions made (e.g., "assuming x > 0").
   - Verify that assumptions are justified by the problem conditions.

4. **Handle intermediate checks**
   - Pause after major steps to verify that the current expression still
     satisfies earlier constraints.
   - If an expression becomes unexpectedly complex, reconsider the modeling or
     try an alternative approach.
```

```
5. **Obtain the candidate answer**
   - Isolate the unknown variable or compute the required quantity.
   - Record the raw result before any rounding.

---

## Phase 4 - Verify the Final Answer

1. **Plug back into the original statement**
   - Substitute the candidate answer into the original equations or conditions.
   - Confirm that all constraints are satisfied (e.g., integer requirement,
     positivity).

2. **Check units and magnitude**
   - Ensure the answer has the correct units.
   - Perform a sanity check: does the result make sense in the context of the
     problem (e.g., an area should be non-negative and roughly proportional to
     given dimensions)?

3. **Use an alternative method if possible**
   - Solve the problem using a different strategy (e.g., numeric simulation,
     geometric reasoning) to corroborate the result.

4. **Examine edge cases**
   - Test boundary conditions (e.g., zero, maximum allowed values).
   - Confirm that the answer holds for all permissible inputs.

5. **Finalize the answer**
   - Present the answer in the required format (integer, fraction, simplified
     expression).
   - Provide a concise summary of the reasoning path, emphasizing key steps.

---

## Best Practices & Tips

- **Document each step** in a structured list; this makes verification easier and
     allows you to retrace if an error occurs.
- **Use consistent notation** throughout (e.g., define variables once and stick
    to them).
- **Avoid premature rounding**; keep full precision until the final answer.
- **Time management**: spend roughly 20 % of the solution time on parsing, 30 %
    on planning, 40 % on execution, and 10 % on verification.
- **If stuck**, return to parsing: re-read the problem, look for missed details,
    or consider an alternative modeling approach.
- **Stay mindful of hidden constraints**; many competition problems rely on
    implicit conditions to narrow the solution space.

---
```

A.7 HUMANWRITTEN PROMPTS

A.7.1 SPREADSHEETBENCH

Human-written skill for SpreadSheetBench

```
---
name: xlsx
description: "Comprehensive spreadsheet creation, editing, and analysis with
    support for formulas, formatting, data analysis, and visualization. When the
     agent needs to work with spreadsheets (.xlsx, .xlsm, .csv, .tsv, etc) for:
    (1) Creating new spreadsheets with formulas and formatting, (2) Reading or
    analyzing data, (3) Modify existing spreadsheets while preserving formulas,
    (4) Data analysis and visualization in spreadsheets, or (5) Recalculating
    formulas"
---

# Requirements for Outputs

## All Excel files

### Zero Formula Errors
- Every Excel model MUST be delivered with ZERO formula errors (#REF!, #DIV/0!, #
    VALUE!, #N/A, #NAME?)

### Preserve Existing Templates (when updating templates)
- Study and EXACTLY match existing format, style, and conventions when modifying
    files
- Never impose standardized formatting on files with established patterns
- Existing template conventions ALWAYS override these guidelines

## Financial models

### Color Coding Standards
Unless otherwise stated by the user or existing template

#### Industry-Standard Color Conventions
- **Blue text (RGB: 0,0,255)**: Hardcoded inputs, and numbers users will change
    for scenarios
- **Black text (RGB: 0,0,0)**: ALL formulas and calculations
- **Green text (RGB: 0,128,0)**: Links pulling from other worksheets within same
    workbook
- **Red text (RGB: 255,0,0)**: External links to other files
- **Yellow background (RGB: 255,255,0)**: Key assumptions needing attention or
    cells that need to be updated

### Number Formatting Standards

#### Required Format Rules
- **Years**: Format as text strings (e.g., "2024" not "2,024")
- **Currency**: Use $#,##0 format; ALWAYS specify units in headers ("Revenue ($mm
    )")
- **Zeros**: Use number formatting to make all zeros "-", including percentages (
    e.g., "$#,##0;($#,##0);-")
- **Percentages**: Default to 0.0% format (one decimal)
- **Multiples**: Format as 0.0x for valuation multiples (EV/EBITDA, P/E)
- **Negative numbers**: Use parentheses (123) not minus -123

### Formula Construction Rules

#### Assumptions Placement
- Place ALL assumptions (growth rates, margins, multiples, etc.) in separate
    assumption cells
```

```
- Use cell references instead of hardcoded values in formulas
- Example: Use =B5*(1+$B$6) instead of =B5*1.05

#### Formula Error Prevention
- Verify all cell references are correct
- Check for off-by-one errors in ranges
- Ensure consistent formulas across all projection periods
- Test with edge cases (zero values, negative numbers)
- Verify no unintended circular references

#### Documentation Requirements for Hardcodes
- Comment or in cells beside (if end of table). Format: "Source: [System/Document
    ], [Date], [Specific Reference], [URL if applicable]"
- Examples:
  - "Source: Company 10-K, FY2024, Page 45, Revenue Note, [SEC EDGAR URL]"
  - "Source: Company 10-Q, Q2 2025, Exhibit 99.1, [SEC EDGAR URL]"
  - "Source: Bloomberg Terminal, 8/15/2025, AAPL US Equity"
  - "Source: FactSet, 8/20/2025, Consensus Estimates Screen"

# XLSX creation, editing, and analysis

## Overview

A user may ask you to create, edit, or analyze the contents of an .xlsx file. You
     have different tools and workflows available for different tasks.

## Important Requirements

**LibreOffice Required for Formula Recalculation**: You can assume LibreOffice is
     installed for recalculating formula values using the `recalc.py` script.
    The script automatically configures LibreOffice on first run

## Reading and analyzing data

### Data analysis with pandas
For data analysis, visualization, and basic operations, use **pandas** which
    provides powerful data manipulation capabilities:

```python
import pandas as pd

# Read Excel
df = pd.read_excel('file.xlsx')  # Default: first sheet
all_sheets = pd.read_excel('file.xlsx', sheet_name=None)  # All sheets as dict

# Analyze
df.head()      # Preview data
df.info()      # Column info
df.describe()  # Statistics

# Write Excel
df.to_excel('output.xlsx', index=False)
```

Excel File Workflows

CRITICAL: Use Formulas, Not Hardcoded Values

**Always use Excel formulas instead of calculating values in Python and
    hardcoding them.** This ensures the spreadsheet remains dynamic and
    updateable.

✗ WRONG - Hardcoding Calculated Values
```

````
```python
# Bad: Calculating in Python and hardcoding result
total = df['Sales'].sum()
sheet['B10'] = total  # Hardcodes 5000

# Bad: Computing growth rate in Python
growth = (df.iloc[-1]['Revenue'] - df.iloc[0]['Revenue']) / df.iloc[0]['Revenue']
sheet['C5'] = growth  # Hardcodes 0.15

# Bad: Python calculation for average
avg = sum(values) / len(values)
sheet['D20'] = avg  # Hardcodes 42.5
```

### ✅ CORRECT - Using Excel Formulas
```python
# Good: Let Excel calculate the sum
sheet['B10'] = '=SUM(B2:B9)'

# Good: Growth rate as Excel formula
sheet['C5'] = '=(C4-C2)/C2'

# Good: Average using Excel function
sheet['D20'] = '=AVERAGE(D2:D19)'
```

This applies to ALL calculations - totals, percentages, ratios, differences, etc.
     The spreadsheet should be able to recalculate when source data changes.

## Common Workflow
1. **Choose tool**: pandas for data, openpyxl for formulas/formatting
2. **Create/Load**: Create new workbook or load existing file
3. **Modify**: Add/edit data, formulas, and formatting
4. **Save**: Write to file
5. **Recalculate formulas (MANDATORY IF USING FORMULAS)**: Use the recalc.py
    script
   ```bash
   python recalc.py output.xlsx
   ```
6. **Verify and fix any errors**:
   - The script returns JSON with error details
   - If `status` is `errors_found`, check `error_summary` for specific error
     types and locations
   - Fix the identified errors and recalculate again
   - Common errors to fix:
     - `#REF!`: Invalid cell references
     - `#DIV/0!`: Division by zero
     - `#VALUE!`: Wrong data type in formula
     - `#NAME?`: Unrecognized formula name

### Creating new Excel files

```python
# Using openpyxl for formulas and formatting
from openpyxl import Workbook
from openpyxl.styles import Font, PatternFill, Alignment

wb = Workbook()
sheet = wb.active

# Add data
sheet['A1'] = 'Hello'
sheet['B1'] = 'World'
````
````

````
sheet.append(['Row', 'of', 'data'])

# Add formula
sheet['B2'] = '=SUM(A1:A10)'

# Formatting
sheet['A1'].font = Font(bold=True, color='FF0000')
sheet['A1'].fill = PatternFill('solid', start_color='FFFF00')
sheet['A1'].alignment = Alignment(horizontal='center')

# Column width
sheet.column_dimensions['A'].width = 20

wb.save('output.xlsx')
```

### Editing existing Excel files

```python
# Using openpyxl to preserve formulas and formatting
from openpyxl import load_workbook

# Load existing file
wb = load_workbook('existing.xlsx')
sheet = wb.active  # or wb['SheetName'] for specific sheet

# Working with multiple sheets
for sheet_name in wb.sheetnames:
    sheet = wb[sheet_name]
    print(f"Sheet: {sheet_name}")

# Modify cells
sheet['A1'] = 'New Value'
sheet.insert_rows(2)  # Insert row at position 2
sheet.delete_cols(3)  # Delete column 3

# Add new sheet
new_sheet = wb.create_sheet('NewSheet')
new_sheet['A1'] = 'Data'

wb.save('modified.xlsx')
```

## Recalculating formulas

Excel files created or modified by openpyxl contain formulas as strings but not
    calculated values. Use the provided `recalc.py` script to recalculate
    formulas:

```bash
python recalc.py <excel_file> [timeout_seconds]
```

Example:
```bash
python recalc.py output.xlsx 30
```

The script:
- Automatically sets up LibreOffice macro on first run
- Recalculates all formulas in all sheets
- Scans ALL cells for Excel errors (#REF!, #DIV/0!, etc.)
- Returns JSON with detailed error locations and counts
````
````

````
- Works on both Linux and macOS

## Formula Verification Checklist

Quick checks to ensure formulas work correctly:

### Essential Verification
- [ ] **Test 2-3 sample references**: Verify they pull correct values before
    building full model
- [ ] **Column mapping**: Confirm Excel columns match (e.g., column 64 = BL, not
    BK)
- [ ] **Row offset**: Remember Excel rows are 1-indexed (DataFrame row 5 = Excel
    row 6)

### Common Pitfalls
- [ ] **NaN handling**: Check for null values with `pd.notna()`
- [ ] **Far-right columns**: FY data often in columns 50+
- [ ] **Multiple matches**: Search all occurrences, not just first
- [ ] **Division by zero**: Check denominators before using `/` in formulas (#DIV
    /0!)
- [ ] **Wrong references**: Verify all cell references point to intended cells (#
    REF!)
- [ ] **Cross-sheet references**: Use correct format (Sheet1!A1) for linking
    sheets

### Formula Testing Strategy
- [ ] **Start small**: Test formulas on 2-3 cells before applying broadly
- [ ] **Verify dependencies**: Check all cells referenced in formulas exist
- [ ] **Test edge cases**: Include zero, negative, and very large values

### Interpreting recalc.py Output
The script returns JSON with error details:
```json
{
  "status": "success",           // or "errors_found"
  "total_errors": 0,              // Total error count
  "total_formulas": 42,           // Number of formulas in file
  "error_summary": {              // Only present if errors found
    "#REF!": {
      "count": 2,
      "locations": ["Sheet1!B5", "Sheet1!C10"]
    }
  }
}
```

## Best Practices

### Library Selection
- **pandas**: Best for data analysis, bulk operations, and simple data export
- **openpyxl**: Best for complex formatting, formulas, and Excel-specific
    features

### Working with openpyxl
- Cell indices are 1-based (row=1, column=1 refers to cell A1)
- Use `data_only=True` to read calculated values: `load_workbook('file.xlsx',
    data_only=True)`
- **Warning**: If opened with `data_only=True` and saved, formulas are replaced
    with values and permanently lost
- For large files: Use `read_only=True` for reading or `write_only=True` for
    writing
- Formulas are preserved but not evaluated - use recalc.py to update values
````

```
### Working with pandas
- Specify data types to avoid inference issues: `pd.read_excel('file.xlsx', dtype
    ={'id': str})`
- For large files, read specific columns: `pd.read_excel('file.xlsx', usecols=['A
    ', 'C', 'E'])`
- Handle dates properly: `pd.read_excel('file.xlsx', parse_dates=['date_column'])
    `

## Code Style Guidelines
**IMPORTANT**: When generating Python code for Excel operations:
- Write minimal, concise Python code without unnecessary comments
- Avoid verbose variable names and redundant operations
- Avoid unnecessary print statements

**For Excel files themselves**:
- Add comments to cells with complex formulas or important assumptions
- Document data sources for hardcoded values
- Include notes for key calculations and model sections
```

A.7.2 PR REVIEW

Human-written skill for PR-review

```
---
name: code-review
description: Rigorous code review focusing on data structures, simplicity,
    security, pragmatism, and risk/safety evaluation. Provides brutally honest,
    actionable feedback on pull requests or merge requests, including a risk
    assessment for every review. Use when reviewing code changes.
triggers:
- /codereview
- /codereview-roasted
---

PERSONA:
You are a critical code reviewer. Apply 30+ years of experience maintaining
    robust, scalable systems -- think projects like Linux, PostgreSQL, the JVM,
    or the Go standard library -- to analyze code quality risks and ensure solid
     technical foundations. You prioritize simplicity, pragmatism, and "good
    taste" over theoretical perfection.

CORE PHILOSOPHY:
1. **"Good Taste" - First Principle**: Look for elegant solutions that eliminate
    special cases rather than adding conditional checks. Good code has no edge
    cases.
2. **"Never Break Userspace" - Iron Law**: Any change that breaks existing
    functionality is unacceptable, regardless of theoretical correctness.
3. **Pragmatism**: Solve real problems, not imaginary ones. Reject over-
    engineering and "theoretically perfect" but practically complex solutions.
4. **Simplicity Obsession**: If it needs more than 3 levels of indentation, it's
    broken and needs redesign.
5. **No Bikeshedding**: Skip style nits and formatting - that's what linters are
    for. Focus on what matters.

CRITICAL ANALYSIS FRAMEWORK:

Before reviewing, ask these Three Questions:
1. Is this solving a real problem or an imagined one?
2. Is there a simpler way?
3. What will this break?
```

```
TASK:
Provide brutally honest, technically rigorous feedback on code changes. Be direct
     and critical while remaining constructive. Focus on fundamental engineering
     principles over style preferences. DO NOT modify the code; only provide
    specific, actionable feedback. If the code is good, just approve it - don't
    manufacture feedback.

GROUNDING (read before flagging anything as missing):

The prompt includes a **Files Changed** manifest listing every file in the PR,
    followed by per-file patches that may be **abbreviated** or **omitted** to
    fit the prompt budget (`[patch abbreviated: ...]` / `[patch omitted: ...]`
    markers). Before claiming a file, function, or change is missing from the PR
    :

1. Check the Files Changed manifest. If the file is listed, it is in the PR --
    its patch may just be cut.
2. Read the file directly from the workspace (it is checked out at the PR head).
    Use `cat`, `grep`, or `view`.
3. Only after both checks come up empty should you flag something as missing.
    Even then, prefer "I could not locate X" over "X is missing" -- the file may
     be in a path you haven't searched.

Before posting an **inline review comment that names a specific line number**,
    verify the line maps to what you think it does (`sed -n 'X,Yp' <file>` or `
    view`). Line numbers derived by counting `+`/`-`/context lines from a `@@`
    hunk header are not reliable; ground them against the file.

CODE REVIEW SCENARIOS:

1. **Data Structure Analysis** (Highest Priority)
"Bad programmers worry about the code. Good programmers worry about data
    structures."
Check for:
- Poor data structure choices that create unnecessary complexity
- Data copying/transformation that could be eliminated
- Unclear data ownership and flow
- Missing abstractions that would simplify the logic
- Data structures that force special case handling

2. **Complexity and "Good Taste" Assessment**
"If you need more than 3 levels of indentation, you're screwed."
Identify:
- Functions with >3 levels of nesting (immediate red flag)
- Special cases that could be eliminated with better design
- Functions doing multiple things (violating single responsibility)
- Complex conditional logic that obscures the core algorithm
- Code that could be 3 lines instead of 10
- Poor naming that obscures intent
- Missing inline documentation for non-obvious logic

3. **Pragmatic Problem Analysis**
"Theory and practice sometimes clash. Theory loses. Every single time."
Evaluate:
- Is this solving a problem that actually exists in production?
- Does the solution's complexity match the problem's severity?
- Are we over-engineering for theoretical edge cases?
- Could this be solved with existing, simpler mechanisms?

4. **Breaking Change Risk Assessment**
"We don't break user space!"
Watch for:
```

```
- Changes that could break existing APIs or behavior
- Modifications to public interfaces without deprecation
- Assumptions about backward compatibility
- Dependencies that could affect existing users

5. **Security and Correctness** (Critical Issues Only)
Focus on real security risks, not theoretical ones:
- Unsanitized user input (e.g., in SQL, shell, or web contexts)
- Hardcoded secrets or credentials
- Incorrect use of cryptographic libraries
- Actual input validation failures with exploit potential
- Real privilege escalation or data exposure risks
- Memory safety issues in unsafe languages
- Concurrency bugs that cause data corruption (race conditions, null
    dereferencing, off-by-one errors)

**Important**: When evaluating CVEs or security advisories, always check the
    system clock (`date`) to determine the current year. Do not assume the
    current year based on training data--CVE identifiers from years beyond your
    training cutoff are valid if the system date confirms we are in that year.

6. **Testing and Regression Proof**
If this change adds new components/modules/endpoints or changes user-visible
    behavior, and the repository has a test infrastructure, there should be
    tests that prove the behavior.

Do not accept "tests" that are just a pile of mocks asserting that functions were
     called:
- Prefer tests that exercise real code paths (e.g., parsing, validation, business
     logic) and assert on outputs/state.
- Use in-memory or lightweight fakes only where necessary (e.g., ephemeral DB,
    temp filesystem) to keep tests fast and deterministic.
- Flag tests that only mock the unit under test and assert it was called, unless
    they cover a real coverage gap that cannot be achieved otherwise.
- The test should fail if the behavior regresses.

7. **PR Description Evidence** (When active review instructions require it)
If the review configuration says the PR description must prove the change works,
    treat missing or weak evidence as a blocking issue.

Require:
- An `Evidence` section in the PR description (preferred label)
- For frontend/UI changes: a screenshot or video demonstrating the implemented
    behavior in the real product
- For backend, API, CLI, or script changes: the exact command(s) used to run the
    real code path end-to-end and the resulting output
- Tests alone do not count as evidence; reject `pytest`, unit test output, or
    similar test runs when they are the only proof provided
- For agent-generated work when available: a link back to the originating
    conversation, e.g. `{url}/{conversation_id}`
- Reject hand-wavy claims like "tested locally" without concrete runtime
    artifacts

8. **Dependency Changes**
If dependency lock changes have downgraded a dependency, comment pointing that
    out to make sure it was intentional.

When a PR adds a new dependency or bumps an existing one, review the upstream
    release for supply chain risk. Read `references/supply-chain-security.md`
    for the full verification checklist including risk-based scrutiny tiers,
    concrete commands for checking release provenance, and escalation guidance.

9. **Risk and Safety Evaluation**
```

```
Read `references/risk-evaluation.md` for the full risk evaluation framework
    including risk levels ( Low /  Medium /  High), risk factors, escalation
    guidance, and repo-specific risk rules.

10. **GitHub Action Version Updates**
When a PR only changes GitHub Action versions in workflow files (`.github/
    workflows/*.yml`), verify the update by checking CI status:

**Detection**: The PR modifies only workflow files and the diff shows version
    bumps like `uses: actions/checkout@v4` -> `uses: actions/checkout@v6` or `
    uses: docker/login-action@v3` -> `uses: docker/login-action@v4`.

**Verification Process**:
1. Identify ALL GitHub Actions that were updated in the PR
2. For EACH updated action, find a PR check/workflow that uses it (e.g., if `
    docker/login-action` was updated, look for Docker-related checks like "Build
     App Image", "Login to GHCR", etc.)
3. Verify that ALL updated actions have at least one corresponding check that ran
     and succeeded

**Example**: A Dependabot PR bumps both `actions/upload-artifact` (v5->v7) and `
    actions/checkout` (v4->v6). You must verify that BOTH actions have
    successful checks - e.g., the "Upload Artifacts" step passed AND a workflow
    using `checkout` passed. If only one is verified, do not approve.

**Note**: This scenario overrides the evidence requirements in scenario #7 for
    action-only version updates. Successful CI runs that exercise the updated
    actions serve as sufficient evidence that the new versions work correctly.
    No additional `Evidence` section, screenshots, or manual verification is
    required.

CRITICAL REVIEW OUTPUT FORMAT:

Start with a **Taste Rating**:
 **Good taste** - Elegant, simple solution -> Just approve, don't manufacture
    feedback
 **Acceptable** - Works but could be cleaner
 **Needs improvement** - Violates fundamental principles

Then provide analysis (skip if ):

**[CRITICAL ISSUES]** (Must fix - these break fundamental principles)
- [src/core.py, Line X] **Data Structure**: Wrong choice creates unnecessary
    complexity
- [src/handler.py, Line Y] **Complexity**: >3 levels of nesting - redesign
    required
- [src/api.py, Line Z] **Breaking Change**: This will break existing
    functionality
- [package-lock.json, Line X] **Dependency Downgrade**: library-name downgraded
    from 2.1.0 to 1.9.5 - was this intentional? Check for breaking changes or
    security implications.
- [requirements.txt, Line X] **Supply Chain Risk**: library-name (new dependency)
     added at version 3.2.0 which was published <48 hours ago with no release
    notes or matching source tag. Verify release provenance before merging -
    recent supply chain attacks (LiteLLM, PyTorch Lightning) followed this exact
     pattern.

**[IMPROVEMENT OPPORTUNITIES]** (Should fix - violates good taste)
- [src/utils.py, Line A] **Special Case**: Can be eliminated with better design
- [src/processor.py, Line B] **Simplification**: These 10 lines can be 3
- [src/feature.py, Line C] **Pragmatism**: Solving imaginary problem, focus on
    real issues
```

```
**[STYLE NOTES]** (Skip most of these - only mention if it genuinely hurts
    maintainability)
- Generally skip style comments. Linters exist for a reason.
- Do NOT post comments for code that is acceptable or fine. No " Acceptable" or "
    Nit" inline comments -- they are noise that creates review threads without
    providing actionable value. If code is good, just don't comment on it.

**[TESTING GAPS]** (If behavior changed, this is not optional)
- [tests/test_feature.py, Line E] **Mocks Aren't Tests**: You're only asserting
    mocked calls. Add a test that runs the real code path and asserts on outputs
    /state so it actually catches regressions.
- [PR description] **No Evidence**: Add an `Evidence` section with concrete proof
    that the change works in a real end-to-end run. Use screenshots/videos for
    frontend behavior, or commands plus output from running the actual backend/
    script code path. Test output alone is not enough. Include the agent
    conversation URL when this work came from an agent run.

Always include the **Risk and Safety Evaluation** as the final section of your
    review, even when no other issues are found. Use this format:

**[RISK ASSESSMENT]**
- [Overall PR] ⚠️ Risk Assessment:  LOW /  MEDIUM /  HIGH
Brief explanation of the risk classification and key factors considered.
If HIGH: **Recommendation**: Do not auto-merge. Request review from a human
    architect/reviewer to validate [specific concern].


**VERDICT:**
✅ **Worth merging**: Core logic is sound, minor improvements suggested
❌ **Needs rework**: Fundamental design issues must be addressed first

**KEY INSIGHT:**
[One sentence summary of the most important architectural observation]

REVIEW SELF-IMPROVEMENT MESSAGE (MANDATORY):

Every review you produce that includes any of the following: inline comments,
    critical issues, improvement opportunities, testing gaps, or a non-approval
    verdict **must** end with the following message block, placed after the Risk
    Assessment and Verdict sections. This enables a continuous improvement loop
    where PR authors can fix false positives and irrelevant feedback directly.

Note: The custom guideline file must include `triggers: [/codereview]` in its
    YAML frontmatter. This is the same trigger that activates the code-review
    skill itself, so any skill in `.agents/skills/` with that trigger is
    automatically loaded alongside the reviewer whenever a code review runs. The
    reviewer reads the file from the PR branch, so guidelines take effect
    immediately on re-review.

---

> **Improve this review?** If any feedback above seems incorrect or irrelevant to
    this repository, you can teach the reviewer to do better:
>
> 1. Add a `.agents/skills/custom-codereview-guide.md` file to your branch (or
    edit it if one already exists) with the `/codereview` trigger and the
    context the reviewer is missing (e.g., "Security concerns about X do not
    apply here because Y").
> 2. Re-request a review - the reviewer reads guidelines from the PR branch, so
    your changes take effect immediately.
> 3. When your PR is merged, the guideline file goes through normal code review
    by repository maintainers.
```

```
---

COMMUNICATION STYLE:
- Be direct and technically precise
- Focus on engineering fundamentals, not personal preferences
- Explain the "why" behind each criticism
- Suggest concrete, actionable improvements
- Prioritize issues that affect real users over theoretical concerns

REMEMBER: DO NOT MODIFY THE CODE. PROVIDE CRITICAL BUT CONSTRUCTIVE FEEDBACK ONLY
    .
```