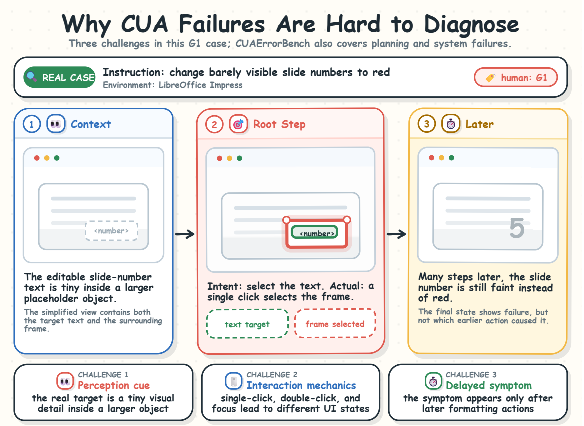
CUADebug: Diagnosing and Repairing Computer-Use Agent Failures

Anonymous ACL submission

Abstract

Computer-use agents (CUAs) operate real desktop and web interfaces through screenshots, mouse and keyboard actions, and stateful UI feedback, yet their failures remain difficult to diagnose and repair. Unlike text-only agents, CUA failures arise from coupled visual perception, spatial grounding, low-level interaction, task reasoning, and environment dynamics, making debugging a distinctive multimodal causal localization problem. We introduce **CUADebug**, a framework for diagnosing and repairing CUA failures. CUADebug includes a CUA-specific error taxonomy, CUAErrorBench, a human-annotated OSWorld failure benchmark, and CUADebugger, a tool-augmented debugger. Instead of prompting over the full trajectory once, CUADebugger actively inspects suspicious steps with paired before/after screenshots and action traces, then submits a structured diagnosis containing the root-cause step, error type, grounded evidence, and corrective strategy for re-execution. Human annotations over 184 failed trajectories show that task reasoning and control is the largest failure family (110/184), while perception and grounding/interaction account for another 61 cases. On the main Claude-agent split, CUADebugger improves joint subtype-and-step diagnosis from 11.2% to 19.6% with Gemini 2.5 Pro and improves consistently across debugger backbones. In single re-execution package evaluation, RCA-based conditions achieve higher task completion than history-only continuation (28.47% with machine RCA and 29.90% with our method, versus 13.89%); in continual re-execution, our method improves success from 12.2% to 25.86%, while human RCA reaches 29.21%. These results show that CUA root-cause diagnosis can provide actionable repair signals rather than merely post-hoc explanations.



Instruction: change barely visible slide numbers to red
Environment: LibreOffice Impress

The editable slide-number text is tiny inside a larger placeholder object.
The simplified view contains both the target text and the surrounding frame.

Intent: select the text. Actual: a single click selects the frame.

Many steps later, the slide number is still faint instead of red.
The final state shows failure, but not which earlier action caused it.

the real target is a tiny visual detail inside a larger object

single-click, double-click, and focus lead to different UI states

the symptom appears only after later formatting actions

Figure 1: Teaser example of an early UI-selection mistake causing later formatting failure.

1 Introduction

Computer-use agents (CUA agents) are a natural interface for applying language and vision models to real software, because they can operate desktop, web, and mobile GUIs through screenshots and mouse or keyboard actions. Compared with text-only or tool-only agents, CUA agents must jointly solve visual perception, spatial grounding, low-level interaction mechanics, and long-horizon state tracking. These coupled requirements make failures difficult to diagnose: a final wrong file, chart, or UI setting may only be the visible symptom of an earlier perception, grounding, reasoning, or environment error. Benchmarks such as OSWorld (Xie et al., 2024), WebArena (Zhou et al., 2024), VisualWebArena (Koh et al., 2024), Mind2Web (Deng et al., 2023), AndroidWorld (Rawles et al., 2025), and Windows Agent Arena (Bonatti et al., 2024) make end-to-end success measurable, but they do not by themselves explain where failed CUA trajectories go wrong.

Existing agent evaluation pipelines are therefore not sufficient for CUA debugging. End-to-end task success only tells us whether an agent finished

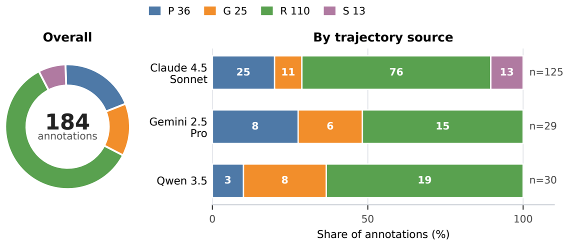


Figure 2: CUAErrorBench annotation statistics by source.

a task, not why it failed. Error taxonomies and failure-attribution studies expose useful categories of agent mistakes (Zhu et al., 2025; Cemri et al., 2025; Shah et al., 2026; Zhang et al., 2025a; Zhu et al., 2026; Ma et al., 2026), but CUA trajectories require causal judgments over screenshots, spatial targets, low-level actions, and temporal screen transitions. As illustrated in Figure 1, the actionable error may occur before the final failure signal. Because these signals are difficult to infer from rewards alone, CUA debugging needs human-grounded supervision over the root-cause step, supporting evidence, and actionable correction.

This paper studies CUA root-cause debugging through three research questions. First, where do CUA agents fail? We answer this with a CUA-specific taxonomy and human failure analysis over annotated OSWorld trajectories. Second, can a tool-augmented RCA agent improve over naive prompting while matching human root-cause judgments? We compare CUADebugger, our debugging agent, against a prompt-only baseline that predicts the same RCA record from the full trajectory and taxonomy, using both taxonomy and step-localization metrics against human annotations. Third, can RCA guide re-execution? We test whether diagnostic context can improve continual or localized re-rollout rather than merely producing plausible explanations.

To support these questions, we introduce CUAErrorBench, a human-annotated OSWorld failure benchmark. Each example labels the root-cause step, L1/L2 taxonomy label, supporting evidence, corrective strategy, and confidence. CUAErrorBench is not another task-success benchmark; it converts failed trajectories into causal supervision for comparing human and model root-cause judgments. The current active pool contains 184 usable annotations across Claude 4.5 Sonnet, Gemini 2.5 Pro, and Qwen 3.5 trajectory sources.

We also introduce CUADebugger, a tool-augmented RCA agent for CUA trajectories (Figure 3). Rather than asking a model to infer the root cause from the full trajectory in one pass, CUADebugger runs a ReAct (Yao et al., 2023) loop with two CUA-specific debugging tools: a *multimodal step-inspection tool* that returns paired before/after screenshots together with the action, reasoning, and execution status at any chosen step; and a *structured root-cause submission tool* whose schema enforces coarse-to-fine taxonomy selection together with grounded evidence, a concrete correction, a calibrated confidence score, and per-step summaries that are passed forward to re-rollout. When episodic memory is enabled, CUADebugger also retrieves transferable diagnoses distilled from prior failures and human annotations as optional RCA context. This design links diagnosis with re-execution: a useful diagnosis should not only describe a failure, but also provide a repair signal that can be tested by re-rollout.

Our experiments currently support three main findings. First, human annotations show that task reasoning and control is the most common failure family (110/184), followed by perception (36/184) and grounding/interaction (25/184). Second, CUADebugger improves fine-grained and joint diagnosis on the main Claude-agent split: Tag+Step Exact rises from 11.2% to 19.6% for Gemini 2.5 Pro, from 7.6% to 14.6% for Qwen 3.5, and from 4.9% to 15.3% for Claude 4.5 Sonnet. Third, diagnostic context can support repair: single re-rollout packages with structured RCA achieve 28.47% task completion with machine RCA and 29.90% with our method, compared with 13.89% for history-only continuation, while continual re-rollout improves from 12.2% to 25.86% with our method and 29.21% with human root-cause guidance.

Our core contributions are three fold. First, we analyze where CUA agents fail and introduce a CUA-specific root-cause taxonomy covering perception, grounding and interaction, task reasoning and control, and external/system failures. Second, we introduce CUAErrorBench, a human-annotated benchmark that turns failed OSWorld trajectories into root-cause supervision. Third, we develop CUADebugger, a tool-augmented RCA agent that combines a ReAct loop with multimodal step-inspection, structured root-cause submission, and optional episodic memory retrieval, and evaluate it against a naive prompt baseline and human judgments.

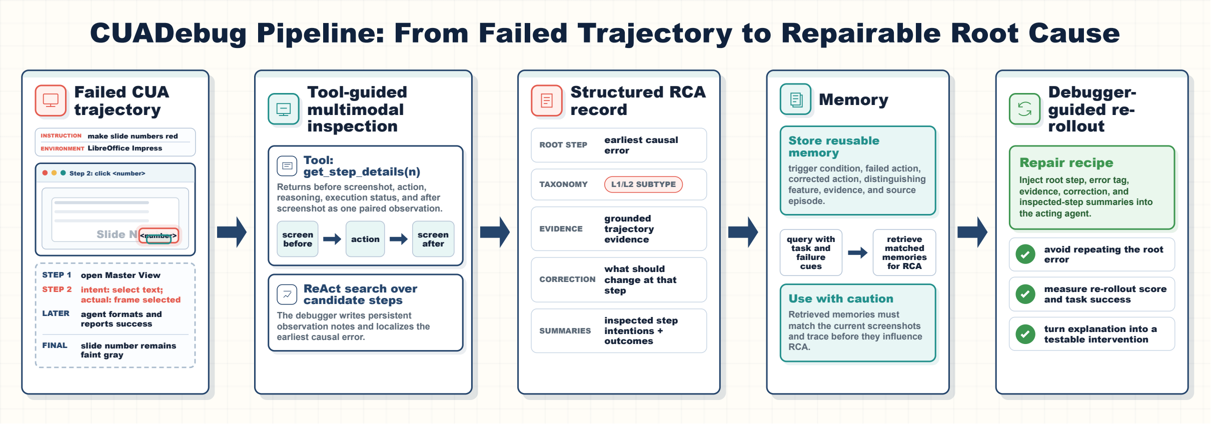


Figure 3: CUADebugger pipeline: inspect failed steps, produce structured RCA, retrieve memories when enabled, and guide re-rollout.

2 CUAErrorBench

We construct CUAErrorBench from failed OS-World trajectories produced by three CUA agent sources. Each trajectory contains a task instruction, visual observations, screenshots, model reasoning traces, low-level actions, rewards, and terminal status. The active benchmark excludes the deprecated 15-step split and contains 184 usable annotations: 125 Claude 4.5 Sonnet 50-step trajectories, 29 Gemini 2.5 Pro trajectories, and 30 Qwen 3.5 trajectories. Figure 2 shows that the largest human-labeled failure family is task reasoning and control, followed by perception and grounding/interaction.

The annotation pipeline stores debugger-generated RCA proposals together with human annotations; CUAErrorBench uses the human fields as reference labels. For each failed trajectory, annotators provide five fields: (1) the root-cause step, (2) a diagnostic label, (3) textual evidence, (4) a corrective action or strategy, and (5) confidence. All 184 paper-usable examples are annotated by multiple annotators; disagreements are discussed and resolved into one final reference label. Appendix A.4 summarizes the human annotation instructions and consent procedure, and Appendix A.5 gives one complete annotated example.

Benchmark curation and annotation protocol. Annotators identify the root cause by inspecting the task instruction, terminal failure signal, action and reasoning trace, and before/after screenshots. They first verify the final failure symptom, then work backward to find the earliest step that introduces a new causal mistake rather than merely inheriting an earlier error. Evidence must cite concrete trajectory content, such as a visible UI state, selected element, action code, or mismatch between the agent's stated intention and the screen outcome. The correction field describes an actionable alternative at that step, while confidence records whether competing root causes remain plausible.

Taxonomy labels. We use a two-level CUA error taxonomy as the common label space for annotation, RCA, memory retrieval, and evaluation. The top level identifies the causal module that determines the repair family, while the subtype captures the concrete failure pattern needed for fine-grained evaluation and memory retrieval. Table 1 shows the four top-level categories; Appendix A.6 gives the full 29-subtype taxonomy.

Code	Error module	Diagnostic question	Typical evidence
P	**Perception**	Did the agent misunderstand what was visible in the observation?	Hallucinated or missed UI content, OCR mistakes, cross-region binding errors, semantic misreadings.
G	**Grounding and Interaction**	Did the agent know the intended operation but execute it on the wrong target or with the wrong mechanics?	Wrong coordinates or element, hidden or disabled target, incorrect click, drag, gesture, or text-entry behavior.
R	**Task Reasoning and Control**	Did the agent choose, maintain, or revise the wrong plan?	Constraint violations, missing subgoals, action-intent mismatch, memory loss, progress misjudgment, failed self-correction.
S	**External/Syst**	Did the environment, tool, or benchmark setup prevent otherwise valid progress?	Rendering or timing failures, unexpected system dialogs, resource limits, tool/API failures, benchmark artifacts.

Table 1: Top-level CUA error taxonomy used in *CUAErrorBench*.

3 Method

CUADebugger operationalizes CUA root-cause annotations in three stages: root-cause prediction, memory retrieval, and debugger-guided re-rollout.

3.1 Root-Cause Analysis

RCA predicts the root-cause step in a failed trajectory. CUADebugger performs this prediction by running a ReAct (Yao et al., 2023) agent loop equipped with CUA-specific debugging tools, rather than a one-shot prompt over the full trajectory. The agent is initialised with the task instruction, trajectory metadata, terminal failure step F, L1/L2 taxonomy definitions, an output schema, and optionally retrieved memories. At each turn it decides which step to inspect next; older screenshots are compressed out of the context window, while the agent's textual observation notes are retained.

Multimodal step-inspection tool. The core inspection primitive, `get_step_details`(n), returns a single *paired* observation for step n: the action code, the agent's reasoning, the execution error and reward, and two screenshots: the *input* screenshot showing the screen state before acting and the *result* screenshot showing the state after the action executed. Pairing the screenshots against the agent's stated intention exposes perception, grounding, and outcome-misinterpretation errors that text traces alone would hide. After each inspection, the agent writes a short observation note that persists across context compression.

Structured root-cause submission tool. The agent terminates by calling `finish`$(\cdot)$. Its schema enforces the full RCA record in one structured object: the root-cause step N, L1/L2 taxonomy tag, grounded evidence, concrete correction, calibrated confidence score in $[0, 1]$, and one per-step summary for each inspected step. The per-step summaries are passed forward to re-rollout (Section 3.5) so the acting agent inherits the debugger's interpretation of the failed trajectory.

Memory retrieval. When episodic memory is enabled (Section 3.4), the initial prompt also includes the top-k retrieved memories, marked as *candidate* evidence: the agent is instructed to compare each memory's trigger condition against the current trajectory and discard non-matching memories rather than copy them. Appendix A.1 gives the RCA prompt template, memory-retrieval prompt block, and tool schemas.

3.2 Error Taxonomy Module

The error taxonomy module makes RCA a coarse-to-fine diagnosis. Given a candidate root-cause step, the debugger first selects one of the four top-level modules in Table 1, then selects a subtype within that module from Appendix A.6. This structure separates causal source from terminal symptom, while the subtype indexes memories for retrieval and the top-level module gives the acting agent an interpretable warning during re-rollout.

3.3 Multimodal Trajectory Inspection

CUADebugger treats screenshots and action traces as paired evidence rather than separate inputs. For each tool-inspected step, the debugger records the visible state before the action, the intended UI operation, the executed low-level action, and the observed state afterward. This inspection exposes failures that are easy to miss from text traces alone, such as selecting the wrong chart subtype, clicking a nearby control, or assuming that a dialog changed state when the screenshot shows otherwise. The resulting step summaries are then used by RCA, memory distillation, and re-rollout guidance.

3.4 Episodic Memory and Retrieval-Augmented Debugging

Episodic memory converts individual trajectories into reusable debugging knowledge. For each failed trajectory, CUADebugger stores an episode with task and application metadata, local error context, inferred agent intention, screen-outcome summary, taxonomy tag, RCA output, and optional human annotation. This episode is the durable reference object: later retrieval can return not only an abstract rule, but also the concrete trajectory evidence from which the rule was distilled.

Retrieval-augmented debugging exposes this memory through a tool-use interface. From each episode, CUADebugger distills a memory with a short title, trigger condition, failed action, corrected action, taxonomy tag, distinguishing feature, evidence, and episodic references. Memories are indexed by transferable fields such as application id, taxonomy tag, trigger condition, failed action, and memory text. During RCA or re-rollout, the debugger or acting agent may query similar failures, but retrieved memories are treated as candidate evidence rather than authoritative labels: the model must compare each trigger and distinguishing feature against the current screenshots and action trace. In cold start, memories are distilled from human annotations; after cold start, the same representation can absorb successful RCA outputs or contrastive failure/success trajectory pairs. The current experiments instantiate this design through a Machine RCA condition and our memory-enabled condition.

3.5 Debugger-Guided Re-Rollout

Re-rollout tests whether RCA helps the agent act better, not only explain better. Both the direct baseline and the debugger-guided method replay the original OSWorld trajectory to the same cutoff state and continue with the same remaining step budget. The debugger-guided method keeps this state and budget fixed, but augments the acting prompt with the previous trajectory summary, debugger-inspected step summaries, and a repair recipe containing the root-cause step, error type, evidence, and correction. In the retrieval-enabled variant, the acting agent can additionally query episodic memory for similar prior failures before choosing its next action. Appendix A.2 lists the re-rollout prompt variants and visual side-channel tools.

This design tests whether RCA is actionable: if the diagnosis contains causal repair information, injecting it during re-rollout should reduce repeated failure patterns; otherwise, it remains plausible prose without behavioral effect.

4 Experiments

4.1 Evaluation Protocol

We evaluate CUA debugging along three research questions. First, failure analysis asks where CUA agents fail by measuring the human-labeled distribution of root-cause modules across trajectory sources. Second, RCA evaluation asks whether CUADebugger improves over a naive prompt baseline and matches human root-cause judgments. Third, re-rollout evaluation asks whether RCA can guide subsequent execution. Unless otherwise stated, human annotations are the reference labels, and empty cells indicate runs or post-analysis metrics that are not yet available.

4.2 Failure Analysis

Failure analysis uses the human labels in CUAErrorBench to characterize where CUA agents fail. We report top-level taxonomy distributions by agent trajectory source in Figure 2. This analysis answers a descriptive question rather than a model-ranking question: it provides the reference view of CUA failure modes against which model debugger behavior is later compared.

Agent traj.	RCA method	Debugger	L1	L2	Step Exact	Step ±2	Tag+ Step
Claude 4.5	Naive	Gemini 2.5 Pro	51.0	30.1	21.0	38.5	11.2
Claude 4.5	**CUADebug**	Gemini 2.5 Pro	53.8	37.1	29.4	47.6	**19.6**
Claude 4.5	Naive	Qwen 3.5	44.4	19.4	23.6	34.7	7.6
Claude 4.5	**CUADebug**	Qwen 3.5	57.3	33.0	25.2	46.6	**14.6**
Claude 4.5	Naive	Claude 4.5	49.3	14.6	22.2	38.2	4.9
Claude 4.5	**CUADebug**	Claude 4.5	61.3	31.4	29.2	45.3	**15.3**
Gemini 2.5	**Naive**	Qwen 3.5	46.7	20.0	26.7	50.0	**6.7**
Gemini 2.5	CUADebug	Qwen 3.5	44.8	20.7	24.1	41.4	3.4
Gemini 2.5	**Naive**	Claude 4.5	53.3	23.3	26.7	50.0	**10.0**
Gemini 2.5	CUADebug	Claude 4.5	38.5	15.4	23.1	53.8	7.7
Gemini 2.5	Naive	Gemini 2.5 Pro	50.0	20.0	30.0	40.0	**10.0**
Gemini 2.5	CUADebug	Gemini 2.5 Pro	53.3	16.7	40.0	50.0	**10.0**
Qwen 3.5	Naive	Gemini 2.5 Pro	53.3	23.3	33.3	43.3	10.0
Qwen 3.5	**CUADebug**	Gemini 2.5 Pro	54.5	36.4	45.5	63.4	**18.2**
Qwen 3.5	Naive	Qwen 3.5	46.7	20.0	16.7	33.3	3.3
Qwen 3.5	**CUADebug**	Qwen 3.5	44.0	20.0	20.0	36.0	**12.0**
Qwen 3.5	**Naive**	Claude 4.5	56.7	30.0	30.0	36.7	**10.0**
Qwen 3.5	CUADebug	Claude 4.5	45.5	36.4	27.3	36.4	9.1

Table 2: RCA accuracy against human annotations (percent). CUADebug denotes our tool-augmented RCA method; Tag+Step requires exact subtype and exact root-cause-step match.

4.3 RCA Evaluation

The RCA evaluation reports both taxonomy and localization accuracy. We compare two RCA methods. The *Naive Baseline* receives the task, full trajectory package, and taxonomy definitions, and produces the RCA record in one pass. CUADebugger receives the same task metadata and taxonomy definitions but must use the step-inspection and structured-submission tools described in Section 3.1 to inspect step details before producing the same RCA record. Table 2 reports both methods. L1 accuracy measures top-level taxonomy agreement with the human annotation, while L2 accuracy measures exact subtype agreement. Step Exact measures exact root-cause-step match, and Step ±2 gives credit when the predicted root-cause step is within two steps of the human-labeled root-cause step. Tag+Step Exact requires both L2 subtype match and exact root-cause-step match.

4.4 Single Re-Rollout Evaluation

Single re-rollout isolates whether a diagnosis can support behavioral repair. The rollout agent restarts one step before the root-cause step and runs with a fixed step and token budget. The Machine RCA condition uses the machine-predicted root-cause step and machine-generated diagnosis without memory retrieval to test the automatic RCA-to-repair pipeline, while the baseline, self-debug,

Condition		Cost			Task	Root error
Method	RCA source	Budget	Input tok.	Output tok.	Acc. (%)	New+ Fixed
Baseline	none	28.01	272.7K	3.5K	13.89	20+11
Self-debug	self	31.53	324.4K	3.9K	15.28	27+12
Machine RCA	machine RCA (model TBD)	23.03	290.8K	2.9K	28.47	29+35
Human oracle	human annotation	22.37	296.9K	3.7K	20.10	43+29
Our method	RCA + memory	28.28	319.1K	2.9K	**29.90**	44+24

Table 3: Single re-rollout results on Claude 4.5 Sonnet trajectories.

human-oracle, and our method conditions use the human-labeled root-cause step to fix the restart point and compare diagnostic context. This controls for localization error: if the restart step itself is wrong, the re-rollout may be uninformative regardless of the debugging method. Because the rows differ in diagnostic prompt format, Table 3 should be read as a re-execution-package comparison rather than a clean ablation of RCA source quality. The main root-error metric is New+Fixed: a post-analysis judge compares the original human annotation, including root-cause step, evidence, and correction, against the new trajectory; the first number counts cases where the original critical error is fixed but a new critical error appears, and the second counts cases where the original critical error is fixed without a new critical error. When the budget reaches a task-level outcome we additionally report Re-rollout Acc as task completion rate. The agent trajectory and rollout agent are fixed to Claude 4.5 Sonnet.

4.5 Continual Re-Rollout Evaluation

Continual re-rollout tests whether diagnostic context helps an agent continue after failure. We use Claude 4.5 Sonnet 50-step failures, restart Claude 4.5 Sonnet from step 51, and compare direct continuation, self-debugging, Machine RCA, our full method, and human annotation guidance. Re-rollout accuracy measures whether the task is completed after re-execution, while New+Fixed uses the same order as Table 3: fixed with a new critical error, followed by fixed without a new critical error.

Method	Original turn	Spend turns	Input	Output	Acc.	New+ Fixed
Baseline	50	38.92	390.3K	3.8K	12.20	26+7
Self-debug	50	43.76	401.6K	4.4K	14.20	30+11
Machine RCA	50	47.13	405.9K	5.1K	21.53	21+44
Our method	50	47.44	426.2K	5.4K	25.86	**27+58**
Human oracle	50	45.75	419.1K	4.8K	**29.21**	29+44

Table 4: Continual re-rollout results on Claude 4.5 Sonnet failures.

5 Results and Analysis

CUA failures are concentrated in reasoning/control but remain multimodal. Figure 2 shows that task reasoning and control accounts for the largest share of human-labeled failures in the active benchmark (110/184, 59.8%). This pattern is consistent across the three trajectory sources: Claude 4.5 Sonnet has 76 reasoning/control failures out of 125 annotations, Gemini 2.5 Pro has 15 out of 29, and Qwen 3.5 has 19 out of 30. At the same time, the remaining failures are not reducible to text-only planning mistakes: perception accounts for 36 cases, grounding and interaction for 25 cases, and external/system issues for 13 cases. This distribution motivates a CUA-specific taxonomy that explicitly separates visual perception, spatial interaction, reasoning/control, and system causes.

CUADebugger improves joint diagnosis on the main annotated split. Table 2 shows that the tool-augmented RCA agent improves RCA on the most complete Claude 4.5 Sonnet split. Relative to the naive prompt baseline, CUADebugger improves L2 accuracy and Tag+Step Exact for all three debugger backbones. Gemini 2.5 Pro improves from 30.1% to 37.1% L2 accuracy and from 11.2% to 19.6% Tag+Step Exact; Qwen 3.5 improves from 19.4% to 33.0% L2 and from 7.6% to 14.6% Tag+Step; Claude 4.5 Sonnet improves from 14.6% to 31.4% L2 and from 4.9% to 15.3% Tag+Step. This suggests that actively inspecting step details with paired before/after screenshots is more useful than simply providing the full trajectory and taxonomy in one prompt. Figure 4 further shows where the debugger agrees with human labels at the category level. Appendix A.7 reports the corresponding subtype-frequency comparison.

Cross-agent RCA remains an important stress test. The Gemini-agent and Qwen-agent rows show that RCA behavior changes with both the failed agent and debugger backbone. Unlike the

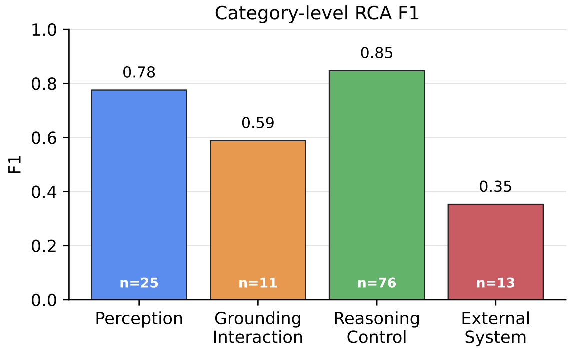


Figure 4: Category-level debugger F1 on the Claude 4.5 Sonnet split ($N = 125$).

Claude split, CUADebugger is not uniformly better on every cross-agent row; for example, Qwen 3.5 on Gemini-agent trajectories has a higher Tag+Step score under the naive baseline than under CUADebugger, while Gemini 2.5 Pro on Qwen-agent trajectories benefits from CUADebugger despite the smaller evaluated subset. These mixed rows also highlight a dependency of CUADebugger on the underlying model's capability: tool inspection can surface the relevant evidence, but the model must still interpret that evidence and revise its causal hypothesis. Stronger backbones appear better able to turn inspected evidence into corrected diagnoses, whereas weaker backbones may show limited gains or no improvement on small cross-agent splits. Because these cross-agent rows have uneven sample sizes, we treat them as a stress test rather than as a final model ranking. Across settings, Tag+Step remains low, confirming that matching both the fine-grained cause and the exact root step is substantially harder than predicting a coarse error family.

Single re-rollout: structured RCA improves behavioral repair. Table 3 measures whether a diagnosis lets the rollout agent fix the original root-cause mistake. The structured RCA conditions provide the strongest task-level recovery: Machine RCA reaches 28.47% Re-rollout Acc, and our method reaches 29.90%, compared with 13.89% for the history-only baseline and 15.28% for self-debug. In root-error outcomes, the same root error is fixed but followed by a new error in 29 Machine RCA cases and 44 our-method cases, and is fixed without a new critical error in 35 and 24 cases respectively. This repair setting reflects the same capability dependence as RCA: pointing out the root error is most useful when the rollout model can translate the diagnosis into a different action sequence; weaker models may still repeat the failure or only partially avoid it. Because these rows differ in restart source and diagnostic prompt format, we treat these gains as evidence that structured repair packages can be useful, not as a direct comparison of diagnosis quality against human annotations.

Continual re-rollout currently shows headroom from accurate RCA. Table 4 reports continual re-rollout from Claude 4.5 Sonnet 50-step failures. Direct continuation reaches 12.20% task success, and self-debugging gives a modest gain to 14.20%. Machine RCA improves success to 21.53%, while our method reaches 25.86%, narrowing much of the gap to the human-oracle row at 29.21%. The same pattern appears in root-error repair: our method fixes the first critical error in 85 cases total (27 with a new critical error and 58 without), compared with 33 total for the baseline and 41 total for self-debugging.

Why human annotation: evolved taxonomies are path-dependent. A natural alternative to human annotation is to let a strong LLM evolve its own taxonomy directly from failed trajectories. We test this directly by running the debugger in an empty-start configuration: Gemini 2.5 Pro sees no taxonomy seed and must commit one of five operations per case (Reuse, Discover, Rename, Split, Merge) over the CUAErrorBench 50-step Claude split, with a running taxonomy state that persists across cases. We run four random case orderings; seed 3 completes all 144 cases on a paid API quota, while seeds 0 to 2 each cover 47 to 55 cases before truncation. Three findings emerge. **(A) Evolved taxonomies do not converge across orderings.** Across the $\binom{4}{2} = 6$ seed pairs, an LLM-as-judge under name-strict alignment finds zero matching subtypes; seed 0 organizes failures as `P-⋆/OI-⋆/I-⋆`, while seed 3 uses an eight-way `ACTION/POLICY/EVALUATOR/VISUAL` scheme, and the two cannot be unified by name. **(B) The complete seed does not saturate.** Seed 3 ends with 40 subtypes after 144 cases with the curve still rising, ruling out a data-quantity explanation for the divergence. **(C) Semantic structure partially converges even though naming does not.** Projecting each run through a separate LLM-as-judge mapping back into CUAErrorBench's label space recovers 15% to 28% of human subtypes per seed and roughly 40% in the four-seed union. Together, these results frame human annotation not as a labor convenience but as a structural requirement: autonomous discovery finds error structure, but cannot find *a stable schema*. CUAErrorBench's value is therefore not just the 204 labels but the canonical schema those labels anchor.

6 Related Work

Computer-Use Agent Benchmarks. Recent benchmarks evaluate agents in realistic computer-use environments, including desktop systems, web interfaces, and mobile applications (Xie et al., 2024; Zhou et al., 2024; Koh et al., 2024; Deng et al., 2023; Rawles et al., 2025; Bonatti et al., 2024). These benchmarks have made end-to-end task success measurable, but they do not by themselves explain why a failed trajectory failed. CUAErrorBench complements these benchmarks by adding human root-cause annotations over failed trajectories, and CUADebugger uses those annotations as corrective supervision.

Agent failure analysis and debugging. Prior work studies agent failure taxonomies, multi-agent failure attribution, and CUA-specific trust or diagnosis problems (Cemri et al., 2025; Shah et al., 2026; Zhang et al., 2025a; Zhu et al., 2026; Ma et al., 2026; Liu et al., 2026; Shi et al., 2026; Zhai et al., 2026). Most closely, AgentErrorBench and AgentDebug study root-cause debugging for general LLM agents over ALFWorld, GAIA, and WebShop (Zhu et al., 2025). Our work focuses on the CUA setting, where decisive evidence is often multimodal and interaction-level: screenshots, target grounding, temporal screen transitions, and low-level GUI mechanics. CUADebugger pairs CUA-specific human annotations with visual inspection tools, episodic memory over agent intentions and screen outcomes, and OSWorld state replay with re-execution guidance.

Multimodal information flow. Recent modular multimodal systems decouple visual evidence extraction from downstream reasoning. SeeingEye, for example, uses an agentic vision translator with tools such as OCR and crop to produce structured intermediate representations for text-only LLM reasoning (Zhang et al., 2025b). CUADebugger similarly separates visual trajectory inspection from causal reasoning, but targets failed CUA trajectories rather than single-query multimodal answering.

Learning from failure. Language agents can improve from reflection, experience, retrieval, and memory (Yao et al., 2023; Shinn et al., 2023; Zhao et al., 2024; Wang et al., 2024; Hu et al., 2026). BacktrackAgent is especially related because it improves GUI agents through online error detection and backtracking during task execution (Wu et al., 2025). CUADebugger addresses a different problem: rather than adding an online rollback mechanism to a running agent, we build a human-annotated CUA failure benchmark and a tool-augmented RCA agent that performs after-execution causal localization. The output is a structured diagnosis with evidence, taxonomy label, and correction, which can then be evaluated through controlled re-rollout and stored as reusable memory.

7 Conclusion

We studied CUA root-cause debugging through three questions: where CUA agents fail, whether a tool-augmented RCA agent improves over naive prompt-based diagnosis while matching human root-cause judgments, and whether RCA can guide re-execution. We introduced a CUA-specific taxonomy, CUAErrorBench as a human-annotated failure benchmark, and CUADebugger as a tool-augmented RCA agent that produces re-execution guidance. Current experiments show that CUA failures are dominated by reasoning/control but still include substantial perception and grounding/interaction cases, that CUADebugger's tool-based step inspection improves joint diagnosis on the main annotated split, that RCA-based re-rollout packages improve task-level recovery under the current protocol, and that our method substantially improves continual re-execution over direct continuation while approaching the human-oracle result. Effective CUA debugging should therefore not stop at explaining failures; it should convert explanations into reusable corrections that can be tested by re-execution.

Limitations

The current results are strongest for the completed failure analysis, Naive-vs-CUADebugger RCA comparison, single re-rollout task accuracy, and continual re-rollout results on Claude 4.5 Sonnet failures. RCA sample sizes remain uneven outside the Claude-agent trajectory split, so cross-agent debugger rankings should be interpreted as preliminary. Single re-rollout now includes task-accuracy and critical-error outcome cells for the baseline, self-debug, Machine RCA, human-oracle, and our method settings, but these conditions are not fully matched: they vary in restart-step source and diagnostic prompt format as well as in diagnosis source. The debugger model for Machine RCA is still pending final confirmation. Future work should also tighten the connection between subtype prediction, exact root-step localization, evidence quality, and behavioral repair during re-execution through matched re-rollout ablations. While there exist other Computer Use Agent benchmarks, we focus on using OSWorld to annotate and test our method; future work can try to expand to other benchmarks.

References

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Jang, and Zack Hui. 2024. Windows agent arena: Evaluating multi-modal OS agents at scale. *arXiv preprint arXiv:2409.08264*.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2Web: Towards a generalist agent for the web. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023), Datasets and Benchmarks Track*.

Zhanghao Hu, Qinglin Zhu, Runcong Zhao, Di Liang, Hanqi Yan, Yulan He, and Lin Gui. 2026. Beyond rag for agent memory: Retrieval by decoupling and aggregation. *arXiv preprint arXiv:2602.02007*.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. 2024. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 881–905, Bangkok, Thailand. Association for Computational Linguistics.

Xunzhuo Liu, Bowei He, Xue Liu, Andy Luo, Haichen Zhang, and Huamin Chen. 2026. Visual confused deputy: Exploiting and defending perception failures in computer-using agents. *arXiv preprint arXiv:2603.14707*.

Xuyan Ma, Xiaofei Xie, Yawen Wang, Junjie Wang, Boyu Wu, Mingyang Li, and Qing Wang. 2026. Demystifying the lifecycle of failures in platform-orchestrated agentic workflows. *Preprint*, arXiv:2509.23735.

Chris Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. 2025. AndroidWorld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*.

Mehil B Shah, Mohammad Mehdi Morovati, Mohammad Masudur Rahman, and Foutse Khomh. 2026. Characterizing faults in agentic ai: A taxonomy of types, symptoms, and root causes. *Preprint*, arXiv:2603.06847.

Yucheng Shi, Wenhao Yu, Jingyuan Huang, Wenlin Yao, Wenhu Chen, and Ninghao Liu. 2026. Towards trustworthy gui agents: A survey. *Preprint*, arXiv:2503.23434.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*.

Qinzhuo Wu, Pengzhi Gao, Wei Liu, and Jian Luan. 2025. Backtrackagent: Enhancing gui agent with error detection and backtracking mechanism. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 4250–4272.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024), Datasets and Benchmarks Track*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Yuwen Zhai, Runze Li, Liang Wang, Nian Shi, Liwu Xu, Wei Zhang, Ran Lin, Bo Xu, and Benlei Cui. 2026. Guide: Interpretable gui agent evaluation via hierarchical diagnosis. *Preprint*, arXiv:2604.04399.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. 2025a. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In *Forty-second International Conference on Machine Learning*.

Weijia Zhang, Zijia Liu, Haoru Li, Haoqi Chen, and Jiaxuan You. 2025b. Seeingeye: Agentic information flow unlocks multimodal reasoning in text-only llms. *Preprint*, arXiv:2510.25092.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. Expel: Llm agents are experiential learners. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'24/IAAI'24/EAAI'24. AAAI Press.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*.

Chenyang Zhu, Spencer Hong, Jingyu Wu, Kushal Chawla, Yuhui Tang, Youbing Yin, Nathan Wolfe, Erin Babinsky, and Daben Liu. 2026. RAFFLES: Reasoning-based attribution of faults for LLM systems. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7659–7688, Rabat, Morocco. Association for Computational Linguistics.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. 2025. Where LLM agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*.

A Appendix

A.1 RCA Prompt and Tool Schemas

This appendix summarizes the prompt and tool interface used by the RCA methods in Section 3.1. The implementation sources are `debugger/profiling/run_plain.py`, `debugger/rca.py`, and `debugger/tools/__init__.py`. The taxonomy block in the live prompt is the subtype inventory in Appendix A.6; we reference it below to avoid duplicating the full taxonomy table.

The naive RCA baseline is a single-call prompt with no tool access and no memory. It receives the task, the textual trajectory, the taxonomy table, and a small trailing screenshot window, then returns the same RCA JSON fields used by CUADebugger.

RCA prompt-only baseline

```
System:
You are a debugging assistant analysing a
     failed GUI-agent trajectory.

Your job is to identify:
  (a) the root error step: the single step
     number where the failure
      originates, not just where it surfaces;
  (b) the taxonomy_tag: the subtype code from
     the taxonomy table that
      best describes the root cause.

Error Taxonomy:
[P/G/R/S subtype table from Appendix CUA Error
      Taxonomy Subtypes]

Return ONLY a single valid ASCII JSON object
     with no markdown fences,
no prose, and no trailing text:
{
  "root_error_step": [int],
  "taxonomy_tag": [str; one subtype code],
  "evidence": [str; 1-2 sentences citing
     action/error/state evidence],
  "correction": [str; what the agent should
     have done at that step],
  "confidence": [float between 0 and 1]
}

User:
[task instruction]
[full textual trajectory: per-step action type,
      action code, error,
 reward/done status, reasoning, and optional
     tool use]
[result screenshots of the last N steps, when
     enabled]

Identify the root error step and taxonomy tag.
      Output JSON only.
```

Tool-augmented RCA system prompt

```
<role>
You are an expert GUI agent trajectory
     debugger. Use the provided tools to
analyze a trajectory step by step, then submit
      your findings.
</role>

<workflow>
1. The user first message includes a step
     index showing each step's action_type,
   whether it had an execution error, and
     whether a screenshot exists. Use it
   to pick which steps to inspect.
2. For each suspicious step, such as a step
     with an execution error, a step after
   a failed step, a repeated action, or the
     terminal step of a failed
   trajectory, call get_step_details.
3. get_step_details returns the input
     screenshot (what the agent saw before
   acting) and the result screenshot (screen
     state after the action). Compare
   them to identify perception, grounding,
     interaction, or outcome errors.
4. Look for patterns across steps: repeated
     errors without correction, stale
   variable references, wrong UI targets, or
     an environment that was not ready.
5. When you have inspected all relevant steps,
      call finish() with the complete
   structured RCA record.
</workflow>

<taxonomy>
Use the CUA taxonomy appendix: P, G, R, and S
     top-level labels with their
fine-grained subtype codes.
</taxonomy>

<annotation_rules>
- Label the root-cause error, even when
     downstream effects manifest differently.
- Perception vs. Grounding: misunderstanding
     visible content is P; understanding
  the content but targeting the wrong location
     or interacting incorrectly is G.
- Prefer fine-grained subtype codes (P1, G2,
     R10, S4, etc.).
</annotation_rules>

## RCA Mode: Find the Single Root Error Step

Your task is Root Cause Analysis, not general
     debugging.

Goal: identify the single earliest step N such
      that N <= F, where F is the
Terminal Failure Step (the last step of a
     failed trajectory), and the mistake at
step N is the direct cause of subsequent
     failures.

Backward-tracing procedure:
1. Use get_step_details to inspect steps. It
     returns textual details plus the
   input screenshot and result screenshot.
2. Identify F = the terminal failure step from
      the trajectory summary.
3. Walk backwards from F. For each candidate
     step, call get_step_details to
   check whether it contains an independent
     mistake or merely inherited the
   failure from an earlier step. Compare the
     input screenshot against the
   action to identify perception or grounding
     errors.
4. Stop when you find the earliest step that
     introduced a new mistake. That is
   step N = root_error_step.
5. Call finish() with root_error_step=N,
     taxonomy_tag, evidence, correction,
   confidence, and per_step_summaries.

Per-step summaries for rerollout:
For every step inspected with get_step_details,
      include one entry in
per_step_summaries:
- intent_summary: infer the agent's intended
     action from action code, reasoning,
  and tool use.
- outcome_summary: infer the observable result
      by comparing the input screenshot
```

```
  against the result screenshot and using
     execution metadata only as support.
- summary_source: always set to "
     debugger_inspected".

After each get_step_details result, write a
     short plain-text observation before
requesting more tools. Older screenshots may
     be compressed out of history, but
the written observation remains.

Confidence guidance:
- 0.9-1.0: clear single root cause with a
     strong causal chain.
- 0.7-0.89: likely root cause with some
     ambiguity.
- 0.5-0.69: multiple plausible candidates;
     this is the best guess.
- < 0.5: weak evidence; still call finish and
     note the uncertainty.
```

Initial user prompt and retrieved memories

```
Perform Root Cause Analysis on this failed
     trajectory:

[trajectory summary with task, app, step index,
      terminal status, and failure step]

The trajectory is already loaded. You can
     directly call get_step_details to
inspect steps. Work backwards from the
     Terminal Failure Step, then call finish()
with the root_error_step, taxonomy_tag,
     evidence, correction, and confidence.

If retrieved memories are available:
Below are some past memories retrieved as
     potentially relevant based on semantic
similarity. They are not guaranteed to apply;
     evaluate each one against the
current trajectory before using it. If none
     match the current situation, ignore
them and reason from scratch.

Example k:
[memory title, trigger condition, failed
     action, corrected action,
distinguishing feature, evidence, taxonomy tag,
      and source episode reference]
```

RCA tool schemas

```
Tool: get_step_details
Description:
  Get full details for a specific step: action
     code, execution error, agent
  reasoning, reward, done status, plus two
     screenshots: the input screenshot
  before the action and the result screenshot
     after the action. In RCA mode,
  these details are used to write
     intent_summary and outcome_summary.
Input schema:
  {
    "type": "object",
    "properties": {
      "step_num": {
        "type": "integer",
        "description": "The step number to
     inspect."
      }
    },
    "required": ["step_num"]
  }

Tool: finish
Description:
  Submit the final RCA result after
     identifying the root error step.
Input schema:
  {
    "type": "object",
    "properties": {
      "root_error_step": {
        "type": "integer",
        "description": "Earliest root-cause
     step N, where N <= terminal step F."
      },
      "taxonomy_tag": {
        "type": "string",
        "enum": [
          "P1", "P2", "P3", "P4", "P5",
          "G1", "G2", "G3", "G4",
          "R1", "R2", "R3", "R4", "R5", "R6",
     "R7", "R8",
          "R9", "R10", "R11", "R12", "R13",
          "S1", "S2", "S3", "S4", "S5", "S6",
     "S7",
          "P", "G", "R", "S"
        ],
        "description": "Prefer a fine-grained
     subtype code."
      },
      "evidence": {
        "type": "string",
        "description": "Grounded evidence from
      action code, errors, or screenshots."
      },
      "correction": {
        "type": "string",
        "description": "Concrete action or
     strategy that should have been used."
      },
      "confidence": {
        "type": "number",
        "minimum": 0.0,
        "maximum": 1.0,
        "description": "Confidence that this
     is the true root-cause step."
      },
      "per_step_summaries": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "step_num": {"type": "integer"},
            "intent_summary": {"type": "string
     "},
            "outcome_summary": {"type": "
     string"},
            "summary_source": {
              "type": "string",
              "enum": ["debugger_inspected"]
            }
          },
          "required": [
            "step_num",
            "intent_summary",
            "outcome_summary",
            "summary_source"
          ]
        }
      }
    },
    "required": [
      "root_error_step",
      "taxonomy_tag",
      "evidence",
      "correction",
      "confidence",
      "per_step_summaries"
    ]
  }
```

When a lesson table is injected into the RCA prompt, the implementation exposes three extra retrieval tools in addition to `get_step_details` and `finish`. These tools return candidate evidence only; the model is instructed to verify each retrieved lesson against the current screenshots and action trace.

Additional RCA lesson tools

```
Tool: lookup_lessons_by_taxonomy
Purpose:
  Return additional distilled lessons that
     share a taxonomy code.
Input:
  taxonomy_tag: subtype or top-level code, e.g.
      "G1" or "R10"
  top_k: optional integer in [1,10], default 3

Tool: search_lessons_by_app
Purpose:
  Return lessons from a specific app,
     optionally filtered by taxonomy.
Input:
  app_id: application identifier, e.g. "chrome
     " or "vs_code"
  taxonomy_tag: optional subtype or top-level
     code
  top_k: optional integer in [1,10], default 3

Tool: follow_episodic_ref
Purpose:
  Resolve a lesson’s episodic_ref into a
     compact source-trajectory summary.
Input:
  episodic_ref: UUID string carried by a
     retrieved lesson
```

A.2 Re-Rollout Prompt Variants and Tools

This appendix summarizes the re-rollout prompt variants used in our experiments. All variants replay the original environment to the cutoff state before continuing the task; they differ only in what diagnostic context is prepended to the acting agent.

The direct baseline receives the task and previous action history, but no external RCA, taxonomy label, debugger summary, or correction.

History-only re-rollout prompt

```
RE-ROLLOUT CONTEXT
Task: [task instruction]
Previous result score: [score] (target: 1.0)

The agent previously attempted this task for [
     N] steps but did not complete it
successfully. You are continuing from step [
     cutoff+1].

PREVIOUS TRAJECTORY SUMMARY (Steps 1-[N])
  Step  1: [OK/ERR/ ] [action_code]
           Error: [execution error, if any]
  ...

NOTE: No repair recipe available for this
     trajectory.
Review the step history above to identify what
      went wrong.

RE-ROLLOUT INSTRUCTIONS
- The environment is now in the state it was
     in after step [cutoff].
- You are starting from step [cutoff+1].
- Complete the task as efficiently as possible.
```

The self-debug baseline receives the same replayed state and action history as the history-only baseline, but explicitly asks the acting model to diagnose the previous attempt itself before choosing the next action.

Self-debug re-rollout prompt

```
SELF-DEBUG RE-ROLLOUT CONTEXT
Task: [task instruction]
Previous result score: [score] (target: 1.0)

A previous attempt reached step [N] and failed
      or only partially succeeded.
The VM has replayed that attempt through step
     [cutoff].
You are now continuing from the current screen
      state.

PREVIOUS TRAJECTORY SUMMARY (Steps 1-[N])
  Step  1: [OK/ERR/ ] [action_code]
           Error: [execution error, if any]
  ...

SELF-DEBUG INSTRUCTIONS
No expert RCA, taxonomy label, human
     correction, or debugger diagnosis is
available. Do not assume the previous attempt
     was correct just because it was
replayed.

At your first inference step, inspect the
     current screen and briefly self-debug:
  1. What was the previous attempt trying to
     accomplish?
  2. What likely went wrong or remains
     unfinished?
  3. Does the current screen contain side
     effects that need recovery?
  4. What is the smallest concrete next action?


Then continue the task from the current state.
      Avoid repeating a failed action
pattern unless the screen clearly shows it is
     now valid. Call DONE only when the
original task is visibly satisfied. Call FAIL
     only when the current environment
prevents completing the requested task.
```

Machine RCA, human-oracle RCA, and memory-augmented RCA use the same re-rollout template, with different RCA sources. Machine RCA fills the repair recipe from machine RCA without memory retrieval, the human-oracle row fills it from the human annotation, and our method uses the memory/RAG-augmented RCA package.

RCA-guided re-rollout prompt

```
RE-ROLLOUT CONTEXT
Task: [task instruction]
Previous result score: [score] (target: 1.0)
The agent previously attempted this task for [
     N] steps but did not complete it.
You are continuing from step [cutoff+1].

PREVIOUS TRAJECTORY SUMMARY (Steps 1-[N])
  Step  1: [OK/ERR/ ] [action_code]
  ...

DEBUGGER-INSPECTED STEP SUMMARIES
  Step [k] [debugger_inspected]
    Intent: [intent_summary from RCA]
    Outcome: [outcome_summary from RCA]
```

```
REPAIR RECIPE
Historical root error: Step [root_error_step]
     (error type: [taxonomy_tag])
[confidence: confidence]
Rerollout start state: the VM has replayed the
      original trajectory through
step [cutoff]. This is the current state to
     repair; inspect the screen before
acting.

Failed pattern to avoid:
  [RCA evidence]

Do instead:
  [RCA correction]

Next action policy:
  1. First check whether the current screen
     still contains side effects from
     the failed path.
  2. If recovery is needed, do the smallest
     recovery step before applying the
     correction.
  3. Then execute the corrected approach
     directly; do not repeat the failed
     pattern above.

Completion check:
  - The final visible/application state must
     satisfy the original task.
  - Save or apply changes when the task
     modifies a file or setting.
  - Once the success condition is visible,
     call DONE instead of continuing.
```

All re-rollout variants use the base OSWorld computer-use action interface for mouse, keyboard, screenshot observation, and terminal DONE/FAIL actions. For RCA-guided re-rollout runs, the runner additionally invokes visual helpers after each action and appends their outputs as `[SCREEN UPDATE step t]` context. These are side-channel observations generated by the runner, not free-form tools that the acting model calls directly.

Re-rollout visual side-channel tools

```
Base acting interface (all methods)
  computer-use actions: mouse clicks, keyboard
      input, screenshots
  terminal actions: DONE, FAIL

smart_grid_caption_from_bytes(screenshot,
     query)
  Summarizes visible screen evidence relevant
     to the current action intent.

local_visual_search_from_bytes(screenshot,
     query, target)
  Searches local screen regions for the
     intended target or related evidence.

ground_text_bbox_from_bytes(screenshot,
     text_query)
  Returns bounding boxes for visible text
     matching the query.

text_region_ocr_from_bytes(screenshot)
  Extracts visible text snippets and
     approximate screen locations.

gui_grounding_from_bytes(screenshot, query,
     model, screen_w, screen_h)
  Predicts target pixel coordinates for the
     current semantic action intent.

Runner-added context:
  [SCREEN UPDATE step t]
  Screen: [caption/evidence]
  Target: [local visual-search evidence]
  Text bbox: [text grounding output]
  Visible text: [OCR snippets]
  Target pixel: ([x], [y])
```

Algorithm 1: Retrieval-augmented debugging with episodic memory.

```
Input: Training trajectories $\mathcal{D}_{train}$, new trajectory $\tau$,
       episodic memory $M$, retrieval size $k$, mode
       $m \in \{\mathrm{RCA}, \text{re-rollout}\}$
Output: An RCA report or re-rollout guidance
for $\tau_i \in \mathcal{D}_{train}$ do
    $r_i \leftarrow$ RCA output or human annotation for $\tau_i$
    $e_i \leftarrow$ episode with task, app, steps, root-cause
      step, local error context, intention, tag,
      evidence, and correction
    $\ell_i \leftarrow \textsc{DistillMemory}(e_i, r_i)$
    $M.\textsc{Add}(\ell_i)$
end
$q \leftarrow \textsc{BuildQuery}(\tau, m)$ from the task instruction,
  app, observed state, suspected failed action, or
  tentative error type
$L \leftarrow \texttt{memory\_query}(M, q, k)$
for $\ell \in L$ do
    inspect the current trajectory and compare $\ell$'s
      trigger condition and distinguishing feature
      against the current evidence
    if $\ell$ matches the current failure pattern then
        use $\ell$'s tag rationale, evidence, and
          correction as candidate guidance
    end
end
return a final RCA report if $m = \mathrm{RCA}$; otherwise
  return re-rollout guidance that avoids the matched
  failed pattern
```

A.3 Retrieval-Augmented Debugging Procedure

A.4 Human Annotation Details, Instructions, and Consent

Six trained researchers annotated the failed CUA trajectories used in CUAErrorBench. Before annotation, annotators were briefed on the goal of root-cause debugging, the CUA error taxonomy, and the distinction between a terminal failure symptom and the earliest causal mistake. Annotations were collected with a Streamlit-based front-end tool implemented in `debugger/vis`. The tool loads assigned tasks and linked trajectory artifacts from a trial directory, opens the selected trajectory directly for inspection, and exposes the task instruction, terminal status, debugger RCA proposal, optional recording video, and a step-by-step trajectory view with screenshots, click markers, action code, execution metadata, accessibility-tree text, and the agent's LLM reasoning. Annotators entered or updated the root-cause step, taxonomy label, evidence, correction, confidence, and notes directly in the interface; the tool saved per-annotator entries into the shared annotation files.

Figure 5 shows the annotation platform used in this process. The interface was designed to make trajectory-level causal inspection practical for annotators: it presents the task, trajectory metadata, agent reasoning, tool-use traces, screenshots, recordings, and annotation fields in one workspace, so annotators can move step by step through a failed agent trajectory and record the root cause without switching between separate files.

Annotators were instructed to first verify the final failure symptom and then trace backward through the trajectory to identify the earliest step that introduced a new causal error rather than a downstream consequence of an earlier error. For each trajectory, annotators recorded the root-cause step, the most specific taxonomy label available, grounded evidence from the trajectory, an actionable correction, and a confidence value. Evidence was required to cite concrete trajectory information, such as visible UI state, selected elements, action code, execution errors, or mismatches between the agent's stated intention and the observed screen outcome. Corrections were required to describe what the agent should have done differently at the root-cause step.

Annotation assignments were load-balanced across annotators, with all 184 paper-usable examples assigned to multiple annotators. When annotations disagreed, annotators discussed the case and resolved it into a final decision used as the reference label. All annotators were members of the research team, were informed that their annotations would be used for research and publication, and consented to this use. The annotation data contains task and trajectory-level labels rather than private information about annotators.

A.5 Example Human Annotation

Table 5 shows one CUAErrorBench example from an OSWorld LibreOffice Calc task. The example illustrates why root-cause annotation requires localizing the step where the trajectory becomes causally wrong rather than only describing the final failed spreadsheet.

A.6 CUA Error Taxonomy Subtypes

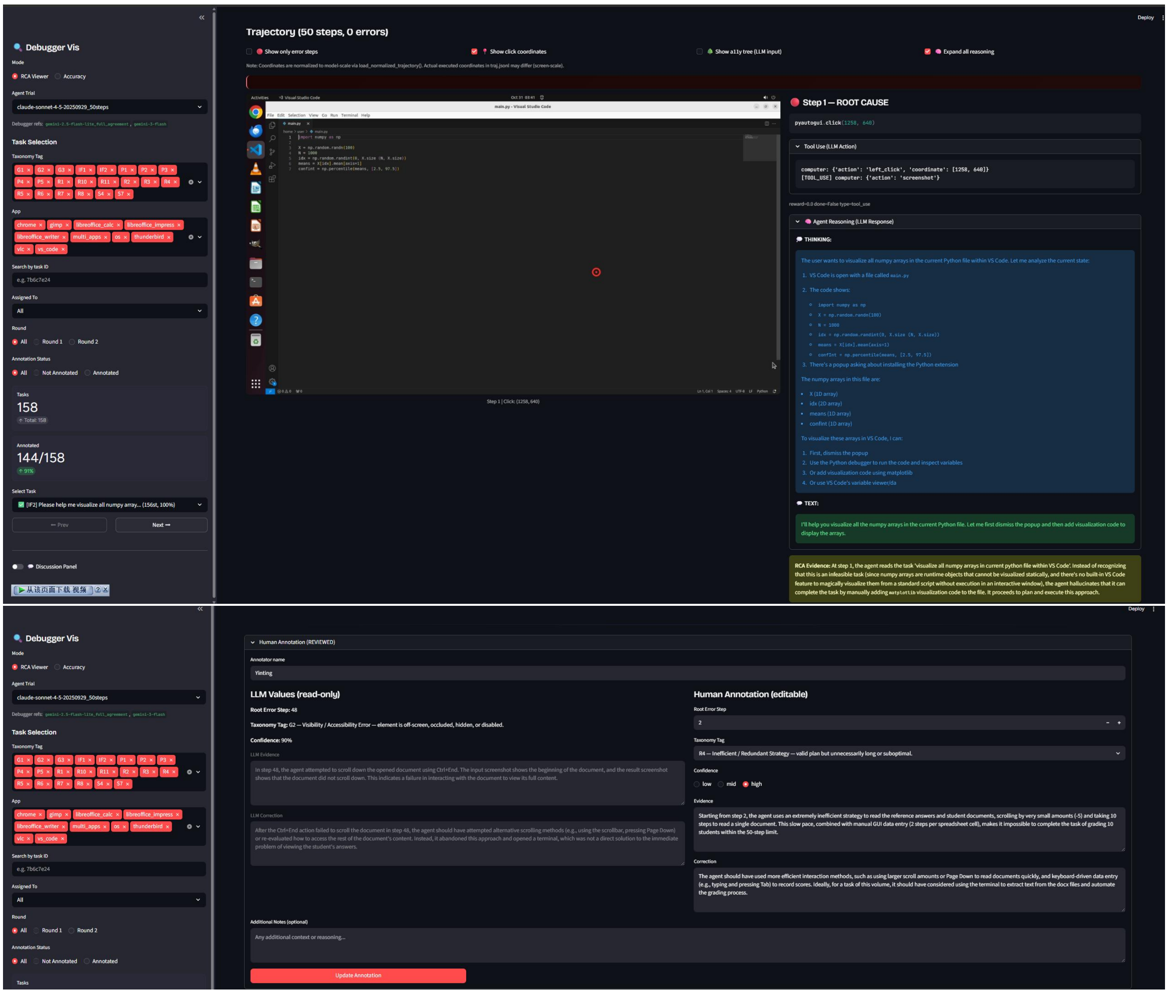
Table 6 lists the fine-grained subtype definitions used for L2 evaluation and memory indexing.

Field	Annotation content
Task	Compute monthly total sales in a new row named "Total" and create a line chart with months on the x-axis.
Trajectory source	OSWorld, LibreOffice Calc, 50-step CUA trajectory.
Human agreement	Two human annotators assigned the same root-cause step, taxonomy subtype, and high confidence.
Root-cause step	Step 21.
Taxonomy label	P2: Misrecognition / OCR error.
Evidence	The agent reasoned that the "Points and Lines" chart subtype was already selected, but the screenshot showed that "Points Only" was highlighted. It then continued through the chart wizard, producing a chart with points but no connecting lines.
Correction	Select the "Points and Lines" subtype before continuing in the chart wizard.
Why this is root-cause supervision	The final artifact is a wrong chart, but the root-cause step is the earlier visual recognition mistake in the chart-type dialog; repairing that step changes the downstream execution.

Table 5: Example CUAErrorBench human annotation.

A.7 Additional Debugger Behavior Analysis

Figure 6 compares debugger-predicted subtype frequencies against human labels on the main Claude split.



Mode: ● RCA Viewer ○ Accuracy

Agent Trial: claude-sonnet-4-5-20250929_50steps

Debugger refs: `gemini-2.5-flash-lite_full_agreement`, `gemini-3-flash`

Task Selection

Taxonomy Tag: G1, G2, G3, IF1, IF2, P1, P2, P3, P4, P5, R1, R10, R11, R2, R3, R4, R5, R6, R7, R8, S4, S7

App: chrome, gimp, libreoffice_calc, libreoffice_impress, libreoffice_writer, multi_apps, os, thunderbird, vlc, vs_code

Search by task ID: e.g. 7b6c7e24

Assigned To: All

Round: ● All ○ Round 1 ○ Round 2

Annotation Status: ● All ○ Not Annotated ○ Annotated

Tasks: 158 (Total: 158)

Annotated: 144/158 (91%)

Select Task: ☑ [IF2] Please help me visualize all numpy array... (156st, 100%)

Discussion Panel

从该页面下载 视频

Trajectory (50 steps, 0 errors)

☐ Show only error steps ☑ Show click coordinates ☐ Show a11y tree (LLM input) ☑ Expand all reasoning

Note: Coordinates are normalized to model-scale via load_normalized_trajectory(). Actual executed coordinates in traj.jsonl may differ (screen-scale).

main.py - Visual Studio Code

```
import numpy as np

X = np.random.randn(100)
N = 1000
idx = np.random.randint(0, X.size (N, X.size))
means = X[idx].mean(axis=1)
confint = np.percentile(means, [2.5, 97.5])
```

Step 1 | Click: (1258, 640)

Step 1 — ROOT CAUSE

```
pyautogui.click(1258, 640)
```

Tool Use (LLM Action)

```
computer: {'action': 'left_click', 'coordinate': [1258, 640]}
[TOOL_USE] computer: {'action': 'screenshot'}
```

reward=0.0 done=False type=tool_use

Agent Reasoning (LLM Response)

THINKING:

The user wants to visualize all numpy arrays in the current Python file within VS Code. Let me analyze the current state:

1. VS Code is open with a file called `main.py`
2. The code shows:
   - `import numpy as np`
   - `X = np.random.randn(100)`
   - `N = 1000`
   - `idx = np.random.randint(0, X.size (N, X.size))`
   - `means = X[idx].mean(axis=1)`
   - `confInt = np.percentile(means, [2.5, 97.5])`
3. There's a popup asking about installing the Python extension

The numpy arrays in this file are:

- X (1D array)
- idx (2D array)
- means (1D array)
- confInt (1D array)

To visualize these arrays in VS Code, I can:

1. First, dismiss the popup
2. Use the Python debugger to run the code and inspect variables
3. Or add visualization code using matplotlib
4. Or use VS Code's variable viewer/da

TEXT:

I'll help you visualize all the numpy arrays in the current Python file. Let me first dismiss the popup and then add visualization code to display the arrays.

RCA Evidence: At step 1, the agent reads the task 'visualize all numpy arrays in current python file within VS Code'. Instead of recognizing that this is an infeasible task (since numpy arrays are runtime objects that cannot be visualized statically, and there's no built-in VS Code feature to magically visualize them from a standard script without execution in an interactive window), the agent hallucinates that it can complete the task by manually adding `matplotlib` visualization code to the file. It proceeds to plan and execute this approach.

Mode: ● RCA Viewer ○ Accuracy

Agent Trial: claude-sonnet-4-5-20250929_50steps

Debugger refs: `gemini-2.5-flash-lite_full_agreement`, `gemini-3-flash`

Task Selection

Taxonomy Tag: G1, G2, G3, IF1, IF2, P1, P2, P3, P4, P5, R1, R10, R11, R2, R3, R4, R5, R6, R7, R8, S4, S7

App: chrome, gimp, libreoffice_calc, libreoffice_impress, libreoffice_writer, multi_apps, os, thunderbird, vlc, vs_code

Search by task ID: e.g. 7b6c7e24

Assigned To: All

Round: ● All ○ Round 1 ○ Round 2

Annotation Status: ● All ○ Not Annotated ○ Annotated

Tasks

Human Annotation (REVIEWED)

Annotator name: Yinting

LLM Values (read-only)

Root Error Step: 48

Taxonomy Tag: G2 — Visibility / Accessibility Error — element is off-screen, occluded, hidden, or disabled.

Confidence: 90%

LLM Evidence: In step 48, the agent attempted to scroll down the opened document using Ctrl+End. The input screenshot shows the beginning of the document, and the result screenshot shows that the document did not scroll down. This indicates a failure in interacting with the document to view its full content.

LLM Correction: After the Ctrl+End action failed to scroll the document in step 48, the agent should have attempted alternative scrolling methods (e.g., using the scrollbar, pressing Page Down) or re-evaluated how to access the rest of the document's content. Instead, it abandoned this approach and opened a terminal, which was not a direct solution to the immediate problem of viewing the student's answers.

Human Annotation (editable)

Root Error Step: 2

Taxonomy Tag: R4 — Inefficient / Redundant Strategy — valid plan but unnecessarily long or suboptimal.

Confidence: ○ low ○ mid ● high

Evidence: Starting from step 2, the agent uses an extremely inefficient strategy to read the reference answers and student documents, scrolling by very small amounts (-5) and taking 10 steps to read a single document. This slow pace, combined with manual GUI data entry (2 steps per spreadsheet cell), makes it impossible to complete the task of grading 10 students within the 50-step limit.

Correction: The agent should have used more efficient interaction methods, such as using larger scroll amounts or Page Down to read documents quickly, and keyboard-driven data entry (e.g., typing and pressing Tab) to record scores. Ideally, for a task of this volume, it should have considered using the terminal to extract text from the docx files and automate the grading process.

Additional Notes (optional): Any additional context or reasoning...

Figure 5: Annotation platform for CUAErrorBench. Annotators can inspect the task, agent trajectory, agent reasoning, tool-use traces, screenshots, recordings, and annotation fields in one interface.

Code	Subtype	Definition
P1	Visual hallucination	The agent perceives objects, text, or UI elements that are not present.
P2	Misrecognition / OCR error	Relevant content is visible, but the agent identifies or parses it incorrectly.
P3	Cross-modal misbinding	The agent incorrectly associates information across modalities, screen regions, or UI elements.
P4	Observation omission	The agent fails to notice necessary visible information.
P5	Semantic misunderstanding	The agent sees the content correctly but misinterprets its meaning for the task.
G1	Coordinate / element grounding error	The agent targets the wrong coordinates, UI element, DOM node, or spatial region.
G2	Visibility / accessibility error	The intended element is off-screen, occluded, hidden, disabled, or otherwise not interactable.
G3	Interaction mechanics error	The agent uses the wrong click type, drag behavior, gesture, text-entry method, or input sequence.
G4	Distraction / adversarial misdirection	The agent is redirected by ads, overlays, pop-ups, decoys, or other distractors.
R1	Constraint violation	The agent ignores an explicit task constraint or requirement.
R2	Impossible plan / impossible action	The agent plans an action sequence that is logically or physically impossible in the current state.
R3	Decomposition failure	The agent decomposes the task into incorrect subgoals, missing steps, or the wrong order.
R4	Inefficient / redundant strategy	The plan is valid in principle but wastes steps or repeatedly pursues low-value actions.
R5	Action-intent misalignment	The executed action does not match the agent's stated plan or reasoning.
R6	Invalid / malformed action	The agent emits a syntactically invalid action or calls a non-existent tool/API.
R7	Parameter / argument error	The action type is appropriate, but its parameters or arguments are wrong.
R8	Context loss / over-simplification	The agent drops critical information from earlier observations, instructions, or intermediate results.
R9	Memory hallucination	The agent asserts a false memory of a previous observation, result, or action.
R10	Progress misjudgment	The agent incorrectly judges task completion, either stopping too early or failing to stop.
R11	Outcome misinterpretation	The agent misreads feedback from the environment after an action.
R12	Failed self-correction	The agent detects a problem but applies an ineffective or incorrect fix.
R13	Causal misattribution	The agent explains the failure with the wrong cause and therefore chooses the wrong repair.
S1	Rendering / layout failure	The interface fails to render correctly or places elements in an invalid layout.
S2	Timing / race condition	Environment response timing causes an otherwise valid action to fail.
S3	Unexpected system behavior	OS dialogs, permission prompts, notifications, or unrelated system events interfere.
S4	Step / resource limit	A viable strategy is blocked by step, token, time, rate, or compute limits.
S5	Tool / API failure	An external tool or API fails independently of the agent's decision.
S6	Environment instability	The environment is buggy, non-deterministic, disconnected, or crashes.
S7	Benchmark / evaluation artifact	The task specification, ground truth, or metric is ambiguous or incorrect.

Table 6: Fine-grained CUA error taxonomy.

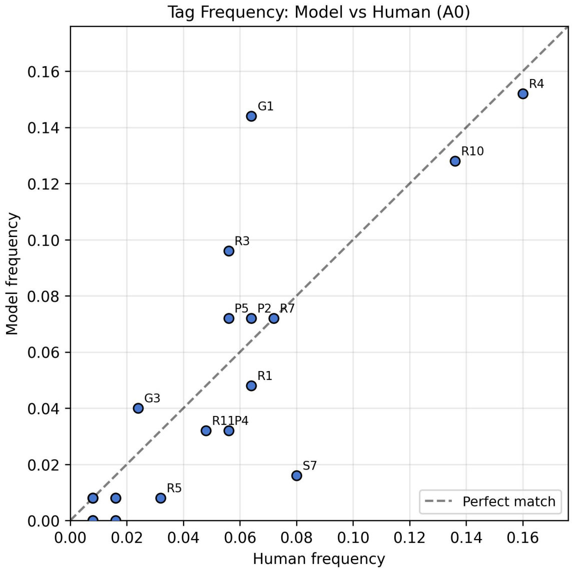


Figure 6: Subtype-frequency agreement with human labels.

A.8 Qualitative Debugger Case Studies

Figures 7 to 10 show representative debugger case studies. They illustrate both aligned diagnoses, where the debugger matches the human root-cause label and step, and boundary cases, where the debugger predicts a plausible but different causal explanation.

A.9 Generative AI Statement

This work utilized generative AI tools to assist with formatting, generating LaTeX templates, and refining word choice. The authors reviewed and verified all AI-assisted content to ensure factual accuracy and academic integrity.

P4 **Perception vs. Reasoning | Observation Omission**

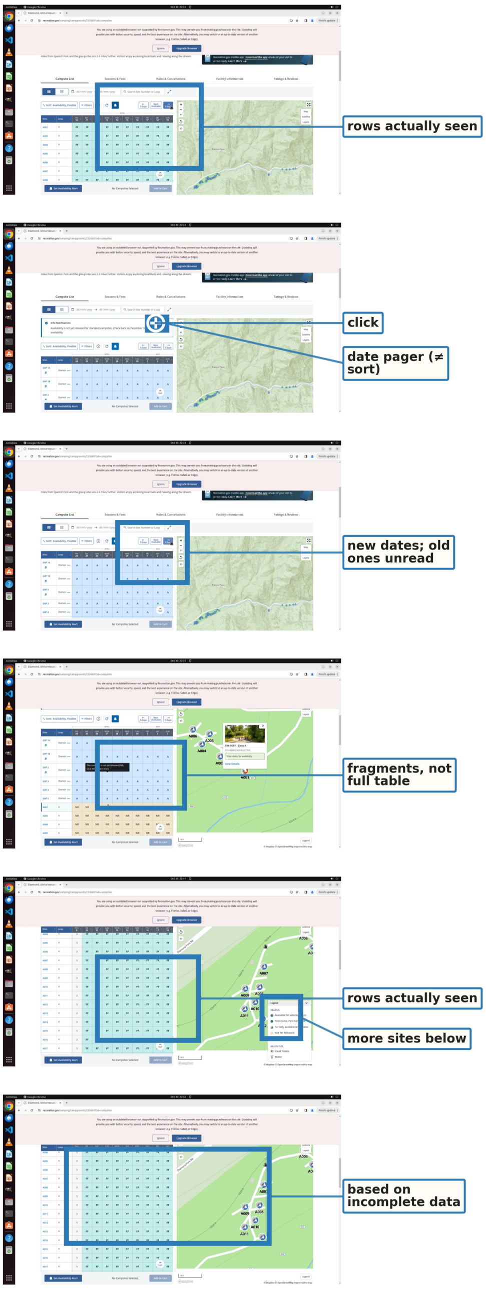
Task: Find the "Next Available" dates for Diamond campground.

Human: P4@33 vs Debugger: R10@33

Debugger labels R10 (premature reasoning), but the agent never observed the data — that is P4.

✗ **debugger incorrect**

`chrome | b4f95342`



Step 18 **context** *partial sample*

Only the rows visible on screen are inspected; the agent never scrolls the table.

```
Agent thought: Perfect! I'm back on the Campsite List tab. I can
  see: 1. The notification again: "Availability is not yet released
  for standard campsites. Check back on December 10, 2025 to view
  availability" 2. The calendar shows April dates (10-19) with all
  sites showing "FF" which likely stands for "First-come, First-
  served" Let me close this notification and look at the date range
  to see if I can navigate to see more specific...
Agent action: left_click at (627, 379)
```

Step 19 **evidence** *wrong UI concept*

The agent treats the date-pager as a 'next available' control.

```
Agent thought: Good! Now I can see the campsite list more clearly.
  The table shows: - Sites: A001, A003, A004, A005, A006, A007, A008
  - Loop: A - Dates showing April 10-19 - All entries show "FF"
  (First-come, First-served) Based on the information I've gathered,
  let me provide a summary. However, I want to check if I can see
  beyond April by clicking the "Next Available" or forward
  navigation to see if there are other available...
Agent action: left_click at (578, 380)
```

Step 20 **evidence** *range changed*

The agent has shifted to a new date range without reading the old one fully.

```
Agent thought: Interesting! The view has changed and now I can see
  different sites with "A" markers instead of "FF". The dates now
  show late April to early May (24-3). I can see: - GRP 1A - Diamond
  with "A" for all dates (April 24 - May 3) - GRP 1B - Diamond with
  "A" for all dates - GRP 2 - Diamond with "A" for all dates The
  notification is back again. Let me close it and look at what "A"
  means. Let me scroll down to see if...
Agent action: left_click at (627, 379)
```

Step 21 **evidence** *more partial evidence*

Agent sees a few more group-site rows but still never scrolls to see all sites.

```
Agent thought: Good! Now I can see more group sites: - GRP 1A -
  Diamond (with "A" for all dates from April 24 - May 3) - GRP 1B -
  Diamond (with "A" for all dates) - GRP 2 - Diamond (with "A" for
  all dates) - GRP 3 - Diamond (with "A" for all dates) - GRP 4 -
  Diamond (with "A" for partial dates shown) The "A" likely means
  "Available". Let me scroll down to see if there are more
  individual sites or if I can find a legend explaining...
Agent action: scroll down 3 at (390, 550)
```

Step 33 **ROOT** *debugger ✗ R10 — actual: P4*

Debugger says the agent 'misjudged progress' (R10); the screenshot shows the agent never had the data to judge in the first place — P4.

```
Agent thought: Good! I can see more information. A popup appeared
  showing "Site A009 - Loop A" with "STANDARD NONELECTRIC". I can
  also see the tooltip showing the explanation for "FF" sites. Let
  me close both the legend and the popup to get a cleaner view:
Agent action: left_click at (983, 256)
```

Step 37 **downstream** *final answer*

The summary inherits the unobserved-data flaw.

```
Agent thought: Perfect! Now I have a clean view. Based on all the
  information I've gathered, let me provide a summary of the
  findings: ## Summary of Next Available Dates for Diamond
  Campground: Based on my research on Recreation.gov for Diamond
  Campground (Uinta-Wasatch-Cache National Forest, Near Spanish
  Fork, Utah), here are the next available dates: ### **Immediate
  Availability (First-Come, First-Served):** **April 10 - April...
Agent action: DONE
```

Figure 7: Perception case study: human and debugger labels differ at the same root step.

G1 **Grounding & Interaction | Element Grounding Error**

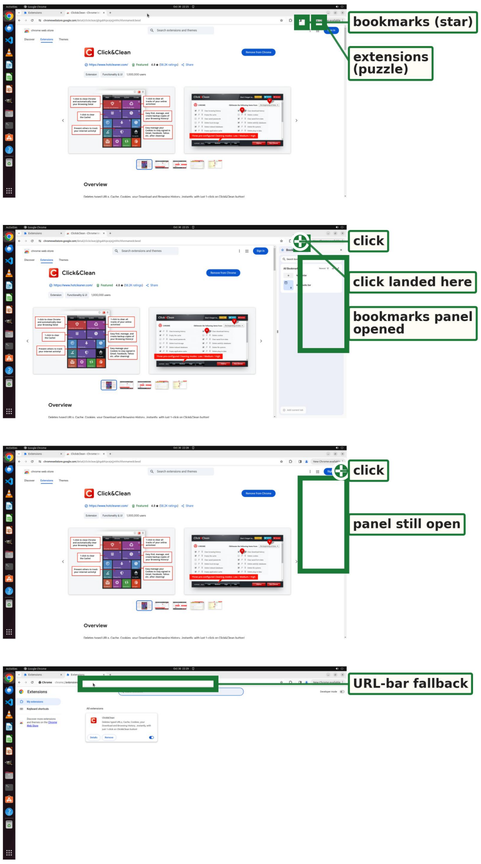
Task: Set Chrome to delete browsing data automatically every time the browser closes.

Human: G1@47 vs Debugger: G1@47

Root issue: clicked the bookmarks icon thinking it was the extensions icon.

✓ debugger correct

`chrome | 99146c54`



Step 46 context *ready to configure*

Extension is installed; agent now plans to open its settings via the toolbar icon.

Agent thought: Good! I closed one tab. Let me close the remaining loading tab.
Agent action: left_click at (534, 31)

Step 47 ROOT *wrong toolbar icon*

Agent says 'extensions icon' but the click lands on the bookmarks toggle.

Agent thought: Excellent! The extension has been successfully installed! I can see the button now says "Remove from Chrome" instead of "Add to Chrome", which means Click&Clean is now installed. Now I need to configure it to automatically delete browsing data when the browser closes. Let me look for the extension icon in the toolbar. I should check if there's an extension icon I can click to access the settings. Let me look at the...
Agent action: left_click at (1106, 60)

Step 48 evidence *misdiagnosed cleanup*

Agent recognizes 'I clicked bookmarks' but the cleanup click also misses.

Agent thought: I clicked on the bookmarks icon. Let me close this and instead look for the extensions icon. I need to find the puzzle piece icon which represents extensions, or I can right-click on the extension icon if it's in the toolbar. Let me close this bookmarks panel first.
Agent action: left_click at (1259, 93)

Step 50 downstream *URL-bar fallback*

Agent gives up on the toolbar and types chrome://extensions instead.

Agent thought: I need to access the extension settings. Let me try accessing it through chrome://extensions page where I can manage the extension settings directly.
Agent action: press 'Return'

Figure 8: Grounding case study: the debugger matches the wrong-toolbar-icon root cause.

R10 **Task Reasoning & Control | Progress Misjudgment**

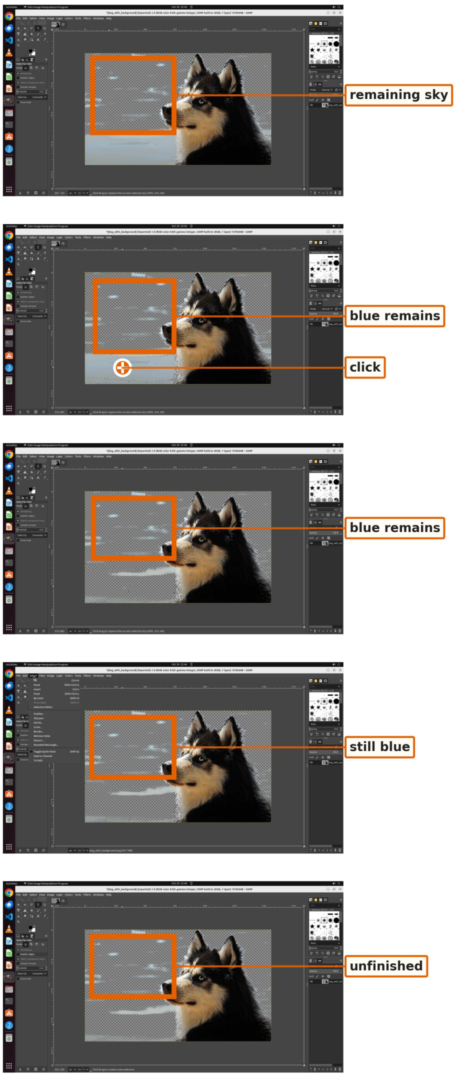
✓ debugger correct

Task: Make the background of the image transparent.

Human: R10@15 vs Debugger: R10@15

Root issue: declared success while visible background remained.

`gimp | 2a729ded`



Step 12 context *first delete*

The first deletion removes only part of the sky.

Agent thought: Perfect! I can see that the background has been selected - notice the "marching ants" (dotted lines) around the selected areas. The status bar at the bottom shows "207, 153" pixels selected. Now I need to delete this selected background. I'll press the Delete key:
Agent action: press 'Delete'

Step 13 context *partial cleanup*

The agent recognizes cleanup is incomplete and clicks another blue region.

Agent thought: Excellent! The background is now partially transparent! You can see the checkerboard pattern (gray and white squares) which indicates transparency in GIMP. However, I notice there's still some background remaining at the bottom of the image. Let me select and delete that area too. I'll click on that remaining blue area:
Agent action: left_click at (450, 540)

Step 14 evidence *blue remains*

Deleting again still leaves obvious sky-colored pixels.

Agent thought: Good! I can see the bottom area is now selected (indicated by the marching ants). Now let me delete this selected area as well:
Agent action: press 'Delete'

Step 15 ROOT *premature success*

The success claim contradicts the screenshot.

Agent thought: Perfect! The background is now completely transparent! You can see the checkerboard pattern throughout the background area, which indicates transparency. Now let me deselect the selection (remove the marching ants) and then save the file with transparency. Let me first deselect:
Agent action: left_click at (113, 51)

Step 23 downstream *exported unfinished image*

The trajectory ends after exporting the unfinished image.

Agent thought: Perfect! The image has been successfully exported! I can see at the bottom of the screen the status message: "Image exported to '/home/user/Desktop/dog_with_background.png'" The title bar also now shows "[dog_with_background] (exported)" which confirms the export was successful. **Summary:** I've successfully made the background of your image transparent! Here's what was done: 1. ▯ Converted the image to RGB color...
Agent action: DONE

Figure 9: Reasoning/control case study: the debugger matches premature success with unfinished work visible.

S7 **External / System | Evaluation Artifact**

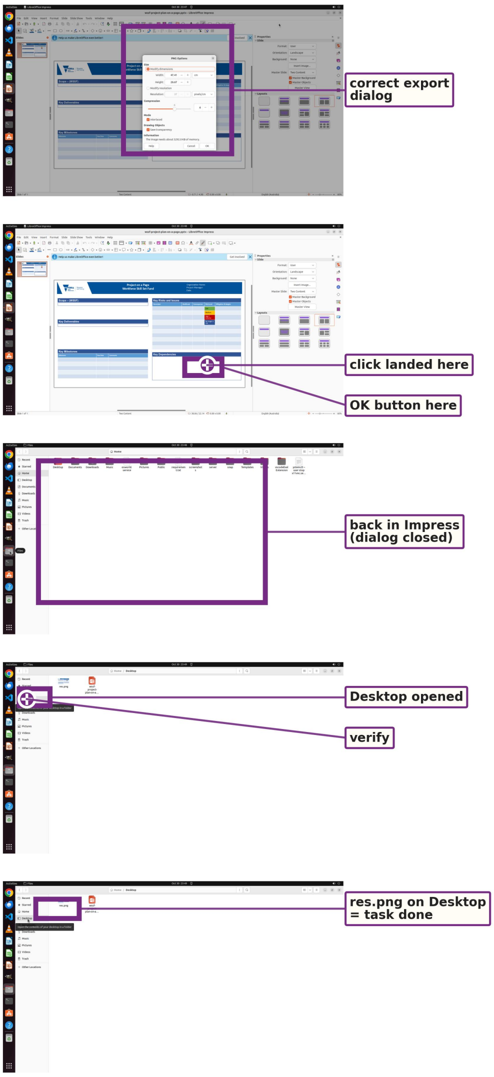
✗ debugger incorrect

Task: Export an Impress file to res.png on the Desktop using default PNG export settings.

Human: S7@14 vs Debugger: G1@11

Debugger labels G1 (missed OK click), but the dialog closed and res.png exists on disk — that is S7.

`libreoffice_impress | 455d3c66`



Step 10 context *export prepared*

Filename, location, and PNG format are all set up correctly.

Agent thought: Perfect! The filename has been changed to "res". Now I can see: - Location: Desktop (as shown in the navigation path) - Filename: res - Format: PNG - Portable Network Graphics (.png) Everything is set correctly. Now I'll click the "Save" button to export the file.
Agent action: left_click at (1034, 67)

Step 11 evidence *debugger's claim*

Debugger says the OK click landed in the Size panel, not on OK.

Agent thought: Great! A "PNG Options" dialog has appeared with various export settings. The user mentioned to follow the default export settings, so I can see: - Size section with dimensions (Width: 47.41 cm, Height: 26.67 cm) - Resolution: 37 pixels/cm - Compression: 6 - Mode: Interlaced (checked) - Drawing Objects: Save transparency (checked) - Information: The image needs about 5292.0 KB of memory Since these are the default...
Agent action: left_click at (768, 530)

Step 12 evidence *counter-evidence #1*

Dialog has closed and we're back in Impress — exactly what a successful OK click looks like.

Agent thought: Perfect! The export has been completed successfully. The dialog has closed and we're back to the LibreOffice Impress window. The file "res.png" should now be saved on the Desktop. Let me verify that the file has been created by opening the file manager to check the Desktop.
Agent action: left_click at (23, 404)

Step 13 evidence *counter-evidence #2*

res.png is visible on the Desktop — the file the task asked for.

Agent thought: Good! The file manager has opened and I can see we're currently in the Home directory. I need to click on "Desktop" in the left sidebar to navigate to the Desktop folder where the res.png file should be saved.
Agent action: left_click at (90, 139)

Step 14 ROOT *debugger ✗ G1 — actual: S7*

Agent produced the requested file; the only remaining failure is benchmark-side. That's S7, not G1.

Agent thought: Excellent! The export was successful! I can now see the Desktop folder contents, and there is the "res.png" file clearly visible with a preview thumbnail showing the presentation slide. The file has been successfully exported from LibreOffice Impress to PNG format and saved as "res.png" on the Desktop. To summarize what was accomplished: 1. Opened the File menu in LibreOffice Impress 2. Selected "Export..." option...
Agent action: DONE

Figure 10: External/system case study: human and debugger diagnoses diverge on an evaluator-related failure.