SKILL TRANSFER: ADAPTING AGENT SKILLS FROM STRONG TO WEAK AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent skills are modular bundles of instructions, code, and resources that an agent loads on demand, increasingly authored and validated on frontier models and shared as an open standard. The same skill behaves differently across models: a strong model turns it into a large performance gain, while a weak model realizes little of that gain and sometimes does worse than with no skill at all. Cost and resource constraints, however, often restrict real deployments to exactly these weaker agents. We hypothesize that the strong model silently supplies procedural steps the skill leaves implicit, which the weak model omits, and formalize closing this gap as *Skill Transfer*: holding the skill's capability ceiling and the model's parameters fixed, we adapt the skill so a specific weak agent can execute it well. Our method abstracts strong and weak trajectories into typed execution structures, localizes their structural divergence with a graph edit distance (GED), and repairs the skill through an iterative diagnoser-patcher loop. Across three benchmarks and two model families, Skill Transfer raises pass rate over the unmodified skill significantly and well over other skill-evolution baselines. The adapted skill also improves the strong agent it was never adapted for, and transfers out of distribution to an unseen task.

1 INTRODUCTION

Large Language Model agents solve complex tasks by interleaving reasoning (Wei et al., 2022; Wang et al., 2022; Yao et al., 2023; Lan et al., 2026) with actions over an external environment (Yao et al., 2022; Schick et al., 2023; Yang et al., 2024), demonstrating excellent performance. However, they often lack knowledge of some specific domains where fine-tuning is expensive and harmful to general capability. To solve this, agent skills have been proposed, serving as reusable file systems where procedural domain knowledge is structurally packaged (Anthropic, 2025a). Specifically, an agent skill is a self-contained bundle of natural-language instructions, runnable scripts, and reference material that the agent loads only when a task matches. Skills are compositional, interpretable, and portable. An agent can accumulate a library of them and reuse it on unseen tasks (Wang et al., 2023; Xia et al., 2026; Ni et al., 2026), and recent open standards let a skill authored once be shared across different agents and providers (Anthropic, 2025b; OpenAI, 2025b). By design, skills are expected to work effectively across various agents, agnostic to specific harnesses and models.

However, Li et al. (2026) report that the same curated skills yield various levels of gains across models and harnesses, demonstrating that some agents improve substantially while others gain little. In addition, Yu et al. (2026) show that skills of different granularity suit different agents, and a skill that benefits one agent can even harm another. Furthermore, we find a sharper form of this mismatch. Given one fixed skill, a strong and a weak agent not only differ in task success rate but also **follow markedly different procedures**, a divergence we quantify with graph edit distance (GED) over their execution trajectories (Sanfeliu & Fu, 1983; Gao et al., 2010). Figure 1 illustrates this mismatch. The skill itself is effective for the strong agent. The weak agent leaves much of its value unrealized because it omits procedural steps that the strong agent supplies implicitly. This raises a natural question:

Can the full value of the skill be recovered for the weak agent without retraining it?

We formalize the problem of closing this gap as *Skill Transfer*. Specifically, given a skill that is already effective for a strong agent, we modify it so that a specific weak agent can use it effectively. We

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.

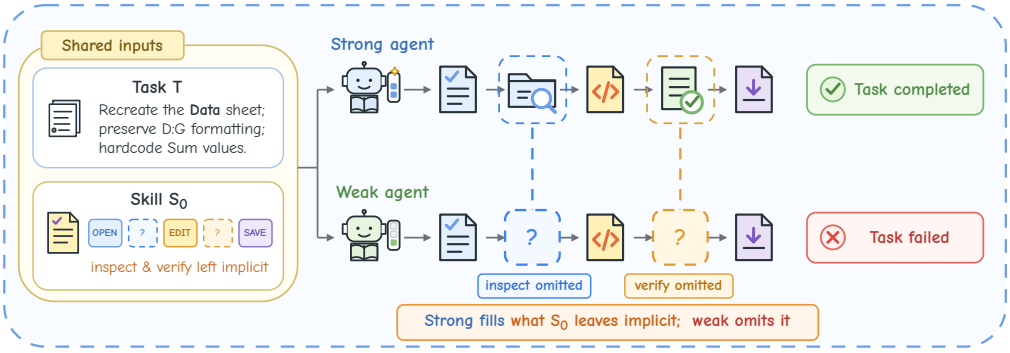


Figure 1: Motivation for Skill Transfer. Given the same task and skill $\mathcal{S}_0$, the strong agent supplies inspection and verification steps that the skill leaves implicit and completes the task. The weak agent omits those same steps and fails. This gap motivates Skill Transfer.

realize Skill Transfer by abstracting the strong and weak agents' trajectories into a shared structural representation, localizing their procedural gaps, and iteratively patching the skill to close them.

This problem matters for deployment. Real-world systems increasingly rely on smaller, cheaper agents to meet cost, latency, and resource constraints (Zheng et al., 2025b; Xu et al., 2024), whereas the strongest models remain expensive and resource-intensive. Skill Transfer lets such deployments reuse the large and growing corpus of skills authored for frontier models, rather than discarding them or retraining the model.

Our contributions are as follows:

- We formalize *Skill Transfer*, the problem of adapting a skill that already works well for a strong agent so that a weak agent can use it effectively.
- We propose a trajectory-comparison method that abstracts strong and weak trajectories into typed execution structures, localizes their structural gaps, and repairs the skill with a diagnoser–patcher agent pair.
- We show across multiple benchmarks and agents that Skill Transfer substantially closes the strong–weak gap without modifying the weak agent.

2 Related Work

Memory Learning. A line of work improves an agent without weight updates by accumulating experience in an external memory: storing and retrieving observations, reflections, or long-term records (Park et al., 2023; Packer et al., 2024; Zhong et al., 2024; Xu et al., 2026; Chhikara et al., 2025), and distilling past trajectories into reusable, transferable insights (Zhao et al., 2024; Shinn et al., 2023; Chang et al., 2026; Wang et al., 2026a). Such memory is auxiliary experience gathered at test time; in contrast, a skill is an authored, self-contained procedure that an agent loads on demand, and our work transfers this skill to a target agent rather than growing an experience store.

Skill Evolution. Several works grow or optimize a skill library so that an agent solves more tasks: accumulating executable skills through self-practice (Wang et al., 2023; Zheng et al., 2025a), discovering and refining them from execution feedback (Alzubi et al., 2026; Zhang et al., 2026; Ni et al., 2026; Wang et al., 2026b), or evolving them via reinforcement learning (Xia et al., 2026). These methods raise a skill's absolute capability to solve more tasks, and do not distinguish the agent a skill was written for from the agent that has to run it. We instead adapt an existing skill effective on the strong agent to a weak agent, taking the gap between strong and weak trajectories as the signal. MASA (Yu et al., 2026) also claims that a skill should be adapted to different backbones, but conditions skill evolution on a pre-specified model card, a coarse profile that is largely unavailable for closed-source models.

Agent Distillation. Distillation is a complementary route to transfer capability into a smaller model's parameters—from classical output (Hinton et al., 2015; Hsieh et al., 2023) to fine-tuning smaller LLMs on teacher-generated agent trajectories for tool use and reasoning (Zeng et al., 2024;

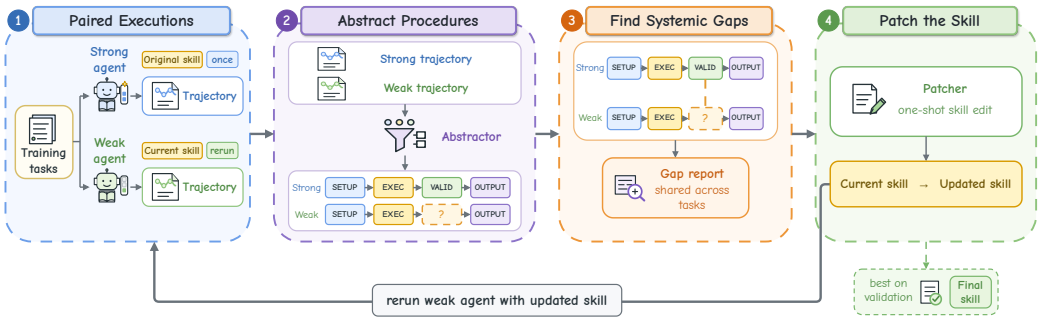


Figure 2: Skill Transfer overview. (1) The strong agent runs once under the original skill $\mathcal{S}_0$, while the weak agent reruns each iteration under the current skill $\mathcal{S}_k$ on the same training tasks. (2) The Abstractor turns both trajectories into typed node sequences (§3.2). (3) Aligning the paired structures localizes the gaps on each task and synthesizes the systemic ones into a gap report (§3.3–§3.4). (4) The patcher edits only the skill, in one pass, to obtain $\mathcal{S}_{k+1}$, which is used to rerun the weak agent. After the iterations, $\mathcal{S}^\star$ is selected on $\mathcal{V}$.

Chen et al., 2023; Kang et al., 2025). All of these modify the model's weights and require training, whereas Skill Transfer is training-free, freezing the weak model and editing only the shared, human-readable skill. In this sense, Skill Transfer can be viewed as a form of training-free distillation: it distills the strong agent's procedural competence into the transferred skill rather than into model weights.

3 Method

Skill Transfer proceeds in four steps. It first abstracts weak and strong trajectories into typed execution structures (§3.2), then localizes where the weak procedure structurally diverges from the strong one on each task (§3.3), then synthesizes the systemic gaps across the whole training batch (§3.4), and finally repairs the skill by turning each diagnosed gap into a targeted edit (§3.5). §3.6 states which agent carries out each step and how the steps iterate until the skill converges. Figure 2 gives an overview.

3.1 Problem Formulation

Let π_s and π_w denote a strong (teacher) and a weak (student) agent that share the same agentic configurations and, at inference time, the same skill; they differ only in their backbone model. A task is a pair (t, y) of a task specification and its ground-truth answer, drawn from a distribution $\mathcal{D}$ over the target domain. Running an agent π equipped with skill $\mathcal{S}$ on t produces a trajectory τ, a sequence of thoughts, tool calls, and observations, together with a final output o. A task metric scores the output against the ground truth, $R(o, y) \in [0, 1]$, with $R{=}1$ for a fully correct output. The two play different roles below: $R(o, y)$ alone decides whether a task is solved, while τ is what we abstract and compare (§3.2–§3.3). Let $\mathcal{S}_0$ be an original skill that is already effective for the strong agent. We define the objective of Skill Transfer as the weak agent's expected success over the domain,

$$J(\mathcal{S}) = \mathbb{E}_{(t,y)\sim\mathcal{D},\ (\tau,o)\sim\pi_w(t;\mathcal{S})}\big[\, R(o, y)\,\big], \tag{1}$$

and adapt $\mathcal{S}_0$ to maximize $J(\mathcal{S})$ while leaving the weak model's parameters and the skill's intended capability unchanged. Because a skill is free-form text and $\mathcal{D}$ is unknown, we cannot maximize J directly. Instead, from a small set of training tasks $\mathcal{T} = \{t_1, \ldots, t_N\}$ that the strong agent solves, an iterative diagnoser-patcher loop proposes a sequence of candidate skills $\mathcal{S}_0, \ldots, \mathcal{S}_K$, from which we select the adapted skill $\mathcal{S}^\star$ on a held-out validation set.

3.2 Execution Structure Abstraction

Raw trajectories are too noisy to compare directly, especially across different agents, because they use different tool syntax, follow different reasoning styles, and present the same step in different surface forms. We instead compare a typed intermediate representation, the *execution structure*, obtained by abstracting each trajectory with an LLM (i.e., the *Abstractor*).

Given a task t and a trajectory τ, the Abstractor emits an ordered sequence of *procedure nodes* $X(v_1, \ldots, v_m)$, one per distinct procedural step (Figure 2b). Each node v_i is a triple

$$v_i = \big(c_i^1,\, c_i^2,\, d_i\big). \tag{2}$$

Here, c_i^1 denotes a *tier-1* label from a small, fixed, domain-agnostic set of six procedural roles: SETUP (preparation), EXECUTE (the primary domain action), VALIDATE (a correctness check before committing), RECOVER (responding to failure), OUTPUT (persisting a result), and OTHER (anything fitting none of the above). c_i^2 denotes a *tier-2* label, a short free-text tag the Abstractor invents to name the specific step (e.g. `borderless_table_fallback`, `formula_write_or_repair`). Tier-1 gives a fixed vocabulary for cross-domain alignment, and tier-2 carries the domain detail a patch needs. d_i holds the step's supporting detail: its completion criterion, the tool calls, code snippets, and observations that realize it, spliced in verbatim from the trajectory, and, when the step activates a skill, the agent's stated reason for that choice. Only the two labels enter the GED calculation of §3.3, but d_i is what the gap analysis there reads and cites. Appendix B shows a complete execution structure.

We abstract a strong and a weak trajectory jointly in one pass so equivalent steps get the same tier-2 label; abstracting them separately lets the same technique drift to different labels and breaks the gap analysis below.

3.3 Structural Gap Localization

Given strong and weak structures $X_s = (v_1, \ldots, v_m)$ and $X_w = (v'_1, \ldots, v'_n)$ for the same task, we first quantify their divergence through graph edit distance (ged), and then localize it to specific steps. Figure 3 shows an example.

Graph edit distance. We treat each execution structure as a path graph ordered by execution time and measure dissimilarity with graph edit distance (Sanfeliu & Fu, 1983; Gao et al., 2010), computed by the standard sequence-alignment dynamic program with a substitution cost $\delta(v, v')$ graded on the two-level category rather than binary match/mismatch:

$$\delta(v, v') = \begin{cases} 0.00 & c^1{=}c'^1 \wedge c^2{=}c'^2 \\ 0.50 & c^1{=}c'^1 \wedge c^2{\neq}c'^2 \\ 0.75 & c^1{\neq}c'^1 \wedge c^2{=}c'^2 \\ 1.00 & c^1{\neq}c'^1 \wedge c^2{\neq}c'^2 \end{cases} \tag{3}$$

and an insertion/deletion cost of 1.0. An alignment $\mathcal{A}$ pairs each node of X_s with at most one node of X_w, and preserves order: if one step of X_s precedes another, their partners in X_w appear in the same order. GED is the cost of the cheapest alignment,

$$\mathrm{GED}(X_s, X_w) = \min_{\mathcal{A}} \Big[\sum_{(i,j)\in\mathcal{A}} \delta(v_i, v'_j) \;+\; \big(m - |\mathcal{A}|\big) \;+\; \big(n - |\mathcal{A}|\big) \Big], \tag{4}$$

where the last two terms charge one unit for every node left unmatched, deleted from X_s or inserted into X_w; the minimum is found by the standard $O(mn)$ dynamic program. Matching nothing costs $|X_s| + |X_w| = m + n$, the worst case and hence the normalizer:

$$\widehat{\mathrm{GED}}(X_s, X_w) = \frac{\mathrm{GED}(X_s, X_w)}{m + n} \in [0, 1]. \tag{5}$$

GED is a single scalar summarizing how much the two procedures disagree overall; it is a diagnostic signal, not a success criterion, so a task is scored correct from its output alone.

Gap type. A single scalar cannot say where the two procedures disagree, so we also localize divergence to individual steps. We write $C(X)$ for the multiset of tier-1 roles occurring in X, so that a role performed twice counts twice, $K(X)$ for the skills activated in X (read off the activation nodes' d), and $\mathcal{M}$ for the label matching that pairs v_i with v'_j when $(c_i^1, c_i^2) = (c_j'^1, c_j'^2)$, falling back to tier-1 alone when no tier-2 counterpart exists. Unlike the alignment $\mathcal{A}$ of Eq. 4, $\mathcal{M}$ need not preserve order, which is what makes step reordering visible. Comparing matched and unmatched nodes yields five gap types (Figure 2c):

- **missing_node**: $C(X_s) \setminus C(X_w) \neq \emptyset$, as a multiset difference. The weak run performs a tier-1 role fewer times than the strong run, either never at all (e.g. no VALIDATE step) or fewer times than needed when the role recurs.

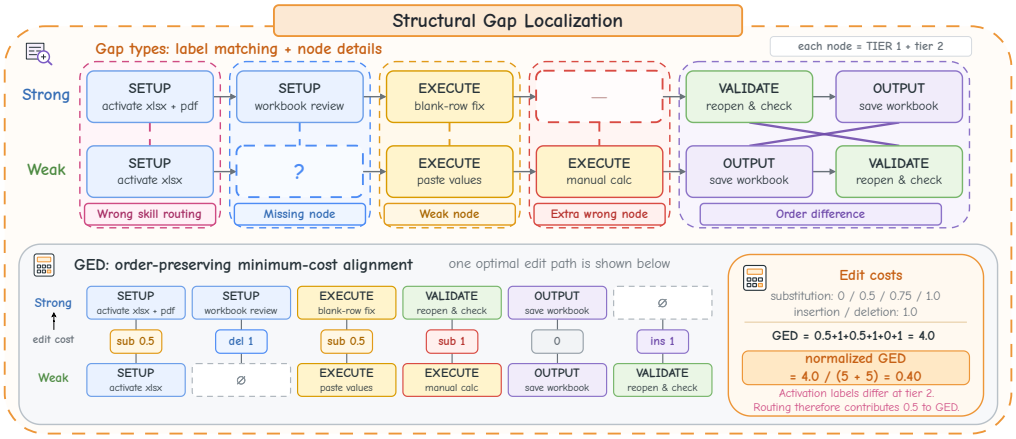


Figure 3: Structural gap localization on one task. *Top*: matching the strong and weak node sequences by their tier-1/tier-2 labels surfaces the five gap types; `wrong_skill_routing` is read from the activated skills recorded in d. *Bottom*: one minimum-cost alignment of the same pair, charging δ (Eq. 3) per matched pair and one per unmatched node, which gives $\mathrm{GED} = 4.0$ and $\widehat{\mathrm{GED}} = 4.0/(5{+}5) = 0.40$.

- **weak_node**: some $(i, j) \in \mathcal{M}$ has $d'_j \prec d_i$. The role is present but executed to a lower bar: a looser completion criterion, fewer artifacts, a worse technique, or a correct tool with an incorrect computed result.
- **extra_wrong_node**: some j is unmatched by $\mathcal{M}$. The weak run takes a step with no counterpart in X_s that fails.
- **order_difference**: some $(i, j), (i', j') \in \mathcal{M}$ have $i < i'$ but $j > j'$. Both runs contain the two steps, in opposite relative order.
- **wrong_skill_routing**: $K(X_s) \setminus K(X_w) \neq \emptyset$. When a task may draw on more than one skill, the weak agent's activated set is missing one the strong agent used.

The ordering $\prec$ on step is judged from the cited d, and every reported gap must quote the specific code snippet or observation behind it, falling back to the raw trajectory only when the structures are insufficient.

3.4 Multi-Case Gap Synthesis

§3.3 types the gaps on a single task. However, a skill edit that chases one task will not generalize, so this step applies that typing to every task in the training batch $\mathcal{T}$ and synthesizes the results. Besides the paired structures $\{X_s^{(t)}\}, \{X_w^{(t)}\}$ and the current skill, it reads a per-task validator report, a finer-grained comparison of π_w's output against π_s's than R (Appendix D), as well as earlier iterations' gap reports and patch history.

Per-task gaps describing the same weakness are grouped across tasks into a set of systemic gaps $G = \{g_1, g_2, \dots\}$. Each $g \in G$ carries a gap type, metadata (label, description), and two case sets: $F_g \subseteq \mathcal{T}$, the tasks where π_w failed with the gap present, and $P_g \subseteq \mathcal{T}$, the tasks where π_w passed despite it. A gap is admitted for repair if $|F_g| \geq 1$ or $|P_g| \geq 2$. The second condition catches a gap that has not yet caused a failure but recurs in runs that passed anyway, which we treat as a latent robustness risk rather than a non-issue.

Strategy selection and patch hint. Each gap admitted for repair is assigned some of the six mutually-exclusive strategies. MANDATE adds a new required workflow step with an explicit completion criterion; GUIDANCE sharpens an existing but under-specified step; PROHIBITION forbids a specific action the weak agent takes that the strong agent never does; HARD-STOP forces the agent to detect and discard a wrong result rather than report success on it; REORDER fixes step sequencing; and DESCRIPTION_PATCH extends a skill's description (rather than its main content) using both agents' stated reasons for choosing a skill, to fix wrong-skill routing. Table 1 maps each gap type to the strategies that repair it; when the gap has been seen in an earlier iteration, its status further constrains the choice (next paragraph and Table 2). The chosen strategy is written into a patch hint,

Table 1: Gap type → patch strategy mapping written into the patch hint.

Gap type	Strategy
missing_node	MANDATE (+ PROHIBITION if the tempting wrong path is non-obvious)
weak_node	GUIDANCE / PROHIBITION / HARD-STOP / MANDATE, chosen by root cause
extra_wrong_node	PROHIBITION
order_difference	REORDER
wrong_skill_routing	DESCRIPTION_PATCH

Table 2: Cross-iteration gap status and how it overrides the strategy of Table 1.

Status	Trigger	Effect on the patch hint
NEW	no iteration-$(k-1)$ match	take the strategy from Table 1; no prior-patch critique
RESOLVED	$\lvert F_g\rvert=0$ and $\lvert P_g\rvert < 2$	drop the gap; emit no patch
IMPROVEMENT	$\Delta < 0$	keep the previous strategy and supplement it for the cases still failing
PERSISTS	$\Delta = 0$	GUIDANCE is forbidden; escalate to a different mechanism and *replace* the standing instruction rather than append to it
REGRESSION	$\Delta > 0$	revert or rescope the harmful edit first, then re-derive the strategy from the root cause

free-form prose covering four elements: *target*, the file and section to edit; *strategy*, the chosen label; *wording*, the instruction text to install, which must be phrased generally enough to cover cases beyond the ones cited; and, when a previous iteration already patched this gap, a critique of why that patch failed to close it. This hint is the primary handoff to the repair step (§3.5), which executes it without redoing the diagnosis. Appendix C shows one gap entry with its hint.

Status. At iteration $k{=}0$ every gap is trivially NEW, since no prior report exists to match against. From $k{\geq}1$ on, each current gap is instead matched to its counterpart in iteration $k{-}1$'s report by label, and classified by the change in its failing-case count $\Delta = |F_g^{(k)}| - |F_g^{(k-1)}|$. Table 2 shows the resulting categories and how each overrides the default strategy choice. Eligible gaps are then ranked for repair by status tier (REGRESSION > PERSISTS > NEW > IMPROVEMENT), then by failing-case count descending within a tier.

3.5 One-Shot Skill Patching

Given the ranked gap report of §3.4, this step applies the edits it describes in a single consolidated pass:

- **Read and group**: read every admitted gap's patch hint and group hints by the target file section they name.
- **Synthesize**: for each affected section, design one coherent edit satisfying every hint touching it.
- **Apply**: replace the old text that needs editing in all sections in one multi-section edit.
- **Verify**: re-read the patched skill, confirm every gap in the repair queue has been fixed and no section contradicts another.

3.6 Iterative Loop

Two LLM agents carry out the steps above. A *diagnoser* performs localization and synthesis (§3.3–§3.4), reading the paired structures and emitting the ranked gap report; a *patcher* performs the repair (§3.5), and is the only one of the two that may edit the skill. Skill Transfer alternates the two over the fixed batch $\mathcal{T}$. At iteration 0, $\mathcal{T}$ is restricted once to tasks π_s solves within up to M independent attempts (excluding tasks the strong agent still fails after M tries) and then held fixed for every later iteration. In each iteration, π_w reruns under $\mathcal{S}_k$ on $\mathcal{T}$ first, the diagnoser then produces the gap report, and the patcher turns $\mathcal{S}_k$ into $\mathcal{S}_{k+1}$. The loop stops at a fixed iteration budget $K{=}$`max_edits`, or earlier if the skill is left unchanged for two consecutive iterations. We take the adapted skill $\mathcal{S}^\star$ to be the iterate with the highest pass rate on a held-out validation set $\mathcal{V}$,

$$\mathcal{S}^\star = \arg \max_{k \in \{0,\ldots,K\}} \mathrm{acc}_{\mathcal{V}}(\mathcal{S}_k). \tag{6}$$

Algorithm 1 summarizes the full loop. Appendix A shows an ablation replacing the whole diagnosis-and-repair framework with the same strong model reading the raw paired trajectories directly, holding

Algorithm 1 Skill Transfer

Require: original skill $\mathcal{S}_0$, already effective for the strong agent π_s; weak agent π_w; candidate training tasks $\mathcal{T}_0$; validation set $\mathcal{V}$; iteration budget K; strong-agent attempts M
Ensure: a skill adapted to π_w

Strong-agent reference, run once
Run π_s on every task in $\mathcal{T}_0$ under $\mathcal{S}_0$, retrying a task up to M times
$\mathcal{T} \leftarrow$ the tasks π_s solved ▷ the rest offer no reference procedure
Adaptation loop, with the weak agent rerun every iteration
$\mathcal{S} \leftarrow \mathcal{S}_0$; $\mathcal{S}^\star \leftarrow \mathcal{S}_0$ ▷ current and best-so-far skill
for $k = 1$ **to** K **do**
  Run π_w on $\mathcal{T}$ under $\mathcal{S}$
  Abstract both strong and weak trajectories jointly ▷ §3.2
  if π_w solved every task in $\mathcal{T}$ **then**
    break ▷ no gap left to close
  Localize the gaps in each strong–weak structure pair ▷ §3.3
  Group the per-task gaps into systemic ones; rank them and write one patch hint each ▷ §3.4
  $\mathcal{S} \leftarrow$ apply all patch hints to $\mathcal{S}$ in a single pass ▷ §3.5
  if $\mathcal{S}$ scores higher than $\mathcal{S}^\star$ on $\mathcal{V}$ **then**
    $\mathcal{S}^\star \leftarrow \mathcal{S}$
  if $\mathcal{S}$ has been left unchanged in two consecutive iterations **then**
    break ▷ converged
return $\mathcal{S}^\star$

the models, the tasks, and the iteration budget fixed. The results demonstrate the framework we design can better improve the weak agent.

4 Experiments

4.1 Setup

Data. We evaluate on three benchmarks, each with 20 and 30 further cases for training and validation. SpreadsheetBench (Ma et al., 2024) collects real-world spreadsheet-manipulation requests from online forums; each task gives the agent an input workbook and an instruction (e.g. insert a formula, reformat a range, filter rows); we use a 200-case test split. OfficeQA (Opsahl-Ong et al., 2026) poses grounded numerical-reasoning questions over historical U.S. Treasury Bulletin documents, requiring the agent to locate the relevant table across long, noisy source text and compute a numeric answer; we use a 70-case test split. DABench (Hu et al., 2024) poses data-analysis questions over tabular CSV files, requiring the agent to write and execute analysis code and report a closed-form answer; we use a 150-case test split.

Agents. We run the experiments under two model families. For GPT, the backbone of strong agent π_s is `gpt-5.4`; weak agents π_w use `gpt-5.4-mini` and `gpt-4.1-mini`. For Qwen3.5, we run SpreadsheetBench and DABench with `Qwen3.5-122B-A10B-FP8` as π_s and `Qwen3.5-4B` (SpreadsheetBench) or `Qwen3.5-2B` (DABench) as π_w, with π_s and π_w each served on a single H200 GPU; we switch to the smaller `Qwen3.5-2B` on DABench because `Qwen3.5-4B` was already too close to π_s to leave a meaningful gap to close. All agents are implemented as a tool-using agent loop on the OpenAI Agents SDK (OpenAI, 2025a). For the GPT experiments, the Abstractor, Diagnoser, and Patcher run on `gpt-5.4` and the Validator on `gpt-5.4-mini`; for Qwen3.5, all four run on `Qwen3.5-122B-A10B-FP8`.

Skills and baselines. Each benchmark starts from an original skill $\mathcal{S}_0$ that is already effective for π_s, reused unchanged across both model families. For SpreadsheetBench, $\mathcal{S}_0$ is Anthropic's official xlsx skill (Anthropic, 2025b). For OfficeQA, $\mathcal{S}_0$ is a pair of third-party public skills finance (Neuman, 2026) and ripgrep (xdg-claude, 2026). For DABench, $\mathcal{S}_0$ is a pair of third-party public skills data-scientist (borghei, 2026) and statistical-analysis (Giro03k, 2026). We compare four methods that all start from the same $\mathcal{S}_0$: *no skill*, running without any skill; *original skill*, the unmodified $\mathcal{S}_0$; *EvoSkill* (Alzubi et al., 2026), a skill-evolution baseline that optimizes the skill's capability; and our method Skill Transfer, run for an iteration budget of $K{=}6$ with up to $M{=}5$ strong-agent attempts (Algorithm 1). EvoSkill is given the same budget as ours: its proposer and generator run on the same

Table 3: SpreadsheetBench, OfficeQA, and DABench. Within each weak agent, **bold** is the best method for that column and underline the second best. CM = Cell Match; G = $\widehat{\text{GED}}$; $\leq X$ is accuracy within $X\%$ relative tolerance of the gold numeric answer; ByQ/PrQ are DABench's own accuracy-by-sub-question and accuracy-proportional-by-sub-question metrics.

		SpreadsheetBench			OfficeQA					DABench			
Agent	**Method**	**P% ↑**	**CM ↑**	**G ↓**	**P% ↑**	**≤.1 ↑**	**≤1 ↑**	**≤5 ↑**	**G ↓**	**P% ↑**	**ByQ ↑**	**PrQ ↑**	**G ↓**
Strong	No skill	59.0	0.717	–	52.9	57.1	62.9	67.1	–	82.0	86.3	85.0	–
`gpt-5.4`	Original skill	67.5	0.807	–	57.1	57.1	67.1	72.9	–	86.0	89.8	88.6	–
	No skill	15.5	0.468	0.280	17.1	17.1	24.3	27.1	0.335	74.7	80.4	79.9	0.204
Weak	Original skill	25.5	0.486	0.203	8.6	12.9	15.7	15.7	0.297	68.7	72.9	74.3	0.218
`gpt-4.1-mini`	EvoSkill	36.5	0.541	0.187	14.3	17.1	24.3	25.7	0.251	79.3	82.8	83.2	0.185
	Skill Transfer	**48.5**	**0.638**	**0.154**	**18.6**	**18.6**	**25.7**	**28.6**	**0.177**	**80.7**	**83.2**	**85.2**	**0.158**
	No skill	48.0	0.662	0.227	21.4	24.3	25.7	30.0	0.275	77.3	85.2	84.3	0.243
Weak	Original skill	39.5	0.632	0.111	14.3	18.6	22.9	28.6	0.168	81.3	87.6	86.0	0.126
`gpt-5.4-mini`	EvoSkill	50.0	0.683	0.099	27.1	28.6	**35.7**	**40.0**	0.127	**86.0**	87.9	87.2	**0.083**
	Skill Transfer	**51.5**	**0.691**	**0.095**	**30.0**	**31.4**	**35.7**	37.1	**0.109**	**86.0**	**89.7**	**89.9**	0.109

Table 4: Qwen3.5 on SpreadsheetBench.

Agent	**Method**	**P% ↑**	**CM ↑**	**G ↓**
Strong	No skill	55.0	70.5	–
`Qwen3.5-122B`	Original skill	59.5	72.2	–
	No skill	22.0	38.1	0.281
Weak	Original skill	15.5	35.5	0.222
`Qwen3.5-4B`	EvoSkill	17.0	33.3	0.223
	Skill Transfer	**26.0**	**42.5**	**0.204**

Table 5: Qwen3.5 on DABench.

Agent	**Method**	**P% ↑**	**ByQ ↑**	**PrQ ↑**	**G ↓**
Strong	No skill	72.7	88.9	87.8	–
`Qwen3.5-122B`	Original skill	74.0	89.4	88.0	–
	No skill	19.3	47.6	43.0	0.259
Weak	Original skill	7.3	48.8	45.8	0.273
`Qwen3.5-2B`	EvoSkill	13.3	42.6	45.5	0.252
	Skill Transfer	**22.0**	**48.9**	**48.4**	**0.243**

backbone and reasoning-effort setting as our Abstractor, Diagnoser, and Patcher, and its executor is the weak agent under test. Both methods therefore use a strong model to edit the skill and differ only in what that model is shown and how its edits are guided.

Metrics. Beyond task pass rate, we report benchmark-specific fine-grained metrics (cell match ratio for SpreadsheetBench, accuracy within a relative tolerance of $\leq$0.1%/$\leq$1%/$\leq$5% for OfficeQA, and DABench's own accuracy-by-sub-question / accuracy-proportional-by-sub-question for DABench), and average $\widehat{\text{GED}}$ (Eq. 5) to measure how procedurally close the weak agent's run is to the strong agent's.

4.2 Main Results

Table 3 reports pass rate and the fine-grained metrics for the GPT family on all three benchmarks; Tables 4 and 5 report the Qwen3.5 family on SpreadsheetBench and DABench (we omit OfficeQA for Qwen3.5: its source documents routinely exceed Qwen3.5's 262,144-token context window, so the task is not fully well-defined for these backbones). Across all eight weak configurations, Skill Transfer achieves the best pass rate, beating the skill-evolution baseline EvoSkill by up to 12.0pp and the original skill by 4.7–23.0pp. EvoSkill's proposer sees only the weak agent's own failures, never a strong-agent trajectory to diff against, so its edits add generic content rather than closing the specific gap the weak agent has.

The original skill's own effect on weak agents is not uniformly positive: it underperforms no skill in six of the eight weak configurations (e.g. 48.0→39.5 for `gpt-5.4-mini` on SpreadsheetBench, 17.1→8.6 for `gpt-4.1-mini` on OfficeQA), which shows that a skill written around the strong agent's implicit steps can actively mislead a weaker one that lacks them. Where the original skill does help, Skill Transfer still adds to it. On SpreadsheetBench `gpt-4.1-mini` goes from 15.5 with no skill to 25.5 with the original skill and 48.5 with Skill Transfer, closing 55% of the distance the original skill leaves to `gpt-5.4`. On DABench `gpt-5.4-mini` goes 77.3 → 81.3 → 86.0 and closes that distance entirely, matching `gpt-5.4`'s pass rate and its ByQ/PrQ (89.7/89.9 against 89.8/88.6). Appendix E shows the accumulated edits between an original skill and the skill Skill Transfer produced from it.

Table 6: Strong agent (`gpt-5.4`) on SpreadsheetBench, using a skill adapted for a weak agent.

Method	Adapted for	Pass % ↑
No skill	–	59.0
Original skill	–	67.5
Skill Transfer	`gpt-4.1-mini`	76.5
Skill Transfer	`gpt-5.4-mini`	81.3

Table 7: Out-of-distribution: WikiTableQuestions, using the xlsx skill adapted on SpreadsheetBench.

Weak agent	Skill	Acc % ↑
`gpt-4.1-mini`	Original skill	44.4
	Skill Transfer	**69.2**
`gpt-5.4-mini`	Original skill	75.6
	Skill Transfer	**77.6**

4.3 Structural Convergence

$\widehat{\text{GED}}$ decreases monotonically from no skill through original skill and EvoSkill to Skill Transfer in 4 of 8 configurations (both gpt-based weak agents on SpreadsheetBench and OfficeQA), and Skill Transfer reaches the lowest $\widehat{\text{GED}}$ in 7 of 8. Where the pattern holds, it shows that Skill Transfer closes the procedural gap identified in §3.3 rather than improving outputs incidentally. The weak agent has learned some of the procedural steps the strong agent performs.

4.4 Skill Transfer Also Benefits the Strong Agent

Although Skill Transfer targets the weak agent's procedural gap, Table 6 shows the adapted skill also raises the strong agent's own pass rate on SpreadsheetBench, from 67.5% to 76.5–81.3% depending on which weak agent the skill was adapted for. A patch that makes an implicit step explicit does not only help a weak model that would skip the step totally; it also helps a stronger model that performs the step inconsistently. The adapted skill converts latent or inconsistent capability into more reliable behavior, benefiting the strong agent too.

4.5 Out-of-Distribution Generalization

We test whether a skill adapted on one benchmark transfers to a structurally related but distinct task: WikiTableQuestions (Pasupat & Liang, 2015) (WTQ), where the agent answers a natural-language question about a table by writing the result into a spreadsheet cell rather than editing a workbook. Table 7 reports results for both weak agents, using the xlsx skill adapted on SpreadsheetBench. For `gpt-4.1-mini`, accuracy on 250 held-out WTQ questions rises from 44.4% to 69.2% (+24.8pp). For `gpt-5.4-mini`, accuracy rises from 75.6% to 77.6% (+2.0pp) – a much smaller gain, consistent with `gpt-5.4-mini`'s already-high accuracy under the original skill leaving less headroom to close. The skill was never trained on WTQ; the gains come from the same xlsx-handling procedure (e.g. correct cell addressing, formula evaluation, output placement) that Skill Transfer made explicit for SpreadsheetBench. The generalization demonstrates that Skill Transfer does not overfit on specific tasks, but truly closes the procedural gaps between weak and strong agents.

5 Conclusion

We formalized *skill transfer*: given a skill already effective for a strong agent, transfer it to a specific weak agent without touching that agent's parameters. Our method abstracts both agents' trajectories into typed execution structures, localizes their gaps with a graded graph edit distance, and repairs the skill through an iterative diagnoser–patcher loop. Across three benchmarks and four weak agents from two model families, Skill Transfer raises pass rate over both the unmodified original skill and a skill-evolution baseline, and $\widehat{\text{GED}}$ decreases toward the strong agent's structure in most configurations, so the gain comes from closing the procedural gap of §3.3. The adapted skill also improves the strong agent it was never adapted for, and transfers to a related task it was never trained on. However, two conditions bound the method. First, it needs a strong agent and a small set of tasks that the agent solves, since the strong trajectories are the reference the diagnoser diffs against; without such a reference there is no gap to localize. Second, the gain depends on why the weak agent fails. Procedural failures are largely recoverable by making the missing steps explicit, whereas failures bound by capability the weak model does not have are difficult to recover from. Within these bounds, skill transfer offers a training-free route to reusing the growing corpus of skills authored for frontier models on the smaller, cheaper agents that real deployments often require.

AI USE STATEMENT

We used generative AI tools for code implementation and code editing, including the pipeline, the evaluation scripts, and the dataset-preparation code. All such code was reviewed by the authors. We also used generative AI tools to polish the writing. Large language models are also the object of study here rather than only an authoring aid. The agents under evaluation, and the Abstractor, Diagnoser, Patcher, and Validator components of our method, are all LLM-driven, as described in §3 and §4. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. Evoskill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766*, 2026.

Anthropic. Introducing agent skills. Anthropic News, 2025a. `https://www.anthropic.com/news/skills`.

Anthropic. Agent skills. `https://github.com/anthropics/skills`, 2025b. GitHub repository. Accessed: 2026-06-29.

borghei. data-scientist (claude agent skill). `https://github.com/borghei/Claude-Skills`, 2026.

Yurui Chang, Yiran Wu, Qingyun Wu, and Lu Lin. Memcollab: Cross-model memory collaboration via contrastive trajectory distillation. *arXiv preprint arXiv:2603.23234*, 2026.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 13(1):113–129, 2010.

Giro03k. statistical-analysis (claude agent skill). `https://github.com/Giro03k/claude-statistical-analysis-skill`, 2026.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alex Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 8003–8017, 2023.

Xueyu Hu, Ziyu Zhao, Shuang Wei, Ziwei Chai, Qianli Ma, Guoyin Wang, Xuwu Wang, Jing Su, Jingjing Xu, Ming Zhu, et al. Infiagent-dabench: Evaluating agents on data analysis tasks. *arXiv preprint arXiv:2401.05507*, 2024.

Minki Kang, Jongwon Jeong, Seanie Lee, Jaewoong Cho, and Sung Ju Hwang. Distilling LLM agent into small models with retrieval and code tools. *arXiv preprint arXiv:2505.17612*, 2025.

Yifan Lan, Yuanpu Cao, Hanyu Wang, Lu Lin, and Jinghui Chen. The illusion of reasoning: Exposing evasive data contamination in LLMs via Zero-CoT truncation. *arXiv preprint arXiv:2605.21856*, 2026.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, et al. SkillsBench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026.

Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. Spreadsheetbench: Towards challenging real world spreadsheet manipulation. *arXiv preprint arXiv:2406.14991*, 2024.

Travis Neuman. finance (claude agent skill). https://agent-skills.md/skills/travisjneuman/.claude/finance, 2026.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

OpenAI. Openai agents sdk. https://openai.github.io/openai-agents-python/, 2025a.

OpenAI. Agent skills. https://github.com/openai/skills, 2025b. GitHub repository. Accessed: 2026-06-29.

Krista Opsahl-Ong, Arnav Singhvi, Jasmine Collins, Ivan Zhou, Cindy Wang, Ashutosh Baheti, Owen Oertell, Jacob Portes, Sam Havens, Erich Elsen, Michael Bendersky, Matei Zaharia, and Xing Chen. Officeqa pro: An enterprise benchmark for end-to-end grounded reasoning, 2026. URL https://arxiv.org/abs/2603.08655.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1470–1480, 2015.

Alberto Sanfeliu and King-Sun Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(3):353–362, 1983.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

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

Hanyu Wang, Yuanpu Cao, Lu Lin, and Jinghui Chen. PreFlect: From retrospective to prospective reflection in large language model agents. *arXiv preprint arXiv:2602.07187*, 2026a.

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

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=_VjQlMeSB_J.

xdg-claude. ripgrep (claude agent skill). https://claude-plugins.dev/skills/@xdg/xdg-claude/ripgrep, 2026.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026.

Jiajun Xu, Zhiyuan Li, Wei Chen, Qun Wang, Xin Gao, Qi Cai, and Ziyuan Ling. On-device language models: A comprehensive review. *arXiv preprint arXiv:2409.00088*, 2024.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 38:17577–17604, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Jianxiang Yu, Jiapeng Zhu, Bochen Lin, Qier Cui, Zichen Ding, and Xiang Li. Skill is not one-size-fits-all: Model-aware skill alignment for llm agents. *arXiv preprint arXiv:2605.30723*, 2026.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. Agenttuning: Enabling generalized agent abilities for llms. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3053–3077, 2024.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, et al. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, et al. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025a.

Yue Zheng, Yuhao Chen, Bin Qian, Xiufang Shi, Yuanchao Shu, and Jiming Chen. A review on edge large language models: Design, execution, and applications. *ACM Computing Surveys*, 57(8):1–35, 2025b.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 19724–19731, 2024.

A ABLATION: REWRITING THE SKILL FROM RAW TRAJECTORIES

To test whether the structural machinery of §3 is necessary, we run a *raw-trajectory* arm on SpreadsheetBench with `gpt-4.1-mini` as π_w. Everything outside the framework is held fixed: the same `gpt-5.4` backbone and reasoning-effort setting for diagnosis and patching, the same 18 training tasks (the 20-task training set after the strong-solves filter), the same validation set, K=6, the same $\mathcal{S}_0$, and byte-identical iteration-0 evidence. The raw arm removes trajectory abstraction and GED, the validator reports, the gap taxonomy, the cross-iteration statuses and the gap-type→strategy mapping, and the diagnoser's survey/deep-dive/synthesis steps with their admission rule. It keeps the raw strong and weak trajectories with the per-task pass/fail verdict, both meta-skills, and cross-iteration memory (its own previous report, the previous skill diff, and a per-task pass/fail table). Its diagnoser writes a free-form issue list with a patch hint per issue; its patcher applies the hints with the same editing tools as ours.

Results. Table 8 evaluates the three skills on the same test batch. The raw arm gains 5.5 points over the original skill, against 23.0 for Skill Transfer. The per-task view separates them further: the raw arm fixes 26 tasks the original skill fails but breaks 15 it passes, while Skill Transfer fixes 48 and breaks 2 Of the 48. Our method can fix more cases and shows less regression compared with directly using powerful agent to edit the skill.

Skill	P% ↑	CM ↑	Fixed	Broken
Original skill $\mathcal{S}_0$	25.5	0.486	–	–
Raw-trajectory rewrite	31.0	0.505	26	15
Skill Transfer	**48.5**	**0.638**	**48**	**2**

Table 8: Raw-trajectory ablation on SpreadsheetBench, `gpt-4.1-mini`, 200 test tasks. Fixed/Broken count tasks flipped relative to $\mathcal{S}_0$.

Analysis. Both diagnosers read the same trajectories and name many of the same problems. They differ in which gaps are expressible, which are admitted, and what form the resulting patch takes; Table 9 attributes each observed difference to the stage of §3 that produces it.

Stage	What its removal costs the raw arm
Typed execution structure (§3.2)	Nodes carry a completion criterion, so a step that runs but stops short is expressible as a `weak_node` rather than as a symptom. The raw patcher appends two steps to the original six-step workflow; ours rewrites it into thirteen, each with a criterion, and states nine hard-stop conditions against four.
Order-preserving alignment (§3.3)	Matching typed nodes exposes steps that are absent or superfluous, which leave no error trace to read off: the three gaps ours finds and the raw arm does not are all `missing_node` or `extra_wrong_node` (no requirement review before scripting; inspection that never becomes an edit; helper sheets left in the deliverable). The same alignment is scored by $\widehat{\text{GED}}$, which falls 0.203→0.154 in this configuration (Table 3).
Admission rule (§3.4)	$\lvert P_g\rvert \geq 2$ keeps a gap that recurs in runs that passed anyway. The raw diagnoser re-reads only the currently failing tasks (18 pairs at iteration 0, then 14, 9, 6, 8, 9), so the four gaps ours admits on passing evidence alone ($\lvert F_g\rvert$=0) lie outside its view.
Cross-iteration status (§3.4)	Tasks regress after a patch in every iteration of the raw run (one to three per iteration), but 2 of its 29 hints refer to a prior patch, against ten REGRESSION gaps in ours that order the patcher to replace the rule it previously added. Raw guidance therefore accumulates: 198 bullets, with "repair the whole repeated range" restated in seven places against two in ours.
Gap type → strategy (§3.4)	The hint fixes the form of the edit, not only its content. Without that mapping the patcher defaults to prohibition: 35 occurrences of "never"/"do not" against 12 in ours, and 69 lines of code against 123.

Table 9: Each stage of §3 removed in the raw arm, and what is lost with it.

We also find that the form of the guidance, not only its content, decides whether the weak agent complies. Both skills forbid reading an `.xlsx` with `head` or `cat`. The ablated skill states the rule as a bullet inside a later section, 190 lines before its first line of workbook-opening code, and 44% of its runs still read the file as text; ours states it in the overview, eight lines above the inspection code

the agent is meant to run instead, and the rate falls to 9%. Three of the 15 tasks the raw arm breaks are that failure verbatim: `head -40` on the input, gibberish, and a stop within five steps. Which form a patch takes is exactly what the gap-type→strategy mapping fixes.

B EXECUTION STRUCTURE EXAMPLE

Listing 1 shows the execution structure the Abstractor produced for the weak agent (`gpt-4.1-mini`) on SpreadsheetBench case 510-3 at iteration 0. Each entry of `nodes` is one procedure node v_i of Eq. 2: `category_tier1` is c_i^1 and `category_tier2` is c_i^2, while `completes_when` and `tool_calls` are the supporting detail d_i, whose `code_snippet` and `observation` are spliced in verbatim from the raw trajectory. The remaining fields are auxiliary: `id` and `label` let a gap report point at individual nodes (Listing 2), `agent_intent` and `purpose` record the agent's stated reason for the step and the call, `produces` and `consumes` give lightweight data flow, and `skill_selection_reason` carries the verbatim `reason` argument of an `activate_skill` call, which is what makes wrong-skill routing detectable. Long snippets, observations, and absolute paths are abbreviated with `...` for space.

Node `node_003` is the weak VALIDATE step that §3.3 calls a `weak_node`: the role is present, but its completion criterion stops at "recalculated successfully without processing errors" rather than reopening the workbook and checking the edited region.

Listing 1: Execution structure abstracted from a weak-agent trajectory (SpreadsheetBench case 510-3, iteration 0).


```
{
  "agent_id": "weak_iter0",
  "task": "Recreate the 'Data' sheet, preserve D:G formatting, hardcode Sum values, and save
      as output.xlsx",
  "total_steps": 5,
  "outcome": "success",
  "nodes": [
    {
      "id": "node_000", "label": "skill_activation",
      "category_tier1": "setup", "category_tier2": "skill_activation",
      "agent_intent": "Prepare the spreadsheet tooling needed to modify the Excel workbook.",
      "completes_when": "when the xlsx skill is activated and its spreadsheet handling
          instructions are available for use",
      "produces": ["xlsx_skill_context"], "consumes": [],
      "skill_selection_reason": "The task involves reading, modifying, and saving an Excel
          workbook while preserving formatting and hardcoding calculated values, which is
          best handled with the xlsx skill.",
      "tool_calls": [{
        "tool": "activate_skill",
        "purpose": "Activate the spreadsheet-editing skill required for workbook modification",
        "code_snippet": "{\"name\": \"xlsx\", \"reason\": \"The task involves reading,
            modifying, and saving an Excel workbook file ...\"}",
        "observation": "<activated_skill name=\"xlsx\"><instructions># Requirements for Outputs
            ...",
        "outcome": "success"}],
      "outcome": "success"
    },
    {
      "id": "node_001", "label": "task_requirement_review",
      "category_tier1": "setup", "category_tier2": "task_requirement_review",
      "agent_intent": "Understand the requested transformation and constraints before
          generating a script.",
      "completes_when": "when the sheet recreation requirements, formatting constraints, Sum
          calculation, and output filename are known",
      "produces": ["task_requirements"], "consumes": ["instruction_file"],
      "tool_calls": [{
        "tool": "read_file",
        "purpose": "Read the case instructions to capture the requested workbook edits and
            output requirements",
        "code_snippet": "{\"file_path\": \".../cases/510-3/case_input/INSTRUCTION.md\"}",
        "observation": "How can I calculate the total number of days from a date provided in
            column D of my 'Data' sheet ...",
        "outcome": "success"}],
      "outcome": "success"
    },
    {
      "id": "node_002", "label": "sheet_recreation_attempt",
      "category_tier1": "execute", "category_tier2": "sheet_recreation_attempt",
```

```
    "agent_intent": "Carry out the main workbook transformation by generating and running
        automation code.",
    "completes_when": "when a script has successfully recreated the Data sheet, preserved
        the required D:G formatting, hardcoded Sum values, and saved the modified workbook",

    "produces": ["transformation_script", "recreated_data_sheet", "hardcoded_sum_values", "
        output_workbook"],
    "consumes": ["task_requirements", "input_workbook"],
    "tool_calls": [{
     "tool": "write_file",
     "purpose": "Create a Python script to load the workbook, recreate the Data sheet,
         preserve formatting, calculate Sum values, and save",
     "code_snippet": "from openpyxl import load_workbook, Workbook\nfrom openpyxl.utils
         import get_column_letter\nimport openpyxl.styles as styles\n\ninput_path = ...",
     "observation": "Successfully wrote 3131 characters to .../weak_outputs/script.py",
     "outcome": "success"}, {
     "tool": "shell",
     "purpose": "Execute the generated Python script to produce the modified workbook",
     "code_snippet": "python3 .../weak_outputs/script.py",
     "observation": "exit_code: 0",
     "outcome": "success"}],
    "outcome": "success"
   },
   {
    "id": "node_003", "label": "workbook_recalculation_check",
    "category_tier1": "validate", "category_tier2": "workbook_recalculation_check",
    "agent_intent": "Validate that the produced workbook is usable after modification.",
    "completes_when": "when the generated workbook has been recalculated successfully
        without processing errors",
    "produces": ["recalculated_workbook_check"], "consumes": ["output_workbook"],
    "tool_calls": [{
     "tool": "shell",
     "purpose": "Recalculate the workbook to check whether the produced file opens cleanly
         and formulas or values evaluate without immediate errors",
     "code_snippet": "python3 .../working_skills/xlsx/scripts/recalc.py .../weak_outputs/
         output.xlsx",
     "observation": "stdout:\n{\"status\": \"success\", \"total_errors\": 0, \"
         total_formulas\": 0}\n\nexit_code: 0",
     "outcome": "success"}],
    "outcome": "success"
   },
   {
    "id": "node_004", "label": "output_workbook_save",
    "category_tier1": "output", "category_tier2": "output_workbook_save",
    "agent_intent": "Deliver the modified workbook in the requested output location.",
    "completes_when": "when output.xlsx exists in the output directory after the sheet
        recreation and Sum hardcoding steps",
    "produces": ["output.xlsx"], "consumes": ["output_workbook", "
        recalculated_workbook_check"],
    "tool_calls": [],
    "outcome": "success"
   }
  ]
}
```


C GAP REPORT EXAMPLE

Listing 2 shows one gap entry from a diagnoser gap report on SpreadsheetBench (`gpt-4.1-mini`, iteration 1; the full report for that iteration contains 15 such gaps). The prior iteration's patch had pushed the weak agent toward recalculating after an edit, but the weak agent still stops at a superficial check (recalc succeeding, or one unrepresentative cell) instead of reopening the workbook and spot-checking the target region; the diagnoser marks this gap `REGRESSION` (Table 2) across seven failing cases, and the `skill_patch_hint`'s PRIOR-PATCH CRITIQUE names four of them individually before prescribing a replacement verification protocol, following the TARGET/STYLE/WORDING/PRIOR-PATCH-CRITIQUE structure described in §3.4.

Listing 2: One gap from a diagnoser gap report (`gap_type=weak_node`, `status=REGRESSION`).


```
{
  "gap_id": "gap_005",
  "gap_type": "weak_node",
  "sub_dimension": "no value-level spotcheck",
  "target_skill": "xlsx",
```

```
  "category_tier1": "validate",
  "category_tier2": "output_verification",
  "label": "recalc_without_targeted_readback_verification",
  "description": "The previous patch moved the agent toward recalculation, but current weak
       runs still often stop at recalc success, formula-only inspection, or a one-cell check
       instead of reopening the saved workbook and checking the actual visible results in the
        task-relevant region. This now appears in more failing cases than in iter 0, so the
       prior patch direction was right but measurably insufficient.",
  "weak_evidence": {
    "267-21": "python3 .../scripts/recalc.py .../weak_outputs/output.xlsx 30",
    "399-14": "print('Checking cells N12 and O12:', ws['N12'].value, ws['O12'].value)",
    "44628": "VALUES D4:G4: [20, 15, 35, 35]",
    "496-15": "ls -l .../weak_outputs/output.xlsx",
    "50631": "python3 .../scripts/recalc.py .../weak_outputs/output.xlsx",
    "50916": "print('Sample cell formulas and values:'); cells_to_check = ['C12','D12','E12','
       F12','G12','D13','E13','D14','E14']; [(c, ws_formulas[c].value, ws_values[c].value)
       for c in cells_to_check]",
    "57612": "FORMULA D4: =IF(AND(C4>=5,C4<=6),5,IF(AND(C4>=3,C4<=4),3.75,IF(AND(C4>=1,C4<=2)
       ,1,0)))"
  },
  "strong_evidence": {
    "267-21": "for r in ws.iter_rows(values_only=True): print(r)",
    "399-14": "for r in [12,15,16,18,21,22,7]: print(r, ws[f'N{r}'].value, ws[f'O{r}'].value,
       ws[f'P{r}'].value)",
    "44628": "print([ws[c].value for c in ['D3','E3','F3','G3']])",
    "496-15": "for row in range(1, ws.max_row + 1): print(row, d, e)",
    "50631": "for coord in ['B36','B37','B38','D7','D38','F7','F38','H7','H30','H36','H37','
       H38']: print('value', coord, ws2[coord].value)",
    "50916": "for r in [12,13,14,17,18,19,20,21]: print(r, [ws.cell(r,c).value for c in range
       (1,9)])",
    "57612": "for r in range(4,16): print(r, wsv[f'C{r}'].value, wsv[f'D{r}'].value)"
  },
  "strong_node_ref": {"267-21": "node_007", "399-14": "node_004", "44628": "node_004",
    "496-15": "node_005", "50631": "node_004", "50916": "node_004", "57612": "node_004"},
  "weak_node_ref": {"267-21": "node_009", "399-14": "node_005", "44628": "node_006",
    "496-15": "node_003", "50631": "node_006", "50916": "node_005", "57612": "node_004"},
  "strong_completes_when": "when the saved workbook has been reopened in both formula and
       data-only views and the task-relevant edited region has been explicitly spot-checked",
  "weak_completes_when": "when recalc succeeds, the file exists, or a tiny unrepresentative
       slice looks plausible, without verifying the actual target range or final visible
       values",
  "missing_produces": ["verification_report", "spotcheck_results"],
  "skill_patch_hint": "TARGET: patch xlsx/SKILL.md in the Recalculating formulas section and
       the Common Workflow after recalc. STYLE: MANDATE, and because this gap is REGRESSION,
       explicitly replace the prior weak mechanism rather than merely intensifying it.
       WORDING: add a post-recalc hard-stop checklist saying the agent must reopen the
       canonical output.xlsx twice: once with data_only=False to verify formulas or
       structural edits landed in the intended addresses, and once with data_only=True to
       verify visible values in the actual destination region. Require four checks: the named
        target cells from the prompt, the first edited row, one middle or edge row, and the
       final edited or summary row. Also require one unchanged control row or cell to confirm
        no unintended spillover. Add a hard-stop sentence: 'A zero-error recalc report, save
       message, or file-exists check is not completion. If readback shows wrong placement,
       blank results, untouched ranges, wrong totals, or the wrong sheet layout, repair and
       re-verify.' PRIOR-PATCH CRITIQUE: the iter-0 patch successfully made recalc common,
       but current structures show the agent satisfies the letter while skipping the real
       semantic check. 496-15 stops at ls -l. 50631 ends after recalc and an H38 patch
       without checking the full date extent. 50916 executes a list comprehension that prints
        nothing, then claims success. 44628 reads back the wrong cells and accepts D4:G4
       instead of D3:G3. The new patch must therefore switch mechanisms from generic recalc
       guidance to a mandatory readback protocol with explicit target-region coverage.",
  "failed_case_ids": ["267-21", "399-14", "44628", "496-15", "50631", "50916", "57612"],
  "pass_case_ids": ["84-40"],
  "status": "REGRESSION"
}
```


D VALIDATOR REPORT EXAMPLE

Listing 3 shows a validator report for a SpreadsheetBench case (48608, iteration 0). The validator compares the weak agent's `output.xlsx` against the strong agent's own output, not the gold answer (§3.4). Its top-level `verdict` is PASS only if every expected file matches exactly; PARTIAL if a file is present but imperfect (as here) or one expected file is entirely missing; FAIL if most expected files are missing or severely wrong. The `scores` block gives `file_presence_ratio` (share of expected files present in any form), `content_match_ratio` (share of present files that are an exact match), and their weighted blend `overall_quality_score` $= 0.4 \times$

presence + 0.6 × content. Within a file's `detail`, `row_match_ratio` counts whole strong rows found anywhere in the weak output, `cell_match_ratio` is the finer-grained cell-by-cell match, and `per_column_match_ratio` breaks the latter down by column. Here, the file is present with matching sheet structure, but a 0.0 match ratio on the PASSES column, yielding `verdict=PARTIAL`.

Listing 3: Validator report for case 48608, iteration 0 (`verdict=PARTIAL`).


```
{
  "strong_agent": "strong",
  "weak_agent": "weak_iter0",
  "verdict": "PARTIAL",
  "result_quality": {
    "files": {
      "present": [
        {
          "filename": "output.xlsx",
          "matched_weak_filename": "output.xlsx",
          "content_match": false,
          "comparator": "xlsx",
          "detail": {
            "sheets_match": true,
            "strong_sheets": ["Sheet1"],
            "weak_sheets": ["Sheet1"],
            "cell_match_ratio": 0.9091,
            "per_sheet": {
              "Sheet1": {
                "present": true,
                "row_match_ratio": 0.125,
                "cell_match_ratio": 0.9091,
                "strong_rows": 8,
                "weak_rows": 17,
                "missing_rows": 7,
                "extra_rows": 16,
                "per_column_match_ratio": {
                  "Name": 1.0,
                  "Session": 1.0,
                  "Result": 1.0,
                  "": 1.0,
                  "Passes (Hidden)": 1.0,
                  "Passes": 0.0
                }
              }
            }
          }
        }
      ],
      "missing_from_weak": [],
      "extra_in_weak": []
    },
    "scores": {
      "file_presence_ratio": 1.0,
      "content_match_ratio": 0.0,
      "overall_quality_score": 0.4
    }
  },
  "analysis": "The weak agent produced the required workbook file, and the workbook structure
       matches the reference sheet name. However, the sheet contents do not fully match the
      reference. The overall cell match is high, but the key 'Passes' column does not match
      the reference, and the row alignment is poor, with many extra and missing rows. This
      indicates a partially correct spreadsheet output rather than an exact reproduction.",
  "verdict_reasoning": "There is one expected file and it is present, so file presence is
      complete. The content is not an exact match due to differences in the workbook rows
      and the 'Passes' column, so content_match_ratio is 0. Because the output is present
      but not correct, the verdict is PARTIAL."
}
```


E ADAPTED SKILL EXAMPLE

Listing 4 shows the full diff between the original xlsx skill $\mathcal{S}_0$ and the skill adapted for `gpt-5.4-mini` on SpreadsheetBench: four hunks, 63 lines removed and 79 added. The edits fall into three kinds: unconditional rules become conditional on what the prompt asks, vague instructions become explicit scoping steps that must precede an edit, and generic verification advice

becomes a mandatory readback gate on the saved workbook together with rules for what to do when it fails.

Listing 4: `skill_changes_xlsx.diff` for `gpt-5.4-mini` on SpreadsheetBench. Header paths are shortened and long lines are cut at a sentence boundary, marked `[...]`.

```
--- skill_baseline_xlsx/SKILL.md
+++ skill_final_xlsx/SKILL.md
@@ -15,8 +15,9 @@
 - 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
+- Inspect and EXACTLY match the workbook's structure, styles, merges, dimensions, and
conventions before editing
+- If the prompt says recreate, rebuild, or preserve a sheet exactly, capture the template
first and treat sheet copy/recreation as a required step before value fixes
+- Never treat in-place edits as equivalent to recreation when template fidelity, split
fields, or preserved layout matter
 - Existing template conventions ALWAYS override these guidelines

 ## Financial models
@@ -96,57 +97,26 @@

 ## Excel File Workflows

-## CRITICAL: Use Formulas, Not Hardcoded Values
+## CRITICAL: Choose formulas vs hardcodes explicitly

-**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.
+- If the task asks to build, fix, copy, extend, summarize, or preserve spreadsheet behavior
-- or the workbook should update when source cells change -- write Excel formulas into the
workbook, not Python-computed results
+- If the prompt explicitly asks for static final values, hardcoded outputs are allowed
because the user requested a non-dynamic result
+- Do not choose Python-computed cell values merely because they are easier when workbook
logic is part of the deliverable
+- Dynamic example: write a lookup, SUMIF, or fill-down formula into the target cells
+- Static exception example: the prompt explicitly asks for final pasted values with no
formulas
+- When the dynamic branch applies, reopen the saved workbook with normal loading and confirm
 the target cells contain formulas, not just displayed numbers

 ## 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 scripts/recalc.py script
+3. **Audit target before editing**: identify the editable deliverable sheet, any source-data
 sheets, and any example/reference/oracle sheets before changing cells. Do not overwrite
sheets named like Desired Results, Example, Template, or Reference unless the prompt
explicitly names them as the sheet to modify. Then identify the exact destination range,
condition or lookup columns, first and last affected row, and whether the task is whole-
column, bounded-range, table, merged-region, summary-section, or sheet-recreation work. If
the task deletes, filters, copies, or keeps rows based on only some columns inside a wider
visible band, write down the exact named condition columns first and treat adjacent numbering
, helper, or metadata columns as out of scope unless the prompt names them explicitly. If the
 destination subset is smaller than the source table, derive destination start/end rows
separately from source lookup bounds. For existing summary blocks, identify the label column
and the result column before writing anything; prove the mapping on the first label row and
on one later label row so you know the real destination cells. Check at least one sample near
 the start and one near the end of the intended span. Do not infer bounds from the prompt
alone or from a tiny preview. If the prompt says recreate or preserve a sheet exactly,
inspect styles, merged cells, and layout anchors before touching data
+4. **Modify the workbook state**: apply the requested spreadsheet change itself; do not
substitute code text, macro text, instructions, comments, or note cells as the main solution.
 For formula repairs, copied patterns, schedule grids, or repeated rules, state the intended
destination start row, end row, start column, and end column before writing, then apply the
change across that exact region. If the prompt names one cell as an example but describes a
whole column, repeated rule, copied formula pattern, or schedule grid, derive the workbook-
defined destination span from the target area itself and fill every target row in that span,
e.g. `for row in range(start_row, end_row+1): if ws[f'{driver_col}{row}'].value is not None:
ws[f'{target_col}{row}'] = make_formula(row)`. For report strips, pass lists, or other
bounded destinations, destination bounds are separate from source lookup bounds. If a row-
selection rule depends on a named subset of columns, build the predicate from that exact
subset rather than the whole inspected slice: bad = checking every visible column because it
is nearby; good = checking only the explicitly named subset, then applying deletion/filter/
copy logic from that predicate. Read back one kept-row boundary and one deleted-row boundary
to prove the subset-column rule, not just the final row count. If merged cells block direct
fill, build the full editable non-merged target set and fill every editable target cell
across the whole intended grid; merged cells are not permission to retreat to a tiny safe
subset. For workbook summary sections, preserve the existing label/result layout: if rows
already contain labels like `Sum A`, `Sum B`, or similar, write formulas into the established
 result cells adjacent to those labels rather than into a nearby blank column. Verify the
same label-to-result offset on more than one summary row before broad fill. For schedule/
calendar formulas, classify each reference before fill as fixed-row, fixed-column, or
relative. Keep the day/cycle key on the current row in the day column, let the return range
move with the destination period column, and build one anchor formula that survives both
horizontal and vertical translation, e.g. `=IFERROR(INDEX(C$2:C$8,MATCH($B12,$B$2:$B$8,0))
,"")`; then prove it on the anchor cell, one horizontally shifted neighbor, and one next-row
cell before broad fill. Do not translate a legacy nested-IF or copied formula whose anchor
already points at the wrong row, header, day field, or prior output cell, and do not build
chain formulas that match against earlier filled schedule cells. For mixed text-and-number
summary cells like category codes with numeric suffixes, build the formula from the actual
data-cell range itself, not from summary labels or an oversized sheet rectangle: compare the
category token to `LEFT(source_range,1)`, parse the numeric suffix from that same `
source_range`, and sum with an error-tolerant conversion so blanks and nonmatches contribute
zero. Do not compare whole cells in a broad block to a bare category letter, and do not parse
 suffixes from label rows or blank-leading regions. If readback or validation shows the
visible divergence is in a repeated date-series column, rebuild that exact date family first
from an explicit row offset from the first target row, e.g. `=IF(start_date+offset<=end_date,
start_date+offset,"")`, rather than repairing a nearby formula family that merely coexists in
 the workbook. After rebuilding the date family, verify the first date, the last intended
date, and the first blank cell after the series. One-cell or sample-cell patches are not
sufficient unless the prompt explicitly names one cell as the entire target
+5. **Save**: Write to file
+6. **Recalculate formulas (MANDATORY IF USING FORMULAS)**: Use the scripts/recalc.py script
   ```bash
   python scripts/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
+7. **Read back the final file (MANDATORY ACCEPTANCE GATE)**: after every edit, reopen the
saved workbook and verify the exact target cells against the actual task invariant; do not
use readback as logging only. If formulas were inserted or changed, run recalc first, then
use `data_only=True` to check final values and normal loading when formula text or formatting
 also matters. For small edited regions, check every target output cell. For large edited
regions, check the first edited cell, the last edited cell, one boundary case, and one logic-
proving case that can expose the likely failure mode. Mark each check `PASS` or `FAIL`
```

```
against the business rule, not merely against "cell has a value," "format looks right," or "
file opens." Treat readback as a go/no-go gate: if a class cell reads `False` or blank where
a class label should appear, a repaired summary reads all zeros despite visible matching
source data, a latest-four result disagrees with a manual row check, a date-series checkpoint
 still contradicts the requested progression, the first untouched row now contains the filled
 formula, or any other target invariant fails, the workbook is not deliverable. Return to
editing, repair the failing logic, and repeat save -> recalc -> readback. The agent MUST NOT
deliver the workbook until every readback check passes

 ### Creating new Excel files

@@ -204,6 +174,22 @@
 wb.save('modified.xlsx')
 ```

+Before modifying an existing workbook:
+- Identify sheet roles first: which sheet is the editable deliverable target, which sheets
are only source data, and which sheets are examples, reference tabs, templates, or desired-
results/oracle tabs
+- Use example/reference/desire-results sheets to infer the required transformation or
compare outcomes, but do not overwrite them unless the prompt explicitly tells you to edit
that sheet; bad: copying the desired ordering into the reference sheet, good: reordering the
working sheet so it matches the reference arrangement
+- Derive the full affected region from the workbook itself: source/destination sheet, start
row, end row, start column, end column, stopping rule, and any condition columns or table
bounds
+- If row deletion, filtering, or copying depends on only some columns inside a wider band,
identify the exact named subset columns before editing and keep nearby numbering/helper
columns out of the predicate unless the prompt explicitly includes them
+- If the task says recreate a sheet, rebuild a template, or preserve a tab exactly,
duplicate or rebuild the target structure first, preserve row/column layout plus style
anchors and merges, then populate repaired values or formulas
+- If the prompt names one cell but describes a repeated rule, copied formula pattern, or
whole target column, derive the workbook-defined first/last target rows plus start/end
columns from the destination area and fill every row in that span rather than only the named
example; state `start_row`, `end_row`, `start_col`, and `end_col` before writing. Reusable
pattern: `for row in range(start_row, end_row+1): if ws[f'{driver_col}{row}'].value is not
None: ws[f'{target_col}{row}'] = template_for(row)`
+- If merged cells block direct fill, enumerate the full editable non-merged target set
across the intended grid and fill every editable target cell; encountering merged cells is
not permission to abandon the rest of the grid
+- For existing summary sections, identify the label column and the result column from the
sheet structure before writing formulas. Prove the destination mapping on the first label row
 and one later label row, then write into those established result cells rather than a nearby
 blank column
+- For schedule/calendar formulas, classify references before filling: fixed-column, fixed-
row, or relative. Keep the match key on the current row in the day column, let the return
range move with the destination period column, and build one anchor formula that translates
correctly in both directions, e.g. `=IFERROR(INDEX(C$2:C$8,MATCH($B12,$B$2:$B$8,0)),"")`;
then test the anchor cell, one horizontally shifted neighbor, and one next-row cell before
broad fill
+- Do not translate a legacy nested-IF or copied formula whose anchor already points at the
wrong field (for example the wrong row, header, date column, day field, or prior output cell)
; translation spreads wrong logic faithfully
+- If writing category totals from mixed cells like A2, B1, or F3, build the formula from the
 actual data-cell range itself, not from summary labels or the whole sheet rectangle. Compare
 `LEFT(source_range,1)` to the category token, parse the numeric suffix from that same `
source_range`, and sum with an error-tolerant conversion so blanks and nonmatches contribute
zero; generic pattern: match the prefix from `source_range`, extract the suffix from `
source_range`, and aggregate only those parsed matches
+- Do not compare whole cells in a broad block to a bare category letter, and do not parse
suffixes from label rows or blank-leading regions; those anti-patterns collapse the logic to
mismatches and zero totals
+- Zero totals across every summary row are a red flag when the visible source table contains
 matching mixed letter-number entries
+- If only a subset of rows belongs in the output, derive destination start/end rows
separately and bound the fill to that subset rather than extending to the whole sheet by
default
+
 ## Recalculating formulas

 Excel files created or modified by openpyxl contain formulas as strings but not calculated
 values. Use the provided `scripts/recalc.py` script to recalculate formulas:
@@ -224,27 +210,57 @@
 - Returns JSON with detailed error locations and counts
 - Works on both Linux and macOS

+Recovery rules after recalc:
+- If `status` is `errors_found`, inspect `error_summary`, fix the specific cells, and rerun
recalc immediately
```
```

```
+- Do not perform multiple near-identical formula rewrites that keep the same failing
coercion or parsing pattern
+- After the second recalc failure on the same target cells, stop whole-range rewrites. Make
one real mechanism switch on one pilot target cell, prove it on the saved workbook, and only
then fill the region
+- Do not cycle through TAKE/FILTER, LARGE/IF, SUMPRODUCT-with-LARGE, AGGREGATE, or similar
array-heavy variants that all chase the same underlying idea without first proving one engine
-compatible mechanism on the pilot cell. Superficial syntax swaps do not count as a new
mechanism
+- Change mechanism based on the error class; for mixed text/number parsing across ranges,
use an error-tolerant formula pattern so blanks, unmatched cells, or nonnumeric entries
contribute zero instead of propagating errors
+- For row-wise "latest N nonblank values" tasks, do not use `COUNT(range)` alone to choose a
 contiguous tail slice unless the workbook guarantees nonblank entries are left-packed. When
blanks may appear between valid entries, derive the cutoff position of the Nth-latest
populated cell first from a nonblank-position mask such as `LARGE((range<>"")*COLUMN(range),
MIN(N,COUNT(range)))`, then aggregate only cells at or after that cutoff with a scalar
pattern such as `SUMPRODUCT((range<>"")*(COLUMN(range)>=cutoff)*range)/MIN(N,COUNT(range))`
+- For latest-N recovery, write the candidate formula to one representative row, run recalc,
reopen with `data_only=True`, compare the numeric result to a manual latest-N check for that
row, and only then fill the remaining rows
+- Do not attempt a third retry unless the mechanism has actually changed and the pilot-cell
proof succeeded
+- After recalc succeeds, still reopen the saved workbook and read back one or two target
cells to confirm no formula errors and plausible values
+
 ## 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
+### Scope before writing
+- [ ] Derive the exact target span from the workbook: sheet, start/end row, start/end column
, stopping rule, and any condition/lookup columns
+- [ ] If the destination subset is smaller than the source table, derive destination start/
end rows separately and state them before filling
+- [ ] Check at least one sample near the start and one near the end of the intended span
+- [ ] Confirm column mapping and row offsets before filling formulas
+- [ ] If preserving or recreating a template, inspect styles, merges, and layout anchors
before editing
+
+### Building and filling formulas
+- [ ] Test 2-3 sample references before broad fill
+- [ ] Verify dependencies and cross-sheet references exist
+- [ ] State the destination `start_row`, `end_row`, `start_col`, and `end_col` before
writing any repeated-rule fix
+- [ ] If the prompt names an exemplar cell but describes a repeated rule, fill the full
workbook-defined region rather than just the sample cell
+- [ ] If a row condition uses only some columns in a wider visible band, define and use that
 exact named subset rather than the whole slice
+- [ ] If merged cells exist, enumerate all editable non-merged targets in the intended grid
and fill all of them
+- [ ] For existing summary rows, identify the label column and the result column, prove the
offset on the first label row and one later label row, and write only into those established
target cells
+- [ ] For copied calendar/schedule formulas, decide fixed-column vs fixed-row vs relative
references before fill; keep the match key on the current row day column, let the return
range move with the destination period column, and test one anchor cell, one horizontally
shifted neighbor, and one next-row cell
+- [ ] Do not copy a schedule formula whose anchor already points at the wrong field or a
prior output cell
+- [ ] For mixed text+number category summaries, compare the category token to the prefix
from the real data-cell range and parse the numeric suffix from that same range; do not
aggregate from label rows or oversized sheet rectangles
```

```
+- [ ] For latest-N-nonblank logic, reject a `COUNT(range)`-based contiguous tail unless
nonblank cells are guaranteed to be left-packed; prove the formula selects positions of the
last N populated cells
+- [ ] Before broad fill after a formula-family repair, prove one pilot row against a manual
check on the recalculated workbook
+- [ ] Bound the fill to the true target subset; do not extend to unrelated rows by default
+- [ ] Use error-tolerant patterns when blanks or nonnumeric text may appear
+
+### Readback after save
+- [ ] Run `scripts/recalc.py` after formula changes
+- [ ] Reopen with `data_only=True` for final values and normal loading when formula text or
formatting must also be checked
+- [ ] Compare observed exact target cells to the task-specific acceptance rule, not just to
"file opens," "format looks right," or "cell has something"
+- [ ] Small target: check every output cell. Large target: check the first edited cell, the
last edited cell, one boundary case, and one logic-proving case
+- [ ] Mark every readback check `PASS` or `FAIL` explicitly
+- [ ] Treat `False` or blank class cells where labels should appear, all-zero summaries
despite visible matching source data, latest-four results that fail a manual row check, date
progressions that still diverge, content mismatches hidden by correct formatting, `#VALUE!`,
`None`, wrong booleans, numerically wrong invariants, or a changed first-untouched row as
failure signals that require more editing before delivery
+- [ ] For summary blocks, read back the real destination cells, not a nearby column that
merely contains plausible computed values
+- [ ] If repeated date-series columns still diverge, repair that exact date family first and
 re-check the first date, last intended date, and first blank after the series
+- [ ] For row-deletion/filter tasks based on subset columns, read back one kept-row boundary
 and one deleted-row boundary that prove the predicate used the right columns
+- [ ] For pilot-cell recovery after repeated recalc failures, prove one target cell
calculates correctly before filling the rest of the region
+- [ ] For dynamic-workbook tasks, confirm the target cells still contain formulas, not
hardcoded numbers
+- [ ] For recreated sheets, verify both preserved structure/formatting and repaired content
fields

 ### Interpreting scripts/recalc.py Output
 The script returns JSON with error details:
```