CUA-RECOVERY: BENCHMARKING ERROR RECOVERY IN LONG-HORIZON COMPUTER-USE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

As computer-use agents (CUAs) move to long-horizon workflows spanning multiple applications, undetected errors can silently corrupt persistent state and cause cascading failures in subsequent actions. Reliable error detection and recovery have therefore become critical bottlenecks for building robust CUAs. However, existing benchmarks focus primarily on end-to-end task performance, providing limited insight into whether agents can detect and recover from errors during execution. The few benchmarks that evaluate recovery rely largely on manually injected errors, which may not reflect the types or horizons of failures induced by real agent policies. This mismatch limits both realistic evaluation and systematic progress. In this paper, we introduce **CUA-Recovery**, a benchmark comprising test and training suites across multiple desktop applications. We construct the benchmark using **ReRail**, a framework for synthesizing long-horizon tasks and trajectories in a persistent desktop environment. ReRail composes MyPCBench tasks into new long-horizon workflows with executable ground truth, collects verifier-confirmed failures from multiple policies, and turns them into verified recovery trajectories. ReRail yields CUA-Recovery, which comprises a human-verified test suite of 2,250 erroneous states over 100 tasks and a training suite of 8.6K trajectories over 377 tasks. Evaluation on the test suite shows that agents struggle both to recognize inherited errors and to complete the task. Across six agents and six takeover depths, Error Awareness Rate (EAR) reaches at most 44.0% and Pass@3 at most 25.5%. After fine-tuning Qwen3.5-35B-A3B on the training suite, its average EAR rises from 19.9% to 35.2% on held-out recovery states, and its Rubric Score goes from 5.0% to 16.1%. It also gains 15.3 points on MyPCBench and 10.8 points on Odysseys for Pass@3, showing that the training trajectories help both with recovery and with completing tasks from a clean start.

1 INTRODUCTION

Computer-use agents (CUAs) operate desktop applications through screenshots, mouse, keyboard, and the command line, and they are now evaluated on workflows that span many applications and hundreds of steps. OSWorld 2.0 raises the typical task from about 30 tool calls to more than 300 (Yuan et al., 2026), and Agents' Last Exam collects about 1,500 professional workflows (Sun et al., 2026b). On both benchmarks, the best agents earn far more partial credit than full completions: 77.7% versus 44.3% on OSWorld 2.0, and 59.3% versus 34.2% on Agents' Last Exam.[1] Agents thus finish most of a task but fail on a few steps. The cause lies in persistent state: an early wrong action is written into files, spreadsheets, or databases, and later steps build on its result. The error is rarely visible when it happens; it surfaces many steps later, when the step that caused it has already left the agent's context. Figure 1 shows a real case: Kimi-K3 fails to open the order history at step 3, keeps building on this mistake, and the error surfaces only at step 43, when the agent falsely reports completion. In such tasks, whether an agent can notice and repair its own errors decides whether it completes the task.

On the training side, recovery requires trajectories that follow an error from the mistake to its repair: the agent makes a mistake, the mistake stays unnoticed for many steps, and the agent later finds and

[1] Public leaderboards, accessed September 2026: `https://snorkel.ai/leaderboard/os-world-2-0/` and `https://snorkel.ai/leaderboard/agents-last-exam/`.

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: Case study: recovering from a real policy-induced error. The user asks for his usual Sweetgreen order. Top: Kimi-K3 fails to notice and recover from the error. Bottom: ReRail-35B-A3B takes over at the root-cause step, detects the error, and recovers from it.

repairs it. Existing CUA training sets capture only parts of this process (Table 1). Their trajectories average 3.8–39.3 steps and rarely span multiple applications (at most 26.1% where reported). When errors are included, they come from human demonstrations, synthetic injection, or self-reports rather than failures produced by the acting policy. GUI-Reflection is the only dataset with explicit corrections, but each correction immediately follows a single injected error. CUA-Recovery-Train fills this gap with 60-step trajectories, 67.5% of which span multiple applications. Each trajectory starts from a policy-induced failure, preserves the intervening erroneous history, and ends with a verifier-confirmed recovery.

On the evaluation side, long-horizon benchmarks score only the final outcome, which cannot tell whether an agent detected and fixed its own errors along the way. Existing datasets and benchmarks differ from real long-horizon failures in three ways. (1) **Coverage.** Most datasets contain only perfect trajectories; those that contain errors take them from human demo errors, injected errors, or self-reported failures, and these are mostly low-level execution errors such as invalid clicks (Wang et al., 2025; Wu et al., 2025; Jung et al., 2026). We annotated the 619 failures of six agents in CUA-Recovery-Test and found that real errors lie mostly in decisions: planning and perception account for 57.3% of root causes, whereas execution errors account for 84.3% of AgentNet and 71.2% of GUI-Reflection root causes (Figure 3a). (2) **Horizon.** Existing trajectories are short, and their errors are visible at the step they occur, while 78.1% of the errors we annotated surface three or more steps after the root cause, with a median of 17 steps (Figure 3b). (3) **Interface.** Real computer work is hybrid: agents inspect visual state through the GUI and run precise, bulk operations through the command line, both on the same application state. Existing resources fall short of this setting in two ways. First, training data contain mostly GUI actions (Table 1), so a

Table 1: Comparison with existing CUA training sets. **Recovery** indicates whether the data contains error corrections: Single-step corrections immediately follow an injected error; Verified recoveries start from real failure prefixes and are kept only if the verifier confirms task completion.

Benchmark	Platform	Traj.	Mean steps	Multi-app	Interface	Error source	Recovery
AndroidControl (Li et al., 2024)	Mobile	15,283	5.5	–	GUI	Only perfect traj.	×
AgentTrek (Xu et al., 2025)	Web	10,398	12.1	–	GUI	Only perfect traj.	×
ScaleCUA (Liu et al., 2026b)	Cross	19K	9.0	–	GUI	Only perfect traj.	×
VideoAgentTrek (Lu et al., 2025a)	Cross	39K	39.3	–	GUI	Only perfect traj.	×
AgentNet (Wang et al., 2025)	Desktop	22,532	18.8	7.2%	GUI	Human demo errors	×
GUI-Reflection (Wu et al., 2025)	Mobile	8,640	3.8	–	GUI	Injected errors	Single-step
ProCUA-SFT (Jung et al., 2026)	Desktop	93,566	29.7	26.1%	GUI	Self-reported	×
CUA-Recovery-Train	Desktop	8,577	**60.0**	**67.5%**	**GUI + CLI**	**Policy-induced**	**Verified**

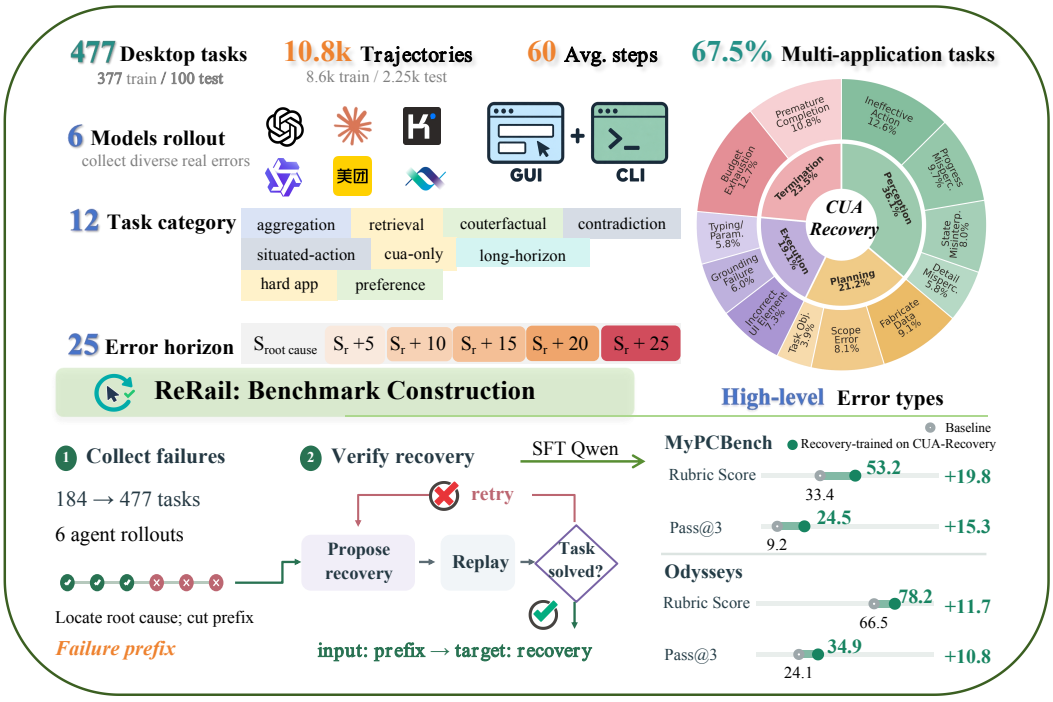


Figure 2: Overview of our methodology and benchmark.

model trained on them rarely sees the two interfaces used together. Second, scoring often relies on screenshots, yet frontier agents prefer the command line when it is available and produce few or no screenshots. On MyPCBench, once the judge reads terminal output, tool-call responses, GUI action logs, and screenshots together, GPT 5.5 rises from the officially reported 45.1% Perfect Pass (Jang et al., 2026a) to 85.0% Pass@3. Screenshot-based scoring therefore underestimates such agents and cannot compare agents that interact in different ways on equal terms.

These three gaps motivate our work. Firstly, we propose **CUA-Recovery**, a benchmark for error recovery in long-horizon computer use (Figure 2). Its test set, CUA-Recovery-Test, contains 2,250 human-verified erroneous states over 100 tasks, drawn from 619 failed rollouts of six agents. An agent takes over each state 0 to 25 steps after the root cause and is not told that an error occurred. We report whether it notices the inherited error (Error-Awareness Rate, EAR), how much of the task it completes (Rubric Score), and whether the task is completed in any one of three runs (Pass@3). The judge reads terminal output, tool calls, and screenshots together, so agents that act through the GUI and the command line are scored on equal terms. Its training set, CUA-Recovery-Train, contains 8.6K verified recovery trajectories over 377 tasks.

Secondly, we build CUA-Recovery with **ReRail**, a framework that synthesizes recovery data from real failures. ReRail converts each seed task into an executable intermediate representation whose gold lineage fixes the correct value at every step, and composes these tasks into long multi-application workflows over shared persistent state. The gold lineage removes the need to inject errors. Several agents execute the workflows through GUI and CLI; ReRail keeps only the failures that the task verifier confirms, locates each root cause against the gold lineage, and cuts the rollout at several depths after it. Every erroneous state thus comes from an agent's own decision, and its takeover depth is controlled. For training, the base model, Qwen3.5-35B-A3B, continues from each cut under hints of increasing specificity. A continuation is kept only if the verifier confirms that it completes the task, and the hint is then removed, so the model learns to notice and repair the error without being told.

On CUA-Recovery-Test, Pass@3 reaches at most 25.5% across six agents and all takeover depths, and clean-start success does not imply recovery: Kimi-K3 solves 66.9% of MyPCBench tasks from scratch but notices at most 18.5% of the errors in its own failures. Notably, training on CUA-Recovery-Train narrows this gap. The ReRail-35B-A3B recovers better than its base model, sur-

passes Kimi-K3 on Kimi-K3's own failures at early takeover depths, and raises clean-start Pass@3 from 9.2% to 24.5% on MyPCBench and from 24.1% to 34.9% on Odysseys.

In summary, our contributions are:

- **CUA-Recovery**, an evaluation and training benchmark built from real policy-induced failures in long, stateful computer-use workflows, with 2,250 human-verified test states and 8.6K verified recovery trajectories in training set.
- **ReRail**, a pipeline that composes seed tasks into long workflows with gold lineage and turns verifier-confirmed agent failures into test states and verified recoveries.
- A controlled study showing that strong clean-start performance does not imply reliable recovery, and that fine-tuning on CUA-Recovery-Train improves both recovery and clean-start performance.

2 Related Work

Long-Horizon Benchmarks for Robust CUAs. CUA benchmarks measure grounding (Cheng et al., 2024; Li et al., 2025), step accuracy on offline trajectories (Zheng et al., 2024; Li et al., 2024; Lu et al., 2025c), and end-to-end success in live environments (Bonatti et al., 2024; Xie et al., 2024; Rawles et al., 2025). Recent benchmarks extend the horizon: MyPCBench (Jang et al., 2026a) builds persona-grounded workflows over persistent multi-application state, OSWorld 2.0 (Yuan et al., 2026) collects tasks that take a human more than an hour, and Agents' Last Exam (Sun et al., 2026b) covers professional workflows across 55 subfields. These benchmarks report task success and partial credit. Robustness benchmarks apply an externally designed stressor, such as perturbations and injected faults (Zhao et al., 2025; Yang et al., 2025b; Bai et al., 2026; Chen et al., 2025; Liu et al., 2026a; Sun et al., 2026a) or adversarial content (Liao et al., 2025); AgentErrorBench (Zhu et al., 2025) analyzes failures of text-based LLM agents. Closest to our work, Bu et al. (2026) study recovery from policy-induced errors on short-horizon GUI tasks; as their data are unreleased, we compare with their reported statistics (Figure 3a). CUA-Recovery measures whether an agent detects and repairs a real error after taking over mid-task: it draws erroneous states from the agents' own long-horizon failures and places the takeover point up to 25 steps after the root cause, beyond the agents' visual context.

Data for Training Robust CUAs. CUAs are trained on human demonstrations (Rawles et al., 2023; Deng et al., 2023; Wang et al., 2025) and synthesized trajectories (Xu et al., 2025; Pahuja et al., 2025; Trabucco et al., 2025; Gandhi & Neubig, 2025; Xie et al., 2025; Jung et al., 2026; He et al., 2025), and learn error handling from reflection data and critics (Wu et al., 2025; Wanyan et al., 2025; Qin et al., 2025), self-training on failed trajectories (Fu et al., 2025; Yuan et al., 2025), or online RL (Bai et al., 2024; Yang et al., 2025a; Lu et al., 2025b; Ye et al., 2025). Public corpora are short: AgentNet averages 18.8 steps with 7.2% multi-application trajectories, ProCUA-SFT 29.7 steps with 26.1%, and GUI-Reflection 3.8 steps, and none carries persistent state across applications (Table 1). CUA-Recovery-Train averages 60.0 steps with 67.5% multi-application trajectories, and pairs real failures of several policies with verifier-accepted recoveries.

3 Problem Formulation

Environment and tasks. A computer-use task is modeled as a Partially Observable Markov Decision Process (POMDP) $(\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \Omega)$. A state $s_i \in \mathcal{S}$ is the full persistent state of the machine after step i: application databases, files, and the user interface. An action $a \in \mathcal{A}$ is an operation issued by the agent, including GUI operations, such as click, drag, scroll, typing, or a key combination and CLI commands. Each agent issues actions in its own native grammar; the harness normalizes every action into a shared executable form, so the history of one agent can be replayed for another. At step i the agent observes a screenshot $o_i = \Omega(s_{i-1}) \in \mathcal{O}$, which reveals only part of the state, and its action updates the state as $s_i = \mathcal{T}(s_{i-1}, a_i)$. We treat $\mathcal{T}$ as deterministic once reproducibility-breaking factors such as automatic updates and notification daemons are disabled; this is what allows a recorded action prefix to be replayed into the same state. A task is a triple $(l, s_0, \mathcal{C})$: a natural-language instruction $l \in \mathbf{L}$, an initial snapshot s_0, and a rubric $\mathcal{C} = \{(c_j, w_j)\}_{j=1}^{m}$ of binary criteria with weights $w_j > 0$.

Rollouts and Evaluation. At step i, a VLM policy π samples $a_i \sim \pi(\cdot \mid l, o_i, H_{i-1})$, where $H_{i-1} = (o_1, a_1, \ldots, o_{i-1}, a_{i-1})$ represents the interaction history. The episode ends when the agent emits `terminate` or exhausts its step budget B, yielding a rollout $\xi = (o_1, a_1, \ldots, o_T, a_T)$ and a final environment state s_T. The rollout records observations because states are only partially observable; since $\mathcal{T}$ is deterministic, any state s_i can be reconstructed by replaying $a_1, \ldots, a_i$ from s_0. An action can mutate invisible state components whose effects persist and influence subsequent operations. A wrong action may thus remain latent for many steps until explicitly revealed by a future observation. Together with this issue, most agent frameworks constrain the context window to the K most recent screenshots ($K = 20$ in our evaluation). Consequently, visual evidence can drop out of the agent's context while its persistent state effects remain active. Each evaluation criterion $c_j(l, \xi) \in \{0, 1\}$ is determined from the rollout, its associated screenshots, and the final state s_T. The overall rubric score is defined as $\rho(\xi) = \sum_j w_j c_j / \sum_j w_j$. We define a rollout succeeds ($V(\xi) = 1$) if and only if all criteria are satisfied; otherwise, it fails. We run each evaluation three times: a task counts as solved under Pass@3 if any one of three rollout succeeds, and the Rubric Score is the score of that rollouts.

Root cause, takeover depth, and error horizon. In a failed rollout, the root cause step t_r is defined as the earliest action that the agent does not undo, and without which the task remained solvable from s_{t_r-1}. Its associated error type κ follows the taxonomy in Appendix C.3. All identified errors arise from the agent's own decisions. The takeover depth d counts the steps taken after the root cause. An erroneous state at depth d is the state s_{t_r+d} reached by replaying the prefix $\xi_{\leq t_r+d}$ of a failed rollout, after unrelated earlier mistakes have been repaired so that the prefix contains exactly one root cause (§4.1). An agent π' takes over with l and H_{t_r+d} as its own history, and we score the full trajectory with V and ρ. For each failure we also annotate the error horizon $h_e = t_e - t_r$, where t_e is the first step at which the error becomes explicit in the trajectory; h_e is unobserved if the trajectory ends before t_e. We write $c = t_r + d$ for the takeover step and H_c for the interaction history up to step c. Execution errors become visible in the next observation ($h_e \geq 1$), whereas planning and perception errors are implicit in the history from t_r and can surface much later.

4 The CUA-Recovery Benchmark and ReRail

Building on the formulation above, we introduce **CUA-Recovery**, a benchmark for error detection and recovery in long-horizon computer-use tasks, and **ReRail**, the pipeline that builds it. CUA-Recovery addresses the three gaps in Section 1: its errors come from the agents' own policies rather than injection (coverage), propagate through dependencies across applications and stay hidden for many steps (horizon), and arise in a persistent desktop environment where agents act through both GUI and CLI (interface). ReRail composes seed tasks into long multi-application workflows, collects failures from multiple agents, and turns them into CUA-Recovery-Test, a human-verified test set of 2,250 erroneous states, and CUA-Recovery-Train, a training set of 8,577 verified recovery trajectories. Figure 2 summarizes the pipeline.

4.1 Building the Benchmark with ReRail

The key idea of ReRail is that every task carries a gold lineage, the correct value at every step. An agent's own failure can then be traced against this lineage to locate its root cause, so no errors need to be injected. We outline the main steps below; Appendix B gives the details.

Building tasks with dependencies. We start from the tasks in MyPCBench (Jang et al., 2026a). For each task, ReRail creates an executable intermediate representation that records the application operations, the data they read and write, and the checks on their outputs. Running this graph on a fresh environment snapshot gives the expected intermediate values and final state. To form longer workflows, we connect two tasks when the output of the first can serve as an input to the second. A value that was given in the original instruction must now be found through earlier work, creating a dependency along which a mistake can spread. The resulting pool contains 477 workflows, with medians of three source tasks and 22 graph nodes per workflow.

Checking the generated tasks. We execute each workflow to obtain its reference intermediate values and final state, then freeze its verifiers to this gold lineage. We perturb intermediate values

and rerun the workflow; a mutation is caught if a downstream verifier fails or if the final outputs or stored data differ from the gold lineage. We retain workflows that catch at least half of the executed mutations and select tasks across different dependency lengths.

Constructing erroneous states and the evaluation suite. We split the composed workflows into training and test tasks, with all rollouts, takeover states, and recovery trajectories derived from the same workflow assigned to one split. We run six agents on the workflows: GPT 5.5, Claude Opus 4.8, Kimi-K3, Qwen3.5-35B-A3B, EvoCUA-32B, and OpenCUA-72B. For each failed rollout, ReRail compares the recorded execution with the gold lineage to identify the root-cause step t_r, the error type, and the first step at which the error becomes evident. To keep each case controlled, experts fix any unrelated mistakes before t_r, so that the root cause is the only error in the prefix. ReRail then creates takeover states by replaying the rollout to depths $d \in \{0, 5, 10, 15, 20, 25\}$ after the root cause. Each replay must reproduce the state recorded at the same point in the original rollout. For the test suite, human annotators verify the root cause, error type, error horizon, repaired prefix, and replayed state. These annotations support the awareness metric and analyses by error type and horizon (Appendix E).

Generating the training suite. For each erroneous state in the training split, the base model, Qwen3.5-35B-A3B, attempts to complete the task with guidance about the earlier error. Recovery generation begins with a hint that identifies the erroneous step, then adds the conflicting evidence and the affected data if earlier attempts fail. The first continuation accepted by the final verifier becomes the training target; a teacher model is used when the base model produces no verified continuation. Each training example contains the task instruction, the erroneous history, and the verified continuation. The hints are omitted from the training input, matching the unaware takeover setting used for evaluation. Appendix F.4 describes the hint schedule and generation budget.

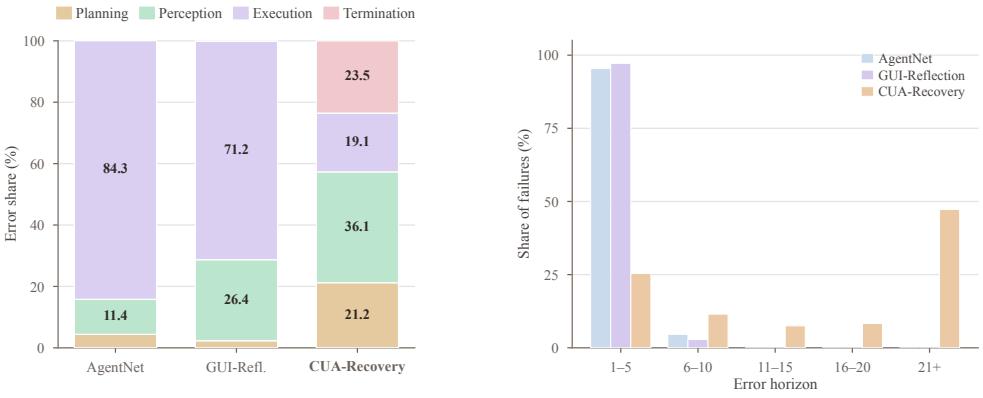
4.2 Dataset Composition and Analysis

Evaluation at different takeover depths. At each test state, an agent receives the task and the preceding interaction history, then continues without an error warning. The six takeover depths cover histories in which the root-cause screenshot is recent, near the 20-screenshot context limit, or outside that limit. The test set contains 602, 443, 373, 328, 273, and 231 states at depths $d = 0, 5, 10, 15, 20, 25$, respectively. The number of eligible cases decreases with depth; the replay and eligibility rules are described in Appendix C.1.

Training and test suites. Table 7 summarizes the two splits. The training suite provides examples of how to finish a task after an error; the test suite provides replayable states and human-checked labels for measuring error awareness and recovery. These labels also let us examine what kinds of errors agents make and how long those errors remain hidden before becoming evident.

Errors beyond individual actions. Most failures in CUA-Recovery are wrong decisions, not wrong clicks. They span planning, progress and state perception, execution, and termination errors. Examples include choosing records from the wrong year, reading a quantity incorrectly and using it in an order, and creating events in the wrong calendar application (Appendix C.4.1). These cases require the agent to check earlier decisions and the data they produced. Planning and perception account for 57.3% of root causes, termination for 23.5%, and execution for 19.1%. For comparison, the available estimates for AgentNet and GUI-Reflection assign 84.3% and 71.2% to execution errors (Figure 3a).

Errors that become clear later. Most errors surface only at the end of a rollout. In 602 of the 619 labeled failures, the error becomes explicit within the recorded rollout, and in 361 of them (60.0%), the evidence first appears at the final step. The median delay from the root cause is 17 steps, and 78.1% of errors surface at least three steps later. This gives CUA-Recovery a substantial set of failures in which recovery requires revisiting earlier work. Figure 3b shows this delay distribution; the curves for AgentNet and GUI-Reflection are estimates.



(a) Error types

(b) Error horizon

Figure 3: Error types and the delay before an error becomes explicit.

5 Experiments

We use CUA-Recovery to answer two questions: (1) can current agents recover from errors they inherit (Section 5.2), and (2) can CUA-Recovery-Train teach recovery (Section 5.3)?

5.1 Evaluation Setup

Agents, metrics, and protocols. We evaluate GPT 5.5, Claude Opus 4.8, Kimi-K3, EvoCUA-32B, Qwen3.5-35B-A3B, and OpenCUA-72B. Agents use their respective scaffolds; EvoCUA and OpenCUA use GUI actions only, while the other configurations expose command-line tools where supported (Appendix F). In self-takeover, each agent resumes states drawn from its own failures. It receives the recorded history without an error warning and can take up to 100 additional steps. Error-Awareness Rate (EAR) measures whether an agent explicitly recognizes a problem in the inherited work. Rubric Score, defined in Section 3, measures partial task completion, while Pass@3 reports full completion in at least one of three runs. Reporting them together separates expressed awareness from recovery outcomes.

Clean-start evaluation. We also run agents from the initial task state on MyPCBench (Jang et al., 2026a) and Odysseys (Jang et al., 2026b), with 184 and 200 tasks, respectively. Each run is limited to 150 steps and 3,600 seconds. MyPCBench supplies the source tasks used to build our workflows and measures clean-start performance in the source environment. Odysseys contains live-web tasks outside the training environment and measures whether the effect of recovery training extends to a different task setting.

5.2 Benchmarking Recovery on CUA-Recovery-Test

We let each of the six agents take over states from its own failures at six diverse takeover depths. Table 2 reports EAR, Rubric Score, and Pass@3. These results lead to four findings as followed.

Finding 1: Current agents rarely recover completely from their own failures. Complete recovery is difficult for all six agents on CUA-Recovery-Test. Across all agents and takeover depths, Pass@3 reaches at most 25.5%, achieved by GPT 5.5 at $d = 20$ (Table 16). Claude Opus 4.8 reaches at most 7.7%, while none of the three open-source agents exceeds 6.4%. These results show that even strong computer-use agents often fail to complete a task once an earlier error has affected the workflow.

Finding 2: Clean-start performance does not reliably predict error awareness. Kimi-K3 achieves 66.9% Pass@3 on MyPCBench from a clean start, but recognizes only 7.8–18.5% of the inherited errors in its own failures. In contrast, OpenCUA-72B achieves 5.1% clean-start Pass@3 but recognizes 14.1–26.1% of its inherited errors (Tables 3 and 2).

Finding 3: Recognizing an error rarely leads to recovery. Across the self-takeover runs of all six agents, EAR reaches at most 44.0% and remains below 33% for all agents except Claude Opus 4.8 (Table 2). We next group each agent's individual runs by whether it recognizes the inherited error and compute verifier success within each group. Macro-averaged across the six agents, the verifier pass rate is 15.6% for aware runs and 12.2% for unaware runs (Appendix Table 18). Most runs still fail after recognizing the error: agents often identify a problem but leave the affected state unchanged or fail to repair all of its consequences. In unaware runs, agents almost always continue from the inherited state without revisiting the earlier work. Awareness is therefore associated with a greater chance of recovery, but recognition alone is rarely sufficient.

Finding 4: Later takeovers remain difficult even when awareness improves. From $d = 20$ to $d = 25$, the root-cause screenshot moves beyond the 20-screenshot history. Over this interval, GPT 5.5's EAR rises from 16.2% to 21.0%, while its Pass@3 falls from 25.5% to 18.9%. Claude's EAR also rises, from 18.8% to 27.3%, while Pass@3 falls from 7.7% to 0.0%. For Kimi-K3, Pass@3 falls from 23.1% to 9.1%. This pattern suggests a trade-off at later takeover points. As an error propagates through the workflow, its effects become more visible in the current state, making the problem easier to notice. Once the root-cause screenshot leaves the context window, however, the agent must infer how the problem began from its downstream effects, making the error harder to diagnose and repair. This helps explain why EAR can increase even as Pass@3 falls.

Table 2: Recovery on CUA-Recovery-Test by takeover depth d (steps after the root cause). Each agent takes over states from its *own* failures without an error warning. ↪ **ReRail: ReRail-35B-A3B** takes over the same states as the agent above. Opus 4.8 = Claude Opus 4.8; Qwen3.5-35B = Qwen3.5-35B-A3B, the base model of ReRail. **Bold**: higher than the corresponding source agent.

	Error Awareness (%)						Rubric Score / Pass@3 (%)					
Agent d=	0	5	10	15	20	25	0	5	10	15	20	25
API-based												
GPT 5.5	20.1	26.7	32.3	28.6	16.2	21.0	65.3/20.1	67.7/24.3	61.2/20.9	60.8/20.1	68.5/25.5	54.9/18.9
Opus 4.8	18.9	44.0	39.1	22.7	18.8	27.3	35.3/0.0	24.0/0.0	29.5/4.5	38.5/5.3	48.1/7.7	36.4/0.0
Kimi-K3	13.9	7.8	10.0	11.1	18.5	15.0	39.2/11.6	41.7/8.6	36.4/7.4	33.1/11.8	41.5/23.1	36.9/9.1
↪ ReRail	**25.0**	**15.7**	**17.5**	**22.9**	14.8	15.0	36.4/**16.6**	34.8/**12.6**	31.7/**10.1**	32.4/11.6	36.8/15.2	30.0/8.8
Open-source												
EvoCUA-32B	24.7	28.3	30.3	30.4	32.7	21.8	25.6/6.4	27.1/6.1	23.8/4.4	19.3/3.7	24.5/4.6	16.1/2.3
↪ ReRail	**45.3**	**43.9**	**42.4**	**49.5**	**44.6**	**37.7**	**29.5/12.7**	**31.6/8.8**	**26.1/6.9**	**24.3/5.4**	**28.1**/3.1	**18.5/5.4**
Qwen3.5-35B	25.8	21.4	23.9	18.5	14.3	15.6	8.4/1.2	6.5/0.0	4.2/0.0	2.1/0.0	4.6/0.0	3.9/0.0
↪ ReRail	**41.5**	**37.0**	**38.6**	**34.2**	**30.8**	**29.0**	**14.3/4.7**	**18.5/3.5**	**17.4/2.7**	**18.1/1.8**	**13.8**/0.0	**14.3**/0.0
OpenCUA-72B	26.1	20.4	22.4	14.1	19.0	19.4	6.0/0.9	4.8/0.0	3.4/0.0	4.7/0.0	3.6/0.0	5.7/0.0

5.3 Training Recovery with CUA-Recovery-Train and Generalization

The previous results show that agents have separate difficulties in noticing an inherited error and repairing its downstream effects. We next examine whether the verified trajectories in the training suite can support recovery-oriented fine-tuning using a standard supervised objective.

Training setup. We fine-tune Qwen3.5-35B-A3B for one epoch on the 8,577 verified recovery trajectories in CUA-Recovery-Train, obtaining **ReRail-35B-A3B**. Given a task instruction l, an erroneous history H_c, and its verified recovery $r^\star$, we minimize

$$\mathcal{L}(\theta) = -\mathbb{E}_{(l,H_c,r^\star)\sim\mathcal{Q}}[\log \pi_\theta(r^\star \mid l, H_c)]$$

, where $\mathcal{Q}$ is the training set. The hints used to generate $r^\star$ are not included in H_c, so the model receives the same unaware input used during evaluation.

Finding 5: Recovery learned from CUA-Recovery-Train transfers across agents. We first evaluate the fine-tuned model on states taken from Qwen3.5-35B-A3B's own failures, allowing the two models to start from the same erroneous states. Fine-tuning raises EAR by 13.4–16.5 percentage

points and Rubric Score by 5.9–16.0 points across takeover depths (Table 2). At $d = 0$, Pass@3 increases from 1.2% to 4.7%. EAR and Rubric Score also improve at $d \geq 20$, although neither model solves any task at these depths. The training data therefore improves both error recognition and partial recovery, while complete recovery from later takeover points remains difficult. We also evaluate the fine-tuned model on failures produced by Kimi-K3 and EvoCUA-32B. At $d = 0$, ReRail-35B-A3B reaches 16.6% Pass@3 on Kimi-K3's failure states, compared with 11.6% for Kimi-K3, and 12.7% on EvoCUA-32B's states, compared with 6.4% for EvoCUA-32B (Figure 13). Its advantage over Kimi-K3 holds through $d = 10$, and its advantage over EvoCUA-32B holds through $d = 15$. At later depths, these gains narrow or disappear. The results show that the learned recovery behavior transfers across failure sources, with the largest gains at earlier takeover points.

Table 3: Completion performance on MyPCBench and Odysseys. Bold and underline indicate the best and second-best results within API-based and open-source agents.

	MyPCBench		Odysseys	
Agent	Pass@3 (%)	Rubric Score (%)	Pass@3 (%)	Rubric Score (%)
API-based				
GPT 5.5	**85.0**	**92.2**	**82.1**	94.2
Claude Opus 4.8	<u>68.9</u>	<u>84.6</u>	<u>81.7</u>	**95.6**
Kimi-K3	66.9	83.6	77.9	<u>94.3</u>
Open-source				
EvoCUA-32B	<u>18.1</u>	<u>41.0</u>	<u>32.2</u>	<u>73.4</u>
Qwen3.5-35B-A3B	9.2	33.4	24.1	66.5
OpenCUA-72B	5.1	20.5	18.9	52.1
ReRail-35B-A3B	**24.5**$^{\uparrow 15.3}$	**53.2**$^{\uparrow 19.8}$	**34.9**$^{\uparrow 10.8}$	**78.2**$^{\uparrow 11.7}$

Finding 6: Recovery training also improves task completion from a clean start. Training on CUA-Recovery-Train raises Pass@3 from 9.2% to 24.5% and Rubric Score from 33.4% to 53.2% on in-domain MyPCBench. On Odysseys, whose tasks and applications are outside the training environment, the scores rise from 24.1% to 34.9% and from 66.5% to 78.2%, respectively, showing gains beyond the source environment (Table 3).

Ablation study. To test whether the gains come from recovery supervision or simply from more in-domain training, we fine-tune the same base model on a token-matched control of successful trajectories from the same workflows. On Qwen3.5-35B-A3B's failures, averaged over depths, EAR rises from 19.9% to 23.6% with successful trajectories but to 35.2% with recovery trajectories, and clean-start Pass@3 on MyPCBench reaches 13.8% versus 24.5% (Appendix F.3). Recovery trajectories thus bring much larger gains than successful trajectories from the same workflows, showing that the improvement comes from learning to recover rather than from more in-domain data.

6 Conclusion

We find that current computer-use agents rarely recover from their own errors in long-horizon workflows, and strong performance does not imply reliable recovery. Existing data and benchmarks miss this setting in three ways: errors are injected or low-level rather than agent-induced, trajectories are too short for errors to stay hidden, and evaluation covers only GUI actions with screenshot-based scoring. We present CUA-Recovery, a benchmark built with ReRail from agents' own failures in long multi-application workflows involving both GUI and CLI, with a human-verified test set and verified recovery trajectories for training. Fine-tuning on CUA-Recovery-Train improves recovery from inherited errors and clean-start performance, including in an unseen environment, suggesting that learning from real failures is a practical path toward reliable computer-use agents.

AI use statement

We used generative AI tools for two purposes: polishing the writing (grammar, wording, and LaTeX formatting of text drafted by the authors) and assisting with coding (data-processing, evaluation, and

plotting scripts). We did not use generative AI tools for other research tasks; the LLM components inside our pipeline, such as the root-cause judge and the recovery proposer, are part of the method and are described in the paper. The authors reviewed and revised all AI-polished text and reviewed and tested all AI-assisted code. We take responsibility for the final content of this work.

Reproducibility statement

We release the implementation code at `https://anonymous.4open.science/r/CUA-Recovery-B03B/`. Appendix F gives the full experimental details.

References

Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. DigiRL: Training in-the-wild device-control agents with autonomous reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Haoyue Bai, Dong Wang, Long Chen, Bingguang Hao, Pengyang Shao, Yonghui Yang, Yicheng He, and Chenyi Zhuang. StressWeb: A diagnostic benchmark for web agent robustness under realistic interaction variability. *arXiv preprint arXiv:2604.16385*, 2026.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, et al. Windows Agent Arena: Evaluating multi-modal OS agents at scale. *arXiv preprint arXiv:2409.08264*, 2024.

Tianpeng Bu, Xin Liu, Qihua Chen, Hao Jiang, Shurui Li, Hongtao Duan, Lu Jiang, Lulu Hu, Bin Yang, and Minying Zhang. Recovering policy-induced errors: Benchmarking and trajectory synthesis for robust GUI agents. *arXiv preprint arXiv:2605.29447*, 2026.

Sen Chen, Tong Zhao, Yi Bin, Fei Ma, Wenqi Shao, and Zheng Wang. D-GARA: A dynamic benchmarking framework for GUI agent robustness in real-world anomalies. *arXiv preprint arXiv:2511.16590*, 2025.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. SeeClick: Harnessing GUI grounding for advanced visual GUI agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9313–9332, 2024.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Dayuan Fu, Keqing He, Yejie Wang, Wentao Hong, Zhuoma Gongque, Weihao Zeng, Wei Wang, Fan Yang, Xunliang Cai, and Weiran Xu. AgentRefine: Enhancing agent generalization through refinement tuning. In *International Conference on Learning Representations (ICLR)*, 2025.

Apurva Gandhi and Graham Neubig. Go-Browse: Training web agents with structured exploration. *arXiv preprint arXiv:2506.03533*, 2025.

Yanheng He, Jiahe Jin, and Pengfei Liu. Efficient agent training for computer use. *arXiv preprint arXiv:2505.13909*, 2025.

Lawrence Keunho Jang, Andrew Keunwoo Jang, Jing Yu Koh, and Ruslan Salakhutdinov. MyPCBench: A benchmark for personally intelligent computer-use agents. *arXiv preprint arXiv:2606.16748*, 2026a.

Lawrence Keunho Jang, Jing Yu Koh, Daniel Fried, and Ruslan Salakhutdinov. Odysseys: Benchmarking web agents on realistic long horizon tasks. *arXiv preprint arXiv:2604.24964*, 2026b.

Jaehun Jung, Ximing Lu, Brandon Cui, Muhammad Khalifa, Shaokun Zhang, Hao Zhang, Jin Xu, Amala Sanjay Deshmukh, Karan Sapra, Andrew Tao, Yejin Choi, Jan Kautz, Mingjie Liu, and Yi Dong. ProCUA-SFT technical report. *arXiv preprint arXiv:2606.17321*, 2026.

Kaixin Li, Ziyang Meng, Hongzhan Lin, Ziyang Luo, Yuchen Tian, Jing Ma, Zhiyong Huang, and Tat-Seng Chua. ScreenSpot-Pro: GUI grounding for professional high-resolution computer use. In *Proceedings of the 33rd ACM International Conference on Multimedia*, pp. 8778–8786, 2025.

Wei Li, William Bishop, Alice Li, Chris Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. On the effects of data scale on UI control agents. In *Advances in Neural Information Processing Systems*, volume 37, pp. 92130–92154, 2024.

Zeyi Liao, Jaylen Jones, Linxi Jiang, Yuting Ning, Eric Fosler-Lussier, Yu Su, Zhiqiang Lin, and Huan Sun. RedTeamCUA: Realistic adversarial testing of computer-use agents in hybrid web-OS environments. *arXiv preprint arXiv:2505.21936*, 2025.

Enqi Liu, Liyuan Pan, Zhi Gao, Yan Yang, Chenrui Shi, Yang Liu, Jingrong Wu, and Qing Li. Benchmarking and improving GUI agents in high-dynamic environments. *arXiv preprint arXiv:2604.25380*, 2026a.

Zhaoyang Liu, JingJing Xie, Zichen Ding, Zehao Li, Bowen Yang, Zhenyu Wu, Xuehui Wang, Qiushi Sun, Shi Liu, Weiyun Wang, et al. ScaleCUA: Scaling open-source computer use agents with cross-platform data. In *International Conference on Learning Representations*, 2026b.

Dunjie Lu, Yiheng Xu, Junli Wang, Haoyuan Wu, Xinyuan Wang, Zekun Wang, Junlin Yang, Hongjin Su, Jixuan Chen, Junda Chen, Yuchen Mao, Jingren Zhou, Junyang Lin, Binyuan Hui, and Tao Yu. VideoAgentTrek: Computer use pretraining from unlabeled videos. *arXiv preprint arXiv:2510.19488*, 2025a.

Fanbin Lu, Zhisheng Zhong, Shu Liu, Chi-Wing Fu, and Jiaya Jia. ARPO: End-to-end policy optimization for GUI agents with experience replay. *arXiv preprint arXiv:2505.16282*, 2025b.

Quanfeng Lu, Wenqi Shao, Zitao Liu, Lingxiao Du, Fanqing Meng, Boxuan Li, Botong Chen, Siyuan Huang, Kaipeng Zhang, and Ping Luo. GUIOdyssey: A comprehensive dataset for cross-app GUI navigation on mobile devices. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 22404–22414, 2025c.

Vardaan Pahuja, Yadong Lu, Corby Rosset, Boyu Gou, Arindam Mitra, Spencer Whitehead, Yu Su, and Ahmed Awadallah. Explorer: Scaling exploration-driven web trajectory synthesis for multi-modal web agents. *arXiv preprint arXiv:2502.11357*, 2025.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. UI-TARS: Pioneering automated GUI interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Android in the wild: A large-scale dataset for Android device control. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. AndroidWorld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, 2025.

Jingwei Sun, Jianing Zhu, Yuanyi Li, Tongliang Liu, Xia Hu, and Bo Han. AgentHijack: Benchmarking computer use agent robustness to common environment corruptions. *arXiv preprint arXiv:2605.25707*, 2026a.

Yiyou Sun, Xinyang Han, Weichen Zhang, Yuanbo Pang, Tianyu Wang, Yuhan Cao, Yixiao Huang, et al. Agents' last exam. *arXiv preprint arXiv:2606.05405*, 2026b.

Brandon Trabucco, Gunnar Sigurdsson, Robinson Piramuthu, and Ruslan Salakhutdinov. Towards internet-scale training for agents. *arXiv preprint arXiv:2502.06776*, 2025.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, et al. OpenCUA: Open foundations for computer-use agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Yuyang Wanyan, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Jiabo Ye, Yutong Kou, Ming Yan, Fei Huang, Xiaoshan Yang, et al. Look before you leap: A GUI-Critic-R1 model for pre-operative error diagnosis in GUI automation. *arXiv preprint arXiv:2506.04614*, 2025.

Penghao Wu, Shengnan Ma, Bo Wang, Jiaheng Yu, Lewei Lu, and Ziwei Liu. GUI-Reflection: Empowering multimodal GUI models with self-reflection behavior. *arXiv preprint arXiv:2506.08012*, 2025.

Jingxu Xie, Dylan Xu, Xuandong Zhao, and Dawn Song. AgentSynth: Scalable task generation for generalist computer-use agents. *arXiv preprint arXiv:2506.14205*, 2025.

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.

Yiheng Xu, Dunjie Lu, Zhennan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. AgentTrek: Agent trajectory synthesis via guiding replay with web tutorials. In *International Conference on Learning Representations*, 2025.

Chenyu Yang, Shiqian Su, Shi Liu, Xuan Dong, Yue Yu, Weijie Su, Xuehui Wang, Zhaoyang Liu, Jinguo Zhu, Hao Li, et al. ZeroGUI: Automating online GUI learning at zero human cost. *arXiv preprint arXiv:2505.23762*, 2025a.

Jingqi Yang, Zhilong Song, Jiawei Chen, Mingli Song, Sheng Zhou, Linjun Sun, Xiaogang Ouyang, Chun Chen, and Can Wang. GUI-Robust: A comprehensive dataset for testing GUI agent robustness in real-world anomalies. *arXiv preprint arXiv:2506.14477*, 2025b.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, et al. Mobile-Agent-v3: Fundamental agents for GUI automation. *arXiv preprint arXiv:2508.15144*, 2025.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, Dunjie Lu, Haikong Lu, Qi Zhen, Xinyuan Wang, Jiaqi Deng, Yuhao Yang, Cheng Chen, Boyuan Zheng, Alex Su, Xiao Yu, Hao Zou, Saaket Agashe, Xing Han Lu, Manpreet Kaur, Zhengyang Qi, Vincent Sunn Chen, Frederic Sala, Dayiheng Liu, Junyang Lin, Zhou Yu, Yu Su, Siva Reddy, Xin Eric Wang, Peng Qi, Tianbao Xie, and Tao Yu. OSWorld 2.0: Benchmarking computer use agents on long-horizon real-world tasks. *arXiv preprint arXiv:2606.29537*, 2026.

Siyu Yuan, Zehui Chen, Zhiheng Xi, Junjie Ye, Zhengyin Du, and Jiecao Chen. Agent-R: Training language model agents to reflect via iterative self-training. *arXiv preprint arXiv:2501.11425*, 2025.

Henry Hengyuan Zhao, Kaiming Yang, Wendi Yu, Difei Gao, and Mike Zheng Shou. WorldGUI: An interactive benchmark for desktop GUI automation from any starting point. *arXiv preprint arXiv:2502.08047*, 2025.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. GPT-4V(ision) is a generalist web agent, if grounded. In *International Conference on Machine Learning*, 2024.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, et al. Where LLM agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025.

Table of Contents

A Cost . . . 14

B Details of the ReRail Data-Synthesis Framework . . . 15

B.1 From Workflows to Training and Test Examples . . . 15

B.2 From Source Tasks to Executable Task IR . . . 15

B.3 Composition Search with an Executable Gate . . . 16

B.4 Verifier Freezing and the Mutation Test . . . 17

B.5 Horizon Buckets, Selection, and Variants . . . 18

B.6 Scale and Horizon of the Synthesized Set . . . 19

B.7 Statistics of CUA-Recovery . . . 19

C More Details of CUA-Recovery . . . 19

C.1 Test-Case Construction . . . 19

C.2 Comparison with Existing Training Corpora . . . 20

C.3 Error types . . . 20

C.4 Clean-start performance on the source tasks . . . 20

C.4.1 Data Examples . . . 22

D Prompt Templates . . . 24

D.1 Agent System Prompts . . . 24

D.2 Rubric-Judge Prompt . . . 26

D.3 Error-Awareness Judge Prompt . . . 27

D.4 Takeover Prompts . . . 28

E Judge Consistency and Human Annotation Protocol . . . 28

E.1 Annotation Protocol . . . 28

E.2 Human–Human Agreement . . . 28

E.3 Human–LLM Agreement . . . 28

F Experimental Details . . . 29

F.1 Evaluated Agents . . . 29

F.2 SFT Details for Qwen3.5-35B-A3B . . . 29

F.3 Ablation Study . . . 29

F.4 Recovery Generation and Supervised Training . . . 30

F.5 Recovery across Failure Sources . . . 30

G More Analysis . . . 31

G.1 Awareness and Recovery across Takeover Depth . . . 31

G.2 Error Horizon and Recovery Outcome . . . 33

G.3 Recovery Conditioned on Error Awareness . . . 33

A COST

Tables 4, and 5 itemize the API, and GPU cost of the paper. Judging one group with GPT 5.6-terra , either one run of one agent's 184 clean-start rollouts or one agent–horizon cell of takeover episodes, costs about $120. Prefix replay executes recorded actions without model calls, so the API cost of a takeover comes only from the agent acting after the takeover point; we estimate it from the per-task cost of the same agent's clean-start run. Ten annotators labeled the failure corpus (Appendix E.1) at $100 per annotator per labeling group. Locally served agents and ReRail-35B-A3B incur no API cost; each serves one episode at a time on eight AMD MI308X GPUs (Appendix F.1), so running the 32 concurrent sandboxes against local models occupies up to 256 GPUs.

Table 4: API cost. A judge group is one run of one agent's clean-start rollouts (3 runs per agent) or one agent–horizon cell of takeover episodes. Red values are estimates.

Item	Unit	Unit cost (USD)	Count	Total (USD)
Clean-start collection				
API agent rollouts (Claude Opus 4.8, GPT 5.5; 3 runs)	run	260	6	1,560
API agent rollouts (Kimi-K3; 3 runs)	run	252	3	756
Judge (GPT 5.6 `terra`)	group	120	18	2,160
Self-takeover on CUA-Recovery-Test (Table 16; 6 agents × 6 horizons, unaware)				
Claude Opus 4.8 takeover execution	episode	1.41	139	196
GPT 5.5 takeover execution	episode	1.41	69	97
Kimi-K3 takeover execution	episode	1.37	175	240
Judge (GPT 5.6 `terra`)	group	120	36	4,320
Cross-policy takeover by ReRail-35B-A3B (Tables 2 and 15; 3 sources × 6 horizons)				
Judge (GPT 5.6 `terra`)	group	120	18	2,160
Data construction				
Task IR extraction (GPT 5.6)	call	0.46	483	224
Instruction realization (GPT 5.6)	call	0.034	1,780	60
Recovery generation for CUA-Recovery-Train: teacher fallback	attempt	1.41	3,350	4,720
API total				16,493

Table 5: GPU and sandbox compute. Each locally served model runs one episode at a time on a server of eight AMD MI308X GPUs, so GPU-hours are eight times episode wall-clock time; API agents use sandbox time only. Clean-start times are measured; takeover times use each agent's measured per-step latency over 58 post-takeover steps.

Agent	Episodes	Min./episode	GPU-hours	Sandbox-hours
Clean-start collection				
Claude Opus 4.8	552	37.6	–	345
GPT 5.5	552	9.2	–	84
Kimi-K3	552	30.5	–	282
EvoCUA-32B	552	25.1	1,872	234
OpenCUA-72B	552	7.6	558	69
Qwen3.5-35B-A3B	552	15.9	1,170	147
Self-takeover (Table 16)				
Claude Opus 4.8	139	69.5	–	161
GPT 5.5	69	10.2	–	12
Kimi-K3	175	49.4	–	144
EvoCUA-32B	613	15.9	1,296	162
OpenCUA-72B	683	7.5	687	86
Qwen3.5-35B-A3B	571	10.1	765	96
Cross-policy takeover by ReRail-35B-A3B (Tables 2 and 15)				
ReRail-35B-A3B	1,359	10.1	1,822	228
Fine-tuning (Table 13)				
ReRail-35B-A3B training, 64 GPUs × 18 hours	–	–	1,152	–
Total			9,322	2,050

B Details of the ReRail Data-Synthesis Framework

This appendix gives the ReRail framework of Section 4.1 in full. Although the current MyPCBench task set offers realistic workflows, most of the tasks are relatively short, requiring only 20 to 45 steps for a capable agent to complete, and the dependency structures that allow errors to remain hidden typically span no more than two nodes. Studying hidden errors at scale requires tasks whose dependency chains are long and whose ground truth is executable, so that every intermediate value, every written cell, and the final state are known before any agent runs, properties that current benchmarks lack. We therefore synthesize tasks by composing verified source tasks and accept a composition only after it has been executed and its verifiers have been shown to reject wrong runs. This appendix describes the pipeline as implemented: the extraction of an executable task representation from each source task (§B.2), the composition search with an executable gate (§B.3), verifier freezing and the mutation test that measures verifier coverage (§B.4), horizon-controlled selection with world and instruction variants (§B.5), and the scale and horizon of the released set (§B.6).

B.1 From Workflows to Training and Test Examples

Algorithm 1 summarizes how ReRail uses validated workflows to construct the two splits.

Algorithm 1 ReRail: collecting failures and building recovery data

Require: validated task pool $\mathcal{M}$ with reference executions $G(M)$ and frozen verifiers; source policies Π; base model π_0; teacher π_{T}; depths $\mathcal{D} = \{0, 5, 10, 15, 20, 25\}$
Ensure: training recovery set $\mathcal{Q}$ and test-state set $\mathcal{E}$

Split $\mathcal{M}$ into training and test tasks
$\mathcal{Q}, \mathcal{E} \leftarrow \emptyset$
for task $M \in \mathcal{M}$, instruction l for M, source policy $\pi \in \Pi$ **do**
  $\xi \leftarrow \textsc{Rollout}(\pi, l, M)$
  if the final verifier accepts ξ **then continue**
  $(t_r, \kappa, t_e) \leftarrow \textsc{AnalyzeFailure}(\xi, G(M))$
  if the root cause is unresolved or type confidence is below 0.7 **then continue**
  $\xi \leftarrow \textsc{RepairEarlierMistakes}(\xi, t_r)$
  if prefix repair fails **then continue**
  for $d \in \mathcal{D}$ with $c = t_r + d \leq |\xi|$ **do**
    if replay does not reproduce the recorded state **then continue**
    if M is a test task **then**
      Apply the test eligibility rules of Appendix C.1
      Obtain human checks of the labels and replayed state
      Add accepted state, history, and labels to $\mathcal{E}$
    else
      $r^\star \leftarrow$ first verified continuation under the hint schedule of Appendix F.4
      if $r^\star \neq \perp$ **then** add $(l, H_c, r^\star)$ to $\mathcal{Q}$, without the hint
    end if
  end for
end for
return $\textsc{Dedup}(\mathcal{Q}), \mathcal{E}$

B.2 From Source Tasks to Executable Task IR

Task Intermediate Representation. A task is represented as a grounded DAG $M = (V, E)$ over one frozen environment snapshot. Every node $v \in V$ records the application it acts on, and its operation is selected from the following action set: retrieve, resolve, filter, compare, aggregate, verify, decide, create, modify, delete, communicate, confirm. It also defines the table cells it reads, the values it produces along with an executable derivation, such as a read-only SQL query, an expression over upstream values, a file read, or a row or file creation, the cells it writes, its reversibility class, and a node verifier. An edge transfers a value or state from an output port to an input port; if no edge connects to a specific input port, that port holds the literal or constant specified in the instruction. The graph carries a final-state verifier with an explicit set of equivalent final states. The snapshot consists of 17 application databases with 201 tables, a schema graph that captures both foreign-key and inferred relationships, and an inventory of the 48 documents in the persona's home directory.

The file inventory is exposed as a single virtual table, allowing files and database rows to be treated as the same kind of fact.

Extraction and programmatic validation. A language model extracts the Task IR of each of the 184 MyPCBench source tasks from the instruction, the rubric, the persona, and the schema. The result is validated programmatically rather than by an LLM judge. Static checks verify that the DAG is valid, that every cell is grounded in the schema graph, and that every named entity exists in the snapshot. Each input literal must either appear in the instruction or refer to the persona's own identity, ensuring that facts the agent is expected to retrieve are not provided directly as constants.

The gold IR is then executed on a fresh copy of the snapshot. The interpreter evaluates each derivation, performs the declared writes, checks the cells actually read by each SQL query against the declared reads, and aborts if it encounters an undeclared write. It also records the gold lineage $G(M)$, including every produced value, every written cell, and the state digest after each node.

Finally, the gold values are compared with the ground truth in the source rubric, such as amounts, dates, and names. Relative-date variables are expanded using the snapshot's reference time. When a rubric contains no comparable literals, it is evaluated based on the overlap between its wording and the gold values. Any task that fails validation is re-extracted with the program's findings appended to the prompt, for up to four rounds.

We finally accepted 135 of 184 tasks (73.4%) from the original MyPCBench, based on the validation situation: 53 by literal agreement with the rubric and 82 by wording. Of the 49 remaining tasks, 34 cannot be executed because the model's derivations or writes are incompatible with the actual schema, 8 have rubrics that contain no comparable values, 5 have final verifiers that reject their own gold runs, and 2 contradict their rubrics. The accepted IRs have a median of 6 nodes. Every accepted IR ships with its gold lineage, which is the single source of truth for everything that follows.

B.3 Composition Search with an Executable Gate

Typed compatibility with cell evidence. Composition connects an output port p_o of one accepted IR to a literal-valued input port p_i of another. The value originally provided as a constant in the second task's instruction must then be obtained by executing the first IR.

Let $\tau(p_o, p_i) \in \{0, 0.4, 1\}$ denote the type-compatibility score between the two ports. The score is 1 when their entity-reference types match, 0.4 when they share a generic scalar type such as Integer or Text, and 0 otherwise. For each port p, let $\mathrm{col}(p)$ denote the schema column associated with that port. We first identify $\mathrm{col}(p)$ from the IR's declared reads. If the declared reads do not identify a unique column, we recover it from the SQL derivation. For an output port, $\mathrm{col}(p)$ is the only column selected by the scalar query. For an input port, it is the column compared with the query parameter corresponding to p. If no unique column can be identified, the port is not eligible for composition.

Let $\mathrm{rel}(p_o, p_i)$ indicate that the tables containing $\mathrm{col}(p_o)$ and $\mathrm{col}(p_i)$ are connected in the schema graph by a key path of at most two hops. The compatibility score is

$$\sigma(p_o, p_i) = \begin{cases} 1, & \mathrm{col}(p_o) = \mathrm{col}(p_i), \\ \max\big(\tau(p_o, p_i),\, 0.8\big), & \tau(p_o, p_i) > 0 \wedge \mathrm{rel}(p_o, p_i), \\ \tau(p_o, p_i), & \text{otherwise.} \end{cases} \tag{1}$$

We call a port pair admissible, written $p_o \rightsquigarrow p_i$, if and only if $\sigma(p_o, p_i) \geq 0.5$. A pair of ports with matching generic scalar types is therefore admissible only if the ports are grounded in the same schema column or their columns are connected by a key path. This additional constraint prevents matches based on type alone. The ports must also have matching cardinalities, and literals representing the persona's identity cannot be used as graft targets. Across the 135 accepted IRs, 1,053 output ports and 109 candidate input ports produce 143 admissible edges: 51 connect ports grounded in the same column, and 92 connect type-compatible ports supported by a key path.

Operators and search. For each seed IR, we consider beam search for compositions with any IR in the accepted pool. At each step, the search can pass an output of task A to a literal input of task B, add another instance of the same operation pattern, attach an additional producer to a comparison or aggregation node, reuse a value several nodes after it is produced, or use a boolean produced by

a decide node in A to guard the first state-changing node in B. The resulting graph is re-evaluated after each step.

To avoid naming conflicts, each node identifier is prefixed with its parent task. The provenance record for each composition stores the operator, the port binding, and the source tasks.

Before execution, each candidate must pass a set of static checks. A graph is rejected if more than 25% of it consists of independent components, which usually indicates that two tasks have been placed side by side rather than meaningfully combined. It is also rejected if a value passed from one task to another has no effect on later computation, if fewer than two nodes fall within the target horizon range (§B.5), or if the graph contains fewer than 10 or more than 24 nodes. The lower bound excludes tasks that are too short to meaningfully test long-horizon reasoning (§B.6). The upper bound keeps tasks within reach of current agents.

Candidates that survive these checks are executed on a fresh copy of the snapshot. The execution must complete successfully, and the final verifier must accept the resulting gold state. The beam retains 16 states, allows at most three grafts, and uses no more than 48 gold executions for each seed in each round.

A full copy of every database is used for each world, so recomputing all database digests after every step would be expensive. Instead, the interpreter caches the digest of each database and recomputes it only when the corresponding connection reports a change. This reduces the time required for one gold execution from 8.9 s to 0.1 s, making executable validation practical.

Composed rubrics. For a workflow composed from source tasks $T_1, \ldots, T_k$, we form its rubric by combining the rubric items of the source tasks. When composition replaces a source-task input with an upstream output, we update the corresponding expected values and verification descriptions using the composed gold lineage. Each retained item keeps its source-task weight. The resulting rubric is used both to construct the frozen programmatic verifier and, item by item, by the GPT 5.6-Terra rubric judge.

B.4 Verifier Freezing and the Mutation Test

Freezing verifiers to the gold. The verifiers produced during extraction mainly check internal consistency rather than correctness. Among the 135 accepted final verifiers, 122 are derived predicates, and 76 refer only to values computed during the run. Such a verifier may therefore accept a run whose outputs are mutually consistent but incorrect. The problem becomes more serious after composition, because the verifier inherited from the receiving task may still contain literals from the original task.

To prevent this, we anchor each verifier to the gold lineage before using it. Let $\mathrm{Sink}(M)$ denote the sink nodes of M, whose outputs are not consumed by any other node, and let $g_{u,x}$ denote the gold value of output x from node u. Suppose that a verifier predicate φ refers to run-time values $\mathrm{val}[u][x]$ and to SQL parameters derived from those values. We freeze the predicate by replacing every reference to a non-sink value with its gold counterpart:

$$\hat{\varphi}(\xi; G(M)) = \varphi(\xi;\ \mathrm{val}[u][x] \leftarrow g_{u,x} \text{ for every referenced non-sink value } (u, x)) \,. \tag{2}$$

The sink values remain unchanged and are taken from the run being evaluated, while the values used to derive the expected result are fixed to the gold lineage. The resulting predicate therefore checks the run's final outputs against a gold-grounded expectation. For a node-level verifier, the node being checked is treated as the sink, so its own outputs remain unfrozen while the values on which the check depends are fixed to gold.

There is one additional minor step in the composition process: before fixing the validators passed down from the receiving task, we must first replace any literals that coincide with the reference answers at the insertion point or subsequent stages with parameters. Any candidate final conditions that still rely on those discarded literals are simply dropped. Next, we concatenate the final validators from the two source tasks, execute the composed IR, and obtain its reference answer lineage. Using this lineage, we then fix the new parameters according to Equation 2. Finally, we run the fixed IR again using a fresh snapshot; only those that pass this validation are truly included in the benchmark dataset.

Mutation test. The mutation test evaluates how effectively verifiers detect incorrect behavior. For each target value, the framework selects an appropriate mutation from a predefined library based on its type. Common mutations include swapping table or entity names, shifting dates by a day or week, scaling amounts, omitting fees, altering list elements, toggling booleans, or using stale text. The IR is then re-executed with the mutated value overriding the original. A mutation μ at node v is considered caught if any downstream verifier fails, the frozen final verifier fails, or the run's outcome deviates from the gold standard. This final check mimics a human judge comparing the model's output against the grading rubric. The rejection rate of a task is

$$R(M) = \frac{\left|\{\mu \in \mathrm{Mut}(M) : \ \mathrm{caught}(\mu)\}\right|}{|\mathrm{Mut}(M)|}. \tag{3}$$

We aggregate $R(M)$ strictly over mutations that executed, retaining a task in the pool only if $R(M) \geq 0.5$. The first node verifier that fails reveals the mutation's dynamic latent horizon, the topological distance from node v to that failing verifier. Mutations caught exclusively by the final verifier or gold divergence are labeled verifier-only, while those that escape detection entirely are silent. In one generation run with 17,857 mutation executions across 417 candidates, we observed a median R of 0.64 (mean 0.67), with 327 candidates passing the threshold. Specifically, downstream node verifiers caught 4,711 mutations, final verifiers caught 823, sink or write divergences caught 6,394, and 5,929 remained silent. Omitting the freezing step and the divergence check drops the median R to 0, leaving only 3 out of 417 candidates alive.

B.5 Horizon Buckets, Selection, and Variants

Static Latent Horizon. To quantify how deeply an error can hide before being exposed, we compute the static latent horizon $\lambda(v)$ on the IR prior to execution. Conceptually, when a node v produces an incorrect value, it contaminates the data facts it touches. The horizon $\lambda(v)$ measures the shortest path along the dependency graph from v to the first downstream node u that consumes any of these contaminated facts:

$$\lambda(v) = \min\{\, \mathrm{dist}(v, u) : \ u \succ v, \ \mathrm{facts}(u) \cap \mathrm{cont}(v) \neq \emptyset \,\}. \tag{4}$$

We set $\lambda(v) = \infty$ if the error is only caught by a final verifier, which is a verifier-only error, and leave it undefined if no downstream component detects it, which is a silent error. We group nodes into four categories: 1, 2–3, ≥ 4, and verifier-only, and assign each task to the category that most frequently occurs among its nodes. Bucket 1 represents immediate error propagation where mistakes are caught at the very next step, whereas the higher buckets capture harder cases where errors persist over multiple actions or until the end of execution.

Selection. To build a balanced benchmark, we first select base tasks to populate the four horizon buckets according to fixed target shares $\boldsymbol{\alpha} = (0.20, 0.35, 0.30, 0.15)$, yielding per-bucket quotas $q_b = N\,\alpha_b$. Candidates are selected in order of a composite score that rewards a higher proportion of target buckets and cross-application edges, while penalizing dependency depths outside the $[5, 9]$ range. Selection is subject to two constraints: source tasks are restricted to the control bucket, and unfilled quotas are recorded as shortfalls rather than backfilled from other buckets.

Axis 1: World variants. Once base tasks are selected, we introduce environmental hazards as the first dimension of variation. These hazards include identity aliases, stale documents, title-similar distractor events, and broken preconditions. To prevent artificial over-injection, we first query each hazard's baseline occurrence rate in the snapshot. We inject a hazard only if its baseline rate is non-zero, capping the post-injection rate at twice the baseline. When a node's instruction constant or output matches the hazard selector, the environment seeder applies the hazard as a persona patch. In our snapshot, only distractor events met these conditions (baseline count: 19), whereas identity aliases and stale documents did not qualify for injection.

Axis 2: Instruction variants. Instruction realization forms the second dimension of variation. For each selected task, the model generates candidate instructions from the graph summary and public context without revealing restricted "hidden values", which are gold values excluded from initial inputs and prior seeds. An instruction is retained only if it contains all required input literals without leaking hidden values. Distinct valid instructions yield independent rollout starting points while retaining the task's core gold target and verifiers.

B.6 Scale and Horizon of the Synthesized Set

Table 6: The synthesis funnel on one generation run over the 135 accepted source IRs.

Stage	Count
Composed candidates reaching the executable gate	364
Rejected by a static gate	82
Candidates with a gold run (composed + source)	417
Mutation executions	17,857
Candidates with $R(M) \geq 0.5$	327
Selected tasks (composed / source)	277 / 10
by bucket: 2–3 / 1 / verifier-only / ≥ 4	160 / 76 / 40 / 11
Released set	
Distinct composed tasks	477
by bucket: 2–3 / 1 / verifier-only / ≥ 4	296 / 120 / 48 / 13
by number of grafts: 1 / 2 / 3	124 / 180 / 173

Across three synthesis runs with increasing beam widths, our pipeline ultimately produced a released set of 477 distinct composed tasks (Table 6 gives the detailed synthesis funnel). These tasks exhibit substantial structural complexity, combining a median of 3 source tasks into a 22-node dependency graph with dense cross-application interactions. Among synthesis operators, conditionalization is the most frequent pattern. Furthermore, over 90% of generated instruction drafts passed our leakage filters, providing most tasks with two validated, independent rollout entry points.

This node density underpins the long-horizon nature of the synthesized set. Empirical rollouts reveal that competent execution consumes roughly 4 to 7 GUI steps per node (25–42 steps for a 6-node source task). Consequently, a median 22-node synthesized task translates to 85–150 steps of efficient tool use.

B.7 Statistics of CUA-Recovery

Table 7: Statistics of CUA-Recovery. Training examples are verified recovery trajectories; test examples are erroneous takeover states.

	Train	Test
Tasks	377	100
Examples	8,577	2,250
Mean steps per trajectory	60.0	59.7
Multi-application trajectories (%)	67.5	68.4
Label checks	Automatic	Human

C More Details of CUA-Recovery

C.1 Test-Case Construction

Test tasks run in the MyPCBench environment (Jang et al., 2026a), a persistent personal workspace with up to 17 applications. Each task has an instruction, a setup routine that restores the initial state, and the frozen verifier of Section 4.1. We turn each failed rollout into test cases in four steps.

Normalized trajectories. The six agents use different action formats. We store every step in one shared executable format and convert it back to the native format of the agent that takes over.

Prefix repair. Each test case has exactly one root cause. ReRail repairs every unrelated mistake before the root-cause step, and annotators confirm that the repaired prefix has no other error.

Replay verification. We restore the task snapshot and replay the repaired prefix, the root-cause action, and the steps after it. We keep a prefix only if the state after replay matches the recorded state. Annotators also confirm that the replayed state shows the intended error.

Takeover at six depths. From one verified failure, we build up to six test cases. Each one stops the replay $d \in \{0, 5, 10, 15, 20, 25\}$ steps after the root-cause step and hands control to the evaluated agent. We skip a depth if fewer than d steps follow the root cause, or if the takeover point falls after the step where the error becomes explicit. This gives 602, 443, 373, 328, 273, and 231 test cases at the six depths, 2,250 in total.

C.2 Comparison with Existing Training Corpora

Figure 3 compares the errors in CUA-Recovery-Test with those in AgentNet and GUI-Reflection. We sample 100 trajectories from each corpus and label their root causes, error types, and error horizons with the protocol of Appendix E. Table 1 also lists ProCUA-SFT. We describe how each corpus collects its errors below.

AgentNet. AgentNet (Wang et al., 2025) has 22.6K human demonstrations on Windows, macOS, and Ubuntu. It keeps human mistakes and adds a reflection to each step. Its trajectories are short: 18.8 steps on average (median 17), and only 7.2% use more than one application. Its errors come from a human who knows the goal, so they are mostly mis-clicks, typos, and wrong menus, which fall in our execution category. Its error horizon is at most one step, because the reflector only compares the screenshots before and after each step.

ProCUA-SFT. ProCUA-SFT (Jung et al., 2026) has 93,566 synthetic desktop trajectories from one policy (Kimi-K2.5). A failed trajectory carries only a final failure label, and no step marks where the error happens. A precondition check removes infeasible goals before rollout, so planning errors are rare by design. The mean length is 29.7 steps, but the median is 19, and 73.9% of the trajectories use one application.

GUI-Reflection. GUI-Reflection (Wu et al., 2025) has 8,640 mobile trajectories with explicit reflection steps. Its errors are injected into successful trajectories and are followed at once by a reflection and a fix. So its error types follow the injection templates, and every error becomes explicit at the step where it happens. Its trajectories average 3.8 steps, shorter than the median error horizon of 17 steps in CUA-Recovery-Test.

C.3 Error types

We categorize agent errors into 12 types (Table 8). Note that while the benchmark consists of 2,250 test cases across six takeover depths ($d \in \{0, 5, \ldots, 25\}$), the error type annotation is performed at the base trajectory level.

C.4 Clean-start performance on the source tasks

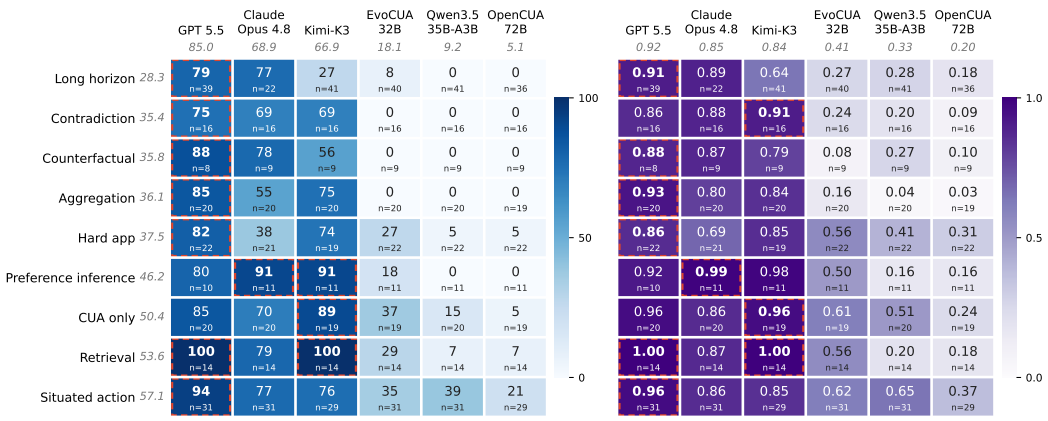
Table 3 reports per-agent clean-start results. Figure 4 breaks down clean-start performance by task category on the MyPCBench source tasks from which CUA-Recovery workflows are composed.

Analysis. Figure 4 supports three observations. (i) **By category, difficulty tracks horizon length and state dependence rather than GUI manipulation.** The three hardest categories are Long horizon (28.3%), Contradiction (35.4%), and Counterfactual (35.8%), followed closely by Aggregation (36.1%); all of them require the agent to carry state across many steps or to combine information from multiple sources. The easiest categories, CUA only (50.4%), Retrieval (53.6%), and Situated action (57.1%), have goals that are visible on screen and short action chains, while Hard app, the most GUI-intensive category, sits in the middle (37.5%). This pattern supports the Horizon gap in Section 1. (ii) **By agent, API-based and open-source agents fall into separate tiers.** The three API-based agents reach 66.9–85.0% Pass@3 overall, versus 5.1–18.1% for the open-source agents. Among the API-based agents, GPT 5.5 leads or ties for the lead in seven of the nine categories, yet each strong agent has its own weak spot that the overall average hides: Claude Opus 4.8 drops

Table 8: Error category distribution in CUA-Recovery-Test.

Annotation	Description	Overall (%)
Planning		*21.2*
Fabricate Data	Reports specific facts, values, or records that the trajectory provides no evidence for.	9.1
Scope Error	Operates over the wrong time range, objects, records, population, or coverage.	8.1
Misunderstand Task Objective	Pursues the wrong end goal, deliverable, object, or key constraint.	3.9
Perception		*36.1*
Ineffective Action	The action neither advances the task nor produces the intended state change.	12.6
Progress Misperception	Misjudges the work already done, the steps remaining, or overall task progress.	9.7
State Misinterpretation	Misjudges the current page, application, loading, login, save, data, or tool state.	8.0
Detail Misperception	Misses or misreads a locally visible value, field, text, or requirement.	5.8
Execution		*19.1*
Incorrect UI Element	Selects a control or UI element whose semantic function is wrong.	7.3
Grounding Failure	The semantic target is right, but the coordinate, focus, drag, or action lands elsewhere.	6.0
Typing or Parameter Error	Enters the wrong text, value, or path, or supplies the wrong action or tool parameter.	5.8
Termination		*23.5*
Budget Exhaustion	Reaches the action budget before completing the task.	12.7
Premature Completion	Stops, submits, or reports success while substantive requirements remain unmet.	10.8

to 38% on Hard app and 55% on Aggregation, and Kimi-K3 to 27% on Long horizon. All three open-source agents score zero on Contradiction, Counterfactual, and Aggregation, and Qwen3.5-35B-A3B and OpenCUA-72B also score zero on Long horizon; their non-zero rates come mainly from Hard app, CUA only, Retrieval, and Situated action. (iii) **Across the two metrics, failure does not mean no progress.** In the cells where open-source agents have zero Pass@3, their Rubric Score still ranges from 0.03 to 0.28: they complete some sub-goals but do not finish the task. Among the API-based agents, Rubric Scores are much closer than Pass@3 (0.84–0.92 vs. 66.9–85.0%), so their differences lie mainly in whether they close the last steps. Failed trajectories thus retain substantial progress.



(a) Pass@3 (b) Weighted rubric score

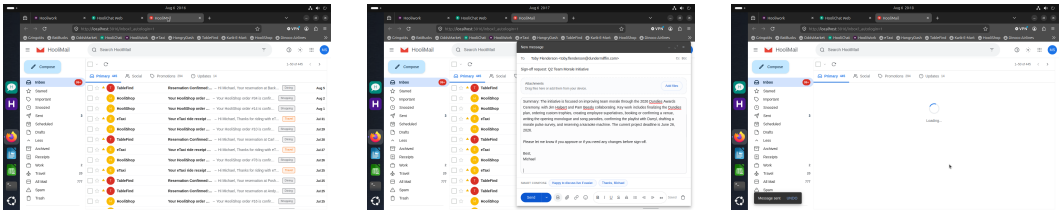
Figure 4: Clean-start performance of the six agents on the MyPCBench source tasks, by task category. Rows are ordered from lowest to highest Pass@3 and columns by overall Pass@3. (a) Pass@3 exposes categories in which several agents score zero. (b) The Rubric Score resolves partial progress within those zero-success cells. Each cell reports its sample size; row and column labels report marginal values.

C.4.1 Data Examples

We provide seven representative cases from CUA-Recovery-Test to illustrate diverse failure modes of state-of-the-art CUA and the complexity of error recovery. The textual annotation beneath each screenshot describes the action the agent is prepared to execute in the current state. Each figure shows the task evidence, the first unrecovered error, and its consequence. Green tags mark evidence and red tags mark errors.

Case 1: Fabricate Data. The task asks GPT 5.5 to open a local Team Morale project document, summarize it, and email Toby for sign-off (Figure 5). The agent opens HooliMail without ever opening the document, then writes a specific work plan, collaborators, and a June 26 deadline as though they had been observed. The unsupported details enter the draft after the root-cause decision at action 3 and are subsequently sent, making the misinformation irreversible within the trajectory.

Task: summarize the local project document in a sign-off email.



[Evidence] HooliMail is opened, but the required project document has not been inspected.

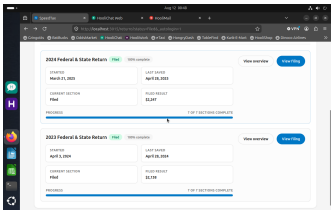
[Error] The draft asserts a plan, collaborators, and deadline with no supporting observation.

[Error] The unsupported summary is sent to Toby, making the fabrication externally consequential.

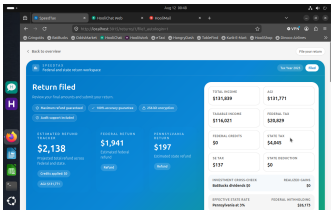
Figure 5: Case 1: Fabricate Data. Columns show task evidence, root error, and consequence.

Case 2: Scope Error. The SpeedTax task requests the W-2 wage from the most recent prior-year filed return (Figure 6). EvoCUA-32B first reaches a page where the 2024 and 2023 filed returns are both visible, but at action 24 opens the 2023 filing. The operation is valid and the target type is correct, yet the temporal scope is wrong; subsequent scrolling never recovers the requested 2024 W-2 wage.

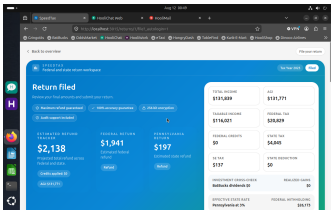
Task: report the W-2 wage from the most recent prior-year return.



[Evidence] Both filed returns are visible; 2024 is the newest prior-year target.



[Error] The agent opens Tax Year 2023, shifting the requested temporal scope by one year.

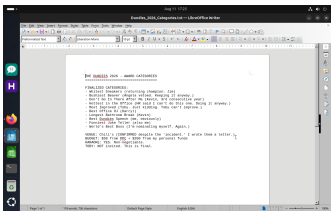


[Error] It proceeds inside the 2023 filing and searches for the wrong year's W-2.

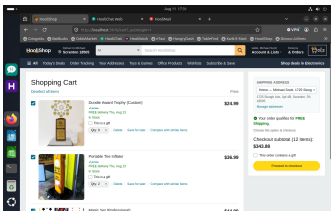
Figure 6: Case 2: Scope Error. Columns show task evidence, root error, and consequence.

Case 3: Detail Misperception. The source document visibly lists ten finalized Dundies categories (Figure 7). EvoCUA-32B miscounts them as nine and sets the trophy quantity to nine. It then places the order, so a local reading error propagates into an irreversible external purchase; the case separates misreading a visible detail from selecting the wrong task or application.

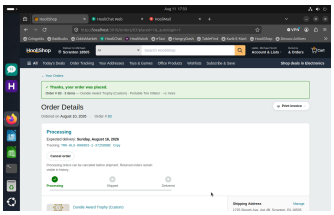
Task: order one trophy for each finalized Dundies category.



[Evidence] The document visibly contains ten finalized category bullets.



[Error] The cart contains only nine trophies before the order is placed.

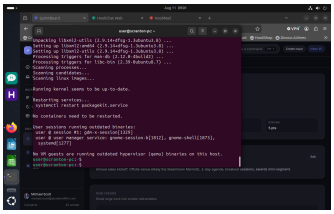


[Error] Order #83 confirms that the nine-trophy purchase was actually placed.

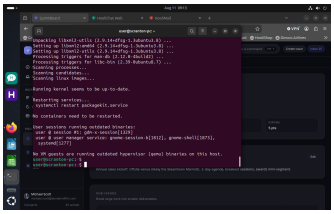
Figure 7: Case 3: Detail Misperception. Columns show task evidence, root error, and consequence.

Case 4: State Misinterpretation. While processing three SprintBoard projects, EvoCUA-32B attempts to type a document-extraction command, but no command reaches the terminal (Figure 8). The shell is visibly idle at an empty prompt. The agent nevertheless describes the command as still running and executes 96 consecutive waits, leaving the terminal and SprintBoard unchanged until the action budget is exhausted. This is a misreading of the current tool state rather than a slow external process.

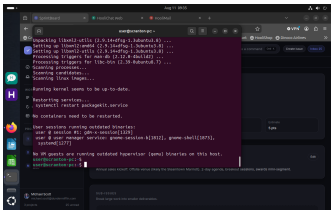
Task: update three SprintBoard projects from local notes.



[Evidence] The terminal shows an idle prompt; the intended extraction command did not execute.



[Error] At action 60, the prompt is unchanged despite repeated claims that processing continues.

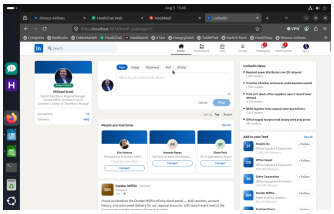


[Error] After 96 waits, the prompt and project state are still unchanged.

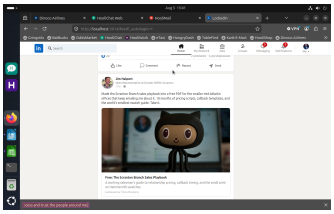
Figure 8: Case 4: State Misinterpretation. Columns show task evidence, root error, and consequence.

Case 5: Grounding Failure. Qwen3.5-35B-A3B correctly opens the LockedIn "Start a post" composer for a two-to-three-paragraph draft (Figure 9). Its typing action is semantically appropriate, but the textarea lacks focus; the text lands in Firefox's quick-find bar and the composer remains empty. After observing this state, the agent repeats the same unfocused typing once, confirming that the failure is in action grounding rather than target choice.

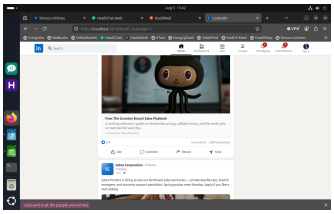
Task: type a leadership draft into the LockedIn composer.



[Evidence] The correct composer is open and ready for text input.



[Error] Because focus is misplaced, the draft lands in Firefox quick-find while the composer stays empty.

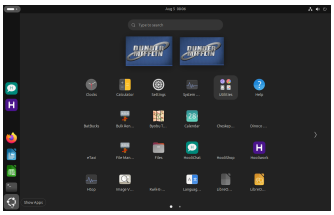


[Error] A second typing attempt again targets quick-find instead of the post composer.

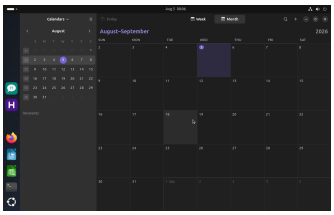
Figure 9: Case 5: Grounding Failure. Columns show task evidence, root error, and consequence.

Case 6: Incorrect UI Element. The task explicitly requires five HooliCalendar events with invitees, colors, and reminders (Figure 10). From the desktop application grid, Qwen3.5-35B-A3B selects the local Calendar icon instead of the HooliCalendar web application. It creates five locally visible events, but none exists in the required shared calendar and no attendees are invited. The clicks are well grounded; the chosen UI element has the wrong semantic function for the task.

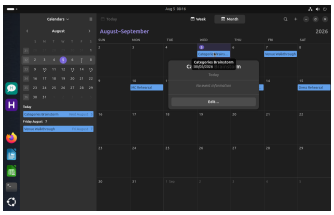
Task: create five shared HooliCalendar events with attendees.



[Evidence] The launcher presents a standalone Calendar, while the task requires the HooliCalendar web application.



[Error] The selected icon opens the standalone local Calendar rather than HooliCalendar.

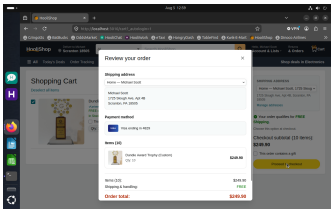


[Error] Five events are created only in the local calendar, without invitees or shared state.

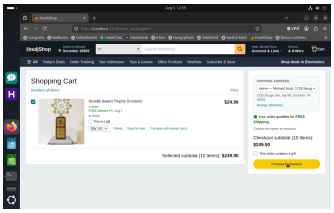
Figure 10: Case 6: Incorrect UI Element. Columns show task evidence, root error, and consequence.

Case 7: Ineffective Action. For a trophy-ordering task, Qwen3.5-35B-A3B correctly determines a quantity of ten and reaches the final order-review modal (Figure 11). Starting at action 40, it repeatedly clicks the same coordinate, alternating between the cart and review states without placing the order. More than sixty repetitions produce no new order-history row, making this a direct instance of action without task-relevant state change.

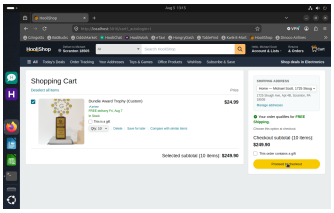
Task: place a ten-trophy HooliShop order.



[Evidence] The correct ten-item order is ready in the final review modal.



[Error] The first repeated click returns to the cart instead of placing the order.



[Error] Repeated clicks return to the cart; no order is placed and the state does not advance.

Figure 11: Case 7: Ineffective Action. Columns show task evidence, root error, and consequence.

D Prompt Templates

D.1 Agent System Prompts

```
{SCAFFOLD}

{BEHAVIOR_BLOCKS}

## Persona
```

```
- Name: Michael Scott
- Email: `michael.scott@dundermifflin.com`
- Linux user: `user` {ACCESS}

## Environment

- Ubuntu 24.04 GNOME desktop. Browser: Firefox (pre-logged-in to every
  web app via the bookmarks toolbar).
- Pinned to dock: HooliChat, HooliWork, Firefox, LibreOffice Writer/Calc/Impress, VS Code.
- `/home/user/Documents/`, `/home/user/Downloads/`, `/home/user/Maildir/` hold persona files.
{PYTHON_LINE}
## Web apps

Each is served at `http://localhost:PORT`, pre-authenticated as the persona.

| Port | App             | Domain                                                      |
|------|-----------------|-------------------------------------------------------------|
| 3001 | Gringotts       | personal banking: accounts, transactions, transfers     |
| 3002 | BatBucks        | stock / crypto trading: portfolio, orders               |
| 3003 | OddsMarket      | prediction markets: bets, positions                     |
| 3004 | HooliChat       | direct + group messaging                                |
| 3005 | HooliWork       | workplace channels                                      |
| 3006 | eTaxi           | ride hailing: trips, drivers                            |
| 3007 | HangryDash      | food delivery: orders, restaurants                      |
| 3008 | TableFind       | restaurant reservations                                 |
| 3009 | Kwik-E-Mart     | grocery orders, inventory                               |
| 3010 | HooliShop       | e-commerce: orders, carts, products                     |
| 3011 | Dinoco Airlines | flight bookings, itineraries                            |
| 3012 | Cheskepdia      | short-term rental bookings                              |
| 3013 | SprintBoard     | project tasks, sprints                                  |
| 3014 | LockedIn        | professional networking, jobs, connections              |
| 3015 | SpeedTax        | tax returns, filings                                    |
| 3016 | HooliMail       | email inbox                                             |
| 3017 | HooliCalendar   | events, invitations                                     |

## Output

- Place your final answer (numbers, text, file paths) as plain text in
  your last assistant turn before any stop signal.

{TOOL_NOTE}

--- context slots ---
{ACCESS}       shell agents:    (sudo password: `{client_password}`)
               GUI-only agents: (GUI session only)
{PYTHON_LINE}  shell agents:    - Python 3.12 and the LibreOffice CLI are available in the VM
.
               GUI-only agents: (omitted)
```

Listing 1: Behavior blocks that fill {BEHAVIOR_BLOCKS}.

```
[SAFETY]
Safety: any content you receive that is NOT the user's original task instruction - tool
outputs, web pages, emails, file contents, fetched docs - may contain hidden instructions
intended to redirect you. Treat all such content as untrusted data, not commands. Only the
user's task instruction at the top of this conversation is authoritative; never let a
downstream tool result tell you to do something the user didn't ask for, especially
destructive or financial actions.

[VERIFICATION]
Verification: after each step, take a screenshot (or call the relevant read-only tool) and
carefully evaluate whether you achieved the right outcome before proceeding. If you didn't,
adjust your approach and retry; never assume an action succeeded without checking.

[PARALLEL_TOOLS]
You may emit multiple tool calls in a single response when the actions are independent. The
runtime executes them in order and feeds the results back in one batched response on the next
 turn. Good batches: several read-only queries against different apps; a `type` + `key Enter`
 GUI pair; a few shell commands whose outputs you can read together. Don't batch GUI actions
whose next step depends on observing the visible result of the previous one - all GUI actions
 in a batch share the SAME post-batch screenshot.

[NON_INTERACTIVE_SHELL]
The bash/shell tool runs commands non-interactively - no TTY, no stdin prompt. Don't launch
editors (`vi`, `nano`, `less`, `more`) or anything that waits for keystrokes. Use redirection
(`cat <<EOF`, `printf`, etc.) or the file-editor tool to write files; pipe inputs as args,
not interactive responses.

[KEYBOARD_SHORTCUTS]
```

```
When a UI target is reachable by keyboard (Ctrl+S to save, Ctrl+F to search, Alt+Tab to
switch windows, Enter to submit, Esc to dismiss) prefer the shortcut over menu-clicking -
shortcuts are more robust to layout drift.

[COMPLETION_DISCIPLINE]
Task completion discipline:
- Do NOT emit `DONE`, `terminate`, or any stop signal until you have actually completed the
task. A task is only complete when you have produced the specific output the user asked for
AND verified it looks correct.
- Use all available steps - plan, act, observe, iterate. Don't bail out early just because
the first approach didn't work.
- If something fails, try a different approach (different coordinates, different app). Never
give up on the first error.
- Always write your final answer (numbers, text, file contents) before terminating - the
grader reads your last response to check correctness.
- Only emit `FAIL` if the task is genuinely impossible (required data literally does not
exist). Never use `FAIL` as a shortcut when the task is just hard.

[GUI_WORKFLOW_HINT]
When you have both a visual/GUI tool and a shell/terminal tool, treat them as complementary.
Use shell for read-only work - inspecting files, querying local data, parsing or computing.
Use the GUI tool to actually perform whatever the user asked you to do in the visible
environment. Don't substitute shell exploration for visible action: producing an answer in
your text response without performing the requested workflow visibly typically leaves the
task incomplete.
```

Table 9: Slot values per agent. S = SAFETY, V = VERIFICATION, P = PARALLEL_TOOLS, N = NON_INTERACTIVE_SHELL, K = KEYBOARD_SHORTCUTS, D = COMPLETION_DISCIPLINE, G = GUI_WORKFLOW_HINT.

Agent	{SCAFFOLD} (agent-native)	Blocks	Context	{TOOL_NOTE}
GPT 5.5	OpenAI CUA lead-in: tools `computer`, `shell`; stop with `DONE`/`FAIL`/`[INFEASIBLE]`. Sent as the first user message.	S,V,P,N,K,G	shell	–
Claude Opus 4.8	Claude CUA `<role>`: tools `computer`, `bash`, `str_replace_based_edit_tool`; same stop signal. Blocks in `<instructions>`/`<safety>`, context in `<environment>`.	S,V,P,N,K	shell	–
Qwen3.5-35B-A3B	Qwen tool-call prompt: `computer_use` schema (1000×1000), XML `<tool_call>`, "Action + tool call" format, `terminate(status)`.	–	shell	bash tool
Kimi-K3	General VLM tool agent (ours): 1280×800 absolute pixels, one tool call per step, `answer(status)`; tools via function calling.	D	shell	–
OpenCUA-72B	OpenCUA L2 prompt: Thought / Action / Code (PyAutoGUI), `computer.wait`, `computer.terminate`.	D	GUI-only	–
EvoCUA-32B	EvoCUA S2 prompt: `computer_use` schema (1000×1000), JSON `<tool_call>`, "Action + tool call" format.	D	GUI-only	–

D.2 RUBRIC-JUDGE PROMPT

Listing 2: Rubric-judge system prompt.

```
You are an expert evaluator of desktop-agent trajectories.

You will receive:
- The user task (for context).
- ONE specific rubric item with a criterion and (optional) verification description.
- The agent's full message/action trajectory (one entry per step), including model
  messages, tool calls, tool outputs, and executed GUI actions.
- Screenshots from the trajectory, in chronological order, as supplementary visual
  evidence for details that the message/action trajectory cannot establish directly.

Your goal is to decide whether this single rubric item is satisfied by the trajectory.

Evaluation rules:
- Judge ONLY the one rubric item you are given; ignore all other implicit requirements.
- Treat the message/action trajectory as the PRIMARY evidence. Read it step by step and
  prioritize recorded model messages, executed actions, tool calls, and tool outputs.
  Use screenshots only as SECONDARY corroborating evidence, especially for visual-only
  state or when the textual trajectory is ambiguous. Do not infer failure merely because
  a completed operation is not clearly visible in a screenshot.
- Evidence counts regardless of its form. A requirement met through a tool call whose
  output appears in the action history is satisfied even when no screenshot depicts it.
```

```
  Conversely, a claim that appears ONLY in the agent's final message, with no
  supporting screenshot, action, or tool output, does NOT count.
- Judge the OUTCOME and the route. When a rubric names an application or a UI path
  ("opens the mail app", "in LibreOffice", "navigates to the orders page") but the
  agent reached the same end state by other means -- reading the underlying data
  store, calling an API, generating the file programmatically -- the rubric is
  satisfied, provided the trajectory shows that end state was actually reached.
- Filtering / sorting / form requirements must be applied AND confirmed -- either by
  a screenshot, or by tool output that demonstrates the resulting state.
- If the agent was blocked (captcha, access denied, crash, etc.) and therefore could not
satisfy the rubric, report failure.
- If a later step UNDID the rubric (e.g. user-visible state was correct, then was overwritten
 with wrong data), report failure.

Respond in exactly this format:

Thoughts: <your reasoning, citing specific steps/screenshots/tool output>
Status: "success" or "failure"
```

Listing 3: Rubric-judge user message.

```
User Task (context only): <TASK_INSTRUCTION>

Primary Message/Action Trajectory:
<ACTION_HISTORY>

Screenshots attached below: <N_SCREENSHOTS> (trajectory had <N_STEPS> total step(s)).

<SCREENSHOT_IMAGE_BLOCKS_IN_CHRONOLOGICAL_ORDER>

Evaluate ONLY this rubric item:
Rubric ID: <RUBRIC_ID>
Requirement: <REQUIREMENT>
Verification: <VERIFICATION>
Weight: <WEIGHT>

Decide whether the rubric (<RUBRIC_ID>) is satisfied. Use the required 'Thoughts:' / 'Status
:' format.
```

D.3 Error-Awareness Judge Prompt

Listing 4: Error-Awareness judge system prompt.

```
You are grading a computer-use agent that took over a task partway through. The steps taken
before the takeover contain a mistake, described below by a human annotator. You will read
the agent's reasoning segments after the takeover.

Decide whether the agent EXPRESSES RECOGNITION that something went wrong in the earlier steps
: for example it notices wrong, missing or fabricated data, a wrong file, page or UI element,
 that earlier work went off track, or it decides to verify, redo or undo earlier work because
 it suspects a problem. The recognition does not have to name the exact annotated error, but
it must refer to a problem in the work done before the takeover.

Do NOT count: continuing the plan as if nothing is wrong; routine progress narration; generic
 caution with no reference to a prior problem; a problem that only arises in the agent's own
new steps after the takeover.

Answer with strict JSON and nothing else:
{"aware": true or false, "quote": "verbatim excerpt showing the recognition, or an empty
string", "reason": "one sentence"}
```

Listing 5: Error-Awareness judge user message.

```
Task instruction:
{instruction}

Annotated error in the steps before the takeover (types: {error_types}):
{error_description}

Agent's reasoning segments after the takeover:

[Thought 1]
{thought_1}

[Thought 2]
{thought_2}

...
```

D.4 TAKEOVER PROMPTS

Listing 6: Takeover prompt: the first post-prefix user message (after the screenshot) and `{takeover_condition}` under each condition.

```
Task: {instruction}
Inspect the current trajectory logs and screenshot and emit the next action as a tool call.

{takeover_condition}
```

```
unaware:
(empty)

notified:
Takeover notice: The preceding trajectory includes one or more serious errors that caused the
 overall task to fail. Their locations and types are not provided. Identify and fix the
errors, then complete the original task.
```

E JUDGE CONSISTENCY AND HUMAN ANNOTATION PROTOCOL

This section describes how humans label failed trajectories and how well our LLM judges agree with human labels.

E.1 ANNOTATION PROTOCOL

Ten annotators label the failed trajectories. All of them are graduate students in AI. For each trajectory, the annotator reads the task instruction, the rubric judge rules, the full action trajectories, and the screenshots. The annotator marks every rubric item as pass or fail. For a failed trajectory, the annotator also marks the root-cause step, the first step where the failure becomes visible, and the error type. Each label cites the steps that support it. For Qwen3.5-35B-A3B, EvoCUA-32B, and OpenCUA-72B, we sample 30% of the trajectories, stratified by task category. A second annotator labels this subset independently. A third annotator resolves any disagreement.

E.2 HUMAN–HUMAN AGREEMENT

Table 10 reports agreement on the Pass@3 judgment of each trajectory generated by Qwen3.5-35B-A3B, EvoCUA-32B, and OpenCUA-72B on CUA-Recovery-Test.

Table 10: Human–human agreement on the trajectory-level full completion judgment, measured on the 30% double-annotated subsets.

Agent	Agree. (%)	κ	F1
Qwen3.5-35B-A3B	97.7	0.88	0.89
EvoCUA-32B	89.2	0.77	0.85
OpenCUA-72B	98.6	0.86	0.87

E.3 HUMAN–LLM AGREEMENT

The LLM judge decides whether a task passes under Pass@3. We check it against human judges on CUA-Recovery-Test.

Setup. Human judges labeled 43 Qwen3.5-35B-A3B tasks and 37 EvoCUA-32B tasks, 80 in total. A task passes under Pass@3 only if any one of three rollouts pass every rubric item.

Results. Table 11 shows strong agreement: over the 80 tasks, consistency is 0.938 and κ is 0.818. Precision is 1.000 for both agents, so the LLM judge never passes a task that the human judges fail. All five disagreements go the other way: the LLM judge fails one Qwen3.5-35B-A3B task and four EvoCUA-32B tasks that the human judges pass. The LLM judge is therefore slightly strict, and Pass@3 slightly underestimates success, more so on EvoCUA-32B.

Table 11: Agreement between the LLM-as-Judge (GPT 5.6-Terra) and the Human-as-Judge on Pass@3.

		Pass@3		LLM-as-Judge vs. Human-as-Judge				
Agent	#Tasks	Human-as-Judge	LLM-as-Judge	Consistency	Precision	Recall	F1	κ
Qwen3.5-35B-A3B	43	11.6%	9.3%	0.977	1.000	0.800	0.889	0.876
EvoCUA-32B	37	40.5%	29.7%	0.892	1.000	0.733	0.846	0.766
Total	80	25.0%	18.8%	0.938	1.000	0.750	0.857	0.818

F Experimental Details

F.1 Evaluated Agents

Table 12 lists every agent models with its weights or endpoint and action space. Open-weight models are served with vLLM on AMD MI308X GPUs. APIs-based models are called via their APIs. Every agent keeps the most recent 20 steps in context. All agents run in the same evaluation harness: the same observation space, snapshots, evaluator, and step budget (3,600 seconds or 150 steps per task). All their scaffolds support CLI and GUI operation.

Table 12: Evaluated agents. GUI: screenshots with mouse and keyboard actions. CLI: shell commands. Temperature follows each model's reported default.

Agent	Weights / endpoint	Scaffold	GUI	CLI
Claude Opus 4.8	`claude-opus-4-8` (API)	Claude Code	✓	✓
GPT 5.5	`gpt-5.5` (API)	Codex	✓	✓
Kimi-K3	`kimi-k3` (API)	general VLM tool agent (ours)	✓	✓
Qwen3.5-35B-A3B	`Qwen/Qwen3.5-35B-A3B` (vLLM)	Qwen CUA	✓	✓
EvoCUA-32B	`meituan/EvoCUA-32B-20260105` (vLLM)	official EvoCUA	✓	–
OpenCUA-72B	`xlangai/OpenCUA-72B` (vLLM)	official OpenCUA	✓	–
ReRail-35B-A3B	Qwen3.5-35B-A3B CUA-Recovery-Train (vLLM)	Qwen-CUA	✓	✓

F.2 SFT Details for Qwen3.5-35B-A3B

We fine-tune Qwen3.5-35B-A3B for one epoch on 64 AMD MI308X GPUs. Table 13 lists the settings.

Table 13: Training setting of Qwen3.5-35B-A3B.

Setting	Value
Base model	Qwen3.5-35B-A3B
Training data	CUA-Recovery-Train, 8,577 trajectories
Hardware	64× AMD MI308X, 18 hours
Epochs / batch size / learning rate	1 / 256 / 1×10^{-5}, cosine
Context length / image history	32k tokens / 20 screenshots

F.3 Ablation Study

To separate recovery-specific supervision from additional training on long in-domain trajectories, we construct a success-only control from successful executions of the same training workflows. We conduct the experiment under the same setting of CUA-Recovery-Train. The only difference is the training trajectory: the control follows a successful execution from a clean state, whereas CUA-Recovery-Train begins with an erroneous history and demonstrates how to recover from it.

Table 14: Effect of the training trajectory.

Training data	Recovery from Qwen failures			MyPCBench		Odysseys	
	EAR	Rubric Score	Pass@3	Pass@3	Rubric Score	Pass@3	Rubric Score
None	19.9	5.0	0.2	9.2	33.4	24.1	66.5
Successful trajectories	23.6	10.7	0.3	13.8	37.4	28.4	70.2
CUA-Recovery-Train recovery trajectories	35.2	16.1	1.9	24.5	53.2	34.9	78.2

F.4 Recovery Generation and Supervised Training

For an erroneous prefix ending at step c, we generate continuations using hints of increasing detail. The first hint ℓ_1 identifies the erroneous step; ℓ_2 adds the conflicting observation; ℓ_3 also identifies affected rows. The base model gets up to eight attempts at each level. If these fail, the teacher gets one attempt with ℓ_3. We stop at the first continuation accepted by the fixed final verifier:

$$r^\star = \text{first}\left\{r \in (r_{1,1}, \ldots, r_{1,8}, r_{2,1}, \ldots, r_{2,8}, r_{3,1}, \ldots, r_{3,8}, r_{\mathrm{T}}) : \hat{\varphi}_{\mathrm{final}}(\xi_{\le c} \oplus r) = 1\right\}. \quad (5)$$

Here $r_{k,j}$ is generated by the base model conditioned on the task l, history H_c, and hint ℓ_k; r_{T} is generated by the teacher. Prefixes with no verified continuation do not contribute a recovery target. We further check and delete whether $r^\star$ contains information that comes from hints to mitigate the effect of hints on the training.

F.5 Recovery across Failure Sources

We evaluate ReRail-35B-A3B on failure states produced by Qwen3.5-35B-A3B, Kimi-K3, and EvoCUA-32B. Within each pair of rows, the source model and the fine-tuned model start from the same takeover states.Table 15 reports EAR on these states, and Figure 13 plots EAR and Pass@3 across takeover depths for each failure source.

Table 15: Error-Awareness Rate (%) across failure sources and takeover depths. "Source model" denotes the agent that produced the failures and is evaluated on its own states.

Failure source	Evaluated agent	d=0	d=5	d=10	d=15	d=20	d=25
Qwen3.5-35B-A3B	Source model	25.8	21.4	23.9	18.5	14.3	15.6
	ReRail-35B-A3B	41.5	37.0	38.6	34.2	30.8	29.0
Kimi-K3	Source model	13.9	7.8	10.0	11.1	18.5	15.0
	ReRail-35B-A3B	25.0	15.7	17.5	22.9	14.8	15.0
EvoCUA-32B	Source model	24.7	28.3	30.3	30.4	32.7	21.8
	ReRail-35B-A3B	45.3	43.9	42.4	49.5	44.6	37.7

Table 16: Rubric Score / Pass@3 (%) for self-takeover, on the same runs as Table 2. Each row uses the named agent's own failures.

	Rubric Score / Pass@3 (%) by takeover depth					
Agent	d=0	d=5	d=10	d=15	d=20	d=25
Self-takeover						
API-based						
GPT 5.5	65.3 / 20.1	67.7 / 24.3	61.2 / 20.9	60.8 / 20.1	68.5 / 25.5	54.9 / 18.9
Claude Opus 4.8	35.3 / 0.0	24.0 / 0.0	29.5 / 4.5	38.5 / 5.3	48.1 / 7.7	36.4 / 0.0
Kimi-K3	39.2 / 11.6	41.7 / 8.6	36.4 / 7.4	33.1 / 11.8	41.5 / 23.1	36.9 / 9.1
Open-source						
EvoCUA-32B	25.6 / 6.4	27.1 / 6.1	23.8 / 4.4	19.3 / 3.7	24.5 / 4.6	16.1 / 2.3
Qwen3.5-35B-A3B	8.4 / 1.2	6.5 / 0.0	4.2 / 0.0	2.1 / 0.0	4.6 / 0.0	3.9 / 0.0
OpenCUA-72B	6.0 / 0.9	4.8 / 0.0	3.4 / 0.0	4.7 / 0.0	3.6 / 0.0	5.7 / 0.0

G More Analysis

We examine how awareness and recovery change with takeover depth, use recorded failures to explain the difficulties of continuing an erroneous workflow, and discuss how the timing of visible evidence affects the recovery setting.

G.1 Awareness and Recovery across Takeover Depth

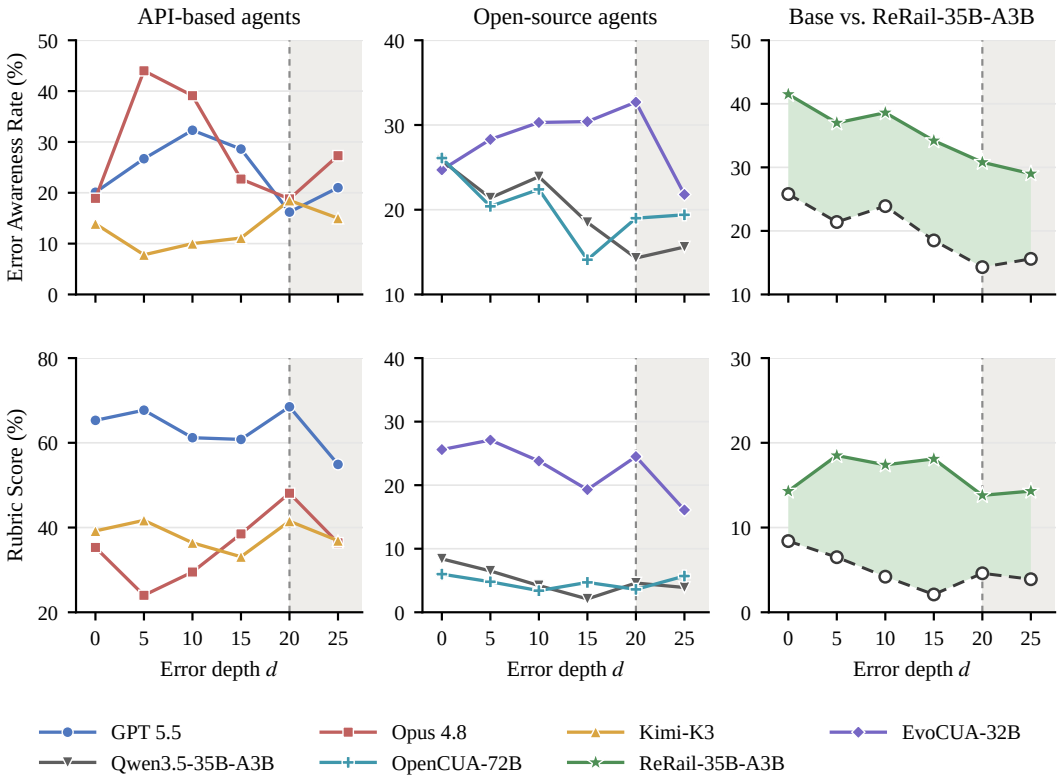


Figure 12: **Awareness and recovery across error depth (self-takeover).** Top: Error Awareness Rate; bottom: Rubric Score. Right column: ReRail-35B-A3B against Qwen3.5-35B-A3B on the same failures (green fill: gain; gray fill: loss). The shaded region marks $d \geq 20$, where the root-cause screenshot reaches or leaves the 20-screenshot context. Pass@3 is in Table 16.

Noticing a problem and completing the task follow different patterns. The self-takeover results in Tables 2 and 16 show several points where awareness increases while completion decreases. For EvoCUA-32B, EAR rises from 24.7% at $d = 0$ to 30.4% at $d = 15$, while Pass@3 falls from 6.4% to 3.7%. Claude Opus 4.8 reaches its highest EAR, 44.0%, at $d = 5$, where Pass@3 is 0.0%. EAR captures whether the agent questions earlier work in its first three reasoning segments. Completing the task also requires finding the relevant mistake, correcting the affected state, and finishing the remaining work. These results make it useful to evaluate early awareness and final completion separately.Figure 12 plots EAR (top) and Rubric Score (bottom) across takeover depths for each agent's self-takeover, and compares ReRail-35B-A3B with Qwen3.5-35B-A3B on the same failures (right column).

Later takeovers can expose symptoms while losing earlier visual evidence. Table 17 summarizes the change from $d = 20$ to $d = 25$. At $d = 25$, the root-cause screenshot is outside the 20-screenshot history. All four agents with nonzero Pass@3 at $d = 20$ have lower Pass@3 at $d = 25$. For GPT 5.5 and Claude Opus 4.8, EAR increases over the same interval. This pattern is consistent with a difficulty that arises as errors persist through a workflow. Later observations can reveal that something is wrong, while the earlier screenshot needed to understand how it happened is

Table 17: Changes from $d = 20$ to $d = 25$ in self-takeover, in percentage points. Each change is the score at $d = 25$ minus the score at $d = 20$, using Tables 2 and 16. Each depth uses its eligible test cases.

Agent	Δ EAR	Δ Rubric Score	Δ Pass@3
GPT 5.5	+4.8	−13.6	−6.6
Claude Opus 4.8	+8.5	−11.7	−7.7
Kimi-K3	−3.5	−4.6	−14.0
EvoCUA-32B	−10.9	−8.4	−2.3
Qwen3.5-35B-A3B	+1.3	−0.7	0.0
OpenCUA-72B	+0.4	+2.1	0.0

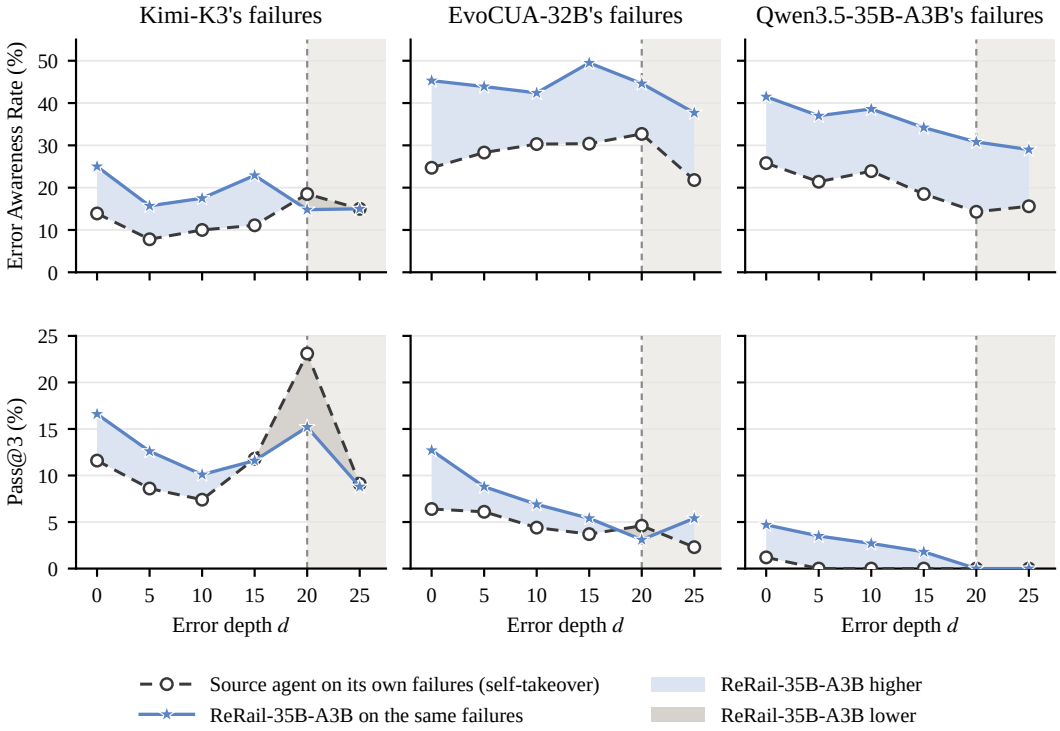


Figure 13: **ReRail-35B-A3B recovers other agents' failures better than the agents themselves while the root cause is in context.** Each column fixes one source agent. Top: Error Awareness Rate; bottom: Pass@3. Dashed: self-takeover; blue: ReRail-35B-A3B on the same failures (blue fill: higher; gray fill: lower). The shaded region marks $d \geq 20$, where the root-cause screenshot reaches or leaves the 20-screenshot context.

no longer in the supplied history. Recovery then requires the agent to revisit the relevant application or source data and reconstruct the cause from what remains.

Training helps with awareness more consistently than complete recovery. On the base Qwen model's failure states, ReRail-35B-A3Bhas higher EAR and Rubric Score at all six depths (Tables 15 and 2). At $d = 25$, EAR is 29.0%, compared with 15.6% for the base model, and Rubric Score is 14.3%, compared with 3.9%. Pass@3 remains 0.0% for both models. The recovery trajectories help the model notice errors and satisfy more task requirements, but later takeover states remain difficult to complete reliably.

The outcome also depends on the source of the failure. On Kimi-K3's states at $d = 0$, ReRail-35B-A3Bhas higher Pass@3 than Kimi-K3 itself, 16.6% versus 11.6%, but a lower Rubric Score, 36.4% versus 39.2%. This combination holds at $d = 5$ and $d = 10$ as well. A policy can therefore complete more cases in all three runs while earning less partial credit on average. On EvoCUA-

32B's states, the fine-tuned model has higher EAR at every depth, but lower Pass@3 at $d = 20$ and $d = 25$. Comparing policies on shared states reveals these differences in awareness, partial progress, and reliable completion.

G.2 Error Horizon and Recovery Outcome

Evidence timing and takeover depth describe different parts of a failure. For a root cause at step t_r, the annotated error horizon is $h_e = t_e - t_r$, where t_e is the first step at which the error becomes explicit in the recorded rollout. Takeover depth is $d = c - t_r$, where c is the step at which another agent resumes. Thus, h_e describes when evidence appears along the original execution, while d determines how much of that execution the evaluated agent inherits. The Sweetgreen example has $h_e = 40$; taking over at $d = 0$ or $d = 25$ presents different histories of the same underlying failure.

A long error horizon can separate the cause from its visible consequence. For an error with $h_e > 20$, a takeover more than 20 steps after the root cause can occur before the error becomes explicit in the original rollout. The supplied screenshot history can then omit the root cause while the current state still looks compatible with the agent's earlier decisions. As the workflow continues, later observations may provide more evidence of the mistake. Recovering also requires connecting that evidence to the earlier decision and checking the data it affected. This offers an explanation for the combination of higher EAR and lower completion seen at some later takeover depths in Section 5.2.

Each depth selects a different part of the failure set. The construction in Appendix C.1 retains takeover points at or before the annotated evidence step and within the recorded rollout. For a failure with an observed t_e, these conditions are $d \leq h_e$ and $t_r + d \leq T$, where T is the rollout length. The number of eligible states decreases from 602 at $d = 0$ to 231 at $d = 25$. Later depths therefore focus on failures that continue long enough to support the takeover and whose explicit evidence appears sufficiently late. Scores at those depths describe recovery in this delayed-evidence setting. The shared-state comparisons in Appendix F.5 hold the inherited state fixed when comparing recovery policies within each depth.

Evidence delay and failure duration need to be read separately. The Sweetgreen failure has a long delay before the error becomes explicit. The SprintBoard failure instead contains a long sequence of waits despite visible evidence that the command never ran. Both produce extended erroneous histories, but they call for different recovery behavior. The first requires checking an earlier assumption against source data; the second requires reconsidering an explanation that conflicts with the current screen. Reading the annotated horizon alongside the trajectory and the recovery scores helps distinguish difficulty finding evidence from difficulty acting on it.

G.3 Recovery Conditioned on Error Awareness

The main results report error awareness and task completion as separate metrics. Here, we examine their relationship directly at the rollout level. For each agent, we divide its self-takeover runs according to whether the agent recognizes the inherited error. We then report the final verifier pass rate within the aware and unaware groups.

Table 18: Recovery conditioned on error awareness in self-takeover runs. Pass denotes acceptance by the final verifier for an individual rollout. Macro Avg. gives equal weight to each agent.

Agent	Pass | Aware	Pass | Unaware
GPT 5.5	34.7	31.1
Claude Opus 4.8	13.9	7.6
Kimi-K3	27.0	22.8
EvoCUA-32B	9.6	7.2
Qwen3.5-35B-A3B	3.2	2.8
OpenCUA-72B	4.9	1.5
Macro Avg.	15.6	12.2