AGENTIC SKILL SELF-CREATION VIA MULTIMODAL PROACTIVE EXPLORATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic skills, which encode workflows, application, function specific conventions, and instructions, have become an indispensable component of LLM-based computer-use agents. However, whether written by humans or generated automatically, skills are often incomplete: their creators cannot easily anticipate which details an agent lacks, such as the location of a specific button or differences across application versions. Existing skill refinement pipelines also remain inefficient, as they typically require the AI agent to roll out a large number of task trajectories before invoking a separate refinement module to extract lessons from these trajectories. In this paper, we argue that the AI agent should take a more proactive role in creating and refining skills, because it is best positioned to identify what information is missing and can actively gather that information by interacting and exploring and searching resources, whereas prior work have much more limited access to such information. We therefore propose PROACTSKILL, task-specific agentic multimodal skill self-creation pipeline that, emulating human researchers, enables the AI agent to create skills for itself by searching application documentation and online resources, and to continuously refine these skills by exploring and interacting with the application during task execution. The created skills are multimodal, so they incorporate visual information throughout the pipeline to make the resulting skills more informative and actionable. We evaluate PROACTSKILL on agentic benchmarks and show that our proactive skill creation pipeline produces higher quality skills than existing skill creation baselines.

1 INTRODUCTION

Computer-use agents based on large language models (LLMs), which operate applications by interacting with graphical user interfaces, are becoming increasingly important. A common approach to improving these agents is to equip them with AI agent **skills**, which compactly encode workflows, application- or function-specific conventions, and instructions on how and when to use the skill (Anthropic). The performance of AI agents therefore depends heavily on the quality of these skills. Ideally, a high-quality skill provides all the information an AI agent needs to accomplish a task in a form that the agent can readily understand and use.

However, existing skill creation pipelines often fail to produce such high-quality skills due to two major challenges. The first challenge lies in **skill creation**. Specifically, many existing skills are curated by human experts, who may inadvertently omit details they consider common sense but that are in fact unfamiliar to AI agents. Moreover, human experts may compile skills in a way that is optimized for human understanding rather than for AI agents. For example, a skill may simply instruct "Export as a PDF", leaving implicit that a dialog will appear, that the save location and file name needs to be set, and that the export happens only after clicking a save or confirmation button. A human is able to complete these steps without thinking, but an agent may stall at the unexpected dialog or never confirm the export.

The second challenge lies in **skill refinement**. Most existing refinement methods follow the same paradigm: the agent first rolls out many trajectories with its current skill, and these trajectories are then distilled into an improved skill. Methods differ mainly in how this distillation is done, whether by reflecting on failed attempts, summarizing successful ones, reusing them as in-context examples, or deriving rules through trial and error (Shinn et al., 2023; Wang et al., 2024; Kagaya et al., 2024;

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.

Zheng et al., 2024). For most skill refinement methods, the resulting knowledge comes only after many costly rollouts, and it is limited to what the agent has already experienced from the trajectory dataset.

Essentially, these two challenges stem from the same mismatch. The agent has the clearest view of what information it lacks, and it is best positioned to obtain that information by interacting with the application and searching online. Yet in existing pipelines, the agent's role is limited to attempting the task, while the knowledge/information critical steps (skill creation and refinement) are often performed offline from static guides (Jiang et al., 2026; Chen et al., 2026) or recorded trajectories (Zhang et al., 2026; Alzubi et al., 2026; Yang et al., 2026). This stands in sharp contrast to how humans solve tasks: when they encounter a problem, they do not simply stop and wait for further instructions or retry the same method repeatedly, but instead explore different menus in the application interface and search online for tips and potential solutions. These observations motivate us to explore the following research question: can skill creation pipelines produce better skills by enabling AI agents to proactively seek the information they need?

In this paper, we propose PROACTSKILL, a novel paradigm for skill creation in which the AI agent proactively assumes responsibility for information gathering, emulating a human when stuck on tasks. Specifically, upon receiving a task instruction, the agent dynamically constructs a task-specific skill by searching application documentation and online resources and screening the gathered information for relevance. When the agent gets stuck, it does not simply abort the task, but instead keeps exploring and interacting with the application, and refines the skill with its findings, rather than delegating the refinement task to a different module. To make the resulting skills more informative and actionable, we incorporate **visual information** throughout the skill creation pipeline. During online search, the agent searches YouTube videos in addition to textual resources, identifies key visual frames for individual steps and incorporates them into the skill. During skill refinement, the agent similarly collects new screenshots from its exploration of the application and adds them to the skill when necessary.

We evaluate PROACTSKILL on OSWorld tasks spanning multiple application domains. We compare against a no-skill baseline and VisualSkills, which provides pre-existing application guides but does not dynamically search web resources or explore the application to construct task-specific skills. We also compare against SkillRevise Liu et al. (2026), which refines skills from execution traces.

Our contributions are as follows:

1. A **proactive skill construction pipeline** that dynamically builds multimodal, task-specific skills from application documentation, web resources, and YouTube tutorials, with the AI agent actively gathering and screening information based on task relevance.
2. A **skill refinement pipeline** that enables the AI agent to refine the constructed skill during task execution by proactively exploring the application when it encounters failures or discovers missing information.
3. An **empirical evaluation** on OSWorld showing that skills created by PROACTSKILL substantially outperform both curated application-level skills (VisualSkill) and acting without skills. An ablation over pipeline stages shows that most of this gain comes from research-based skill construction, while refinement makes skills quicker to execute. In a controlled setting with deliberately broken skills, PROACTSKILL's exploration-based refinement repairs them about as well as trajectory-based revision (SkillRevise), without first needing a recorded failure.

2 RELATED WORK

2.1 COMPUTER-USE AGENTS AND BENCHMARKS

OSWorld (Xie et al., 2024) evaluates LLM agents on real desktop applications, while WebArena (Zhou et al., 2024) and Mind2Web (Deng et al., 2023) focus on web interaction; SkillsBench (Li et al., 2026) compares shell/file-based agents with and without curated skills. Common strategies reflect on previous attempts, distill successful trajectories, use them as in-context examples, or accumulate rules from trial and error (Shinn et al., 2023; Wang et al., 2024; Kagaya et al., 2024; Zheng et al., 2024), but typically require expensive task rollouts. We instead draw on how humans and agents consult web search and documentation while planning (Agashe et al., 2025).

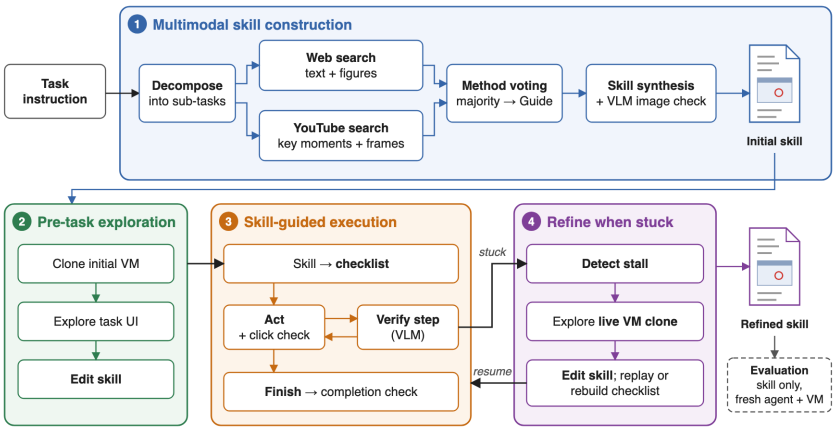

Figure 1: **Overview of PROACTSKILL.** (1) A research agent builds a multimodal skill from web and YouTube sources. (2) An exploration agent checks the skill against the real interface on a VM clone. (3) The executor follows the skill as a VLM-verified checklist. (4) When progress stalls, the agent explores a live VM clone and refines the skill. The refined skill is then evaluated on its own.

2.2 AI AGENT SKILL CREATION

Skills augment agents with knowledge the base model does not reliably capture. CUA-Skill (Chen et al., 2026) builds a library of desktop skills as parameterized execution graphs, and VisualSkill (Jiang et al., 2026) builds multimodal application-level guides from documentation and UI exploration. We adopt this guide-based representation but construct *task-level* skills containing only the relevant application knowledge. Other methods derive skills from interaction trajectories: MMSkills (Zhang et al., 2026) distills public trajectories into multimodal skill packages, EvoSkill (Alzubi et al., 2026) retains skills proposed from execution failures, Voyager (Wang et al., 2023) grows an executable skill library in Minecraft, and Agentic Proposing (Jiao et al., 2026) composes modular reasoning skills. In contrast, PROACTSKILL requires no pre-collected trajectories: it synthesizes skills on demand from human-facing web pages and YouTube tutorials and grounds them through interaction with the target application.

2.3 AI AGENT SKILL REFINEMENT

Several methods improve skills from execution feedback. SkillOpt (Yang et al., 2026) and EvoSkill (Alzubi et al., 2026) edit skills from scored rollouts or failures and keep changes that improve validation performance; Skill-R1 (Vishe et al., 2026) and SAGE (Liang et al., 2025) use reinforcement learning or reward signals over prior experience; and SkillRevise (Liu et al., 2026) diagnoses skill defects from traces, retrieves repair principles from a general memory, and keeps the best revision found across repeated re-executions within a fixed budget. SEAgent (Sun et al., 2025) learns unfamiliar software through autonomous exploration, but updates the agent policy rather than an explicit skill. PROACTSKILL instead keeps the agent fixed and refines the skill *online*, within a single episode: when progress stalls at a checklist step, it explores a clone of the live environment and edits the skill with what it observes, without rollouts, training, or reward design.

3 METHOD

PROACTSKILL equips a computer-use agent with a task-specific skill that the agent constructs itself and refines as it executes the task. Given a task instruction, PROACTSKILL proceeds in four stages (Figure 1): (1) *skill construction*, in which a research agent gathers textual and visual knowledge from the web and YouTube and synthesizes it into a multimodal skill; (2) *pre-task exploration*, in which an agent explores the target application's interface and records its findings in the skill; (3) *skill-guided execution*, in which the execution agent performs the task while the skill's steps are tracked as a checklist; and (4) *stuck-triggered exploration and refinement*, in which the agent

explores a copy of the live environment whenever progress stalls and updates the skill with what it learns. All components are driven by the same underlying model.

Skills can be reused and can be retrieved via embedding similarity to the task instruction. (Note that for our experiments, we assume that skills are freshly made for each task.) Each candidate skill is represented for matching by its name and description only (not its full content); the query instruction and every candidate document are encoded in one batched call with a shared sentence-embedding model and ranked by cosine similarity. If the top-ranked candidate exceeds a single global similarity threshold, it is used directly; otherwise the skill-creation procedure described above is triggered instead.

Agents interact with the environment and the skill pipeline exclusively through tools exposed via the Model Context Protocol (MCP): a *skill-authoring* server (`websearch`, `youtube_search`, `create_skill`, `edit_skill`) and a *desktop-control* server (GUI actions, `screenshot`, `check_progress`, `finish`). Candidate URLs are found with the agent harness's built-in web search. Checklist tracking, verification, stuck detection, and VM cloning are performed by the harness and are not exposed to the agents as tools.

3.1 Multimodal Initial Skill Creation

When no existing skill is relevant enough for the task instruction, the agent invokes a skill-creation procedure, driven by an agent MCP calls a *skill-creation meta-skill* to research and create a new skill from scratch. The created skills contain not only textual descriptions of the procedure, but also images of the user interface showing where to click or what to expect.

Skill Decomposition: The agent first determines whether the instruction needs to be decomposed into several distinct subtopics/sub-tasks, which each require separate research. It can also plan different queries that target alternative approaches for sub-tasks.

Web search: For each sub-task, the agent issues three to five queries and selects candidate pages from the web search results. A scraping tool retrieves each page, strips non-content elements, and renders the page to PDF. Candidate figures (such as embedded images, vector drawings, and tables) are extracted from the rendered page, and a vision-language model (VLM) judges to keep crops that can help complete the task. The text and figures are then summarized by the model into per-source steps, key facts, required tools, and caveats. A cross-source summary is also kept as well.

Youtube Search: We want to extract information from various good quality videos but we want to have tractable parsing: so we retrieve 15 candidate videos, discard the videos that are longer than ten minutes, and keep the five that are the most-viewed. For each video, we transcribe the audio and prompt the model to segment the transcript into five to ten key moments, and to retrieve the start time, end time, title, and summary that correspond to each moment. For each key moment, we also extract a before visual frame at its start time, and an after visual frame at its end time. A VLM checks each frame and keeps it only if it clearly shows the relevant interface and would help a viewer understand where to look or what to click. A key moment is retained if at least one of its frames passes the VLM check.

Method Voting: Since the different sources can often describe different ways of completing a task, we prompt the model to group the sources by method, and pick the method that the majority of sources use. The remaining methods are classified as alternative methods (Other Methods) for each sub-task.

Skill Synthesis: The skill is then synthesized in three calls: one writes the front matter and introduction; one writes the Guide from the winning method, placing screenshots from the retained frames at the steps they illustrate; and one writes the Other Methods section from the minority methods. Finally, a VLM verifies every image placed in the Guide against the step it is attached to, and images that do not support their step are removed. We save a snapshot of the skill at this point, which we refer to as the *initial skill*.

In addition, VisualSkill (Jiang et al., 2026) has already produced a number of general application guides. As additional context, the research agent receives the section of an existing whole-application guide whose heading best matches the instruction (by embedding similarity), which it may use to inform its research.

3.2 PRE-TASK EXPLORATION

Documentation and tutorials frequently target a different application version or omit interface details. Before the task begins, PROACTSKILL therefore runs an exploration agent on a clone of the task's virtual machine, which starts from the same initial state as the task. The agent is instructed to explore the parts of the application's interface relevant to the task (menus, toolbars, panels, and dialogs) so that their layout and controls are documented for the executor. Exploration is restricted to applications related to the task. When finished, the agent writes up what it did and observed, and an editing step integrates these findings into the skill, rewording, splitting, or removing steps that do not match the actual interface and adding alternatives to the Other Methods section. The editing prompt instructs the model to include only findings that were actually observed. The clone is then discarded, so the task itself begins from an unmodified environment.

3.3 SKILL-GUIDED EXECUTION

The execution agent receives the full text of the skill together with the task instruction. The skill is parsed into a step-by-step *checklist*, using the model to rewrite vague steps into concrete, actionable ones and to drop steps irrelevant to the task. Steps already satisfied in the initial screenshot are marked complete.

After each action, a VLM step verifier compares the current screenshot with the active checklist step and labels it `done`, `progress`, `ambiguous`, or `not_done`. The agent moves on to the next step if it is verified as done. The agent can query its checklist position at any time, and it is notified whenever the checklist advances or changes. For coordinate-based actions, a VLM first checks whether the agent's chosen point lies on the intended element (the agent names the element when clicking); if not, a zoomed crop around the point is used to re-localize the target. When the agent claims completion, a VLM completion check must confirm from the final screenshot that the task is complete; otherwise the claim is rejected and the agent continues.

3.4 TRIGGERED EXPLORATION AND REFINEMENT

Rather than aborting or repeating a failing action, PROACTSKILL responds to stalled progress by exploring. The harness monitors execution and triggers exploration when any of the following holds: (1) the agent repeats the same action twice consecutively; (2) two consecutive actions each change less than 2% of the screen's pixels; or (3) the step verifier labels the current step `not_done` twice or `ambiguous` four times. Before exploring on (3), the harness checks whether the task is already complete or whether a later checklist step is already satisfied; if so, it advances the checklist and records the executed actions in the skill instead of exploring. Since there's a max step budget present, we allow stuck actions at most two times to prevent from wasting more of the step budget for the task.

Exploration on a live clone: To explore without disturbing the task, PROACTSKILL clones the VM *in its current state*: the running guest is briefly paused, its full machine state (CPU, memory, and devices) is migrated to a file and its disk is copied, and the live VM is resumed while a new VM restores the saved state in a separate container. An exploration agent then operates the clone, with a budget of 30 actions and 20 minutes. Its prompt states the task, the active skill, and an objective: for triggers (1) and (3), it is told which action or step failed and to find a working path forward without repeating it. For trigger (2), it is told that progress has stalled and to find a way forward from the current screen. The execution agent is paused while exploration runs.

Refining the skill: When exploration finishes, the exploration agent writes up its findings, which are integrated into the skill using the same editing step as in Pre-Task Exploration. The outcome is then applied to the live task in one of two ways. If the stalled step is verified as complete on the clone's final screenshot, the exploration agent's actions are replayed on the live VM and the checklist advances. Otherwise, if the skill was edited, the checklist is rebuilt from the refined skill and the execution agent is notified of the update. Refinement therefore benefits the remainder of the current episode, and the refined skill is kept as the output of PROACTSKILL for the task.

3.5 POST REFINEMENT EVALUATION

The execution harness above bundles the skill with exploration, checklist tracking, and verification. To measure the value of the skill itself, after all of these mechanisms, we evaluate each skill post refinement to see how well the refined skill performs: a fresh agent is given the skill and task in a new virtual machine initialized from the task's starting state, and executes the skill via a MCP call. Coordinates are only rescaled to the native resolution, and the agent's own `finish` call ends the episode.

4 EXPERIMENTS

We design our experiments to answer three questions: **(RQ1)** Do skills created by PROACTSKILL improve agent performance over the existing multimodal skills creation method (VisualSkill) and over acting without skills? **(RQ2)** Which stages of the pipeline account for these gains, and what does each stage cost? **(RQ3)** When a skill contains errors, can PROACTSKILL's exploration-based refinement repair it?

4.1 EXPERIMENTAL SETUP:

We use OSWorld (Xie et al., 2024) and we focus on the four application domains for which curated multimodal skills are available from VisualSkill (Jiang et al., 2026): GIMP, LibreOffice Calc, LibreOffice Impress, and LibreOffice Writer. Because each episode is expensive, we evaluate on a fixed random subsample of 10 tasks per domain (40 tasks in total). All methods run in the same OSWorld Docker environment (Ubuntu, 1920×1080). Agents observe screenshots downscaled to 1280×720, receive no accessibility tree, and act through the same set of `pyautogui`-backed GUI tools exposed via MCP. Every episode is limited to a max of 30 GUI actions (screenshots do not count) and 3 hours of wall-clock time. Success is measured by each task's OSWorld evaluator on the final environment state. Every agent and every auxiliary model call in every condition uses Claude Opus 5. We run three independent trials of each condition and report the mean and standard error of the mean of the success rate.

4.2 SKILL QUALITY AND PIPELINE ABLATION (RQ1, RQ2)

We compare the following:

- **No skill.** The agent receives only the task instruction. It runs in our execution environment with all harness assistance disabled: no checklist tracking, coordinate verification, completion check, or exploration.
- **VisualSkill** (Jiang et al., 2026). The agent receives the curated, whole-application multimodal skill for the task's domain (the stage-2 skill that was created with VisualSkill). The agent loads topic guides on demand through a `load_topic` tool that returns each guide's text with its figures interleaved. We use VisualSkill's own agent loop and prompt without harness assistance; our only changes are an action counter enforcing the same 30-action budget.
- **PROACTSKILL and its stages.** We run the full pipeline for every task: skill construction from the web and YouTube, pre-task exploration, and harness-assisted execution with stuck-triggered exploration and refinement. No skill is reused across tasks or trials; each skill is built from scratch within the run. To attribute gains to individual stages, we measure performance at three points. *Initial skill:* the skill right after construction, before any interaction with the application; *Exploration:* the harness-assisted execution episode itself, during which the skill is refined; *Refined skill* (reported as PROACTSKILL (full or post refinement)): the final skill produced after all refinement.

We evaluate the initial and refined skills in the same way as the baselines: a fresh agent receives the skill and the task in a new environment, without harness assistance. Differences among these conditions therefore reflect only the skill the agent receives. PROACTSKILL's skills are task-specific, whereas VisualSkill provides one general skill per application.

4.3 REFINING CORRUPTED SKILLS (RQ3)

Initial skills from PROACTSKILL are often already strong, which limits how much the main benchmark can reveal about refinement itself. We therefore perform a smaller scale evaluation to evaluate refinement in a controlled setting where each skill contains injected errors known to cause failure.

Corruption pipeline: For each candidate task, we first construct a skill with PROACTSKILL's skill creation stage. We then inject an error into one step of the skill's Guide, chosen uniformly at random from all steps except the first. With probability $2/3$ we apply a *value corruption*, which alters a concrete detail in the step using the first applicable rule: (i) shifting a cell reference by one or two rows (e.g., `B3`→`B4`); (ii) replacing the letter in a `Ctrl+`*letter* shortcut; (iii) swapping a word for its opposite (e.g., *left*/*right*, *open*/*close*, *show*/*hide*, *first*/*last*); or (iv) rotating the characters of a quoted value (e.g., "Bold"→"oldB"). If no rule applies, and otherwise with probability $1/3$, we *delete* the step entirely and renumber the remaining steps. Because an agent may recover from an error on its own, we keep only corruptions that actually break the skill. A fresh, unassisted agent attempts the task with the corrupted skill. If it succeeds, we corrupt an additional step and try again, up to four corrupted steps in total. We repeat this procedure independently in each trial and keep only tasks whose skills are confirmed broken in both trials.

Conditions. Both methods start from the same confirmed-broken skill in each trial.

- **PROACTSKILL.** The corrupted skill is used as the initial skill of the full method: pre-task exploration, and the skill goes through one single round of live stuck-triggered exploration and refinement phase.
- **SkillRevise** (Liu et al., 2026), adapted to GUI skills. From the recorded failed attempt, the method diagnoses which skill steps caused the failure, retrieves relevant repair principles, and rewrites the skill in a single LLM call. We run one revision round.

The two methods draw on different information: SkillRevise sees the text trajectory of one failed attempt, while PROACTSKILL interacts with copies of the live application. We evaluate each repaired skill with the same fresh, unassisted agent. Because every skill fails before repair by construction, the success rate after repair directly measures how well each method fixes the skill.

5 RESULTS

We evaluate PROACTSKILL on 40 OSWorld tasks spanning GIMP and three LibreOffice applications (Calc, Impress, Writer) with 3 trials per task. We compare against two baselines. *VisualSkill* (Jiang et al., 2026) runs the original agent unmodified with its pre-authored multimodal skill library, which was built offline through documentation mining and manual UI exploration rather than generated by the agent at test time. Because this library covers only GIMP and LibreOffice, we restrict the comparison to these domains. *No-skills* runs the same base agent with no external procedural guidance. Unless otherwise noted, we report standard error of the mean. All conditions use Claude Opus 5. All experiments are run with resources: 32 VCPUs and 256 GB of memory.

Main Results: Table 1 reports accuracy by domain and overall. The two baselines are statistically indistinguishable (55.2% vs. 53.9%; $p = 0.821$), suggesting that the skill library developed by

Table 1: Accuracy by Domain and Overall on OSWorld (3 trials each). PROACTSKILL outperforms baselines VisualSkill and No-skills.

Condition	GIMP ($N=10$)	LO Calc ($N=10$)	LO Impress ($N=10$)	LO Writer ($N=10$)	Overall ($N=40$)
VisualSkill	3.3 ± 3.3	53.3 ± 12.4	64.0 ± 10.9	100.0 ± 0.0	55.2 ± 6.9
No-skills	10.0 ± 7.1	60.0 ± 14.7	62.4 ± 14.2	83.3 ± 11.4	53.9 ± 7.3
PROACTSKILL (initial skill only)	56.7 ± 15.8	90.0 ± 10.0	82.4 ± 11.3	96.5 ± 3.4	81.4 ± 5.8
PROACTSKILL (post refinement)	50.0 ± 15.1	86.7 ± 10.2	85.7 ± 10.1	99.9 ± 0.1	80.6 ± 5.8

Table 2: Average Cost (USD) by domain on OSWorld ($N = 10$ tasks per domain, 40 overall, 3 trials each).

Condition	GIMP	Calc	Impress	Writer	Overall
VisualSkill	1.24 ± 0.27	1.82 ± 0.38	2.49 ± 0.57	2.32 ± 0.58	1.97 ± 0.24
No-skills	0.039 ± 0.009	0.034 ± 0.006	0.027 ± 0.002	0.025 ± 0.002	0.031 ± 0.003
PROACTSKILL (initial skill)	0.018 ± 0.002	0.018 ± 0.001	0.021 ± 0.001	0.022 ± 0.002	0.020 ± 0.001
PROACTSKILL (refined skill)	0.017 ± 0.002	0.018 ± 0.001	0.022 ± 0.001	0.021 ± 0.002	0.020 ± 0.001

Table 3: Time (minutes) by Domain and Overall on OSWorld on 40 tasks, each three trials. PROACTSKILL (post-refinement) skill execution outperforms both baselines.

Condition	GIMP ($N = 10$)	LO Calc ($N = 10$)	LO Impress ($N = 10$)	LO Writer ($N = 10$)	Overall ($N = 40$)
VisualSkill	9.0 ± 1.6	12.2 ± 2.1	15.6 ± 3.2	10.5 ± 1.5	11.8 ± 1.1
No-skills	50.7 ± 8.1	60.4 ± 4.9	43.7 ± 5.7	29.6 ± 3.0	46.1 ± 3.3
PROACTSKILL (initial skill)	6.8 ± 1.9	8.0 ± 1.2	16.1 ± 5.4	9.5 ± 1.6	10.1 ± 1.6
PROACTSKILL (post refinement)	6.1 ± 1.5	7.7 ± 1.1	10.4 ± 1.5	8.6 ± 2.2	8.2 ± 0.8

VisualSkill offers little benefit over no skills on this task distribution. On GIMP, VisualSkill even underperforms the no-skills agent (3.3% vs. 10.0%). PROACTSKILL reaches 80.6% $\pm$ 5.8 overall, significantly outperforming both baselines ($p < 0.001$ for each comparison). The largest gains are on GIMP (50.0% vs. at most 10.0%) and Calc (86.7% vs. at most 60.0%), while Writer is near ceiling for all non-exploration conditions.

Ablation over pipeline phases: To understand where these gains come from, we evaluate the skill at each stage of the pipeline: after initial creation, during exploration, and after the final refinement (which we report as PROACTSKILL (full)). Most of the accuracy improvement is attained by the initial skill, which is synthesized from online sources. Accuracy drops during exploration (71.0% $\pm$ 6.3%), and costs averaging \$7.37 $\pm$ 0.53, which is consistent with the agent spending part of its budget on probing edge cases, recovering in cloned environments rather than completing the task directly, and extensive verification checks of steps. The final refinement recovers this drop, returning accuracy to the level of the initial skill.

Time: Refinement makes the skill faster to execute: mean run time falls from 10.1 to 8.2 minutes (Table 3). Relative to the baselines, executing a PROACTSKILL completed refined skill is about $4.5\times$ faster than acting without skills (8.2 vs. 46.1 minutes). These savings come at a one-time cost of building the skill: exploration takes 124 $\pm$9 minutes (due to copying multiple clone copies), an up-front investment that is amortized over subsequent executions of the skill if needed in the future, and does not rely on datasets of trajectory dataset beforehand. Note that tasks were also executed in parallel to speed up experiments, this may have an effect on wall clock times.

Table 4: Refining corrupted skills on OSWorld. Both methods start from the same confirmed-broken skills (initial skill score is 0). Each repaired skill is executed by a fresh agent without harness assistance. Cost and time cover only the refined skill execution.

	PROACTSKILL			SkillRevise		
Domain	Acc. (%)	Cost (\$)	Time (min)	Acc. (%)	Cost (\$)	Time (min)
GIMP ($N = 6$)	25.0 ± 17.1	0.0153 ± 0.0037	3.6 ± 2.1	33.3 ± 21.1	0.0170 ± 0.0033	4.0 ± 2.6
LO Calc ($N = 3$)	66.7 ± 33.3	0.0222 ± 0.0102	9.1 ± 6.4	66.7 ± 33.3	0.0207 ± 0.0068	8.1 ± 5.5
LO Impress ($N = 3$)	33.3 ± 33.3	0.0233 ± 0.0036	10.4 ± 1.3	0.0 ± 0.0	0.0279 ± 0.0043	8.7 ± 2.3
Overall ($N = 12$)	37.5 ± 13.9	0.0190 ± 0.0031	6.7 ± 1.9	33.3 ± 14.2	0.0207 ± 0.0027	6.2 ± 1.9

Comparison with SkillRevise: To study refinement in isolation from skill creation, we compare the runs for PROACTSKILL's completed refined skills against refined skills from a single round of SkillRevise, both refined from the same corrupted skills on 12 tasks over two trials. As shown in Table 4, the two methods reach comparable accuracy (37.5% vs. 33.3%) and run time (6.7 vs. 6.2 minutes), and comparable cost. See paragraph below on Cost Comparison about how these costs are computed.

Cost Comparison Computation: Table 2 reports each condition's per-task cost using whatever measurement its harness's logs actually support. VisualSkill costs are the harness's own self-reported per-task total, which we independently verified against real per-token pricing on a sample of tasks. No-skills costs measure the input/cache-read portion of every turn directly from session logs, then add an estimate of the output-token cost from the character length of each generated turn, priced at the model's output rate. **PROACTSKILL**'s exploration costs are reconstructed directly from session-log token counts, re-priced per model and per call, plus a bounded estimate ($\pm 8\%$) for helper-model calls not directly logged. Where available, a billing-provider-reported spend match takes precedence over all of the above as the most reliable source.

Table 4 compares PROACTSKILL against SkillRevise, so we restrict the comparison to what the two methods actually share: a single clean re-execution of whichever fixed skill each method produced, with no skill-repair cost included on either side. For apples to apples comparison: PROACTSKILL's figure is its post-refinement-phase cost alone (computed as above), not its full production total. SkillRevise figures is computed the same way as No-skills': input/cache-read cost measured directly, output cost estimated from character length.

6 CONCLUSION

Limitations and future work: Our evaluation focuses on GUI-based computer-use agents in desktop applications. Extending PROACTSKILL to web and multi-application workflows, and evaluating it with other underlying models and other types of benchmarks, is an important next step. Exploration is slow due to creating full copies of Docker clone containers. This cost is paid once and can be spread across later reuses of the skill. However, for a fair comparison with the baselines, we build every skill from scratch in our experiments. Future work could evaluate a persistent skill library, measuring how well retrieved skills carry over to related tasks and how quickly the up-front cost pays for itself. Finally, stuck-triggered exploration assumes that the running environment can be cloned. This lets the agent try actions without irreversible effects on real task execution.

We introduced PROACTSKILL, a skill-creation and refinement pipeline. Instead of relying on static, human-written guides or large collections of pre-recorded trajectories, PROACTSKILL lets the agent research a task the way a person would. It searches documentation, web pages, and video tutorials, and turns what it finds into a multimodal, task-specific skill grounded in visual references. When the skill falls short during execution, the agent does not abort or repeat the same failing action. Instead, it explores a copy of the live application and writes what it learns back into the skill, so each episode leaves behind a more efficient skill. Across several desktop applications, skills created this way substantially outperform both curated application guides and acting without skills. Together, these findings suggest that treating the agent as an active researcher and explorer, rather than a passive consumer of skills, is a promising direction for building capable computer-use agents and for AI skill acquisition.

AI USE STATEMENT

In this work, we used generative AI tools in experiments as this is work on AI agent skills that call LLM/VLM tools and MCP calls, creating a figure flowchart and for help with formatting Latex tables, double checking for any missed relevant works in our literature review, creating copies of bash scripts for sending in experiment runs to clusters, and checking writing, helped with code cleaning/finding implementation details in the baselines. We have not used generative AI tools for proposing or refining hypotheses, formulating mathematical claims, generating synthetic datasets, or assisting in writing proofs, or generated references with AI. We have reviewed all AI assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work aims to make computer-use agents more reliable by giving them more complete and accurate skills. It does not involve human subjects, personal data, or crowdsourced annotation. All experiments use the publicly available OSWorld benchmark in sandboxed virtual machines. We note three considerations. Skills are built from untrusted online sources, they may carry incorrect or deliberately malicious instructions, such as prompt-injection attacks, into the agent's behavior. Our pipeline filters sources for relevance and checks retrieved images, but it is not a defense against adversarial content, and deployed systems should combine it with content filtering and human review of the resulting skills. Computer-use agents that act autonomously can cause unintended side effects. PROACTSKILL performs all exploration on cloned copies of the environment, so trial-and-error actions never affect the actual task. Real-world deployments, however, may involve irreversible actions or external services that cannot be cloned, and should keep humans in the loop for such actions. As with other agent automation research, improving agent capability could also be misused; we do not believe PROACTSKILL meaningfully increases this risk beyond that of existing computer-use agents.

REPRODUCIBILITY STATEMENT

The benchmarks, number of tasks, and trials, and which base model is used has been included in the main paper, standard error of the mean has been marked in the tables, and code will be submitted with the submission.

REFERENCES

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Wang. Agent s: An open agentic framework that uses computers like a human. In *International Conference on Learning Representations*, volume 2025, pp. 22924–22946, 2025.

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. URL `https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills`.

Tianyi Chen, Yinheng Li, Michael Solodko, Sen Wang, Nan Jiang, Tingyuan Cui, Junheng Hao, Jongwoo Ko, Sara Abdali, Leon Xu, et al. Cua-skill: Develop skills for computer using agent. *arXiv preprint arXiv:2601.21123*, 2026.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Ziyan Jiang, Li An, Yujian Liu, Jiabao Ji, Qiucheng Wu, Jacob Andreas, Yang Zhang, and Shiyu Chang. Visualskill: Multimodal skills for computer-use agents. *arXiv preprint arXiv:2606.18448*, 2026.

Zhengbo Jiao, Shaobo Wang, Zifan Zhang, Xuan Ren, Wei Wang, Bing Zhao, Hu Wei, and Linfeng Zhang. Agentic proposing: Enhancing large language model reasoning via compositional skill synthesis. *arXiv preprint arXiv:2602.03279*, 2026.

Tomoyuki Kagaya, Thong Jing Yuan, Yuxuan Lou, Jayashree Karlekar, Sugiri Pranata, Akira Kinose, Koki Oguri, Felix Wick, and Yang You. Rap: Retrieval-augmented planning with contextual memory for multimodal llm agents. *arXiv preprint arXiv:2402.03610*, 2024.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026.

Xuechen Liang, Meiling Tao, Yinghui Xia, Jianhui Wang, Kun Li, Yijin Wang, Yangfan He, Jingsong Yang, Tianyu Shi, Yuantao Wang, et al. Sage: Self-evolving agents with reflective and memory-augmented abilities. *Neurocomputing*, 647:130470, 2025.

Yuxuan Liu, Zhaochen Su, Lingyun Xie, Yuhao Zhang, Qing Zong, Jiahe Guo, Zhongwei Xie, Yiyan Ji, Yauwai Yim, Hongyu Luo, et al. Skillrevise: Improving llm-authored agent skills via trace-conditioned skill revision. *arXiv preprint arXiv:2606.01139*, 2026.

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.

Zeyi Sun, Ziyu Liu, Yuhang Zang, Yuhang Cao, Xiaoyi Dong, Tong Wu, Dahua Lin, and Jiaqi Wang. Seagent: Self-evolving computer use agent with autonomous learning from experience. *arXiv preprint arXiv:2508.04700*, 2025.

Yash Vishe, Rohan Surana, Xunyi Jiang, Zihan Huang, Xintong Li, Nikki Lijing Kuang, Tong Yu, Ryan A Rossi, Jingbo Shang, Julian McAuley, et al. Skill-r1: Agent skill evolution via reinforcement learning. *arXiv preprint arXiv:2605.09359*, 2026.

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

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

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

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.

Kangning Zhang, Shuai Shao, Qingyao Li, Jianghao Lin, Lingyue Fu, Shijian Wang, Wenxiang Jiao, Yuan Lu, Weiwen Liu, Weinan Zhang, et al. Mmskills: Towards multimodal skills for general visual agents. *arXiv preprint arXiv:2605.13527*, 2026.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *International Conference on Learning Representations*, volume 2024, pp. 19036–19066, 2024.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

A APPENDIX

This appendix reproduces the exact system prompts and tool/action-space definitions used by the agents evaluated in this paper, for reproducibility. All conditions in the main comparisons (Proact-Skill, SkillRevise-on-Corrupt, and the no-skill baseline) share the same execution substrate and tool schema (Sections A.1–A.2); they differ only in what skill guidance, if any, is injected into the prompt and in how that guidance is authored/repaired (Sections A.4–A.5). The vendored, originally-released VisualSkills baseline uses its own separate prompt and tool schema (Section A.7), included for transparency since it is not our own implementation.

A.1 MAIN TASK-EXECUTION AGENT SYSTEM PROMPT

The executor agent is a Claude-Code-CLI process controlling the VM exclusively through MCP tools (Section A.2); it has no shell, file-edit, or code-execution access. The system prompt below is followed by the active skill's guide text (if any) and the task instruction.

> You are utilising an Ubuntu virtual machine to complete a task autonomously.
>
> You MUST interact with applications through their GUI only. Do NOT open a terminal, run shell commands, edit config files, or use any programmatic workaround – your tools do not include Bash/Read/Write/Edit for exactly this reason.
>
> You have the following MCP tools to interact with the desktop:
>
> - screenshot(): take a screenshot to see the current desktop state.
> - click(x, y, button, description): click at pixel coordinates (”left”/”right”/”middle”). ALWAYS pass description – a short name of the element you're clicking (e.g. ”the Save button”, ”Chrome's three-dot menu”). If this exact click ever needs automatic retry, that description is the ONLY way the recovery system knows what you were aiming for instead of guessing blind.
> - double_click(x, y, description): double-click at coordinates (same description guidance as click()).
> - move_to(x, y): move the mouse without clicking.
> - drag_to(start_x, start_y, end_x, end_y): click and drag.
> - scroll(x, y, clicks): scroll up (positive) or down (negative).
> - type_text(text): type text.
> - key_press(key): press a single key (e.g. ”enter”, ”tab”, ”escape”).
> - hotkey(keys): press a shortcut (e.g. [”ctrl”, ”c”], [”alt”, ”f4”]).
> - check_progress(): check your position in the task's skill checklist and any automatic-recovery findings – call this whenever unsure what to do next.
> - finish(status, summary): the ONLY way to end the run – ”done” once the task's target state is visibly achieved, ”fail” if it cannot be completed. You MUST call this; do not just stop calling tools.
>
> Every action tool automatically returns a screenshot after execution. After each action, carefully evaluate the returned screenshot to confirm you achieved the right outcome before moving on. If a ”done” finish() claim is rejected, keep working and call finish() again once the actual issue named in the rejection is resolved.

The active skill guide (when present) and the task instruction are then appended:

> Relevant skill guide for this task:
> ¡skill guide text¿
> Task: ¡instruction¿

A.2 TOOL / ACTION-SPACE DEFINITION

The executor's complete action space, exposed as MCP tools (matching the descriptions in the system prompt):

- **screenshot()** – returns an image of the current desktop.
- **click(x, y, button, description)** – left/right/middle click at pixel coordinates.
- **double_click(x, y, description)**
- **move_to(x, y)** – move the mouse without clicking.
- **drag_to(start_x, start_y, end_x, end_y)** – implemented as move-to-source then a timed drag.
- **scroll(x, y, clicks)** – positive scrolls up, negative scrolls down.
- **type_text(text)**
- **key_press(key)** – e.g., *enter*, *tab*, *esc*, *backspace*, *delete*, arrow keys, *home*/*end*, *pageup*/*pagedown*, *f1–f12*, *space*.
- **hotkey(keys)** – e.g., [*ctrl*, *c*] or [*alt*, *f4*].
- **check_progress()** – returns the agent's current checklist position and any background stuck-recovery findings.
- **finish(status, summary)** – *status* is either *done* or *fail*; the only way to end a run. Gated by a save-confirmation check and (outside the plain condition) an LLM completion judge (Section A.3).

Every action is executed via *pyautogui* inside the VM; click coordinates are given in a fixed model-facing coordinate space and rescaled server-side to the VM's true screen resolution. This action space is identical across all GUI domains evaluated (GIMP, LibreOffice Calc, LibreOffice Impress) – there is no per-domain tool allow-list; only the injected skill guidance differs by domain and condition.

The executor CLI process is launched with the following disallowed built-in tools, so that it can *only* act through the MCP tools above:

> *--disallowed-tools Bash, Edit, Write, Glob, Grep, Agent, AskUserQuestion, NotebookEdit*

(run with *--dangerously-skip-permissions*, since the process is sandboxed inside a disposable VM with no host filesystem access).

A.3 Completion-Judge (Verification) Prompt

A *finish("done", ...)* claim is never trusted at face value. Outside the no-skill/plain condition, an independent LLM call judges the live screenshot against the full task instruction before the run is allowed to end:

> Task: {instruction}
> Actions taken so far: {history_text}
> The agent just claimed the task is DONE. Look at the attached screenshot of the live VM's CURRENT state. Is the FULL task instruction genuinely and completely satisfied right now – not just attempted?
> If the instruction names a specific target (a filename, exact value, specific setting, particular piece of text/content), you must be able to point to concrete visual evidence of THAT SPECIFIC target in the screenshot – do not answer YES just because something generically similar is present or because the action history sounds plausible. If you cannot confirm it from what's visible, answer NO. Answer with exactly one word: YES or NO.

A rejected claim returns the following message to the agent, which must continue acting before calling *finish* again:

> Your completion claim was rejected: the current screenshot does not yet show the task fully complete. Do not call finish("done", ...) again until the specific target/state the instruction asks for is visibly confirmed.

A separate, non-LLM hard gate additionally blocks *finish("done", ...)* when the active skill's checklist includes a save step but no save keystroke (*Ctrl+S*/*Cmd+S*) appears in the action history.

A.4 PROACTSKILL: SKILL-REFINEMENT PROMPT (OURS)

Our method's skill checklist is revised after a live rollout using observations logged during execution (free exploration, mid-task stuck-recovery probes, and confirmed misclicks). The revision call:

> You are refining an existing SKILL.md based on new observations from actually running the skill on a live VM. Domain: {domain}
>
> The SKILL.md below uses a lightweight format: frontmatter, then a "## Guide" section with numbered steps (optionally followed by a "![Step N](stepNN.png)" image line and a short sentence stating when/why to look at that image – e.g. what UI state it shows and what to check it against), then an "## Other / Possible Methods" section listing condensed alternative approaches. Preserve this format exactly – do not reintroduce Overview/Preconditions/state-card/Test-Cases sections.
>
> == EXISTING SKILL ==
> {existing_skill}
>
> {exploration_summary}
> {new_images_block}
>
> Update the SKILL.md to:
>
> 1. Incorporate any useful UI element observations from the Free Explorer notes directly into the relevant numbered step's text.
> 2. For each Mid-Task Explorer block: it was probed live partway through an actual run of this skill, at a specific step, because progress had stalled there. Its notes describe real UI elements/behavior encountered at that point in the procedure. Fold anything useful into the matching (or nearest) numbered step's text – a corrected description of what to look for or click – rather than leaving it as a generic note; these are concrete findings about what the skill needs at that point, not speculation.
> 3. Fix or clarify any step where the agent got stuck (trigger: repeated_action or fail) – reword the step, split it into two steps if it was doing too much at once, or remove it if it was simply wrong.
> 4. If a resolution was found during exploration for a stuck step, fold it into that step's instructions directly.
> 5. If the exploration surfaced a genuinely different approach worth keeping as a fallback, add or update an entry under "## Other / Possible Methods" instead of blending it into the main numbered guide.
> 6. Keep all steps that worked – do not remove or weaken them.
> 7. Keep any existing "![Step N](stepNN.png)" image lines (and their when/why-to-look-at-it sentence) attached to the same step they were originally under, renumbering only if you insert/remove a step. If a Mid-Task Explorer note reveals what an existing screenshot actually shows (or should show), fold that into its cue sentence too, not just the step text.
> 8. If NEW SCREENSHOTS are listed above, attach each one to whichever numbered step it's actually relevant to (matching by its region/resolved-step description), using the exact filename given, with the same "![Step N](filename.png)" + when/why-to-look-at-it sentence convention as existing images. These are real, verified screenshots – prefer them over inventing a description from text alone where they apply. Do not attach a new screenshot to a step it doesn't actually relate to just because it's available.
> 9. Any NEW SCREENSHOT labeled "CORRECT ICON for ... – the agent misclicked here this run" is a small cropped image of the exact UI element that step needs, taken because a live run actually misclicked there. Attach it to that exact step and rewrite the step's text to explicitly name the earlier mistake and the correct target – e.g. "Click the icon shown below (NOT the

similarly-placed X the agent mistakenly clicked here before)" – so a future run of this skill doesn't repeat the same misclick. This step-level warning matters more than for other screenshots, since it documents a CONFIRMED live failure, not just a UI observation.

Output ONLY the updated SKILL.md content, preserving the exact frontmatter format:

—
name: ...
description: ...
—

When a step remains stuck after refinement, a separate skill-quality critic decides how to repair it (rewrite / split / remove / fetch new research / switch to a recorded alternative method):

You are a skill quality critic for a desktop automation agent.

Task: "{instruction}"

The agent got stuck executing this skill step: "{step_text}"

Diagnosis: {diagnosis}

Actions tried (all failed): {attempts_str}

Decide what to do with this step. Output JSON only – no prose.

- If the step describes the wrong UI or wrong procedure (a web search would fix it): {"action": "web_search", "query": "¡specific query¿", "reason": "¡why wrong¿"}
- If the step is correct but too vague to act on: {"action": "rewrite", "new_step": "¡clearer ¡=20-word version¿", "reason": "¡what was unclear¿"}
- If the step conflates two distinct actions: {"action": "split", "steps": ["¡step 1¿", "¡step 2¿"], "reason": "¡why split¿"}
- If the step is misleading or irrelevant to the task: {"action": "remove", "reason": "¡why remove¿"}
- If the CURRENT METHOD itself (not just this one step) looks fundamentally wrong or unworkable for this app/UI, and a recorded alternative method looks more promising, switch to it instead of patching this step: {"action": "switch_method", "method_name": "¡exact name¿", "reason": "¡why¿"}

Prefer the smallest fix that could work (rewrite/split) over remove, and prefer remove over switch_method unless the whole current method – not just this step – is the problem.

You are NOT shown a screenshot here – you have only the text above. If a rewrite/split needs a specific value (a cell reference, filename, label, number) that you cannot actually confirm from the text given, do NOT invent/guess a literal value and write it into new_step/steps as if it were fact. Instead phrase the step so the agent reads that value off the CURRENT screen itself before acting.

A.5 SkillRevise Baseline: Diagnosis, Principles, and Revision Prompt

SkillRevise-on-Corrupt is our native reimplementation of the cited SkillRevise procedure (diagnose → retrieve principles → revise → utility-gate), used as a baseline skill-repair strategy against which ProactSkill is compared.

A.5.1 Diagnosis (rule-based)

Failure diagnosis is produced deterministically from execution artifacts (steps reached, which step(s) the agent got stuck on, outcome, exception text), not by an LLM call. Representative attribution templates:

- "Task succeeded – no failure to attribute."

- "Round crashed with {exception} before completion – likely an environment/harness fault, not a defect in the skill's instructions (reached step {reached}/{total_steps} first)."
- "The agent repeatedly failed the same checklist step(s) without visible progress: {stuck_list}. Reached {reached}/{total_steps} steps before getting stuck – the skill's instruction for this step is likely too vague, references the wrong UI element, or omits a precondition the agent needs to satisfy first."
- "Run timed out having reached only {reached}/{total_steps} checklist steps, with no single step flagged as repeatedly stuck – the skill's steps may be too coarse (each one bundling multiple real actions) or missing intermediate steps."
- "Agent explicitly signalled FAIL after reaching {reached}/{total_steps} steps – the skill likely assumes a precondition or app state that isn't actually true at task start."
- "Task did not verifiably succeed (outcome={outcome}, score={score}) despite reaching {reached}/{total_steps} checklist steps – the final step(s) likely don't actually produce/confirm the state the evaluator checks for."

A.5.2 Principle Bank

A fixed bank of seven repair principles is retrieved by keyword overlap with the diagnosis (top-3) and injected into the revision prompt:

1. **verifiable-confirm**: Make confirm/save/submit actions verifiable: after any action meant to finalize a change, the step should describe what should visibly change on screen afterward, not just the action itself.
2. **preserve-working-steps**: Do not reorder, merge, or delete a checklist step the agent already reached and passed in a prior attempt – only edit steps at or after the point of failure.
3. **check-before-acting**: Add an explicit check of the current UI state before an action that depends on a prior action's effect – do not assume the previous step succeeded just because no error was shown.
4. **name-concrete-target**: Replace vague step language with the concrete, named UI element the agent should look for.
5. **split-bundled-steps**: If a single checklist step actually requires several distinct clicks/keystrokes, split it into separate steps.
6. **state-precondition**: If a step assumes a particular starting state, state that precondition explicitly rather than leaving it implicit.
7. **terminal-state-check**: The last checklist step should describe the specific final visual state that proves task completion.

A.5.3 Revision Operator Prompt

You are revising a SKILL.md procedural guide for a desktop-GUI automation agent, based on evidence from one real execution attempt that did not fully succeed.
— CURRENT SKILL (SKILL.md) —
{current_skill_text}
— END CURRENT SKILL —
— VERIFICATION SPEC (what success requires) —
{diagnosis.verification_spec}
— END VERIFICATION SPEC —
— FAILURE ATTRIBUTION (what went wrong this attempt) —
{diagnosis.failure_attribution}
— END FAILURE ATTRIBUTION —
— STEPS TO PRESERVE (already reached/working – do not reorder or remove) —
{preserved_block}
— END STEPS TO PRESERVE —

> — REPAIR PRINCIPLES (general guidance retrieved for this failure) —
> {principle_block}
> — END REPAIR PRINCIPLES —
> Revise the skill to address the failure attribution, guided by the repair principles, while keeping every step listed under "STEPS TO PRESERVE" intact and in the same relative order. Make concrete, execution-anchored edits – name specific UI elements, add missing preconditions or intermediate steps, split bundled steps, or clarify vague ones – rather than vague advice. Keep the same SKILL.md structure (frontmatter, ## Procedures with ### method / #### state cards, any ## Other Methods section) as the current skill.
> Respond with ONLY the complete revised SKILL.md content, nothing else – no commentary before or after it.

where *verification_spec* is constructed as:

> Task instruction: {instruction}
> Evaluator: {evaluator_description}
> This round's observed outcome: {outcome} (raw score: {score_raw})

A.6 Fault-Injection ("Corrupted") Condition

The corrupted-skill conditions used in Tables 4 (and elsewhere) are produced by an algorithmic, non-LLM mutation of one numbered guide step in a freshly authored SKILL.md, seeded per task/domain for reproducibility. Two mutation strategies are used:

- **corrupt_value**: a targeted regex-based edit – swapping a cell reference (e.g. B2→B3), a hotkey (Ctrl+X→Ctrl+Z), an opposite-word pair (left/right, save/delete, open/close, enable/disable, show/hide, add/remove, increase/decrease, first/last, top/bottom, before/after), or scrambling a quoted value.
- **delete_step**: removing the step (and its image caption) entirely and renumbering the remaining steps.

If a single corruption does not measurably break a validation run, an escalation procedure applies additional rounds of corruption until it does. Every corruption round is logged (strategy, step number, before/after text, seed, round index) for provenance.

A.7 Baseline: Original VisualSkills Agent (Vendored, for Comparison)

For the baselines-vs-ours comparisons, we also report the originally released VisualSkills agent, run via its own unmodified prompt and tool server (vendored, not our implementation):

> You are utilising an Ubuntu virtual machine to complete a task autonomously. The screen resolution is 1280x720.
> You MUST interact with applications through their GUI only. Do NOT open a terminal, run shell commands, edit config files, or use any programmatic workaround.
> You have the following MCP tools to interact with the desktop:
> - screenshot(): take a screenshot to see the current desktop state.
> - click(x, y, button): click at pixel coordinates ("left"/"right"/"middle").
> - double_click(x, y): double-click at coordinates.
> - move_to(x, y): move the mouse without clicking.
> - drag_to(start_x, start_y, end_x, end_y): click and drag.
> - scroll(x, y, clicks): scroll up (positive) or down (negative).
> - type_text(text): type text.
> - key_press(key): press a single key (e.g. "Return", "Tab", "Escape").
> - hotkey(keys): press a shortcut (e.g. ["ctrl", "c"], ["alt", "F4"]).

> Every action tool automatically returns a screenshot after execution.
> After each action, carefully evaluate the returned screenshot to confirm you achieved the right outcome before moving on. If not correct, try again.
> When you believe the task is complete, simply stop – do not call any signal tool.

This baseline's tools are identical in name to ours (Section A.2) but omit the *description* parameter on *click*/*double_click* and have no *check_progress*/*finish* tools – it signals completion only implicitly, by stopping, rather than through an explicit, judge-gated *finish("done", ...)* call. Its disallowed-tools list is identical to ours (Bash, Edit, Write, Glob, Grep, Agent, AskUserQuestion, NotebookEdit).

A.8 Skill-Authoring Prompts: Task Decomposition and Method Voting

Before a new skill guide is written from web/video research, two LLM calls shape what gets researched and which of possibly several recovered methods becomes the skill's primary procedure.

A.8.1 Decomposition Prompt

Decides whether an instruction is one atomic action or several distinct sub-tasks each needing their own research, and separately flags topics that are ambiguous about *method* (several genuinely different ways to accomplish the same goal) so that candidate approaches can be researched in parallel:

> You are deciding whether a desktop-automation task instruction describes ONE atomic action or ACTUALLY 2-4 distinct sub-tasks that would each need their own dedicated research (different app, different menu area, or a genuinely separate skill someone would look up on its own).
> Domain: {domain}
> Instruction: "{instruction}"
> Separately from that, for EACH resulting topic (whether atomic or a sub-task), decide if the task is AMBIGUOUS ABOUT METHOD – i.e. the instruction names a goal but not which specific tool/menu/technique accomplishes it, so multiple genuinely different approaches are plausible (e.g. "convert PNG to SVG in GIMP" could mean GIMP's Trace Bitmap, GIMP's Export Path, or a GIMP-to-Inkscape hop – these are different METHODS, not just different search phrasings of one method). If so, list 2-3 candidate approaches as their own short search queries; otherwise leave the list empty.
> If the instruction is one atomic action (even if it takes several clicks), output: {"decomposed": false, "sub_topics": [], "query": "¡search-engine-friendly phrasing of the whole task¿", "candidate_approaches": ["¡query for approach 1¿", "¡query for approach 2¿", ...]}
> If it is really 2-4 distinct sub-tasks, output: {"decomposed": true, "sub_topics": [{"label": "¡short human-readable name¿", "query": "¡search-engine-friendly phrasing of just this sub-task¿", "candidate_approaches": ["¡query for approach 1¿", ...]}]} List sub_topics in the order they'd actually be performed. Do not decompose further than the task naturally splits – most instructions are atomic.
> Output ONLY the JSON object, no markdown fences, no explanation.

(When a prior live rollout already learned something about the task, that context is appended and the prompt is instructed to sharpen the query/candidate-approach text with it rather than re-guessing generically.)

A.8.2 Method-Voting Prompt

Once multiple web sources / video transcripts have been summarized, this call treats each source as one independent "voter" for a method and elects a winning method by plurality, keeping every other distinct method on record as a minority method (these become the skill's "Other / Possible Methods" section):

> You are given research (web sources and/or YouTube video summaries) about how to: "{query}"

Research: {research_context}
Each web source and each YouTube video is one independent "voter" describing a method (a sequence of steps) for accomplishing the task. Different voters may describe the same method in different words, or describe genuinely different methods (e.g. different tools, different menu paths, different overall approach).
Your job:

1. Group the voters by which underlying method they describe (methods that only differ in wording/UI labels but do the same thing in the same order count as the SAME method).
2. Determine the WINNING method – the one described by the largest number of voters (a plurality/majority vote). If there is an exact tie, pick the method that appears most complete/detailed.
3. List every other distinct method as a minority method.

Output in this exact format, no extra prose:
Winning Method
Name: ¡short name for the winning method¿
Vote count: ¡N of M voters¿
Steps:
1. ¡step¿
...
Minority Methods
¡method name¿
Vote count: ¡N of M voters¿
Steps:
1. ¡step¿
...
(repeat for each minority method; if none, write "None – all voters agree.")

A.9 Stuck-Recovery Prompts: Region Diagnosis and Free Exploration

When the executor repeats a failed action or otherwise stalls, two further LLM calls (independent of the critic in Section A.4) help it recover without reusing the coordinates that just failed.

A.9.1 Stuck-Region Diagnosis Prompt

Given the screenshot at the point of failure and a description of the action that did not work, this call re-derives, from scratch, what the action was actually aiming for and why it likely missed:

Task: {instruction}
The agent tried this action but it did NOT work: {action_desc}
The coordinates used were WRONG – do not assume they were close. Look at this screenshot fresh and identify, in ONE sentence, the exact visible UI element (with its own precise pixel location, re-derived from scratch) that this action's INTENT was aiming for, and why the previous attempt likely missed it (e.g. wrong row, hidden until hover, wrong icon).

Its output is substituted into the stuck-recovery system prompt ("Where to look instead: {diagnosis}"), which otherwise instructs the agent to try a completely different approach (dismiss popups, scroll, right-click, tab/enter navigation, or a proper drag sequence instead of a click) and explicitly forbids repeating any of the listed failed actions.

A.9.2 Free-Exploration Element-Planner Prompt

Used both for pre-task UI mapping and for open-ended stuck recovery when no targeted diagnosis is available, this call enumerates candidate interactive elements of the foreground application (not the OS chrome or other apps) together with a probe action and a recovery action for each, so exploration can proceed safely without derailing into an unrelated application:

> Task: {instruction}
>
> This is a desktop screenshot. Coordinate space: x in 0-{COORD_SPACE}, y in 0-{COORD_SPACE_Y} (matches the image's actual aspect ratio – it is wider than it is tall, do not treat it as square).
>
> One application window relevant to the task should already be open and occupying most of the screen – that window's own UI (its menus, toolbars, panels, buttons, tabs) is what you're exploring. The goal is to learn that app's UI, not to switch apps or return to the OS.
>
> List ONLY interactive UI elements that belong to that application's own window: its menu bar items, toolbar buttons, panel controls, tabs, dropdowns, input fields, checkboxes, and similar in-app controls.
>
> The application's own UI includes its ENTIRE window chrome, not just the main content pane in the middle: its title bar, menu bar, every toolbar/ribbon (including ones squeezed into a thin strip at the very top or edge), status bar, and any side panels – as well as controls nested inside those (e.g. a browser's address/URL bar and the icons inside or beside it, or a document/image editor's own menu bar and tool ribbon above the canvas). Do not fixate on just the document/page/content area's own buttons and inputs – the surrounding chrome is just as much "the application's own UI" and is often exactly what the task needs.
>
> Rules:
>
> - Do NOT list anything outside the application window: taskbar/dock icons, desktop icons, the OS app-switcher, system tray, clock, or other applications' icons.
> - Do NOT list empty space, backgrounds, or non-interactive decorations.
> - Use exact visual coordinates for each element – look carefully at where it sits.
> - Include up to {max_elements} elements, prioritising those most relevant to the task.
>
> Output a JSON array (no prose):
> {"region_id": "¡short_snake_case_label¿", "description": "¡element type + visible label, e.g. 'New Tab button'¿", "probe_action": "¡one-line pyautogui call with exact x in 0-{COORD_SPACE}, y in 0-{COORD_SPACE_Y} coords¿", "recovery": "¡pyautogui to dismiss/undo, e.g. pyautogui.press('escape') or click away¿"}

Exploration is additionally bounded by an app-boundary guardrail appended to the stuck-recovery system prompt, which permits any dialog, secondary window, or file browser that the task genuinely calls for, but forbids switching to an unrelated application, dock icon, or system menu once stuck:

> This task only involves {APP}. Do NOT switch to, launch, or click the taskbar/dock icon for a different, unrelated application. If you truly can't find what you need, look harder within the current app first (menus, right-click, scrolling, keyboard shortcuts) before considering anything else. If {app} itself isn't open, you may reopen it from the dock/applications menu – but do not open a different application instead.
>
> The rule is task-relevance, not staying frozen on one window: any dialog, secondary window, file/folder browser, or other UI that the task or the current app genuinely calls for is still in scope, whatever form it takes. What's off-limits is reaching for something with NO connection to the task once stuck – an unrelated app, a random dock/taskbar icon, a system menu – not avoiding everything besides the exact window you started in.

A.10 Skill-Authoring Delegation Prompts (Create / Edit)

The prompts in Section A.4 run *inside* an *edit_skill* MCP tool call. That tool call is itself invoked by a separate Claude-Code-CLI sub-agent, which is launched with one of the two short delegation

prompts below telling it which pipeline to follow and which MCP tools it may use; it does the actual web/video research and then calls *create_skill* or *edit_skill* once.

A.10.1 CREATE-SKILL DELEGATION PROMPT

Used the first time a domain/task needs a skill authored from scratch:

> Follow preprocess/create-or-edit-skill/run-pipeline.md in –manual mode (use your own WebSearch to find URLs, then the websearch/youtube_search/create_skill MCP tools to research and write the skill – your own WebFetch is allowed too, as a fallback for a URL the websearch tool's fetch/PDF-render pipeline can't handle, but prefer websearch first since it does real page-text fetch, PDF rendering, and VLM-filtered visual extraction that a plain fetch can't; do not call skill_creation/create_or_edit_skill.py directly) to CREATE a new skill.
>
> Action: create
> Instruction: {instruction}
> Domain: {domain}
> Target skill directory: {output_dir}
>
> Write the finished SKILL.md (and any step screenshots) to that exact directory, then stop – do not run or evaluate the skill.

Disallowed built-ins for this sub-agent: Bash, Edit, Glob, Grep, Agent, AskUserQuestion, NotebookEdit (its own WebFetch remains allowed, as a fallback alongside the dedicated research MCP tools).

A.10.2 EDIT-SKILL DELEGATION PROMPT

Used when an existing skill is being repaired rather than authored from scratch (this is the outer call that wraps the ProactSkill *refine_skill* prompt of Section A.4):

> Follow preprocess/create-or-edit-skill/run-pipeline.md in –manual mode (use the edit_skill MCP tool – and, if fresh research is warranted, your own WebSearch plus the websearch/youtube_search MCP tools; your own WebFetch is disallowed for this run, and do not call skill_creation/create_or_edit_skill.py directly) to EDIT an existing skill.
>
> Action: edit
> Task instruction the skill is for: {instruction}
> Domain: {domain}
> Skill directory to edit: {skill_dir}
> Exploration log with new findings to incorporate: {path}
> (or: No exploration log available – review the existing SKILL.md for correctness/completeness issues and improve it based on general knowledge of the task and domain.)
>
> Back up the existing SKILL.md (e.g. SKILL.md.bak) before overwriting it, matching skill_editor.refine_skill's convention. Then stop – do not run or evaluate the skill.

Disallowed built-ins for this sub-agent: Bash, Edit, Glob, Grep, Agent, AskUserQuestion, NotebookEdit, WebFetch – unlike the create-skill delegation above, WebFetch is *not* allowed here, so any fresh research must go through the dedicated websearch/youtube_search MCP tools rather than free-form fetching.

A.11 TOOL ALLOW-LISTS

- **Main task-execution agent** (all conditions): disallowed built-ins Bash, Edit, Write, Glob, Grep, Agent, AskUserQuestion, NotebookEdit (*--disallowed-tools*, run with *--dangerously-skip-permissions* inside a disposable VM).

- **Skill-creation sub-agent**: disallowed built-ins Bash, Edit, Glob, Grep, Agent, AskUserQuestion, NotebookEdit (own WebFetch permitted as a fallback alongside dedicated research MCP tools *websearch*/*youtube_search*/*create_skill*).
- **Skill-edit sub-agent**: disallowed built-ins Bash, Edit, Glob, Grep, Agent, AskUserQuestion, NotebookEdit, WebFetch (must use the dedicated *edit_skill* MCP tool rather than free-form web fetching).

The GUI action space itself (Section A.2) is identical across all three GUI domains evaluated (GIMP, LibreOffice Calc, LibreOffice Impress); only the injected skill guidance is domain-specific.