SELF-IMPROVEMENT NEEDS ATTRIBUTION: DEPENDENCY-AWARE CAUSAL MEMORY FOR LONG-HORIZON MOBILE GUI AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Mobile GUI agents are moving toward long-horizon tasks that span dozens or even hundreds of steps across multiple applications, where the working context explodes as execution progresses. Information the agent reads on earlier screens is gradually drowned in the expanding context and becomes hard to identify when needed by subsequent steps. Existing systems fold raw histories into task state or structured memory and recall items by recency or similarity. However, merely summarizing or retrieving past context fails to catch silent step errors, leaving active context polluted with unverified steps and further impairing downstream experience distillation. We introduce **CausGUI**, a self-improving framework that empowers mobile GUI agents with causal context and attribution-driven evolution, alongside MobileWorld-XL, an evaluation benchmark derived from the MobileWorld ecosystem to assess the generalization of extracted experience across unseen tasks. At its core, CausGUI maintains a causal ledger to log the causal dependencies across actions and observed evidence, enabling online step auditing, causal-dependency context assembly, and cause-attributed experience curation. Under identical conditions on three public benchmarks with a Qwen3.5-397B-A17B backbone, CausGUI consistently outperforms competitive frameworks, achieving 52.1% on MobileWorld (+16.2 over backbone), 40.6% on MemGUI-Bench (+10.9), and 51.9% on AndroidWorld hard slice (+14.9). Furthermore, task success rises across five rounds of self-evolution from 52.1% to 59.0%, with experience transferring to the unseen tasks of MobileWorld-XL. The code is available at: `https://anonymous.4open.science/r/CausGUI`.

1 INTRODUCTION

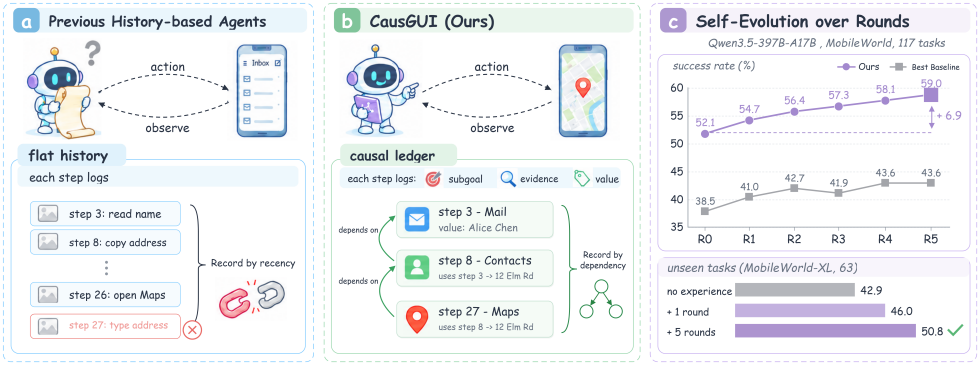


Figure 1: Why long-horizon GUI agents need attribution. (a) History kept by recency leaves a late failure's cause unknown. (b) CausGUI records each step's subgoal, cited evidence and value. (c) Attributed causes compound over self-evolution and carry to unseen tasks (Tables A2 and A10).

Mobile GUI agents have emerged as a powerful paradigm for automating smartphone interactions (Wang et al., 2024a; Wen et al., 2024). While existing agents handle simple, short-horizon

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.

routines effectively, real-world utility increasingly demands tackling complex workflows that span dozens or even hundreds of steps and involve intricate dependencies across execution stages (Kong et al., 2026; Tian et al., 2026). Successfully executing these long-horizon tasks is essential for truly autonomous mobile assistants, as it enables agents to connect scattered information, carry out multi-stage user goals, and sustain coherent interaction over extended horizons.

However, managing such extended trajectories with complex dependencies poses fundamental challenges. Execution errors on long-horizon tasks are subtle and deceptive: actions often take effect on plausible but wrong targets, leaving errors hidden until the task fails many steps later (Lee et al., 2025). Simultaneously, recent studies highlight pervasive context anomalies such as goal drift, progress hallucination, and repeating actions on stale screens (Shi et al., 2026; Liu et al., 2026a;c). As execution progresses, the expanding, unorganized context triggers a severe "lost in the middle" phenomenon (Liu et al., 2024)—earlier critical evidence gets drowned in historical noise while unattributed actions and stale updates pollute working memory. Consequently, distilling experience purely from terminal outcomes retains these unexamined intermediate missteps, hindering effective self-improvement. This raises a fundamental question: how can an agent systematically untangle such errors across long-horizon executions and enable reliable self-evolution?

Existing methodologies attempt to address this problem from distinct angles. On the execution front, modern agent frameworks range from single-model loops grounded in UI documentation (Zhang et al., 2025) to planner–executor designs that monitor screen states and task progress (Ye et al., 2025). However, these approaches focus primarily on local step checks, leaving downstream-failure tracing shallow and unreliable. On the context front, while structured task states dynamically compress execution history (Liu et al., 2026b), such compression merely organizes past events without establishing inter-step causal dependencies. Context selection thus still relies on recency or surface similarity, leaving the logic of accumulated history opaque. On the experience front, skill induction and lesson extraction methods (Wang et al., 2025; Jiang et al., 2025; Li et al., 2026b) distill reusable tips directly from full trajectories. However, distilling lessons directly from unverified execution trails risks propagating undetected missteps, allowing flawed strategies to pollute future learning iterations.

Bridging these gaps presents three main challenges. First, erroneous actions often pass unnoticed during execution and remain hard to localize afterwards, as an action usually appears plausible at the moment while the actual failure surfaces dozens of steps later. Second, determining what context a step requires is governed by explicit information and factual dependencies, which simple history truncation or surface similarity search fails to capture. Third, experience learning remains vulnerable when credit assignment is ungrounded, as fluke successes can disguise latent flaws and uninspected failure trajectories risk re-introducing suboptimal actions into future tasks.

To address these limitations, we present CausGUI, a self-improving framework that empowers mobile GUI agents with causal context and attribution-driven evolution. As illustrated in Figure 1, CausGUI anchors its lifecycle on a unified causal ledger that records the targeted subgoal, cited evidence, and observed values for every step. Over this ledger, the attribution auditor reads backward to verify execution and trace surfaced failures to their root steps, the dependency resolver reads forward to assemble working context from explicit dependencies while reducing stale noise, and the self-evolution curator reads across tasks to synthesize persistent capabilities from causal memory relationships, updating and refining them through reuse. Experiments on long-horizon benchmarks show that CausGUI consistently outperforms strong agent frameworks, with gains that compound over self-evolution and transfer to unseen tasks.

Our main contributions are as follows:

- We establish causal attribution for the long-horizon execution of mobile GUI tasks. Each step is recorded together with the subgoal it serves, the evidence it cites, and the values it observes. A failure that surfaces dozens of steps later can therefore be traced to the step where it began, extending verification from local step checks to trajectory-level root-cause localization.
- We design dependency-aware context management. Working context is assembled from each step's explicit dependencies. Evidence read on earlier screens therefore remains available many steps later and across applications, while stale or out-of-scope entries are filtered out.

- We introduce attribution-driven self-evolution. Experience is admitted only when it is grounded in a localized cause, and its survival is decided by reuse across distinct tasks. The experience store thus improves with use, and entries whose faults recur are retired.

2 Related Work

Architectures and context management. Mobile GUI agents have progressed from single-model ReAct loops (Zhang et al., 2025; Zheng et al., 2024a; Rawles et al., 2025) to hierarchical planner–executor designs (Agashe et al., 2025; Ye et al., 2025; Li et al., 2026b). Hierarchical supervisors help decompose complex tasks but intervene on rigid schedules or vague distress signals, lacking evidence to pinpoint where an execution derailed. Concurrently, context representation has evolved from rolling summaries to structured task states (Shi et al., 2026) and actionable context objects (Liu et al., 2026b; Tian et al., 2026), with retrieval-augmented knowledge shared across a multi-agent pipeline (Zhou et al., 2026). Even so, existing agents still pull past information by recency or surface similarity rather than actual subgoal dependencies, and step verifiers (Dai et al., 2026) or periodic reflectors (Shinn et al., 2023) assess local screen changes rather than goal alignment, letting actions that alter the wrong input field slip through.

Experience learning and self-evolution. Prior research distills reusable GUI experience through diverse paradigms. One line contrasts paired successes and failures into textual heuristics (Zhao et al., 2024; Ouyang et al., 2026; Wang et al., 2025), while another replays full successful trajectories as in-context exemplars (Zheng et al., 2024b). A third direction converts repeated execution paths into structured macro-actions or parameterized skills (Wang et al., 2024b; Jiang et al., 2025; Li et al., 2026b), and recent work explores evolving memory stores that prune stale entries via usage counters (Mi et al., 2026; Tang et al., 2026). Closest to us, self-evolving stores layer workflow, skill and failure memories under online reinforcement learning (Xiao et al., 2026) or distill lessons from failed exploration (Lin et al., 2026). Both still key admission on the outcome of a whole trajectory, whereas we admit an entry only once an auditor has localized the step that produced it and named its evidence, since a trajectory-level label cannot say which of thirty steps earned the credit. Across these paradigms, admission and survival hinge on final task completion or retrieval frequency; yet terminal success is an outcome, not a causal explanation, so fluke successes canonize erroneous steps as guidance. Injecting distilled experience into prompt text likewise offers no verifiable link between what was retrieved and what was used, leaving stores open to noise and negative transfer.

Long-horizon mobile evaluation. While static trajectory datasets (Rawles et al., 2023; Zhang et al., 2024; Li et al., 2024) lack interactive environment feedback, popular interactive benchmarks (Rawles et al., 2025; Xu et al., 2025) consist mostly of brief, single-app workflows. To evaluate agents under realistic multi-app complexity, recent benchmarks like MobileWorld (Kong et al., 2026) and MemGUI-Bench (Liu et al., 2026c) introduce tasks spanning dozens of steps and extensive cross-app value propagation. Under these settings an agent often executes plausibly across several apps, only to fail at delivery because an early-observed identifier was corrupted or forgotten midway. These environments expose silent wrong-target actions, deeply buried root causes, and context pollution that erases early evidence, calling for explicit causal dependencies across steps and working memory governed by task needs rather than surface retrieval.

3 CausGUI

3.1 Framework Overview

As illustrated in Figure 2, CausGUI coordinates an executing Actor and a supervisory Manager around a shared **causal ledger**. The Actor proposes actions with their declared effects and cited ledger grounds, while the Manager intervenes non-interactively to adjust plans, verify subgoals or diagnose failures. Both roles only propose; a deterministic, model-free runtime manages state transitions, mounting and admission under strict ledger rules (Appendix A). Rather than a flat log, the causal ledger binds every entry's effect to its supporting rationale, enabling three unified operations: the **attribution auditor** validates steps provisionally against screen transitions and authoritatively at subgoal boundaries, backtracking along citations to isolate root causes; the **dependency resolver**

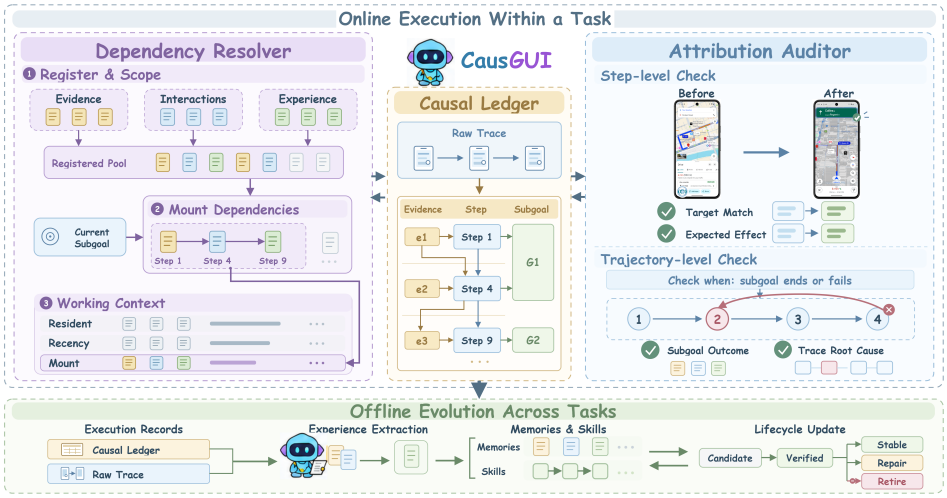


Figure 2: Overview of CausGUI. Within a task (top), execution writes one causal ledger, the resolver mounts what the active subgoal depends on, and the auditor walks a failed subgoal back to the step that introduced the error. Across tasks (bottom), attributed records are distilled under cause, effect and contrast.

projects ledger entries into the Actor's context strictly by the active subgoal's requirements; and the **self-evolution curator** distills verified causal paths into reusable memories and skills across tasks.

3.2 Attribution Auditor

The auditor localizes the step where an error began. It targets errors invisible on the spot that surface at settlement or task end, and rests on the ledger's **support graph**, a **backtracker** that layers the relevant entries, and a two-level **verifier**: a deterministic runtime check on what the Actor declares each step, and the Manager's ruling at each subgoal boundary.

Support graph. The ledger holds three kinds of entries. A *subgoal* is written by the Manager at planning time, together with a declaration of which earlier subgoals' products it depends on. A *step* is the record of one action, $r_t = (s_t, a_t, o_t, e_t, B_t)$: the screen s_t, the action a_t, the target object o_t, the declared expected effect e_t, and the grounds B_t, the evidence and earlier steps this decision cites; every step belongs to the subgoal it serves. A piece of *evidence* $f \in \mathcal{F}$ is a value read from a screen and needed later; it carries the step that read it and every step that used it. All connections point to the past, so the ledger forms a directed acyclic support graph leading a wrong value straight to the step that read it.

Backtracker. Run by the runtime when a subgoal fails, the backtracker organizes the ledger into a two-layer view for verification. Its default scope is the failed subgoal; the Manager can request an earlier one when needed. The *object layer* groups steps by what they revolve around: steps touching the same evidence under that evidence, repeated attempts on one screen under that screen with its dead ends. The *step layer* expands a group on demand into its concrete steps, each with before-and-after screenshots, the declared grounds and expectation, and the actual outcome. Compression comes from grouping: a suspicious value carries under its name the steps that read and used it.

Verifier. Verification runs at two granularities. At the *step level*, the Actor declares target object o_t, expected effect e_t, and grounds B_t before acting. The runtime admits proposals through an action gate and checks the post-action transition:

$$\hat{v}_t \;=\; \mathrm{gate}(a_t) \;\wedge\; \nVdash[\, s_{t+1} \models e_t \,] \;\wedge\; \nVdash[\mathrm{obj}(s_{t+1}) = o_t\,], \tag{1}$$

where $\mathrm{gate}(a_t) = \nVdash[(s_t, a_t) \notin \mathcal{D}] \wedge (\mathrm{irrev}(a_t) \Rightarrow \mathrm{grounded}(B_t))$ ensures that proposals avoid recorded dead ends $\mathcal{D}$ and irreversible actions possess fully sourced grounds. Violations register as dead ends on s_t and trigger retries.

At the *trajectory level*, the Manager rules on every subgoal's product at its boundary, and takes over early on settlement failure, retry exhaustion or Actor request. Inspecting the layered graph, it identifies the root failure r° as the earliest step in dependency order whose execution fails despite valid grounds:

$$r^\circ \;=\; \min_{\preceq}\Big\{\, r \;:\; \mathrm{ok}(r)=0 \;\wedge\; \textstyle\bigwedge_{b\in B(r)} \mathrm{ok}(b)=1 \Big\}, \tag{2}$$

where $\preceq$ denotes the partial order over the support graph and $\mathrm{ok}(\cdot)$ is the Manager's verdict. A corrupted value propagates inspection upstream to its reading step; if every step is valid, the failure is planning-level and the plan is revised. The verdict—defect, recovery action, rollback target—is committed to the ledger to guide later mounting and curation.

3.3 Dependency Resolver

The dependency resolver structures context into three tiers: the **resident zone** holds the instruction, plan state and verified evidence; the **recency zone** keeps recent transitions, the **mount zone** surfaces what the active subgoal needs. Its lifecycle: registration, mounting, arbitration.

Registration. Every write into the ledger, whether an observed piece of evidence, a failed route, a diagnostic verdict, or newly fetched experience, passes through registration to prevent context pollution. Registration assigns every entry x an address in the graph and maps it to an applicability scope $\mathrm{scope}(x)$:

$$\mathrm{scope}(x) \;=\; \begin{cases} \big\{(s,g) \mid g \succ_{\mathcal{G}} g_{\mathrm{src}}(x)\big\}, & x \in \mathcal{F} \quad \text{(evidence)},\\ \big\{(s,g) \mid s = s_{\mathrm{src}}(x),\; g = g_{\mathrm{src}}(x)\big\}, & x \in \mathcal{D}\cup\mathcal{V} \quad \text{(dead ends / verdicts)},\\ \big\{(s,g) \mid \mathrm{pre}(x) \sqsubseteq (s,g)\big\}, & x \in \mathcal{E} \quad \text{(experiences)}, \end{cases} \tag{3}$$

where $\succ_{\mathcal{G}}$ is downstream reachability in the graph and $\mathrm{pre}(x)$ gives app and screen preconditions. Scope boundaries re-index whenever the plan updates, filtering history by structure, not truncation.

Mounting. At the start of each step, mounting assembles the mount zone from current subgoal g_t and screen s_t by taking the union of three scope-checked sources:

$$M_t \;=\; \mathrm{dep}(g_t) \;\cup\; \mathrm{mem}(s_t, g_t) \;\cup\; \mathrm{exp}_k(s_t, g_t) \;\subseteq\; \big\{\, x \;:\; (s_t,g_t) \in \mathrm{scope}(x) \,\big\}. \tag{4}$$

Here, $\mathrm{dep}(g_t)$ mounts the declared upstream products; $\mathrm{mem}(s_t, g_t)$ surfaces the current screen's failed routes and verdicts to prevent repeats; and $\mathrm{exp}_k(s_t, g_t)$ retrieves the top-k deployable cross-task experiences whose scope covers the situation, ranked by a hybrid lexical-embedding score:

$$\mathrm{exp}_k(s_t, g_t) \;=\; \underset{e\,:\,(s_t,g_t)\in\mathrm{scope}(e)}{\overset{k}{\arg\mathrm{top}}} \Big[\, \lambda\, \mathrm{sim}_{\mathrm{lex}}(e, q_t) + (1-\lambda)\, \mathrm{sim}_{\mathrm{vec}}(e, q_t) \,\Big], \tag{5}$$

where q_t encodes the current app, subgoal, and screen state. Crucially, retrieval only nominates entries; scope boundaries govern what is mounted. Mounting also maintains the resident evidence table: fresh readings overwrite stale values, and items overturned by verdicts are purged.

Arbitration. When co-present entries conflict or provide contradictory guidance on the same UI element, the Actor triggers arbitration. Adjudication compares each conflicting entry's provenance, including source task, screen context, lifecycle status, and write timestamp. The losing entry is written back with a tightened scope or a demoted retrieval rank, so identical conflicts do not recur; permanent retirement is deferred to lifecycle management.

3.4 Self-Evolution Curator

The curator runs between tasks in three stages: extraction, reuse, and update.

Extraction. Once a batch finishes, an extraction agent walks it with the ledger as index and produces two kinds of entries. A *memory* comes from a verdict, whether or not the task later succeeded: the fault's situation becomes its scope and the stated fix its content, both pointing at a ledger step;

situation, fault and fix must all be present. An extracted experience $e = (c_e, y_e, \Delta_e)$ enters only when it binds a verified cause c_e, the observed effect y_e and a stated counterfactual contrast Δ_e:

$$\mathrm{admit}(e) \;=\; \mathbb{1}[r^\circ \in \mathrm{root}(\mathcal{V})] \;\wedge\; \mathbb{1}[B(r^\circ) \vdash c_e] \;\wedge\; \mathbb{1}[\mathrm{eff}(r^\circ) = y_e] \;\wedge\; \mathbb{1}[\Delta_e \neq \varnothing]\,, \qquad (6)$$

where $\mathrm{root}(\mathcal{V})$ collects the steps that the verdicts in $\mathcal{V}$ name as root causes, grounding every admitted memory in a localized cause r°. Whether the contrast holds decides the entry's initial state: when the task succeeded after the repair, $\Delta_e \models y^*$ is execution-verified and the entry enters as *verified*; a fix stated in a task that failed or aborted has a pending contrast and enters as a *candidate*. A candidate is held in shadow: stored and counted but never mounted until later tasks corroborate it, so only verified experience reaches the Actor. A *skill* comes from verified trajectories and enters as verified: a repeatedly walked route becomes preconditioned steps with task-varying parameters as slots. Eq. 6 reads on a route through its three terms: the cause is the subgoal class the route serves, the effect is the terminal check its path passed, and the contrast is what distinguishes it from the routes already keyed on that class, so a route that restates an existing one is merged rather than admitted. Causes recurring across apps become *global memory*; app-specific entries form the *per-app tier*.

Reuse. Both tiers are retrieved together, keyed on app, subgoal, and screen, each entry filtered by its own scope; on conflict the more specific tier prevails, and a conflict that survives the tier order goes to arbitration (§3.3). Entries enter only through the resolver's mounting condition and must be cited in the grounds when used; a memory acts as a reminder. A skill executes stepwise, each step gated by its precondition and confirmed by step-level verification; any mismatch aborts to the Actor, and the abort point is written back for end-of-round repair.

Update. Admission binds accountability rather than correctness; correctness is settled by reuse. Each entry m carries two counters: $n^+(m)$, the number of distinct tasks that succeeded with m present, or, for a candidate in shadow, that succeeded on the situation m describes; and $n^-(m)$, the number of distinct tasks in which attribution has charged a verdict to m. A verdict is charged when the described fault recurs with m present or when m is cited as a ground of the root-cause step, so a candidate, never present, cannot be charged. Tasks are counted by identity, so re-running a task in a later round never counts twice, and both counters restart at every state change, so each promotion needs $k_\uparrow$ fresh successes. At round end the counters drive updates: proven entries advance candidate→verified→stable and rise in retrieval order; an aborted skill is repaired in place. The state advances by

$$\sigma_{t+1}(m) \;=\; \big[\, n_t^-(m) < k_\downarrow \,\big] \cdot \min\!\big\{\sigma_t(m) + \mathbb{1}\big[n_t^+(m) \geq k_\uparrow\big],\, \sigma_{\max}\big\}, \qquad (7)$$

so an entry is removed at the end of the round in which the verdicts charged to it reach $k_\downarrow$; $k_\uparrow$ and $k_\downarrow$ are in the appendix. The store advances in immutable generations (Appendix B.1): entries are untouched within a round and updated together at its end, so each round is attributable to one version.

4 Experiments

4.1 Experimental Setup

Benchmarks. Our main testbed is MobileWorld's GUI-only subset, 117 interactive cross-app tasks on a real device,[1] with its deterministic checkers. To probe generalization we introduce MobileWorld-XL, 63 unseen tasks in the same app ecosystem, each with a unique answer checked by the same backend; 44 recombine verified subtasks with a novel one, and 19 are entirely new, reusing no verified subtask (construction and audit in Appendix C). MemGUI-Bench (128 tasks) is a memory-side supplement judged by MemGUI-Eval, and the AndroidWorld hard slice, 27 tasks of crowd-rated complexity at least 2.4, adds a third distribution; both under their official budgets (Appendix B.1).

Baselines. We compare against two groups, both run by us on the same harness and backbones. End-to-end GUI models come first; the backbones themselves appear in this group, anchoring how much a framework rather than the model contributes. Agentic frameworks follow; the set is listed

[1] The release lists 117 GUI-only tasks; the paper reports 116. We use the release and its 50-step budget.

Table 1: Long-horizon results behind Figure A1: MobileWorld, MemGUI-Bench and the AndroidWorld hard slice, no prior experience. SR is single-run success at the task-count resolution of Section 4.1; Steps is mean steps per task. Per column and block, **bold** is best and underline the runner-up.

	MobileWorld		MemGUI-Bench		AndroidWorld	
Agent	**SR**	**Steps**	**SR**	**Steps**	**SR**	**Steps**
END-TO-END MODELS						
GUI-Owl-1.5-32B-Instruct (Ye et al., 2025)	**36.8**	**26.1**	12.5	48.6	**40.7**	**33.8**
Qwen3.5-397B-A17B (Team, 2026)	35.9	27.2	**29.7**	**35.3**	37.0	34.9
MAI-UI-235B-A22B (Zhou et al., 2025)	33.3	26.6	18.0	39.8	33.3	36.7
GUI-Owl-1.5-8B-Instruct (Ye et al., 2025)	30.8	28.4	10.2	44.4	29.6	38.9
Qwen3.5-35B-A3B (Team, 2026)	22.2	27.2	21.1	38.9	25.9	40.8
Qwen3-VL-235B-Thinking (Bai et al., 2025)	13.7	28.6	23.4	38.7	22.2	43.7
ScaleCUA-7B (Liu et al., 2026d)	6.0	33.6	3.9	49.4	3.7	48.6
AGENTIC FRAMEWORKS						
BACKBONE: *Qwen3.5-35B-A3B*						
KnowAct-GUIClaw (Li et al., 2026b)	27.4	**22.6**	21.1	37.1	33.3	34.7
MemGUI-Agent (Liu et al., 2026b)	34.2	25.6	25.0	**34.1**	29.6	35.8
OpenGUI	24.8	25.1	19.5	40.4	25.9	35.9
ClawGUI (Tang et al., 2026)	20.5	23.2	18.8	38.3	18.5	**32.9**
Mobile-Use (Li et al., 2026a)	12.0	24.6	11.7	39.9	11.1	35.1
Mobile-Agent-V3 (Ye et al., 2025)	17.1	27.1	14.8	41.4	18.5	37.1
Agent-S2 (Agashe et al., 2025)	15.4	41.6	15.6	48.3	22.2	43.9
Mobile-Agent-E (Wang et al., 2025)	4.3	48.6	5.5	49.7	0.0	44.2
M3A (Rawles et al., 2025)	16.2	24.7	18.0	40.6	14.8	36.1
T3A (Rawles et al., 2025)	12.8	31.7	15.6	42.2	11.1	37.4
AppAgentx (Jiang et al., 2025)	6.0	46.1	3.1	49.3	3.7	42.7
SeeAct (Zheng et al., 2024a)	2.6	34.9	1.6	46.9	0.0	43.3
CausGUI	**39.3**	26.8	**31.3**	34.3	**37.0**	33.1
BACKBONE: *Qwen3.5-397B-A17B*						
KnowAct-GUIClaw	38.5	**21.1**	32.8	**33.4**	44.4	34.2
MemGUI-Agent	46.2	24.9	35.2	33.6	44.4	33.8
OpenGUI	33.3	23.7	30.5	38.9	37.0	34.7
ClawGUI	28.2	21.6	27.3	33.7	25.9	32.8
Mobile-Use	16.2	23.0	18.8	38.4	18.5	34.6
Mobile-Agent-V3	22.2	25.6	24.2	40.1	22.2	35.2
Agent-S2	23.9	40.0	25.8	46.8	29.6	42.7
Mobile-Agent-E	6.0	45.1	7.0	48.6	3.7	43.1
M3A	23.1	23.3	28.9	39.2	25.9	35.6
T3A	17.1	30.1	22.7	40.8	18.5	36.3
AppAgentx	8.5	44.9	3.9	48.2	11.1	40.3
SeeAct	4.3	33.4	2.3	45.4	0.0	41.8
CausGUI	**52.1**	25.3	**40.6**	33.9	**51.9**	**31.9**

in Table 1. Both blocks share one harness: the 35B block runs Qwen3.5-35B-A3B for the Actor, Manager and post-run extraction alike, so no stronger model hides in the loop, while Qwen3.5-397B-A17B probes the ceiling and is the backbone of every appendix analysis. Two frontier backbones outside the Qwen family, Kimi-K2.5 and Kimi-K2.6, are run in Appendix B.3 to check that the gain neither depends on one family nor vanishes on a far stronger backbone. We fix $N{=}3$ and $k_{\uparrow}{=}k_{\downarrow}{=}2$ throughout; sweeps are in Appendix A.4, three-seed repeats in Appendix B.4, and tokens and calls in Appendix B.5.

Metrics. Success rate is at task-count resolution (pass@1) from the seed-0 run: $k/117$ on MobileWorld, $k/128$ on MemGUI-Bench, and $k/27$ on the AndroidWorld hard slice. One step is one submitted environment action; re-selections after rejected proposals cost no step, whereas a retry after a failed next-screen check does (Appendix B.1). Average steps per task accompany the tables.

4.2 MAIN RESULTS: LONG-HORIZON EXECUTION

(1) Overall. Table 1 reports the main comparison, visualized in Figure A1. CausGUI leads the strongest re-run framework in all six backbone–benchmark cells, by 3.7 to 7.5 points, and on the 397B block it adds 16.2 points on MobileWorld over its backbone where the best baseline adds 10.3, clearing the best end-to-end model. The same ordering holds on the 35B block and on two frontier non-Qwen backbones (Appendix B.3), so it comes from the framework rather than the

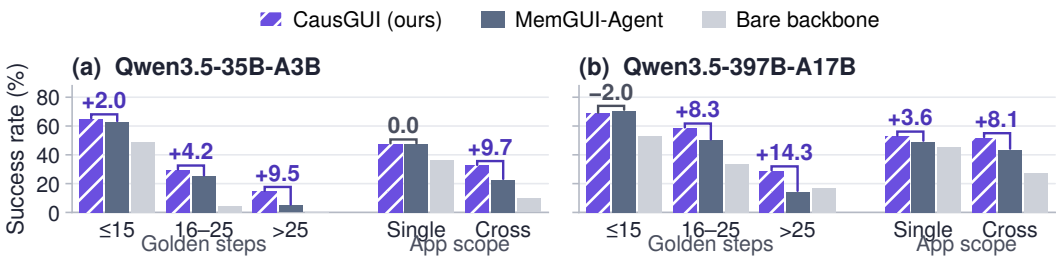


Figure 3: Where the gains come from: MobileWorld success rate by task length (golden steps) and app scope, for CausGUI, MemGUI-Agent (strongest re-run framework) and the bare backbone. Brackets give our margin in points; counts in Table A1.

Table 2: Additive ablation on MobileWorld, two backbones; rows A–D in the text. Mgr. calls / prompt: Manager invocations per task and their prompt tokens; Tokens: prompt and output over all calls. Backbone and hyperparameters fixed; SR in %.

	Config	SR	Steps	Mgr. calls	Tokens (k)	Mgr. prompt (k)
	BACKBONE: *Qwen3.5-35B-A3B*					
A	bare backbone	22.2	27.2	–	276.0	–
B	+ ledger & resolver	33.3	25.9	1.8	220.1	17.2
C	+ auditor	39.3	26.8	6.4	281.0	59.6
D	+ curator (R5)	**42.7**	**25.4**	5.6	268.0	52.1
	BACKBONE: *Qwen3.5-397B-A17B*					
A	bare backbone	35.9	27.2	–	282.0	–
B	+ ledger & resolver	46.2	24.0	1.6	218.2	16.5
C	+ auditor	52.1	25.3	5.8	284.5	60.4
D	+ curator (R5)	**59.0**	**22.7**	4.9	270.5	51.0

model. Three-seed repeats give standard deviations of 0.9–2.2 on the large benchmarks and up to 3.7 on the AndroidWorld slice, with CausGUI ahead of every baseline at every seed (Appendix B.4).

(2) Where the gains come from. The gain concentrates where the mechanisms act: long golden paths and cross-app tasks (Figure 3; counts in Table A1). Beyond 25 golden steps the margin over the strongest re-run framework is 14.3 points on the 397B block and 9.5 on the 35B block, against within two points at $\leq$15 steps: short tasks leave little room for attribution or memory to matter. Across app scope, the margin is 8.1 points on the 397B block and 9.7 at 35B on cross-app tasks, where values must survive transfer, against at most 3.6 on single-app tasks; on the long bucket the three strongest frameworks fall below the bare backbone on the 397B block: folding the history compresses away values needed many steps later. Appendices B.6 and B.7 measure both mechanisms directly.

(3) Accuracy–efficiency. The Manager wakes sparsely: on the 397B block it is called 5.8 times per task (23% of steps), and the full system spends 284.5k tokens per task against 282.0k for the bare backbone (+0.9%), since the zoned prompt saving nearly covers the Manager's calls (Table A6). Extra budget becomes progress rather than repetition: raising the step cap from 50 to 100 adds five tasks for CausGUI and two to three for the baselines (Table A5).

4.3 Ablation Studies

Table 2 builds the system up rather than removing components: no row disables a mechanism another still depends on. Row A is the bare backbone (linear history); row B adds the causal ledger with the dependency resolver; row C adds the attribution auditor, reaching the cold-start system; row D adds the self-evolution curator with the R5 store. The ledger has no row: writing it is part of acting, present from row B onward. The ladder runs on both backbones; the ordering holds.

Each increment attributes one component's share. The ledger with the resolver adds 11.1 points on the 35B block and 10.3 on the 397B block, and already shortens the prompt, since the three zones replace the raw history. The auditor adds 6.0 and 5.9 points: with the context already clean, this is verification and backtracking's share, priced in the Mgr. columns of row C. The curator adds

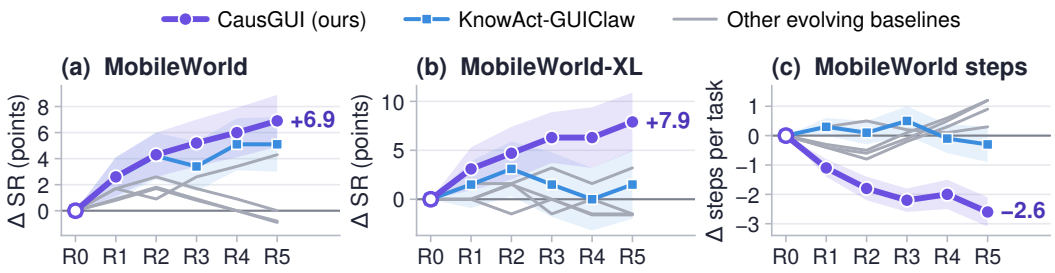


Figure 4: Self-evolution over five rounds (Qwen3.5-397B-A17B; store frozen within a round). (a) Success-rate gain on the 117 MobileWorld tasks over the no-experience run R0. (b) Gain on the 63 unseen MobileWorld-XL tasks: transfer only, the MobileWorld store after each round mounted frozen (hollow marker, no experience). (c) Change in average steps per task. Bands: ±1 std over three seeds for CausGUI and KnowAct-GUIClaw; grey: other evolving baselines (Appendix B.1).

3.4 and 6.9 points while trimming Manager calls at unchanged per-call cost and lowering the total, since skills execute routes without per-step decisions. On both blocks steps fall with the resolver, rise at row C, and fall again at row D: on-the-spot retries are executed actions, so verification buys correctness first and brevity once the store carries the verified routes.

4.4 Self-Evolution and Generalization

Results. Tasks run R1→R5 with the store seeded from the R0 trajectories, on the frameworks that can ingest the executed trajectories (protocol and exclusions in the caption). CausGUI rises from 52.1 to 59.0 SR, improving at every round, while its average steps fall from 25.3 to 22.7 (Figure 4; Table A2); KnowAct-GUIClaw, the strongest baseline, plateaus at 43.6 by R4, and two baselines end below their cold start, since their stores admit every entry and never retire one. CausGUI's shrinking gains track its shrinking admissions, and attributed negative transfer falls from 3.1% to 0.9% as the store converges (Table A8, Appendix B.8). An episodic control that replays the most similar raw trajectories as exemplars (Zheng et al., 2024b) is back at its cold start by R5 with negative transfer at 6.8%: replayed episodes re-inject the failed routes the resolver excludes.

Generalization. MobileWorld-XL splits 70/30 into 44 tasks that chain verified source subtasks with a novel one and 19 that reuse nothing from the source (Table A9). Every column is transfer alone: the store each method evolved on MobileWorld after round r is mounted frozen here, and nothing is learned on MobileWorld-XL. With no experience CausGUI scores 42.9; the round-1 store already lifts it to 46.0 and the round-5 store reaches 50.8, while the baselines gain at most 2 tasks at any round and three end below their cold run (Figure 4b, Table A10). Both strata improve, by 4 and 1 tasks, and the round-1 store gains only in the recombination stratum, where a verified subtask gives a skill something to fire on (Table A11). Splitting the store by entry type, skills alone carry that whole gain of 2 tasks and memories alone add none; memories pay off as the source store matures, putting the full store 2 tasks ahead of skills alone by R5 (Table A10). MobileWorld-XL shares apps and backends with the source set, so we claim compositional generalization, not out-of-distribution robustness.

5 Conclusion

We presented CausGUI, a self-improving framework that gives mobile GUI agents causal context and attribution-driven evolution. Execution writes one causal ledger that three components read: the auditor backward to the failing step, the resolver forward into each step's context, the curator across tasks into reusable experience. Under identical conditions it outperforms competitive baselines on all three benchmarks and backbones, improves over five rounds of self-evolution, and carries to MobileWorld-XL. Grounding experience in verified causes is a practical route to agents that improve with use rather than accumulate noise.

Reproducibility Statement

Appendix A specifies the deterministic runtime: the per-step timeline, the ledger entry templates and effect-matching rules, the extraction tools and frame budgets, and one-at-a-time sensitivity sweeps of every threshold. Appendix B documents the evaluation protocol and step semantics, the per-benchmark budgets and scoring, the full result tables, frontier-backbone checks, three-seed repeats, the accounting of steps, tokens and calls, and the attribution, context and experience-store audits. Appendix C reports the construction and audit of MobileWorld-XL, and Appendix D the failure analysis behind the targeted mechanisms. Appendix E walks two executed trajectories step by step, and Appendix F gives the system prompts of the model roles together with the per-step context template the runtime fills. Every benchmark is public and is run with its official task set, checker and step budget; hyperparameters are fixed across all systems and backbones; and every baseline shares our harness. The experience store is frozen within a round, so each self-evolution round is attributable to one store version, and transfer to MobileWorld-XL is measured with that store held frozen.

AI Use Statement

We used general-purpose AI assistants for literature search, manuscript drafting and language editing, and code development in the framework implementation, the evaluation harness, and the plotting scripts. All AI-assisted material was reviewed by the authors. Research questions, method design, experimental decisions, and the reported measurements remained under the authors' control. The authors take full responsibility for the paper's technical content, results, conclusions, and correctness.

References

Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent s2: A compositional generalist-specialist framework for computer use agents. *arXiv preprint arXiv:2504.00906*, 2025.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, et al. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*, 2025.

Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. V-droid: Advancing mobile gui agent through generative verifiers. 2026.

Wenjia Jiang, Yangyang Zhuang, Chenxi Song, Xu Yang, Joey Tianyi Zhou, and Chi Zhang. Appagentx: Evolving gui agents as proficient smartphone users. *arXiv preprint arXiv:2503.02268*, 2025.

Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, et al. Mobileworld: Benchmarking autonomous mobile agents in agent-user interactive and mcp-augmented environments. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6142–6167, 2026.

Jungjae Lee, Dongjae Lee, Chihun Choi, Youngmin Im, Jaeyoung Wi, Kihong Heo, Sangeun Oh, Sunjae Lee, and Insik Shin. Verisafe agent: Safeguarding mobile gui agent via logic-based action verification. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*, pp. 817–831, 2025.

Ning Li, Xiangmou Qu, Jiamu Zhou, Muning Wen, Kounianhua Du, Xingyu Lou, Qiuying Peng, Jun Wang, and Weinan Zhang. Mobileuse: A hierarchical reflection-driven gui agent for autonomous mobile operation. *Advances in Neural Information Processing Systems*, 38:40361–40388, 2026a.

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. *Advances in Neural Information Processing Systems*, 37:92130–92154, 2024.

Yunxin Li, Jinchao Li, Shibo Su, Zhenran Xu, Chenrui Zhao, Tongshu Bian, Xiaoman Liang, Meishan Zhang, Baotian Hu, and Min Zhang. Knowact-guiclaw: Know deeply, act perfectly, personal gui assistant with self-evolving memory and skill. *arXiv preprint arXiv:2607.12625*, 2026b.

Zichuan Lin, Feiyu Liu, Yijun Yang, Jiafei Lyu, Yiming Gao, Yicheng Liu, Zhicong Lu, Yangbin Yu, Mingyu Yang, Junyou Li, et al. Ui-voyager: A self-evolving gui agent learning via failed experience. *arXiv preprint arXiv:2603.24533*, 2026.

Chen Liu, Ling Chen, Hanzhang Zhou, Xu Zhang, Quyu Kong, Panrong Tong, Wenhao Wang, Xin Yu, Steven Hoi, and Yue Wang. What memory do gui agents really need? from passive records to active task-driving states. *arXiv preprint arXiv:2606.31612*, 2026a.

Guangyi Liu, Gao Wu, Congxiao Liu, Pengxiang Zhao, Liang Liu, Mading Li, Qi Zhang, Mengyan Wang, Liang Guo, and Yong Liu. Memgui-agent: An end-to-end long-horizon mobile gui agent with proactive context management. *arXiv preprint arXiv:2606.19926*, 2026b.

Guangyi Liu, Pengxiang Zhao, Yaozhen Liang, Qinyi Luo, Shunye Tang, Yuxiang Chai, Weifeng Lin, Han Xiao, WenHao Wang, Siheng Chen, Zhengxi Lu, Gao Wu, Hao Wang, Liang Liu, and Yong Liu. Memgui-bench: Benchmarking memory of mobile gui agents in dynamic environments, 2026c. URL https://arxiv.org/abs/2602.06075.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

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*, volume 2026, pp. 62707–62766, 2026d.

Hongze Mi, Yibo Feng, WenJie Lu, Song Cao, Jinyuan Li, Yanming Li, Xuelin Zhang, Haotian Luo, Songyang Peng, He Cui, et al. Darwinian memory: A training-free self-regulating memory system for gui agent evolution. *arXiv preprint arXiv:2601.22528*, 2026.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026.

Chris 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*, volume 2025, pp. 406–441, 2025.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Androidinthewild: A large-scale dataset for android device control. *Advances in Neural Information Processing Systems*, 36:59708–59728, 2023.

Yibo Shi, Jungang Li, Linghao Zhang, Zihao Dongfang, Biao Wu, Sicheng Tao, Yibo Yan, Chenxi Qin, Weiting Liu, Zhixin Lin, et al. Androtmem: From interaction trajectories to anchored memory in long-horizon gui agents. *arXiv preprint arXiv:2603.18429*, 2026.

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

Fei Tang, Zhiqiong Lu, Boxuan Zhang, Weiming Lu, Jun Xiao, Yueting Zhuang, and Yongliang Shen. Clawgui: A unified framework for training, evaluating, and deploying gui agents. *arXiv preprint arXiv:2604.11784*, 2026.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Ziwei Chai, Y Charles, HS Che, Cheng Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Qwen Team. Qwen3. 5-omni technical report. *arXiv preprint arXiv:2604.15804*, 2026.

Shizuo Tian, Hao Wen, Yuxuan Chen, Jiacheng Liu, Shanhui Zhao, Guohong Liu, Ju Ren, Yunxin Liu, and Yuanchun Li. Agentprog: Empowering long-horizon gui agents with program-guided context management. In *Proceedings of the 24th Annual International Conference on Mobile Systems, Applications and Services*, pp. 883–910, 2026.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*, 2024a.

Zhenhailong Wang, Haiyang Xu, Junyang Wang, Xi Zhang, Ming Yan, Ji Zhang, Fei Huang, and Heng Ji. Mobile-agent-e: Self-evolving mobile assistant for complex tasks. *arXiv preprint arXiv:2501.11733*, 2025.

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

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th annual international conference on Mobile computing and networking*, pp. 543–557, 2024.

Han Xiao, Guozhi Wang, Hao Wang, Shilong Liu, Yuxiang Chai, Yue Pan, Yufeng Zhou, Xiaoxin Chen, Yafei Wen, and Hongsheng Li. Ui-mem: Self-evolving experience memory for online reinforcement learning in mobile gui agents. *arXiv preprint arXiv:2602.05832*, 2026.

Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. Androidlab: Training and systematic benchmarking of android autonomous agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2144–2166, 2025.

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.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI conference on human factors in computing systems*, pp. 1–20, 2025.

Jiwen Zhang, Jihao Wu, Teng Yihua, Minghui Liao, Nuo Xu, Xiao Xiao, Zhongyu Wei, and Duyu Tang. Android in the zoo: Chain-of-action-thought for gui agents. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 12016–12031, 2024.

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

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. Gpt-4v (ision) is a generalist web agent, if grounded. *arXiv preprint arXiv:2401.01614*, 2024a.

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

Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, et al. Mai-ui technical report: Real-world centric foundation gui agents. *arXiv preprint arXiv:2512.22047*, 2025.

Yuxiang Zhou, Jichang Li, Yanhao Zhang, Haonan Lu, and Guanbin Li. Mobile-agent-rag: driving smart multi-agent coordination with contextual knowledge empowerment for long-horizon mobile automation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29939–29947, 2026.

APPENDIX OUTLINE

We provide implementation details of the framework in Appendix A; experimental details in Appendix B, including the evaluation protocol (B.1), the full result tables (B.2), the seed-robustness check (B.4), step budget, cost, and mechanism usage (B.5), root-cause attribution (B.6), the context audit (B.7), experience-store dynamics and admission (B.8), transfer of the store (B.9); the construction and full results of MobileWorld-XL in Appendix C; the failure analysis in Appendix D; two annotated case studies in Appendix E; and the system prompts of the model roles in Appendix F.

A IMPLEMENTATION DETAILS

A.1 PER-STEP TIMELINE

One step of one task runs through six stages; the deterministic runtime enforces the sequencing, and the two model roles act only where named.

1. *Mount.* The runtime assembles the Actor's context: the resident zone (task instruction, plan state, and the current evidence table), the recency zone (the last 8 transitions), and the mount zone of Eq. 4.
2. *Propose.* The Actor returns one action a_t with its target object o_t, declared effect e_t, and grounds B_t; every ground must name a mounted ledger entry.
3. *Gate.* The runtime evaluates $\mathrm{gate}(a_t)$: the pair (s_t, a_t) must not match a recorded dead end, and an irreversible action must cite fully sourced grounds. A rejected proposal returns to the Actor with the violated rule; up to $N{=}3$ re-selections consume calls and tokens but no step, and exhaustion wakes the Manager.
4. *Execute.* The action is submitted to the device and the next screen s_{t+1} is observed.
5. *Provisional settlement.* The runtime checks Eq. 1 against s_{t+1}. On failure the transition is recorded as a dead end on s_t and the Actor retries, which does consume a step. Either way the step record r_t and any evidence read from s_{t+1} are written to the ledger before the next mount.
6. *Boundary settlement.* When the plan marks the subgoal complete, the Manager rules $\mathrm{ok}(\cdot)$ on its product. A failed settlement invokes the backtracker and Eq. 2, and the verdict is written to the ledger before execution resumes.

Per MobileWorld task this loop yields on average 25.3 executed actions, 1.0 gate re-selection, 1.7 retries, and 5.8 Manager calls (Table A6).

A.2 CAUSAL LEDGER: TEMPLATES, MATCHING, AND THRESHOLDS

Templates. A *subgoal* entry holds its statement, the declared upstream subgoals, its status, and, once settled, its product. A *step* entry is the record $r_t = (s_t, a_t, o_t, e_t, B_t)$ plus the settlement outcome and pointers to the before-and-after screenshots. An *evidence* entry holds a key, a value, the reading step, the steps that have used it, and a status flag (current or superseded). A *dead end* is a screen signature with the rejected action; a *verdict* names the defect, the root-cause step, the stated fix, and the rollback target. Every field is either copied from the screen or emitted by the declaring role; the runtime writes and links entries but never paraphrases them.

Matching. A declared effect is a typed template (a screen change, a value present, or a state toggle) checked against the UI tree of the next screen: values by normalized string equality (case, width, and number formatting normalized; timestamps truncated to the minute), objects by the element identifier and visible text of the element acted on. Two evidence entries merge when key and source screen coincide; a fresh reading of the same key supersedes the old value instead of coexisting with it.

Thresholds. $N{=}3$ re-selections per step; a recency zone of 8 transitions; $k{=}3$ mounted experiences with $\lambda{=}0.5$ in Eq. 5; lifecycle counters $k_{\uparrow}{=}k_{\downarrow}{=}2$ in Eq. 7 over the three states candidate, verified, and stable. Sensitivity to each is reported in Appendix A.4.

Lifecycle events. Four events govern an experience entry, each with its own evidence. *Registration* needs a localized root cause, the observed effect, and a stated contrast (Eq. 6); an entry missing any of them is not written. *Initial state*: verified when the contrast is execution-verified, that is, the task succeeded after the repair, and for every route, whose path passed its terminal check; candidate otherwise. *Use*: verified and stable entries are mounted and must be cited in the grounds, and a skill executes step by step under step-level verification; a candidate is never mounted or executed. *Promotion and retirement*: $k_{\uparrow}$ distinct successful tasks since the entry's last state change promote it by one state, and $k_{\downarrow}$ verdicts charged to it since its last state change retire it. Both counters count task identities and restart at every state change, so a task re-run in a later round never counts twice and every promotion rests on fresh evidence.

A.3 Extraction Tools and Frame Budget

Evidence enters the ledger through deterministic tools rather than a second model pass: a UI-tree parser reads typed values from the accessibility tree, an OCR fallback covers canvas-rendered screens, and a tree diff between s_t and s_{t+1} isolates the changed region that Eq. 1 matches against the declared effect. The post-task extraction agent walks the ledger as an index and re-opens screenshots only where a verdict or a proposed skill cites them. Frame budgets keep every prompt bounded: the Actor sees the current and the previous screenshot (two frames); the Manager receives the object layer as text and expands one group at a time into its step layer, capped at 8 frames per wake; post-task extraction is capped at 12 frames per task, taken at settlement boundaries. These caps produce the prompt sizes of Table A6: 8.0k mean prompt tokens for the Actor and 10.4k for the Manager.

A.4 Sensitivity

The defaults sit on plateaus. Sweeping one hyper-parameter at a time on the 397B MobileWorld block, $N{=}3$ re-selections, $k{=}3$ mounted experience entries, $\lambda{=}0.5$ in Eq. 5, and $k_{\uparrow}{=}k_{\downarrow}{=}2$ in Eq. 7 are each within one to three tasks of their neighboring settings; the window size of the recency zone and k interact weakly, and no grid cell moves the R1 success rate by more than three tasks. Three-seed variance is reported in Appendix B.4.

B Experimental Details

B.1 Evaluation Protocol

All methods share the same executor, grounding, seeds, and environment resets, and each benchmark's own step budget: 50 steps on MobileWorld, $2.5g{+}1$ steps for a task with g golden steps on MemGUI-Bench, and the per-task budget shipped with AndroidWorld. The Manager's planning, boundary-ruling, and diagnosis calls in CausGUI are fully counted in the cost table. A step is one environment action submitted, following the official MobileWorld semantics; a re-selection after a rejected proposal does not consume a step, but its calls and tokens are counted; a retry after a failed next-screen check is an executed action and consumes a step. We report success rate under each benchmark's own official score (VALID_SCORE on MobileWorld) over scored tasks, excluding infrastructure failures. The tables report the seed-0 run at task-count resolution; the bare backbone, the two strongest re-run frameworks, and CausGUI are repeated with three seeds and reported as mean $\pm$ standard deviation (Appendix B.4); ablations and MobileWorld-XL report the seed-0 run. Self-evolution runs in immutable generations: the store is frozen for the whole of a round and updated once at its end, so every round's result is attributable to one version of the store.

B.2 Full Results

Table A1 gives the task counts behind Figure 3. Figure A1 shows six of the twelve re-run frameworks; Table 1 lists all of them. Table A2 lists the per-round numbers behind Figure 4a,c; and the MobileWorld-XL rounds behind Figure 4b are in Table A10.

Table A1: Task counts behind Figure 3: successful tasks per MobileWorld bucket (bucket sizes in the second header row), for CausGUI, the re-run frameworks MemGUI-Agent, KnowAct-GUIClaw, and OpenGUI, and the bare backbone. A task's golden steps are the minimum number of steps needed to complete it; the app split follows each task's declared app set. Row totals equal the MobileWorld column of Table 1.

		Task length (golden steps)			App scope		
Backbone	**Agent**	**≤15**	**16–25**	**>25**	**Single**	**Cross**	**Total**
	bucket size	*51*	*24*	*42*	*55*	*62*	*117*
Qwen3.5-35B-A3B	CausGUI	33	7	6	26	20	46
	MemGUI-Agent	32	6	2	26	14	40
	KnowAct-GUIClaw	29	2	1	22	10	32
	OpenGUI	26	3	0	21	8	29
	Bare backbone	25	1	0	20	6	26
Qwen3.5-397B-A17B	CausGUI	35	14	12	29	32	61
	MemGUI-Agent	36	12	6	27	27	54
	KnowAct-GUIClaw	31	9	5	26	19	45
	OpenGUI	30	7	2	26	13	39
	Bare backbone	27	8	7	25	17	42

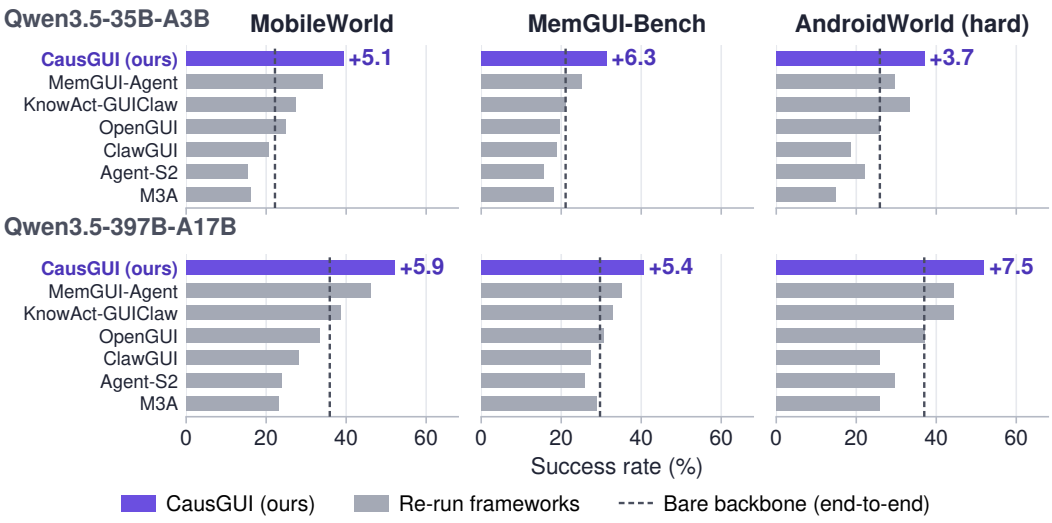


Figure A1: Long-horizon execution with no prior experience. Columns are benchmarks, rows are backbones. Bars are single-run success rates at task-count resolution. Every panel shows CausGUI and, in one fixed order, the six re-run frameworks that rank among the five strongest on at least one benchmark of the 397B block. The number after the CausGUI bar is its margin over the strongest re-run framework of that cell; the dashed line is the bare backbone run end-to-end. Full numbers, including the end-to-end models and average steps, are in Table 1.

B.3 Frontier Backbones

The two Qwen backbones of Table 1 leave two questions open: whether the gain is an artefact of one model family, and whether it survives a backbone already strong enough to solve much of MobileWorld unaided. Table A3 answers both on Kimi-K2.5 and Kimi-K2.6 (Team et al., 2026), two frontier backbones outside the Qwen family, each running the store distilled over five rounds of self-evolution as in Table A2. Both gain: 18 tasks on K2.5 (45.3 to 60.7) and 14 on K2.6 (52.1 to 64.1), against 27 tasks on the 397B backbone (35.9 to 59.0). The framework gain therefore shrinks monotonically as the backbone strengthens, from 23.1 to 15.4 to 12.0 points, but it does not vanish, and the absolute ceiling keeps rising. Mean steps fall alongside success on both backbones, by 4.2 and 3.9, the same direction and magnitude as on the Qwen backbones, so the extra tasks are not bought with longer trajectories. One coincidence is worth naming: bare Kimi-K2.6 scores 52.1,

Table A2: Self-evolution over five rounds on MobileWorld, the numbers behind Figure 4a,c (experience store persisting across rounds; Qwen3.5-397B-A17B backbone). R0 is the no-experience score quoted from Table 1; The store is distilled from the R0 trajectories, so R1 is the first pass that runs with experience mounted. Baselines are the re-run frameworks of Table 1 whose experience pipelines can ingest the executed task trajectories (exclusions in the caption of Figure 4). SR in %; Avg. steps per task.

	MobileWorld SR						MobileWorld Avg. steps					
Agent	**R0**	**R1**	**R2**	**R3**	**R4**	**R5**	**R0**	**R1**	**R2**	**R3**	**R4**	**R5**
KnowAct-GUIClaw	38.5	41.0	42.7	41.9	43.6	43.6	**21.1**	**21.4**	**21.2**	**21.6**	**21.0**	**20.8**
ClawGUI	28.2	29.9	29.1	30.8	31.6	32.5	21.6	21.9	22.1	21.8	21.7	21.9
Agent-S2	23.9	25.6	26.5	25.6	24.8	23.9	40.0	39.6	39.2	39.8	40.3	40.9
Mobile-Agent-E	6.0	6.8	7.7	6.8	6.0	5.1	45.1	44.8	44.6	45.2	45.7	46.3
AppAgentx	8.5	9.4	10.3	9.4	8.5	7.7	44.9	44.5	44.3	44.8	45.4	46.1
CausGUI	**52.1**	**54.7**	**56.4**	**57.3**	**58.1**	**59.0**	25.3	24.2	23.5	23.1	23.3	22.7

Table A3: Frontier non-Qwen backbones on MobileWorld, under the protocol of Table 1: the bare backbone against CausGUI running the store distilled over five rounds of self-evolution (R5, as in Table A2). No baseline framework is re-run on these backbones. SR in %; steps is the mean steps per task.

System	**SR**	**Steps**
BACKBONE: *Kimi-K2.5*		
Bare backbone	45.3	26.3
CausGUI (R5)	**60.7**	**22.1**
BACKBONE: *Kimi-K2.6*		
Bare backbone	52.1	25.4
CausGUI (R5)	**64.1**	**21.5**

exactly what CausGUI reaches on the 397B backbone before any experience is mounted (Table A2, R0), and the framework still adds 12.0 points on top of that starting point.

B.4 SEED ROBUSTNESS

Table A4 repeats the bare backbone, the two strongest re-run frameworks, and CausGUI with three seeds on both Qwen backbones; repeating all thirteen frameworks three times is not affordable, so every other row of Table 1 is a single seed-0 run. Seed 0 is the run reported in Table 1, and means and standard deviations are computed on task counts before conversion to percentages. The standard deviations stay between 0.9 and 2.2 points on the 117- and 128-task benchmarks and between 2.1 and 3.7 points on the 27-task AndroidWorld slice, where a single task moves the score by 3.7 points. CausGUI stays ahead of every baseline under every individual seed, and its margin over the strongest re-run framework exceeds the run-to-run fluctuation in every cell.

B.5 STEP BUDGET, COST, AND MECHANISM USAGE

Step budget. The step budget separates the systems only once there is room to recover (Table A5). At 15 steps all four systems solve 10 to 16 tasks, bounded by the 51 tasks whose golden path fits in 15 steps. At 30 steps CausGUI solves 52 tasks against 38 to 47. Raising the budget from 50 to 100 steps adds 5 tasks for CausGUI and 2 to 3 for the baselines. The baselines have 21 to 34 failed runs that exhaust the 50-step budget, so the extra steps are available to them, but they spend them repeating the same action; our 8 budget-exhausting failures bound our own gain. The mean-length columns of Table A5 are not independent of the Steps column of Table 1: writing $\mathcal{T}$ for the task set and $\mathcal{T}^+, \mathcal{T}^-$ for its solved and failed parts,

$$\bar{L} \;=\; \frac{|\mathcal{T}^+|\,\bar{L}^+ \;+\; |\mathcal{T}^-|\,\bar{L}^-}{|\mathcal{T}|}, \tag{8}$$

so a system's overall mean is fixed once its success count and its two conditional means are.

Table A4: Three-seed robustness check: mean ± std over three seeds for the bare backbone, the two strongest re-run frameworks, and CausGUI, under the protocol of Table 1. Seed 0 is the run reported in Table 1; means and standard deviations are computed on task counts ($n-1$) and converted to percentages. All other rows of Table 1 are single seed-0 runs. SR in %.

System	MobileWorld	MemGUI-Bench	AndroidWorld-Hard
BACKBONE: *Qwen3.5-35B-A3B*			
Bare backbone	21.9 ± 2.2	20.6 ± 1.6	27.2 ± 2.1
KnowAct-GUIClaw	27.9 ± 1.8	21.6 ± 1.6	32.1 ± 2.1
MemGUI-Agent	33.9 ± 1.3	24.7 ± 1.2	29.6 ± 3.7
CausGUI	**39.0 ± 1.3**	**31.5 ± 1.2**	**38.3 ± 2.1**
BACKBONE: *Qwen3.5-397B-A17B*			
Bare backbone	35.3 ± 1.8	30.2 ± 1.6	35.8 ± 2.1
KnowAct-GUIClaw	39.0 ± 1.8	32.3 ± 0.9	44.4 ± 3.7
MemGUI-Agent	45.6 ± 1.0	35.7 ± 1.6	45.7 ± 2.1
CausGUI	**51.9 ± 1.3**	**40.9 ± 1.2**	**50.6 ± 2.1**

Table A5: Step budget and trajectory length on MobileWorld (Qwen3.5-397B-A17B). SR under budgets of 15/30/50/100 steps; mean length of successful and failed trajectories at 50 steps; share of failures that used all 50 steps; share of tasks declared complete but rejected by the checker.

System	15	30	50	100	Succ. len	Fail len	Cap hit %	False compl. %
CausGUI	13.7	44.4	52.1	56.4	22.0	28.9	14.3	5.1
MemGUI-Agent	13.7	40.2	46.2	48.7	19.6	29.4	33.3	15.4
KnowAct-GUIClaw	8.5	32.5	38.5	40.2	18.6	22.6	29.2	12.8
Bare backbone	11.1	32.5	35.9	37.6	22.6	29.8	45.3	15.4

Cost and mechanism usage. Per MobileWorld task, CausGUI makes 26.3 Actor calls and 5.8 Manager calls (token totals per system in Table A6). The Actor calls are the 25.3 executed actions plus 1.0 gate re-selection; the 1.7 retries after a failed next-screen check are executed actions and are already inside the step count. The Actor prompt averages 8.0k tokens, shorter than the backbone's 10.1k because the three zones replace the raw history, so the Actor side, prompt and output together, costs 219.6k tokens, 62.4k less than the backbone. The Manager prompt averages 10.4k tokens, carrying the object layer and the expanded group with its screenshots, and the 5.8 calls add 64.9k tokens (60.4k prompt, 4.5k output). The total is therefore $219.6{+}64.9{=}284.5$k against 282.0k for the backbone (+0.9%): the Manager slightly more than uses up the Actor's saving. Per task the online cost and its overhead over the bare backbone are

$$C \;=\; n_A\big(\bar{p}_A + \bar{o}_A\big) \;+\; n_M\big(\bar{p}_M + \bar{o}_M\big), \qquad \delta \;=\; \frac{C - C_0}{C_0}, \tag{9}$$

with n_A, n_M the Actor and Manager calls, $\bar{p}, \bar{o}$ their mean prompt and output tokens, and C_0 the backbone's total; between-task extraction sits outside C. Of the 5.8 Manager calls, 1.6 are the planning calls that row B of Table 2 already pays without any auditor, and the remaining 4.2 are what the auditor adds, at 17% of steps: 70% of them are boundary rulings on the product of a completed subgoal, 25% answer an Actor request, and 5% follow the exhaustion of N attempts on one step, whether re-selections or retries; 74% of them occur at subgoal boundaries, and the median backtrack spans 4 steps, measured over all calls that backtrack rather than over the failed trajectories of Appendix B.6, where the annotated root cause sits 9 steps back. After five rounds the cost falls to 270.5k, since skills execute routes without step-by-step decisions and the Manager wakes only 4.9 times, at 51.0k prompt tokens.

B.6 Root-Cause Attribution

Does attribution land on the right step? We sample 60 real failed MobileWorld trajectories, 20 each from the seed-0 runs of CausGUI, MemGUI-Agent, and the bare backbone, keeping only failures on which two annotators independently mark the same root-cause step (157 of the 194-trajectory pool); the closed pool, the stratified random draw, and the mechanical truncation rule leave no room to pick trajectories that favor any judge. Each judge receives the same recording, stripped of agent-specific annotations, re-organizes it in its own format under the same Manager model and a comparable token budget; for our auditor the same Manager model first re-declares

Table A6: Cost per MobileWorld task (Qwen3.5-397B-A17B, no prior experience): prompt and output tokens over all model calls made while the task runs. Between-task experience extraction is outside this total and adds 9.7k tokens per task for CausGUI.

System	SR	Tokens (k)
Bare backbone	35.9	282.0
MemGUI-Agent	46.2	268.7
CausGUI	52.1	284.5
M3A	23.1	295.6
OpenGUI	33.3	312.0
KnowAct-GUIClaw	38.5	334.2
Agent-S2	23.9	371.9
Mobile-Agent-V3	22.2	402.5

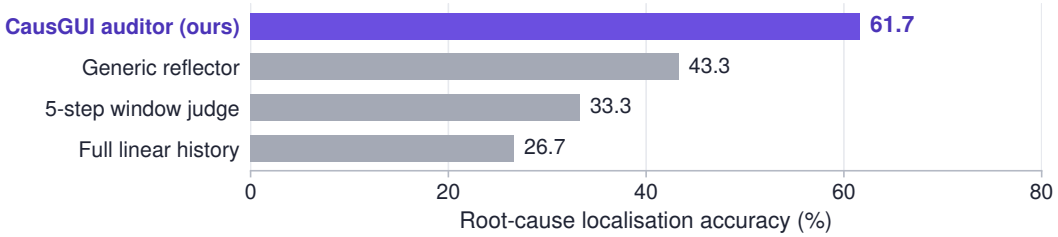


Figure A2: Root-cause localization on 60 real failed MobileWorld trajectories, sampled 20/20/20 from the failures of CausGUI, MemGUI-Agent, and the bare backbone (Qwen3.5-397B-A17B). Only failures whose single root-cause step two annotators mark independently and identically enter the pool; every judge re-organizes the same recording (screenshots, executed actions, task instruction) under the same Manager model, and a localization counts only when the step and the reason are both correct, graded blind by experts.

the grounds and expected effect of every step from that recording alone, so the ledger it backtracks over is reconstructed after the fact and no judge sees annotations from the original run, and names the root-cause step with a reason; experts grade blind, and a verdict counts only if the step and the reason are both right (grader agreement 55/60). The attribution auditor, backtracking over the reconstructed ledger, localizes 61.7% (37/60), against 43.3% for a generic post-hoc reflector distilled from the reflection pattern of the strongest re-run frameworks, 33.3% for a five-step window judge, capped by the share of failures whose root cause lies within its window (the annotated median root-cause distance is 9 steps), and 26.7% for the full linear history: the complete context is the worst organization of the same evidence (Figure A2), and the ordering is unchanged within each source block. The window judge's ceiling is structural rather than a matter of reasoning quality: writing d° for the number of steps between the root-cause step r° and the step at which the failure surfaces, a judge that only ever inspects the last w steps cannot localize a cause that lies further back, so

$$\mathrm{acc}(w) \;\le\; \Pr\big[\, d^\circ \le w \,\big], \tag{10}$$

and at $w{=}5$ against an annotated median d° of 9 steps that bound is already the binding constraint.

B.7 Context Audit

Does the context hold the right evidence? A precise, denoised context beats a complete one, and completeness without discipline is worse than either: dumping the whole ledger into the prompt copies a superseded value in 11.4% of evidence-using actions and scores 23.9, below the bare backbone, while scope-gated mounting copies one in 2.3% and scores 46.2 (Table A7). What hurts is not missing evidence but stale evidence trusted equally, and keeping every current value in view is no answer either: without dependency binding the Actor must still choose among several current readings of the same key (34.2). The gains feed back: verdicts keep the evidence table current, the cleaned context yields grounds worth citing, and attribution built on those citations localizes better; the ladder of Table 2 separates the two shares, the resolver alone adding 10.3 points and the auditor a further 5.9 on top of the cleaned context. The stale-use column counts consumption rather than



Figure A3: Context audit on MobileWorld (Qwen3.5-397B-A17B, no prior experience; attribution off throughout). (a) Mean Actor prompt size at each step, averaged over the trajectories that reached that step and reported by the serving API with screenshots included; steps 1–2 are the frame buffer filling (one frame, then two), not context growth. The undifferentiated dump grows by 0.33k tokens per step and passes 18k, while the two zoned designs stay flat and separate only by the dependency-bound evidence. (b) Share of evidence-using actions that copied a superseded value. Full numbers in Table A7.

Table A7: Context audit on MobileWorld (Qwen3.5-397B-A17B, no prior experience). Three memory designs inside our own framework, on the same backbone, prompts and 50-step budget, with no token cap and the attribution auditor off in every row; the scope-gated row is row B of Table 2. With the auditor off no dead end or verdict is ever written, and with no prior experience the experience store is empty, so of the three mount sources of Eq. 4 only the dependency-bound evidence $\mathrm{dep}(g_t)$ remain, and they are the only input separating working memory from the full design. Undifferentiated memory records the ledger as usual but assembles the history and every recorded entry, superseded values included, into one flat stream. Working memory keeps the resident zone, evidence table included, and the recency zone of the last 8 transitions, and switches the mount zone off; the full design adds the dependency-bound evidence to exactly that. Undifferentiated memory also sees the last 8 transitions, inside its full history. Neither is the bare backbone, which carries a pruned trajectory and records no ledger at all (row A of Table 2). Stale use: share of evidence-using actions that copied a superseded value, with supersession judged by prompting a chat LLM on the paired screens. SR in %.

Context	SR	Steps	Stale use
Undifferentiated memory (flat dump)	23.9	31.8	11.4
Working-memory only	34.2	27.4	5.1
Dependency-scoped memory (CausGUI)	**46.2**	**24.0**	**2.3**

presence: over the steps whose action consumes registered evidence,

$$\rho_{\text{stale}} = \frac{\left|\{\, t \,:\, a_t \text{ consumes } f,\ \text{status}(f) = \textsc{superseded} \,\}\right|}{\left|\{\, t \,:\, a_t \text{ consumes some } f \in \mathcal{F} \,\}\right|}, \tag{11}$$

so a design is charged only when a superseded value actually reaches an action, not when one merely remains visible.

Table A7 compares the three memory designs behind Figure A3, each inside our own framework with the same backbone, prompts and budget, and with the attribution auditor off in every row, so the audit isolates the dependency resolver. Recording the ledger and then dumping it is the worst of the three: the history and every recorded entry, superseded values and abandoned attempts included, arrive as one flat stream; the prompt grows by 0.33k tokens per step and passes 18k by step 40 (Figure A3a); 11.4% of evidence-using actions copy a superseded value, because the old and the new sit side by side and are trusted equally; and success falls to 23.9, below the bare backbone of row A of Table 2, which carries a pruned trajectory and records nothing. Undisciplined memory is thus worse than no memory. Working memory holds visibility fixed: its resident evidence table carries every current value, and a fresh reading supersedes the old one, so stale use halves against the dump (5.1%). What it lacks is the binding of a subgoal to the evidence it consumes. A long cross-app task accumulates several current readings of the same key taken on different screens, such as a rating on a list page and on a detail page, and without the binding the Actor has to choose among

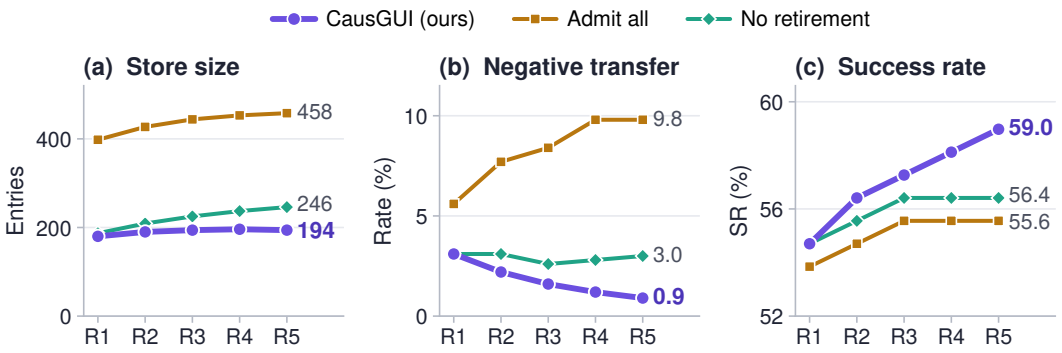


Figure A4: Experience-store dynamics over five rounds on MobileWorld (Qwen3.5-397B-A17B, immutable generations) for CausGUI and two controls: admit every extracted entry while keeping retirement, and causal admission with retirement switched off. (a) Store size after each round. (b) Attributed negative transfer during the round, the share of experience mounts in failed tasks that the verdict names as a ground of the root cause. (c) Success rate. Per-round counts for CausGUI are in Table A8.

them: it re-opens screens to disambiguate, so trajectories stay long (27.4 steps), and when it takes a value from a transition in the recency zone rather than from the table, it copies an older reading, which is where its stale uses come from. Success reaches only 34.2. Dependency-scoped mounting supplies exactly this binding: $\mathrm{dep}(g_t)$ projects each upstream product the active subgoal declared into the step that consumes it, and the Actor's grounds must cite a mounted entry, so the value it copies is the one registered for that subgoal. Stale use falls to 2.3%, the residue being registration and supersession misses rather than policy, and the prompt stays flat (0.03k per step, 8.9k at step 40 against 18.6k for the dump). The dependency-bound evidence is the only input that separates the two zoned rows (Table A7), so the 12.0 points between them are the value of dependency binding rather than of visibility, bought for about 1k prompt tokens per call. Against the dump the design is worth 22.3 points; the resolver's own share, measured against the bare backbone, is the 10.3 points from row A to row B of Table 2.

B.8 Experience-Store Dynamics and Admission

Store dynamics. The store is filtered at the door and converges over rounds (Table A8, Figure A4). Of 342 entries extracted after the cold run R0, 156 pass causal admission; 92 of the rejected ones lack a contrast, 44 an effect, and 32 a cause. The gate is what keeps the store small: the cold run contains 140 error events and the auditor localizes a root cause in 61.7% of the failures it is given (Appendix B.6), so a fault becomes a memory only when a localized cause meets a stated contrast: 34 of the 68 entries proposed from failed tasks survive, and all 34 wait in shadow until a later task corroborates them. Admissions then fall from 31 to 7 per round while recurrence retires 7, 10, 10, 8, and 9 entries, so the store grows from 156 to 196 and settles at 194, with a net change that falls from $+24$ to -2. Each retirement needs two charged verdicts; a round produces only about 90 verdicts (its failed tasks plus its trajectory-level repairs), of which 30 to 40 are charged to an entry, which bounds how fast the store can shrink. Admission and retirement are the only two ways the store changes size, so its trajectory across rounds is the recursion

$$|\mathcal{E}_r| \;=\; |\mathcal{E}_{r-1}| \;+\; \mathrm{adm}_r \;-\; \mathrm{ret}_r, \tag{12}$$

which is what the Admitted, Retired and Store columns of Table A8 report row by row. Negative transfer is measured on mount events rather than on entries: letting $\mathcal{U}_r$ collect the pairs (m, τ) in which entry m was mounted during a failed task τ of round r, and $\mathcal{U}_r^{\mathrm{ch}} \subseteq \mathcal{U}_r$ those pairs whose verdict names m as a ground of the root-cause step,

$$\mathrm{NT}_r \;=\; \frac{|\mathcal{U}_r^{\mathrm{ch}}|}{|\mathcal{U}_r|}, \tag{13}$$

which is the named/mounts ratio printed in that table. Over the same rounds the reuse rate rises from 41% to 68% and the attributed negative-transfer rate falls from 3.1% to 0.9% (12 of 387 mounts in failed tasks to 4 of 428). The route tier plateaus for a structural reason rather than a failure to learn: a

Table A8: Experience-store dynamics of CausGUI over five rounds on MobileWorld (Qwen3.5-397B-A17B, immutable generations: round r uses the store frozen after round $r-1$). Every memory comes from a verdict: in a task that succeeded after the repair its contrast is execution-verified and it enters as verified; in a task that failed or aborted its contrast is pending and it enters as a candidate held in shadow, stored but never mounted until later tasks corroborate it. A skill is one canonicalized route, keyed on an app and a subgoal class rather than on a task, whose path passed its terminal check; a recurrence advances an existing route instead of adding a new one. Reuse = tasks in which at least one mounted entry was cited in the grounds. NT = attributed negative transfer, the share of experience mounts in failed tasks that the verdict names as a ground of the root cause (named/mounts); the auditor localizes 61.7% of root causes (Appendix B.6), so harm it does not localize is never charged here; this is an indicator of attributed negative transfer, not a measured rate of all harm. An entry retires when the verdicts charged to it reach $k_{\downarrow}=2$, and a round supplies only about 90 such verdicts, which bounds how fast the store can shrink. R0 admits 156 of 342 extracted entries (68 memories proposed from failed tasks, 96 from repaired steps of successful tasks, and 178 routes), 34 of them as shadow candidates; of the rejected, 92 lack a contrast, 44 an effect and 32 a cause, and 18 merge into an entry proposed earlier in the same batch. An entry costs 0.118k tokens as a memory and 0.137k as a route once mounted, so the R5 store is 25.1k tokens in all. Lifecycle after R5: 29 candidates in shadow, 62 verified, 103 stable.

Round	Admitted	Retired	Store after (mem./skills)	Reuse %	NT %	SR
R0	156	0	156 (62/94)	–	–	52.1
R1	31	7	180 (72/108)	41	3.1 (12/387)	54.7
R2	20	10	190 (76/114)	52	2.2 (9/409)	56.4
R3	14	10	194 (78/116)	59	1.6 (7/437)	57.3
R4	10	8	196 (79/117)	64	1.2 (5/417)	58.1
R5	7	9	194 (78/116)	68	0.9 (4/428)	59.0

route is keyed on an app and a subgoal class, and the reachable route space of the benchmark is finite, so once 117 routes are verified there are few left to add. Two controls fail in two different ways. Admitting every extracted entry lets the store grow to 458 entries and the attributed negative-transfer rate to 9.8%, and success stalls at 55.6. Switching retirement off keeps the entries recurrence would have removed, holds negative transfer flat near 3.0% instead of letting it fall, and costs three tasks. An episodic control that stores every raw trajectory instead of distilling it reaches 681 episodes by R5, 33 times the tokens of the distilled store, and is back at its cold start with negative transfer rising to 6.8%: replayed episodes re-inject failed routes, relocating into memory the pollution the resolver removes from context.

Three defenses. First, memorization. Promotion counts distinct task identities, so re-solving a task in a later round never advances an entry, and the frozen store lifts the 63 unseen tasks of MobileWorld-XL by 2 tasks with nothing learned during that pass (Table A10), which memorized task-specific answers could not do; both gained tasks lie in the recombination stratum, while the 19 tasks that reuse no verified subtask gain nothing from the frozen store at all (Table A11). Second, wrong memories. The attributed negative-transfer rate falls to 0.9% at R5, against 9.8% when every extracted entry is admitted; because it is only as complete as attribution, we pair it with a measure that needs no verdict. Across rounds CausGUI keeps 87 to 95% of the tasks it solved in the previous round, against 72 to 76% for the bare backbone re-run five times with no store, and 52 tasks are solved in all five rounds, so the evolving store regresses less between rounds than running without one, though it does not remove regressions altogether. Third, scope isolation: no app-tier entry is ever mounted outside its own app, a cross-app hit count of exactly zero; global-tier entries are cross-app by design and are not counted.

B.9 Transfer of the Experience Store

The store evolved on the 117 MobileWorld tasks is mounted, frozen, on MobileWorld-XL, whose 44 recombination tasks chain verified source subtasks with a novel subtask and whose other 19 reuse nothing from the source (Table A10). Splitting the store by entry type separates what travels: skills alone add the same 2 tasks at round 1 as the full store, while memories alone leave the cold score unchanged, because skills execute the familiar segments of a chain directly, whereas task-scoped memories rarely match a recombined task on first sight; both gained tasks place their novel subtask

Table A9: Composition of MobileWorld-XL: 63 tasks over eight app groups, thirty-three of them spanning two or more apps, mirroring the 55/62 single-to-cross split of the 117 source tasks. Every task carries a unique verifiable answer. The benchmark is stratified by novelty: a *recombination* task chains at least one subtask verified on the 117 source trajectories with at least one novel subtask never executed in the source, while an *entirely new* task reuses no verified subtask, so no skill learned on the source can execute any of its segments. The reused operations listed are those of the recombination stratum.

App group	Tasks	Cross	Rec.	New	Reused operations
Files & document reading	10	5	7	3	PDF table read, rename, email w/ attachment
Mattermost	7	4	5	2	reply, create channel, add member
Mail	9	5	6	3	send, forward, reply, conditional
Messages / SMS	8	5	6	2	send value, delete by pattern, code forward
Mastodon	12	4	8	4	follow, unfollow, bookmark, list, profile
Settings & calendar	6	2	4	2	toggle, wallpaper, event, alarm
Mall, maps & misc	6	3	4	2	cart edit, checkout, distance, contact
Cross-app composites	5	5	4	1	multi-app chains
Total	**63**	**33**	**44**	**19**	–

Table A10: Generalization on MobileWorld-XL, 63 unseen tasks derived from the 117 MobileWorld tasks (44 recombining verified source subtasks with a novel one, 19 entirely new; Qwen3.5-397B-A17B backbone), the numbers behind Figure 4b. R0: no experience. R1–R5: the store that each method evolved on the 117 MobileWorld tasks after that round (Table A2) is mounted frozen on MobileWorld-XL. No experience is extracted or admitted on MobileWorld-XL, so every column measures transfer alone. The last three rows run CausGUI with the store restricted by entry type: memories only, skills only, and the full store. SR in %; Avg. steps per task at R5.

	SR						Avg. steps
Agent	**R0**	**R1**	**R2**	**R3**	**R4**	**R5**	**R5**
KnowAct-GUIClaw	30.2	31.7	33.3	31.7	30.2	31.7	**22.4**
ClawGUI	22.2	23.8	23.8	25.4	23.8	25.4	23.3
Agent-S2	19.0	19.0	20.6	17.5	19.0	17.5	42.6
Mobile-Agent-E	4.8	4.8	6.3	4.8	3.2	3.2	47.2
AppAgentx	6.3	6.3	4.8	6.3	4.8	4.8	46.8
CausGUI + memories only	42.9	42.9	44.4	42.9	44.4	46.0	25.4
CausGUI + skills only	42.9	46.0	46.0	44.4	46.0	47.6	25.1
CausGUI + memories & skills	**42.9**	**46.0**	**47.6**	**49.2**	**49.2**	**50.8**	24.6

after a familiar prefix, where mounted experience is already active. Memories pay off as the source store matures: entries admitted in the later MobileWorld rounds anchor the novel segments, and the full store ends 2 tasks ahead of skills alone at R5 (Table A10). Against the five evolving baselines the round-1 store gains 2 tasks for CausGUI and at most 1 for any baseline. A 35B backbone consuming the 397B store recovers about 70% of the gain it obtains from its own store.

C GENERALIZATION: MOBILEWORLD-XL

This experiment tests whether the experience accumulated on the 117 tasks transfers to 63 unseen tasks. Two strata separate the two questions a transfer benchmark has to answer (Table A9): 44 *recombination* tasks, each a chain that reuses at least one subtask verified on the source trajectories and introduces at least one novel subtask, measure whether verified experience can be recomposed; the other 19 are *entirely new*, reusing no verified subtask, and measure whether anything survives when nothing can be recomposed (Table A10). The protocol is transfer only, on the same 397B backbone: R0 runs with no experience, and for each round r the store that self-evolution produced on the 117 MobileWorld tasks after round r is mounted frozen on MobileWorld-XL (R1–R5). Nothing is extracted or admitted on MobileWorld-XL itself, so the whole curve isolates what the source experience carries. We report SR and the average steps per task.

Table A11: MobileWorld-XL by stratum: solved tasks (SR in %) per round for CausGUI and the strongest evolving baseline, on the same runs as Table A10. The recombination stratum holds the 44 tasks that reuse at least one subtask verified on the 117 source trajectories; the entirely-new stratum holds the 19 that reuse none. The two rows of a method sum to its row in Table A10 at every round. Every round is transfer alone: the round-1 gain lands entirely in the recombination stratum, where a skill keyed on an app and a subgoal class has a familiar segment to execute; the entirely-new stratum first gains at R3, once the source store has accumulated causes general enough to matter where no segment is familiar.

	Solved tasks (SR %)						
Agent, stratum	**R0**	**R1**	**R2**	**R3**	**R4**	**R5**	**Δ**
CausGUI, recomb. (44)	20 (45.5)	22 (50.0)	23 (52.3)	23 (52.3)	23 (52.3)	24 (54.5)	+4
CausGUI, entirely new (19)	7 (36.8)	7 (36.8)	7 (36.8)	8 (42.1)	8 (42.1)	8 (42.1)	+1
KnowAct, recomb. (44)	14 (31.8)	15 (34.1)	16 (36.4)	15 (34.1)	14 (31.8)	15 (34.1)	+1
KnowAct, entirely new (19)	5 (26.3)	5 (26.3)	5 (26.3)	5 (26.3)	5 (26.3)	5 (26.3)	+0

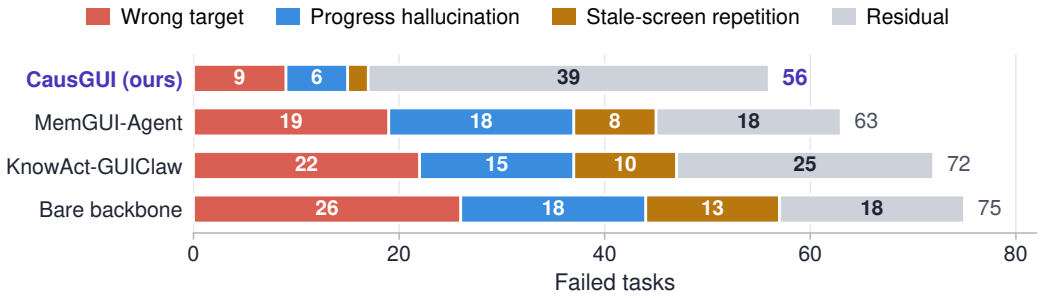


Figure A5: Failed MobileWorld tasks by the mechanism that ends them first, for CausGUI, the two strongest re-run frameworks, and the bare backbone (Qwen3.5-397B-A17B, cold start). The categories are those of Section 1: an action taking effect on the wrong target, progress hallucination (a declared completion the checker rejects), repetition on a stale screen, and a residual. A task is counted once, under the mechanism that ends it, so bar length is the total number of failures and the segments are their composition. The three targeted mechanisms account for 57 of the backbone's 75 failures and 17 of our 56.

The round-1 store lifts CausGUI from 42.9 to 46.0 SR and the round-5 store reaches 50.8 with R5 average steps of 24.6; the baselines improve by smaller margins at every round, and the strongest of them peaks at 33.3 by R2 and falls back to 31.7. At round 1 the gain is 2 tasks, all of it carried by skills: skills alone reach 46.0 while memories alone stay at the cold 42.9 (Table A10). The later rounds show what a maturing source store adds: memories admitted in those rounds anchor the novel segments and pull the full store to 50.8 against 47.6 for skills alone. Table A11 splits these totals by stratum. Both strata end above their cold run, by 4 tasks of 44 and 1 task of 19, and the two gains separate in time as well as in size: the round-1 store moves only the recombination stratum, since a skill keyed on an app and a subgoal class has nothing to execute in a task whose every segment is unfamiliar, while the entirely-new stratum stays at its cold 7 tasks through R2 and first gains at R3, once the source store has accumulated causes general enough to matter where no segment is familiar. The strongest baseline gains 1 task in the recombination stratum and none in the other, so the asymmetry is not a property of the split but of what is being reused. MobileWorld-XL shares apps and backends with the source set, so we phrase the result as compositional generalization, not out-of-distribution robustness; which part of the store travels is analysed in Appendix B.9.

D Failure Analysis

Figure A5 breaks failed tasks down by the mechanism that ends them, over CausGUI, the two strongest re-run frameworks, and the bare backbone, using the categories of Section 1. All three targeted mechanisms shrink. Wrong target, an action whose effect lands on the wrong object, falls from 26 under the bare backbone and 19 to 22 under the frameworks to 9, the difference being caught at settlement rather than at the end of the task. Progress hallucination, a declared completion the

checker rejects, falls from 18 to 6, since a completion claim has to survive both levels of verification. Repetition on a stale screen falls from 13 to 2, and with it the same-screen repeat rate, from 13.8% of the backbone's steps to 2.1% of ours. Together the three fall from 57 of 75 failures under the backbone to 17 of 56 under CausGUI, from three quarters of the failures to under a third. The residual grows as they shrink, for a mechanical reason: a task is counted under the mechanism that ends it first, so a run a baseline loses to a wrong-target action reaches the exploration problem behind it only once that action is repaired, and is then counted as a residual failure. Eight of those residual failures exhaust the 50-step budget, which bounds what a larger budget can recover, and that is what the 100-step run in Appendix B.5 finds.

E CASE STUDIES

Two complete runs, one per benchmark family, show the mechanisms of Section 3 acting on real screens. Every annotation is taken from the run's own ledger: the Actor's reasoning and declared intent, the entries written at that step, what the dependency resolver mounted, and the rulings the runtime issued.

Scope-gated mounting (Figure A6, top). The task collects three attributes for each of six courses, computes a score per course, and opens the leader. Eighteen evidence entries therefore accumulate, and the arithmetic of any one course needs exactly three of them. Two things follow in the figure. Each value is pinned to the screen it was read on, so a later step cites the evidence instead of scrolling the list again; and at step 20 the resolver mounts only the three evidence entries of the course being scored, so those three are the support this step may cite, while the other fifteen stay current in the resident evidence table without being mounted — the $\mathrm{dep}(g_t)$ term of Eq. 4, where an undifferentiated context would instead offer all eighteen as equally available. The derived value enters the ledger with the page that sourced it, which is what lets the end state name the leader and the runner-up.

Unverified reads, refused settlements, and root-cause verdicts (Figure A6, bottom). The task reads two dates from a calendar and sends them as one exactly formatted string. The month grid affords a plausible misread: a bar that looks like October 12–15. That provisional read does enter the ledger, but not as verified evidence, so the resolver will not mount it for the SMS subgoal to consume; the event detail screen then corrects the start date by one day (step 5), which is the difference between a passing and a failing answer. Two guards fire later. At step 9 the Manager rules the date string done and the runtime rejects the settlement, because the deliverable is not visible on screen — a claim is not a settlement (Eq. 1). That failed settlement wakes the Manager, and its verdict places the fault at the step that typed the dates from memory rather than from the screen, naming the step-5 screenshot as the point to resume from, since a single misread character fails the task. The subgoal is re-settled at step 12 and the run closes at step 15.

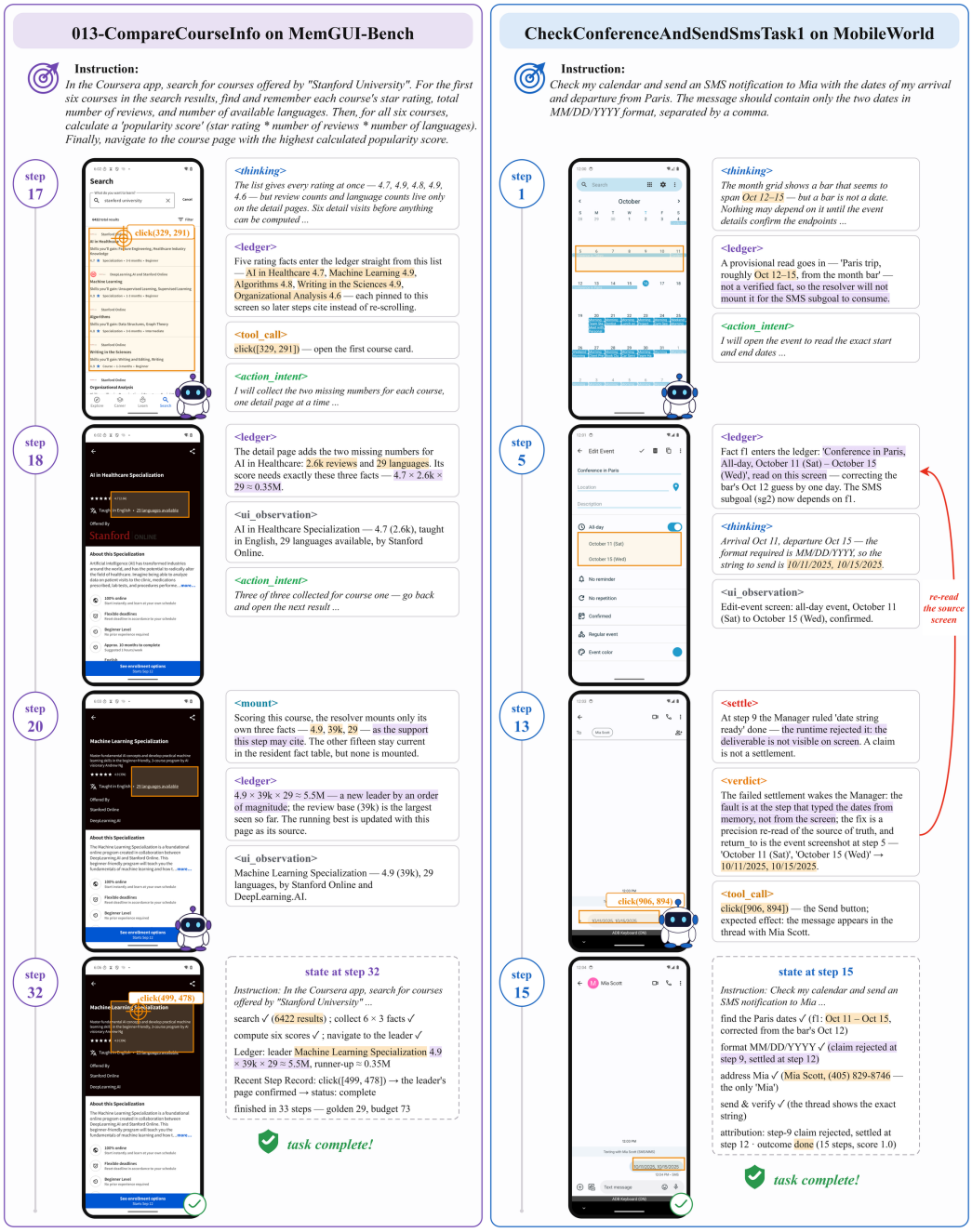


Figure A6: Case studies on both benchmarks (CausGUI); each panel reads top to bottom. *Left*: `013-CompareCourseInfo` on MemGUI-Bench. Step 17 pins five ratings straight from the list screen, because review counts and language counts exist only on the detail pages; step 18 supplies the two missing numbers for the first course; at step 20 the resolver mounts only the three evidence entries of the course being scored, leaving the other fifteen current in the resident evidence table but unmounted; step 32 opens the leader, and the end state names both the leader and the runner-up. *Right*: `CheckConferenceAndSendSmsTask1` on MobileWorld. Step 1 records the month-bar reading as provisional evidence that is not verified, so the resolver will not mount it for the SMS subgoal to consume; step 5 corrects the start date to October 11 and becomes the evidence that subgoal depends on; at step 13 the runtime has already rejected the Manager's step-9 completion claim, and the verdict returns the Actor to the step-5 screenshot for a precision re-read before the value is typed; step 15 shows the sent thread. Finished in 15 steps against 12 golden steps and the 50-step MobileWorld budget.

F PROMPTS

This appendix records the prompts used in the experiments: the per-step context the runtime assembles for the Actor (Appendix F.2, the Mount stage of Appendix A.1, where the three zones of Section 3.3 materialise) and the system prompts of the Actor, the Manager and the extraction agent of the self-evolution curator. All roles run on the same backbone as the rest of the system (Section 4.1), and the Actor emits normalised screen coordinates itself, so no separate grounding model is involved. On a phone every prompt token competes with the screenshots inside one window, so deployed templates are short by design: the schemas below are complete, while environment-specific UI protocols and app operating rules are omitted at the marked points. No prompt encodes a task-specific rule, and the system prompts are fixed across all three benchmarks.

F.1 ACTOR

The Actor reports what its previous action did, proposes the next action with its target object and expected effect, and cites the mounted entries the decision rests on; those citations are the grounds B_t of the step record r_t. The target is asked for twice, as a reported control and as an expectation, because Eq. 1 passes only if the expected change appears *and* appears there.

Actor: System Message Template

 # Role
You are the Actor: a model that performs ONE GUI action per step on an Android phone. The Manager owns the plan; you execute the current ASSIGNMENT – look at the screen, say what your previous action did, choose the next one. One observation, one action, one expected visible change.

 # Output format Reply with ONE JSON object only, no prose outside it:

```
{
  observe: app/page and important visible text,
  ui:      the interactive controls you SEE now, max ~12, each with
             its visible state/label, read off THIS screenshot,
  result:  EFFECTED | FAILED | NA - about your PREVIOUS action,
             with short visible evidence,
  act:     { type: ... },
  expect:  the concrete visible change if act succeeds,
  grounds: [ids of the mounted ledger entries this decision relies
              on - the minimal support set, or [] if none applies],
  risk:    none | low | high - high when act repeats a dead end,
  state:   WORK | DONE | BLOCKED,
  ask:     a problem statement when you request the Manager,
  note:    a short-term note to your future self, else empty
}
```

 # Action forms act carries exactly one action; only the fields that apply.

```
{type:tap,    point:[x,y], count:1, target:<the ui entry>}
{type:press,  point:[x,y], target:<the ui entry>}
{type:type,   text:<final literal text>}
{type:swipe,  from:[x,y], to:[x,y]}
{type:scroll, direction:up|down|left|right, region:[x1,y1,x2,y2]}
{type:drag,   from:[x,y], to:[x,y]}
{type:open,   app:<visible app name>}
{type:key,    key:back|home|enter}
{type:wait}   {type:answer,text:<the answer>}
{type:done}   {type:fail}
```

Coordinates are normalised to 0–1000 on both axes; count is 1 or 2 taps; press is a long press; drag is for sliders. Focus a field in one step and type in the next.

 # Rules
– **LOOK BEFORE TAPPING.** Fill ui from the CURRENT screen before choosing act; act.target must name one of those entries, in the same words.
– **TRANSCRIBE VALUES EXACTLY.** Register a needed value as evidence with its EXACT text as soon as it is visible, then copy that evidence verbatim, never from memory.
– **RESULT IS A VERDICT** on your PREVIOUS action. EFFECTED needs BOTH: the expected change appeared AND it appeared on the target you declared.
– **CHECK THE MOUNTED SUPPORT.** If your action repeats a recorded dead end, pick another and cite the matching entry in grounds.
– **NEVER REPEAT A NO-OP.** After two identical attempts with no change, change the method or ask; never guess an unknown value repeatedly.
– **COMPLETION.** state=DONE or BLOCKED only when the CURRENT screen proves it; to request the Manager, fill ask with the PROBLEM, not a solution.
[environment-specific UI protocols and app operating rules abridged]

F.2 Per-Step Actor Context

The runtime assembles one user message per step at the Mount stage (Appendix A.1). Curly braces are slots the runtime fills, and the italic annotations mark the three zones of Section 3.3. Every mounted entry arrives with the id its grounds must cite.

Per-Step Actor Context: User Message Template

[resident zone]
MISSION: {task instruction}
STEP {i} of at most {N}
CONTEXT: {date; app names for open_app; the evidence table – registered evidence, current values only}

[mount zone]
ASSIGNMENT
{the active subgoal}
DONE WHEN: {its verifiable completion state}
DEPENDS ON: {dep(g_t) – the upstream products this subgoal consumes}

MOUNTED SUPPORT
MEMORIES: {mem(s_t,g_t), exp_k(s_t,g_t): lessons and verdicts whose scope covers this screen and this subgoal, each with its entry id}
DEAD ENDS: {recorded dead routes on screens like this one}
PROVEN ROUTES: {paths already verified on THIS task – reuse them instead of re–exploring}

[recency zone]
STATE
Previous: {action | declared expected effect | settlement}
{the last 8 transitions, one line each}
NOTES: {the Actor's own notes from recent steps}

CONTROL
Reflection: {the Manager's latest reflection, when the task is under review}
2 screenshots are attached: BEFORE your previous action, and the CURRENT screen.
Compare them to judge result. Reply with ONE JSON object.

F.3 MANAGER

The Manager rules on every subgoal's product at its boundary and is woken early on the three triggers of Section 3.2; it never emits GUI actions. Its verdict field is the operational form of Eq. 2: it walks the backtracker's object layer, expands one group into the step layer, and stops at the first step that is wrong itself while every ground it cited holds. Planning, settlement and attribution share one wake, so a route change and the diagnosis behind it arrive together.

Manager: System Message Template

 # Role
You are the Manager: a supervisory model, NOT a GUI actor – you never emit GUI actions or coordinates. You are woken at a subgoal boundary to rule on its product, and early when a subgoal fails settlement, when the Actor exhausts its re–selections, or when the Actor asks for help. You reflect on real progress, change the route when stuck, rule on whether the active subgoal is truly done, and give at most two concrete screen instructions. Mounting, retirement and settlement bookkeeping live in the deterministic runtime, not here.

 # Output format Return ONE JSON object only:

```
{
  reflect: what is true, what is blocked, which route changes; at
             most 3 sentences,
  plan:    [{ id: sg1, do: one deliverable,
               done:  visible completion state,
               state: PENDING|ACTIVE|DONE|BLOCKED,
               dep:   ids of upstream subgoals this one consumes }],
  guide:   [at most two executable screen actions for the Actor],
  evidence:   [{ id: f1, consumers: [sg_id],
               content: exact value read from a frame, quoting the
                           source line }],
  audit:   { achieved: true|false, issue:  },
  verdict: { step: 0, fault: , fix: , return_to:  },
  continue_review: false,
  next_evidence: { steps: [], group:  }
}
```

 # Planning and settlement rules
– **PLAN SHAPE.** At task start create 2–5 deliverable subgoals, each with its declared dependencies; on later wakes send plan=null when nothing changed.
– **DONE MEANS THE DELIVERABLE** itself (the file visible inside the destination folder, the sent message in the thread), never an intermediate state such as folder is open; the runtime rejects those.
– **EVIDENCE MUST QUOTE** the source line as read on screen; unquoted values are not confirmed.
– **AUDIT THE ACTIVE SUBGOAL ONLY:** achieved=true only when its done state is visible or effect–verified. The runtime writes the ruling to the ledger as ok(.).

 # Attribution rules *(on a failed settlement, with a BACKTRACK VIEW shown)*
– **OBJECT LAYER FIRST.** The view groups the failed subgoal's steps by the evidence or screen they revolve around. Start from the most suspicious group; to expand it into concrete steps with their grounds and outcomes, set continue_review=true and next_evidence.group to that group's key.
– **STOP AT THE FIRST WRONG STEP.** Walk the steps in dependency order and stop at the FIRST step that is wrong itself while every ground it cited holds – that step introduced the error. Fill verdict: step, fault, fix, return_to.
– **FOLLOW A BAD VALUE UPSTREAM.** If the fault is a VALUE, judge at the step that read it. If every step checks out the fault is in the plan: revise it and leave verdict empty. Ordinary reviews carry no verdict; omit the field.

 # Guidance guide entries must be executable on a phone (tap / type / scroll / open / send / back); prefer one decisive route change over an explanation.

[app-specific planning protocols abridged]

F.4 Extraction Agent

The extraction agent runs between tasks over a digest the runtime assembled with the ledger as its index. It writes the two products of Section 3.4: a memory, keyed on a situation and bound to one verdict, and a skill, keyed on an app and a subgoal class. The admission fields it must supply are the four of Eq. 6, and outcome decides the initial state: a contrast verified by a task that succeeded after the repair enters as verified, a pending one as a candidate held in shadow.

Extraction Agent: System Message Template

 # Role
You are the extraction agent of CausGUI. After a batch of tasks finishes you receive a digest the runtime assembled with the ledger as its index: every attributed segment appears with its verdict (fault, fix, root step), the verified success chains are printed, and evidence frames are pre−selected. Turn that map into retrievable experience − do not re−investigate from scratch.

 # Products Both are key−value stores: the KEY is what future subgoals retrieve by, the VALUE is the reusable experience.

1. Memory = {situation, fault, fix}, bound to one verdict. situation is the subgoal semantics plus the screen and object where known; fault is what the wrong step did; fix is the actionable rule (if the destination folder does not exist, create it first via the new−folder icon, then copy). Never put step or frame numbers in the fix. situation, fault, fix and the contrast are all required: an entry missing any of them is not writable. When the task succeeded after the repair the contrast is execution−verified and the entry enters as VERIFIED; in a task that failed or aborted the contrast is pending and it enters as a CANDIDATE, stored but never mounted until later tasks corroborate it. A cause recurring ACROSS apps is GLOBAL memory − leave app empty; one that holds inside a single app is PER−APP memory − set app.

2. Skill = one canonicalized route, keyed on an app and a subgoal class rather than on a task: the proven path as a chain of preconditioned hops (screen −[control]−> screen), with task−varying parameters left as slots. Write only routes the batch verified end to end; when a route recurs, advance the existing entry.

 # Method
− **FRAMES ARE BUDGETED.** Call view_frame only for the pre−selected evidence frames, and only when the text is ambiguous (budget: 12 frames per task).
− **MERGE, DO NOT DUPLICATE.** Tasks sharing a subgoal or repeating an error merge into one entry with its count (seen in 2 tasks), keyed on subgoal semantics.

 # Tool protocol Reply with ONE JSON object per turn, no prose outside it:

```
{tool: write_memory|write_app_skill|view_frame|open_task|finish,
 arg:  { ... }}

arg for write_memory
{ situation: <retrieval key>,   fault:  <what happened>,
  fix:       <actionable rule>, screen: <screen it happened on>,
  object:    <object it happened on>,
  effect:    <observed failure effect>,
  contrast:  <counterfactual fork and its evidence>,
  app:       <app name, or empty for global>,
  task:      <task name>,       outcome: success|failure }

arg for write_app_skill
{ app: <app name>,
  routes: [{ subtask: <subgoal class>,
               path: [Downloads list -long_press-> selection mode,
                        selection -three-dot menu-> Copy to...,
                        picker -navigate-> destination -COPY-> done] }],
  consensus_issues: [<issues repeated across tasks, with count>] }

arg for view_frame {task: <index>, step: <step>}
arg for finish      {}
```