KNOWING WHEN TO STOP, RECOVER, AND SEARCH: A MODULAR FRAMEWORK FOR GUI AUTOMATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Autonomous GUI agents face two fundamental challenges: *early stopping*, where agents prematurely declare success without verifiable evidence, and *repetitive loops*, where agents cycle through the same failing actions without recovery. Existing agent planners track progress through generated subtasks, yet in GUI environments, completing planned steps does not guarantee that the user's requirements are met. Moreover, existing anti-loop mechanisms typically detect only local repetition and lack mechanisms to escalate recovery when the agent remains stuck. To this end, we present **GUI-Pro Agent**, a modular GUI agentic framework organized around three decisions: when to `Stop`, `Recover`, and `Search`. To prevent premature termination, a mandatory *Completeness-Aware Planner* tracks UI-observable success criteria throughout execution and independently verifies completion claims before termination. To recover from stalled agent action loops, a mandatory *Loop Breaker* detects repeated ineffective agent actions and progressively forces the agent to change its action strategy. When the current observation is insufficient to determine a viable workflow, an on-demand *Search Agent* supplies external procedural knowledge to the agent. To validate GUI-Pro Agent, we evaluate it on two benchmarks containing Linux and Windows tasks with five LLM backbones, achieving top performance on both (**77.5%** on OSWorld and **61.0%** on WindowsAgentArena). Notably, three of the five backbones surpass human performance (72.4%) on OSWorld in a single pass. Ablation studies further show that completion checking, loop recovery, and search provide complementary gains that vary with backbone capability and interaction budget, while the Loop Breaker nearly halves wasted steps for loop-prone backbones (*e.g.*, Gemini 3 Flash). Moreover, transferring three proposed mechanisms to an existing GUI agent (CoAct-1) also improves it by over 5% to 70.3% on a controlled OSWorld subset, showing the strong portability of these components.

1 INTRODUCTION

The rapid advancement of multimodal large language models (MLLMs) has catalyzed a new generation of autonomous Graphical User Interface (GUI) agents (Qin et al., 2025; Agashe et al., 2025a; Wang et al., 2025b; Wu et al., 2025; Hong et al., 2024) capable of performing desktop tasks by observing screenshots and executing mouse and keyboard actions. Systems such as OSWorld (Xie et al., 2024), WindowsAgentArena (Bonatti et al., 2025), and related benchmarks (Zhang et al., 2025a; Yang et al., 2025a) have established standardized evaluation environments spanning Linux, Windows and macOS, revealing both promises and persistent limitations of current approaches.

Despite steady progress, two fundamental problems remain largely unsolved in this domain. First, GUI agents do not reliably know *when a task is finished*: they routinely declare success prematurely, *e.g.*, after toggling a setting without verifying the state changed—because completion is left to the model's implicit judgment rather than verified against observable UI evidence (Andrade et al., 2026; Cemri et al., 2025). Existing planning modules primarily focus on how to complete intermediate subgoals, but provide limited mechanisms for verifying whether the requested final state has actually been achieved (Agashe et al., 2025a;b). Second, agents fall into *repetitive loops*: cycling through the same failing action without recovery. Existing anti-looping heuristics can detect repeated actions, but often lack a systematic way to change strategy when repeated attempts continue to fail (Yao et al., 2023b; Shinn et al., 2023; Madaan et al., 2023; Li et al., 2023; Kim et al., 2023).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

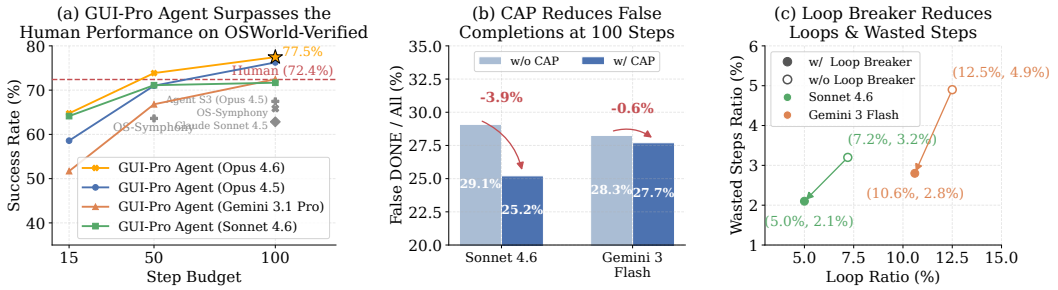

Figure 1: **Performance of GUI-Pro Agent.** (a) Our GUI-Pro Agent (w/ Opus 4.6) achieves the best results (77.5%) on OSWorld-Verified (Xie et al., 2024) and surpasses human performance with one pass. (b) Completeness-Aware Planner (CAP) reduces false completion rates at a 100-step budget. (c) The Loop Breaker reduces repetitive loops and wasted action steps at the 100-step budget.

We present **GUI-Pro Agent**, a modular GUI agent framework that addresses both challenges through three mechanisms, *a Completeness-Aware Planner, a Loop Breaker, and a Search Agent* that enable the system to determine when to `STOP`, `RECOVER` (from repetitive loops), and `SEARCH` online for enhanced performance. To mitigate early stopping, we further introduce the *Completeness-Aware Planner* (CAP), which maintains a set of UI-observable completion criteria throughout execution and uses the status of unmet criteria at each step to guide subsequent decisions. Unlike conventional planners that prescribe a sequence of intermediate steps, CAP lets the agent choose actions dynamically based on the current UI state and what remains to be completed. This design helps prevent the agent from stopping simply because it has finished its planned actions while the requested final state has not yet been achieved. To address repetitive loops, we introduce a mandatory *Loop Breaker* that runs after every action and escalates its intervention as evidence of a loop accumulates—from switching the interaction modality (*e.g.* clicking screen elements or writing Bash commands), to forcing an outright strategy change, to acting on an external loop signal—so that the agent is redirected before repeated failures consume the step budget.

Beyond the two core challenges, GUI agents frequently stall on unfamiliar application workflows, leading to poor out-of-distribution performance. Prior work addresses this via visual browser-based search (Yang et al., 2026a), which requires additional agent steps and a visual grounding pipeline. We instead deploy an on-demand Search Agent that directly queries a capable LLM with search grounding (*e.g.*, Gemini 3 Pro (Google DeepMind, 2025)), returning plain textual knowledge injected into the Manager's context, avoiding the need to navigate a browser to retrieve procedural text. We further integrate a Coding Agent (Coder) for accurate command line executions and a Grounding Agent (Grounder) for visual grounding following existing works (Gonzalez-Pumariega et al., 2026; Yang et al., 2026a; Wang et al., 2024).

We evaluate GUI-Pro Agent on OSWorld-Verified (Xie et al., 2024) and WindowsAgentArena (WAA) (Bonatti et al., 2025) across five LLM backbones to systematically validate the framework across distinct model families and capability tiers. As shown in Figure 1, GUI-Pro Agent reaches 77.5% with Opus 4.6, and Opus 4.5 and Gemini 3.1 Pro also exceed the human reference level of 72.4%, making GUI-Pro Agent the first GUI framework to do so. Furthermore, GUI-Pro Agent generalizes well to WAA, reaching 61.0% and outperforming the strongest baselines by over 4%.

Ablation studies further validate the contribution of each component. The full GUI-Pro Agent system with Sonnet 4.6 achieves 71.67% on OSWorld, while removing the Completeness-Aware Planner, Searcher, or Loop Breaker reduces performance to 68.81%, 70.04%, and 71.63%, respectively. To further examine whether these proposed modules are transferable to other GUI frameworks, we integrate them into CoAct-1 (Song et al., 2026). On a fixed OSWorld subset, adding the Completeness-Aware Planner, Search Agent, and Loop Breaker individually improves CoAct-1 by 3.78%, 0.83%, and 0.93%, respectively, while combining all three improves performance by over 5% (*i.e.*, 65.13% to 70.34%), highlighting their strong potential to transfer to other GUI frameworks.

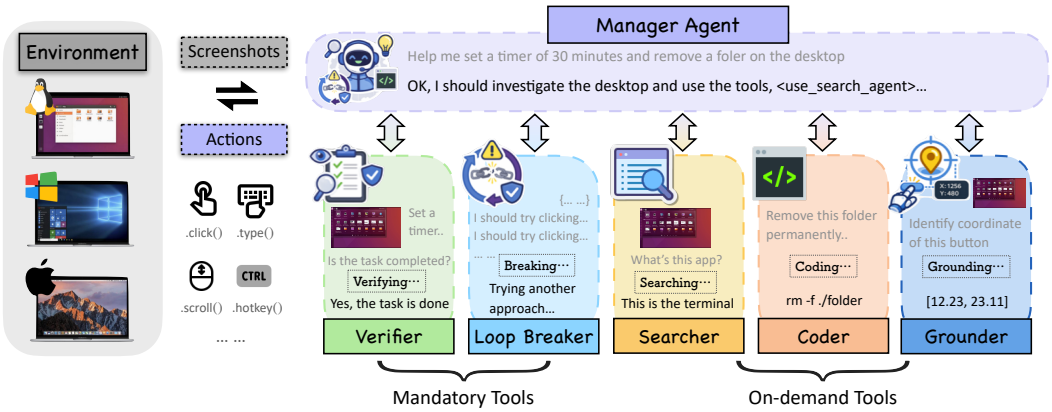

Figure 2: **Overview of GUI-Pro Agent.** The Manager Agent perceives screenshots and acts through basic GUI actions (*e.g.*, `click`, `type`, `scroll`, `hotkey`). Two *mandatory* modules coordinate with the Manager: the Verifier guards against premature completion, and the Loop Breaker interrupts stalled action loops. In the meanwhile, three *on-demand* tools: Search Agent (Searcher), Coding Agent (Coder), and Grounding Agent (Grounder), are triggered by the Manager as needed.

2 Related Work

GUI Agents: Models and Frameworks. End-to-end models trained for GUI interaction—UI-TARS (Qin et al., 2025), AGUVIS (Xu et al., 2025), ShowUI (Lin et al., 2025), CogAgent (Hong et al., 2024), OS-Atlas (Wu et al., 2025), among others (Yang et al., 2025b; Wang et al., 2025b; Fu et al., 2025)—achieve strong grounding without HTML or accessibility trees. Frontier providers have followed with commercial APIs: Claude Computer Use (Anthropic, 2024), OpenAI CUA (OpenAI, 2025), and Seed-1.8 (Bytedance Seed, 2026), the latter serving as our dedicated grounding model. Complementary modular frameworks combine MLLMs with planning, memory, and tools. The Agent S family (Agashe et al., 2025a;b) couples hierarchical planning with experience memory; Agent S3 (Gonzalez-Pumariega et al., 2026) adds best-of-N trajectory selection. OS-Symphony (Yang et al., 2026a) and GTA1 (Yang et al., 2026b) combine memory systems and test-time search, while CoAct-1 (Song et al., 2026) and EvoCUA (Xue et al., 2026) emphasize coding-as-actions and synthetic experience. UFO (Zhang et al., 2025b) and AutoGLM (Liu et al., 2024) target Windows and mobile respectively.

Self-Verification, Termination, and Error Recovery. Reliable termination remains difficult because progress must be inferred from partial, noisy observations. ReAct (Yao et al., 2023b) and Tree of Thoughts (Yao et al., 2023a) enable mid-trajectory correction, while Reflexion (Shinn et al., 2023) and Self-Refine (Madaan et al., 2023) improve policies through explicit self-feedback. For computer-control agents, structured reflection (Li et al., 2023; Kim et al., 2023) has been explored to reduce repeated failures. Complementarily, verifier-based training and step-wise checking (Cobbe et al., 2021; Lightman et al., 2024) improve reliability in reasoning tasks; Pan et al. (Pan et al., 2024) extend autonomous evaluation to digital agents. In GUI settings, Self-Grounded Verification (Andrade et al., 2026) reveals agreement bias in MLLM verifiers, and failure taxonomies (Cemri et al., 2025) identify premature completion and action loops as dominant errors. GUI-Pro Agent directly targets these issues with evidence-grounded Completeness-Aware Planner and a multi-level Loop Breaker that escalates recovery strategies.

3 GUI-Pro Agent

3.1 Overview

GUI-Pro Agent is built around a Manager agent that performs GUI interaction in a perceive–reason–act loop as shown in Figure 2. At step t, its belief state b_t summarizes the user instruction, trajectory history, and current observation, including the screenshot and tool results. From

this state, it selects an action a_t: a UI primitive (*e.g.*, clicking or scrolling through `pyautogui`) or a call to the Search, Coding, or Grounding Agent. Keeping these choices in a shared action space allows the Manager to select the form of assistance appropriate to the current obstacle while retaining responsibility for the task throughout execution.

3.2 COMPLETENESS-AWARE PLANNER

Conventional hierarchical planning decomposes how to execute the task (Agashe et al., 2025a;b). These designs primarily focus on how a task should be decomposed, but they provide weaker signals regarding whether the requested outcome has actually been achieved. Our *Completeness-Aware Planner* (CAP) instead represents what must be true for the task to be complete. It combines a *Completion Gate* for evidence checks with a separate model judge for proposed completion claims.

Completion Gate. To make completion assessable as execution unfolds, the gate first derives K UI-observable success criteria $\mathcal{C} = \{c_1, \ldots, c_K\}$ from the task instruction. Expressing these criteria in terms of the requested final state lets the Manager reconsider its route without losing track of what must ultimately be demonstrated. The criteria do not prescribe an action order or a dependency graph; their evidence status is revisited at each step to identify what still requires execution or verification. Hidden-state requirements are represented by observable proxies where possible, whose adequacy remains a limitation. At each step t, the Manager checks the available evidence against these criteria using its belief state b_t:

$$\text{Gate}(b_t) = \begin{cases} \text{DONE} & \text{if self-check passes } K \text{ criteria and UI is stable} \\ \text{CONTINUE} & \text{otherwise} \end{cases} \tag{1}$$

Because UI changes may lag behind an action, an immediate screenshot can leave its outcome unresolved. Action-specific verification rules ask the Manager to check the expected visual outcome after each action (Table 8 in Appendix A.5), helping it distinguish pending updates from evidence that a requirement has been satisfied.

Completeness Model Judge. Because the Completion Gate only conditions on the initial task instruction, the Manager can still miss potential task requirements, incurring premature task termination even with the Completion Gate. Once the gate outputs DONE, a separate MLLM judge therefore revisits the original instruction alongside the current observation and recent trajectory, rather than relying solely on the initial criterion list. This second perspective is intended to expose missing requirements or unsupported completion claims at the point where they would otherwise end execution.

Given the instruction g, observation o_t, and trajectory context summarized in b_t, the judge returns an accept/reject decision with an explanation. Acceptance requires direct visual evidence for every criterion, visible confirmation of side-effect actions such as saving, exporting, or sending, and a stable UI. Termination requires agreement between the gate and the judge. On rejection, criterion-level evidence and missing-evidence feedback are appended to the trajectory, giving the Manager a basis for further action. Thus, the final decision is binary while the feedback preserves partial progress. The judge's instructions appear in Appendix A.7.2.

3.3 LOOP BREAKER

Repeated failures can reflect an ineffective interaction, an unproductive workflow, or a broader trajectory pattern. The Loop Breaker addresses these at three levels using screen observations o_t and actions a_t. It runs after every Manager action: Tiers 1–2 check local action and screen-state recurrence, while Tier 3 invokes an external model to inspect the recent trajectory and produce a strategy-level signal. We define counters for repeated actions without visible change and recurring screen states:

$$n_t^a = \left|\{i \in [t{-}1,\, t] : a_i = a_t \ \wedge\ o_{i+1} \approx o_i\}\right|, \qquad n_t^o = \left|\{i \in [t{-}2,\, t] : o_i \approx o_t\}\right| \tag{2}$$

These counters and an external reflection signal w_t trigger mandatory intervention, with fixed thresholds $\tau_a = 2$ and $\tau_o = 3$ across tasks and both benchmark platforms.

Tier 1: Modality Switch ($n_t^a \geq \tau_a$). If the same action on the same target repeatedly produces no visible change, the agent must switch interaction mode (*e.g.*, keyboard shortcut → menu click → command-line).

Tier 2: Strategy Change ($n_t^o \geq \tau_o$). If the same screen state recurs three times without progress, the agent must change its overall strategy (*e.g.*, from menu navigation to programmatic file editing). Repetition alone does not establish task infeasibility; failure termination requires separate evidence.

Tier 3: Reflection-Driven Judge. To detect unproductive patterns beyond local repetition, an external model judge inspects the recent trajectory and returns a "keep" or "switch" signal. When w_t signals a switch, a directive at step $t+1$ blacklists the repeated action and requires the Manager to choose an alternative (*e.g.*, `click` $\rightarrow$ `type`, or GUI actions $\rightarrow$ `call_coding_agent`). The prompts appear in Appendix A.7.1.

Together, the three tiers connect local action failures, recurring navigation dead ends, and broader trajectory patterns to changes in interaction mode or strategy.

3.4 Search Agent (Searcher)

Recovery can change the route, but an unfamiliar application may leave the agent without a useful alternative to try. A screenshot reveals the current interface while often leaving the procedure for reaching a goal implicit. In such cases, a short tutorial can supply the missing workflow knowledge and give subsequent interaction a more informed starting point.

Retrieving that knowledge through visual browser search, as in OS-Symphony (Yang et al., 2026a), introduces additional navigation and grounding demands. Since the information sought is procedural text, we expose a direct textual interface through the Search Agent $\mathcal{S}$. The Manager invokes `search(query)` within the same action space as UI primitives, formulating a targeted "How to" question according to guidelines in the tool description. An LLM with native search grounding (*e.g.*, Gemini 3 Pro (Google DeepMind, 2025)) returns a summarized tutorial as plain text, which is added to the belief state for subsequent steps. This avoids browser interaction for retrieval while leaving the Manager to apply the guidance in the current environment.

`Search` is useful when uncertainty concerns a workflow for which external guidance is likely to exist. Accordingly, the tool is invoked on demand when the Manager is uncertain about a GUI workflow and a tutorial is likely to be available, as with well-documented application features.

In addition to the above components, we also equip the Manager agent with a grounding agent (Grounder) for screen elements localization and a coding agent (Coder) for code executions (Gonzalez-Pumariega et al., 2026; Yang et al., 2026a; HCIII Team, Lenovo, 2026).

4 Experiments

4.1 Experimental Settings

Benchmarks. We evaluate our GUI-Pro Agent on two desktop GUI benchmarks. OSWorld-Verified (OSWorld for short) (Xie et al., 2024) is our primary benchmark. Following common practice (Xie et al., 2024), we use the 361-task evaluation set that excludes the eight Google Drive tasks, spanning five application domains (OS, Office, Daily, Professional, and Workflow) in an Ubuntu virtual environment. WindowsAgentArena (WAA) (Bonatti et al., 2025) provides 154 tasks on a Windows environment spanning productivity, file management, and system configuration, serving as our cross-platform generalization testbed.

Implementation Details. In each configuration, the named backbone serves as the Manager, Reflection Agent, Completeness Model Judge, and Coding Agent, with Gemini 3 Pro (Google DeepMind, 2025) or Gemini 3.1 Pro (Google DeepMind, 2026) as the Search Agent. Seed 1.8 (Bytedance Seed, 2026) is the default grounding model, except for one Opus 4.5 variant that uses MAI-UI (Zhou et al., 2025); on WAA, Gemini 3 Flash is the Manager. The Completeness Model Judge within Completeness-Aware Planner uses temperature $T = 0.2$ and all other components use $T = 1.0$. Full model assignments and infrastructure are given in Appendix A.5.

4.2 Main Results

GUI-Pro Agent Surpasses Human-level Performance. With a 100-step budget, GUI-Pro Agent with Opus 4.5 achieves 74.89% and with Opus 4.6 achieves 77.45%, exceeding the reported hu-

Table 1: Success rates (%) on OSWorld-Verified (Xie et al., 2024). Our results use the 361-task evaluation set. We report results of 15, 50, and 100 steps budgets. Baseline results are taken from the OSWorld leaderboard, except OS-Symphony at 100 steps, which is taken from its paper.

Method	**Step**	**Success Rate (%)**					
		OS	Office	Daily	Prof.	Work.	Avg.
Max 15 Steps							
GUI-Pro Agent w/ Gemini 3 Flash	15	79.20	29.92	54.14	57.13	34.00	43.15
GUI-Pro Agent w/ Gemini 3.1 Pro	15	83.30	52.99	54.33	61.22	34.70	51.69
GUI-Pro Agent w/ Sonnet 4.6	15	83.30	**69.72**	58.72	57.13	**60.20**	64.13
GUI-Pro Agent w/ Opus 4.5	15	**87.50**	57.16	59.69	71.43	45.20	58.58
GUI-Pro Agent w/ Opus 4.6	15	83.30	60.65	**66.38**	**79.59**	55.90	**64.75**
Max 50 Steps							
UiPath w/ GPT-5 (UiPath, 2025)	50	73.91	49.52	62.12	71.43	37.30	53.69
CoAct-1 w/ GPT-5 (Song et al., 2026)	50	70.83	60.65	54.09	69.39	42.37	56.39
OS-Symphony w/ GPT-5 (Yang et al., 2026a)	50	78.26	65.74	65.92	79.59	55.30	65.77
GUI-Pro Agent w/ Gemini 3 Flash	50	83.30	69.19	62.75	62.49	51.00	63.14
GUI-Pro Agent w/ Gemini 3.1 Pro	50	83.30	71.51	67.32	67.35	55.90	66.80
GUI-Pro Agent w/ Sonnet 4.6	50	83.30	**79.23**	69.24	57.13	**66.70**	71.11
GUI-Pro Agent w/ Opus 4.5	50	**87.50**	75.97	71.38	79.59	55.60	71.00
GUI-Pro Agent w/ Opus 4.6	50	83.30	78.60	**72.60**	**83.67**	61.30	**73.85**
Max 100 Steps							
Seed-1.8 (Bytedance Seed, 2026)	100	66.67	68.80	67.05	71.43	42.38	61.87
Claude-Sonnet-4.5 (Anthropic, 2025)	100	70.83	72.59	61.35	63.27	49.54	62.84
Kimi K2.6 (Kimi Team, 2026)	100	79.17	80.03	77.11	81.63	54.97	73.07
CoAct-1 w/ GPT-5 (Song et al., 2026)	100	75.00	62.93	57.94	71.43	47.87	59.93
GTA1 w/ GPT-5 (Yang et al., 2026b)	100	79.17	63.91	62.56	79.59	50.91	63.41
OS-Symphony w/ GPT-5 (Yang et al., 2026a)	100	79.17	65.73	67.76	69.23	57.98	65.84
Agent S3 w/ GPT-5 (Gonzalez-Pumariega et al., 2026)	100	77.50	66.30	61.36	69.80	51.37	62.60
UiPath w/ Opus 4.5 (UiPath, 2025)	100	70.83	74.13	68.33	73.47	52.97	67.14
Agent S3 w/ Opus 4.5 (Gonzalez-Pumariega et al., 2026)	100	75.00	75.53	63.94	58.37	57.43	66.00
HIPPO w/ Opus 4.5 (HCIII Team, Lenovo, 2026)	100	87.50	74.27	69.27	**95.92**	64.31	74.49
GUI-Pro Agent w/ Gemini 3 Flash	100	91.70	64.90	74.85	67.33	63.40	68.77
GUI-Pro Agent w/ Gemini 3.1 Pro	100	83.30	76.60	73.76	73.47	62.90	72.47
GUI-Pro Agent w/ Sonnet 4.6	100	83.30	79.24	69.25	57.14	**68.80**	71.67
GUI-Pro Agent w/ Opus 4.5	100	87.50	79.38	**76.57**	81.63	61.00	74.89
GUI-Pro Agent w/ Opus 4.5 + MAI-UI	100	91.67	**84.26**	72.77	83.67	61.09	76.26
GUI-Pro Agent w/ Opus 4.6	100	**91.70**	82.87	75.17	83.67	65.60	**77.45**

man reference level of 72.4%. Notably, three of the five backbones surpass human performance at 100 steps: Opus 4.6 (77.45%), Opus 4.5 (74.89%), and Gemini 3.1 Pro (72.47%), demonstrating that GUI-Pro Agent's framework design generalizes across model families. When varying the Manager Agent backbone, GUI-Pro Agent w/ Sonnet 4.6 achieves an overall 71.67%, with particularly strong results in the OS and Multi-Apps domains (per OSWorld's official task categorization (Xie et al., 2024)), where it surpasses Agent S3 w/ Opus 4.5 by 8.3% and 11.4%, respectively (*i.e.* average 76% *vs.* 66%). Moreover, even Gemini 3 Flash achieves 68.77%, outperforming the best GPT-5-based frameworks despite using a substantially lighter model, providing additional evidence that our GUI-Pro Agent can effectively adapt to models with different capabilities.

Across categories, Opus 4.6 and Gemini 3 Flash both exceed 91% on the OS domain at 100 steps, outperforming the strongest baseline (HIPPO at 87.50%) by 4.2% and other baselines by over 12% (*e.g.*, OS-Symphony at 79.17%), indicating reliable system-level task handling across backbones. The Professional domain shows the largest variance (57.1% Sonnet 4.6 vs. 83.7% Opus 4.6), suggesting that the reasoning depth of the Manager matters more in difficult tasks.

Table 2: Ablation of components on OSWorld. Component effects depend on the backbone and budget.

Method	**Success Rate (%)**	
	50 Steps	**100 Steps**
GUI-Pro Agent w/ Gemini 3 Flash	63.14	**68.77**
– CAP	**66.00**	67.34
– Loop Breaker	58.90	66.95
– Search Agent	62.54	65.82
GUI-Pro Agent w/ Sonnet 4.6	**71.11**	**71.67**
– CAP	68.53	68.81
– Loop Breaker	69.67	71.63
– Search Agent	68.92	70.04

Additionally, GUI-Pro Agent generalizes well to WindowsAgentArena (Table 3). On WAA, GUI-Pro Agent reaches 60.4% overall at 50

Table 3: Results on WindowsAgentArena (WAA) (Bonatti et al., 2025). Our GUI-Pro Agent (Gemini 3 Flash) achieves the best performance across both 50 and 100 step configurations.

Method	Office	Web	Sys.	Code	Media	Util.	Overall
Max 50 Steps							
Qwen3-VL-32B (Bai et al., 2025)	19.1	49.7	54.2	21.1	42.2	25.0	31.7
UI-TARS-2 (Wang et al., 2025a)	-	-	-	-	-	-	50.6
Agent S3 w/ GPT-5 (Gonzalez-Pumariega et al., 2026)	-	-	-	-	-	-	54.1
GUI-Pro Agent	32.6	73.3	87.5	66.7	52.4	75.0	**60.4**
– Completeness-Aware Planner	11.6	73.3	70.8	54.2	61.9	83.3	51.3
– Loop Breaker	27.9	66.7	83.3	58.3	38.1	75.0	52.6
– Search Agent	18.6	63.3	75.0	66.7	38.1	66.7	49.4
Max 100 Steps							
GTA1-32B w/ o3 (Yang et al., 2026b)	-	-	-	-	-	-	51.2
Agent S3 w/ GPT-5 (Gonzalez-Pumariega et al., 2026)	-	-	-	-	-	-	56.6
GUI-Pro Agent	35.0	73.3	87.5	66.7	52.4	83.3	**61.0**
– Completeness-Aware Planner	13.9	73.3	70.8	54.2	57.1	83.3	51.3
– Loop Breaker	30.2	66.7	83.3	58.3	47.6	83.3	55.8
– Search Agent	25.0	63.3	75.0	66.7	47.6	83.3	53.9

steps and 61.0% at 100 steps, which outperforms Agent S3 w/ GPT-5 at both configurations by at least 4.4% (*i.e.*, Agent S3 56.6% at 100 steps), and GTA1 w/ o3 at 100 steps by 9.8%, providing evidence of GUI-Pro Agent's ability to generalize across operating systems and task distributions.

Configuration-Controlled Comparisons. Under our harness, the Sonnet 4.6 base scaffold (w/ coder + grounder) scores 69.19% on the 361 tasks, compared with 71.67% for the full framework, a 2.48% gain with the same backbone, grounding, action interface, and budget.[1] With Opus 4.5, GUI-Pro Agent achieves 74.89% using the Seed 1.8 grounder, already exceeding HIPPO's 74.49% with the stronger MAI-UI grounder (Zhou et al., 2025). Replacing only our grounder with MAI-UI raises GUI-Pro Agent's score to 76.26%, widening the advantage over HIPPO to 1.77% under the same backbone and grounder.

4.3 Verify Each Component

To understand the effect of each component in GUI-Pro Agent, we conduct an ablation study and analyze early stopping and loop behavior. Table 2 reports the results across different MLLM backbones and action budgets on OSWorld. For Sonnet 4.6 at 100 steps, removing Completeness-Aware Planner lowers the score by 2.86% (71.67% to 68.81%), while removing search lowers it by 1.63% and removing loop control by only 0.04%. For Gemini 3 Flash, loop control gives a 4.24% gain at 50 steps and search a 2.95% gain at 100 steps. Verification is not uniformly beneficial: at 50 steps, Gemini 3 Flash scores 66.00% without it, compared with 63.14% in the full system, a 2.86% decrease when verification is included. At 100 steps its contribution is positive (68.77% versus 67.34%). The results support configuration-dependent benefits rather than a claim that every module always improves performance.

On WAA (Table 3), removing Search, verification, and Loop Breaker lowers the 50-step score by 11.0%, 9.1%, and 7.8%, respectively. At 100 steps the corresponding drops are 7.1%, 9.7%, and 5.2%. Removing verification lowers Office accuracy from 32.6% to 11.6% at 50 steps, and removing search lowers it to 18.6%. These stronger effects on WAA complement the smaller or occasionally negative effects on OSWorld.

Completeness-Aware Planner Mitigates False Completion Ratio. Figure 3 analyzes false completions using the 100-step results in Table 10. We report *False DONE/Failed* (FDF), the fraction of failed tasks with false completion, and *False DONE/All* (FDA), the false DONE count divided by all

[1]The public Sonnet 4.6 result of 72.1% (Anthropic, 2026b) uses Anthropic's native computer-use tools and screen-cropping grounding and is therefore a system-level reference rather than a control.

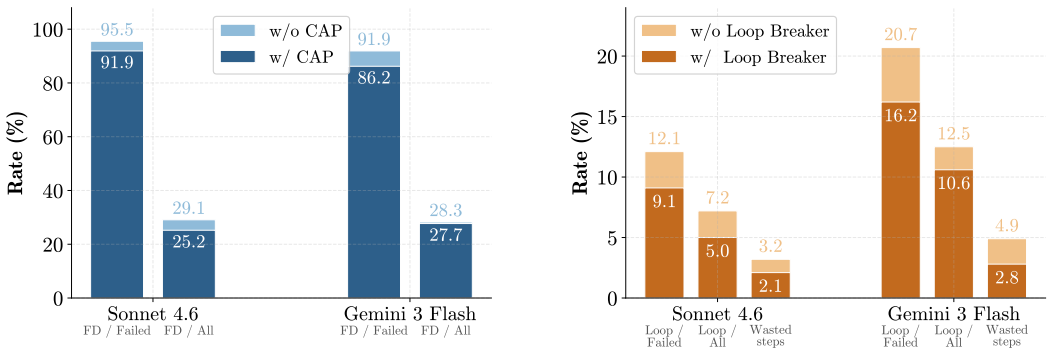

Figure 3: Component analysis on OSWorld at 100 steps. CAP reduces false completions (FD: false DONE), using Table 10; All denotes all 361 tasks. The Loop Breaker reduces loop incidence and the wasted steps ratio.

361 tasks. FDA differs from the complement of DONE accuracy, whose denominator is completion signals. For Sonnet 4.6, CAP reduces false DONE counts from 105 to 91, lowering FDF by 3.6% (95.5% to 91.9%) and FDA by 3.9% (29.1% to 25.2%). For Gemini 3 Flash, false DONE counts decrease from 102 to 100: FDF drops by 5.7% (91.9% to 86.2%), while FDA decreases by 0.6% (28.3% to 27.7%). Even with CAP, FDF remains above 86% for both backbones: most failed tasks still end with an incorrect completion claim. CAP therefore only partially resolves false completion.

Loop Breaker Reduces Loops and Wasted Steps. We similarly analyze loop behavior with three metrics: *Loop / Failed* (LF) (fraction of failed tasks involving loops), *Loop / All* (LA) (fraction of all tasks with loops), and *Wasted Steps Ratio* (WSR) (fraction of total steps spent in detected loops) with results also in Figure 3. The Loop Breaker reduces loop incidence across both backbones. In detail, for Sonnet 4.6, LF drops by 3% from 12.1%, with wasted steps decreasing from 3.2% to 2.1% over all steps. The effect is more pronounced for Gemini 3 Flash, where LF drops by 4.5% (from 20.7% to 16.2%) and wasted steps nearly halve from 4.9% to 2.8%. This backbone-dependent pattern is consistent with the ablation findings: Gemini 3 Flash loops roughly twice as often as Sonnet 4.6 (*i.e.*, 10.6% vs. 5.0% on LA), giving evidence that weaker models are more prone to repetitive behavior and benefit more from the loop detection.

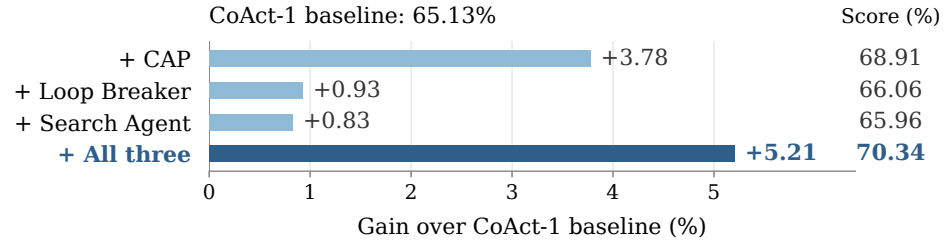

Figure 4: Gains over CoAct-1 (65.13%) on an OSWorld subset with Gemini 3 Flash and a 100-step Manager budget. CAP denotes Completeness-Aware Planner. Each proposed component helps improve CoAct-1's performance, while adding all three yields the highest score.

Transfer to CoAct-1. To understand whether the proposed completion, loop-recovery, and search mechanisms can improve other *off-the-shelf* GUI agents, we integrate these three proposed modules into CoAct-1 (Song et al., 2026) while keeping its other components untouched. We compare the CoAct-1 baseline with each module added individually and with all three added together on a fixed 60-task OSWorld subset. In detail, we use Gemini 3 Flash for the Manager and the CAP backbone, Seed 1.8 for grounding, and a 100-step Manager budget across all experiments; the Coder and Searcher are Gemini 3.1 Pro. We present more details in Appendix A.1. As shown in Figure 4, the CAP gives the largest individual score increase (65.13% *vs.* 68.91%), followed by the Loop Breaker (65.13% to 66.06%) and Search Agent (65.13% to 65.96%). Combining all three raises the score from

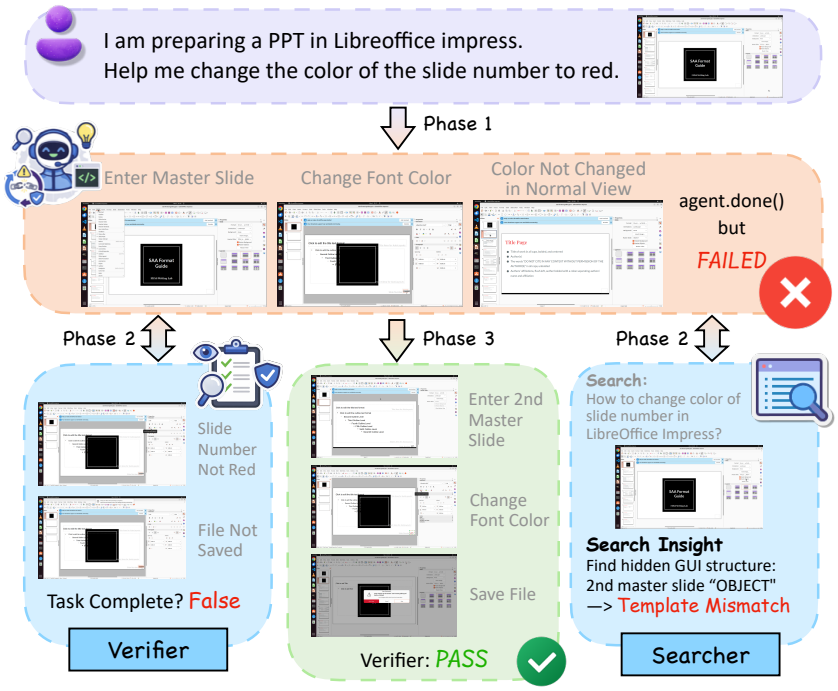

Figure 5: **Case study on OSWorld:** ***Changing slide number color to red in LibreOffice Impress.*** The Manager encounters multiple failures before final completion. CAP rejects two premature `done()` calls, and the Searcher provides critical knowledge about multi-master-slide editing.

65.13% to 70.34% (+5.21%). These results provide evidence that the benefits of proposed modules extend beyond our specific Manager implementation and can serve as truly portable enhancements to other GUI agents. Additionally, the fact that our CAP yields the largest individual gain further validates its effectiveness in addressing early stopping issues, which contributes most to performance gains in even more general GUI scenarios.

4.4 Case Study

To illustrate how components interact to recover from failures, we display a case study from OSWorld in Figure 5. The task instruction is *"The slide number is barely visible. Please change the color of the slide number to red."*. The Manager's first attempt modifies one master slide and calls `done()`, but our CAP rejects it, noting the slide number is still gray and the file has not been saved. Prompted by the rejection, the Manager invokes the Searcher. The retrieved procedural knowledge contains a key insight: *"If your presentation uses different master slides for different sections, repeat for each master slide."* Acting on this, the Manager inspects the slide content and discovers a second master named `OBJECT` that it had overlooked. With this knowledge, the Manager applies the color change to both slides and calls `done()` a second time. The CAP rejects it again: the color is now correct, but the file is still unsaved. Only after an explicit `Ctrl+S` does the Judge accept the result. This case highlights the complementary roles: CAP prevents early termination while the Searcher supplies task-specific knowledge.

5 Conclusion

In this paper, we presented GUI-Pro Agent, a modular GUI agent framework that addresses premature task completion and repetitive action loops in autonomous desktop agents. Its Completeness-Aware Planner, Loop Breaker, and Search Agent push OSWorld performance to 77.5% with Opus 4.6 and reach 61.0% on WindowsAgentArena. Ablations show that each component contributes, with gains shaped by backbone strength and step budget. False completion remains the dominant failure mode, motivating continued research on completion verification.

Ethics Statement

This work studies autonomous agents that operate desktop software through GUI actions. All experiments are conducted in the sandboxed virtual machines provided by OSWorld-Verified and WindowsAgentArena; no human subjects, private user data, or live user accounts are involved. Because agents that act on real desktops can modify, delete, send, or expose information, we discuss the associated risks and the safeguards required for deployment (sandboxing, scoped permissions, confirmation of irreversible operations, and audit logs) in Appendix A.4. The Completeness-Aware Planner and Loop Breaker are reliability mechanisms and are not substitutes for such security and authorization controls.

Declaration of AI Tool Usage

During the preparation of this manuscript, we used OpenAI's GPT-5.6 model for minor language refinement and smoothing of the writing. The AI tool was not used for generating original content, conducting data analysis, or formulating core scientific ideas. All conceptual development, experimentation, and interpretation were conducted independently without reliance on AI tools.

References

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent S: An open agentic framework that uses computers like a human. In *ICLR*, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/394c7c30ea87b5c3521b4d9e9d419071-Abstract-Conference.html.

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. In *COLM*, 2025b. URL https://openreview.net/forum?id=zg5is4GJ3R.

Moises Andrade, Joonhyuk Cha, Brandon Ho, Vriksha Srihari, Karmesh Yadav, and Zsolt Kira. Let's think in two steps: Mitigating agreement bias in MLLMs with self-grounded verification. In *ICLR*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/3e1fb72d876d634b2ac29e9ac2c937c9-Abstract-Conference.html.

Anthropic. Developing a computer use model. https://www.anthropic.com/research/developing-computer-use, 2024.

Anthropic. Introducing Claude Sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, 2025.

Anthropic. Introducing Claude Opus 4.6. https://www.anthropic.com/news/claude-opus-4-6, 2026a.

Anthropic. Introducing Claude Sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6, 2026b.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, Mei Li, Kaixin Li, Zicheng Lin, Junyang Lin, Xuejing Liu, Jiawei Liu, Chenglong Liu, Yang Liu, Dayiheng Liu, Shixuan Liu, Dunjie Lu, Ruilin Luo, Chenxu Lv, Rui Men, Lingchen Meng, Xuancheng Ren, Xingzhang Ren, Sibo Song, Yuchong Sun, Jun Tang, Jianhong Tu, Jianqiang Wan, Peng Wang, Pengfei Wang, Qiuyue Wang, Yuxuan Wang, Tianbao Xie, Yiheng Xu, Haiyang Xu, Jin Xu, Zhibo Yang, Mingkun Yang, Jianxin Yang, An Yang, Bowen Yu, Fei Zhang, Hang Zhang, Xi Zhang, Bo Zheng, Humen Zhong, Jingren Zhou, Fan Zhou, Jing Zhou, Yuanzhi Zhu, and Ke Zhu. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025. URL https://arxiv.org/abs/2511.21631.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Keunho Jang, and Zheng Hui. Windows Agent Arena: Evaluating multi-modal OS agents at scale. In *ICML*, pp. 4874–4910, 2025. URL https://proceedings.mlr.press/v267/bonatti25a.html.

Bytedance Seed. Seed1.8 Model Card: Towards generalized real-world agency. *arXiv preprint arXiv:2603.20633*, 2026. URL https://arxiv.org/abs/2603.20633.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *NeurIPS*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/b1041e52d3be19f0a9bc491657488e4a-Abstract-Datasets_and_Benchmarks_Track.html.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021. URL https://arxiv.org/abs/2110.14168.

Tianyu Fu, Anyang Su, Chenxu Zhao, Hanning Wang, Minghui Wu, Zhe Yu, Fei Hu, Mingjia Shi, Wei Dong, Jiayao Wang, Yuyang Chen, Ruiyang Yu, Siran Peng, Menglin Li, Nan Huang, Haitian Wei, Jiawei Yu, Yi Xin, Xilin Zhao, Kai Gu, Ping Jiang, Sifan Zhou, and Shuo Wang. Mano technical report. *arXiv preprint arXiv:2509.17336*, 2025. URL https://arxiv.org/abs/2509.17336.

Gonzalo Gonzalez-Pumariega, Vincent Tu, Chih-Lun Lee, Jiachen Yang, Ang Li, and Xin Eric Wang. Scaling agents for computer use. *TMLR*, 2026. URL https://openreview.net/forum?id=eve4jBYa8D.

Google DeepMind. Gemini 3 Pro Model Card. https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf, 2025.

Google DeepMind. Gemini 3.1 Pro Model Card. https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-1-Pro-Model-Card.pdf, 2026.

HCIII Team, Lenovo. Muscle-Mem-Agent. https://github.com/wadang/muscle-mem-agent, 2026.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and Jie Tang. CogAgent: A visual language model for GUI agents. In *CVPR*, 2024. URL https://openaccess.thecvf.com/content/CVPR2024/html/Hong_CogAgent_A_Visual_Language_Model_for_GUI_Agents_CVPR_2024_paper.html.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. In *NeurIPS*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/7cc1005ec73cfbaac9fa21192b622507-Abstract-Conference.html.

Kimi Team. Kimi K2.6: Advancing open-source coding. https://www.kimi.ai/blog/kimi-k2-6, 2026.

Tao Li, Gang Li, Zhiwei Deng, Bryan Wang, and Yang Li. A zero-shot language agent for computer control with structured reflection. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 11261–11274, 2023. doi: 10.18653/v1/2023.findings-emnlp.753. URL https://aclanthology.org/2023.findings-emnlp.753/.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *ICLR*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/aca97732e30bcf1303bc22ac3924fd16-Abstract-Conference.html.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Stan Weixian Lei, Lijuan Wang, and Mike Zheng Shou. ShowUI: One vision-language-action model for GUI visual agent. In *CVPR*, pp. 19498–19508, 2025. doi: 10.1109/CVPR52734.2025.01816.

Xiao Liu, Bo Qin, Dongzhu Liang, Guang Dong, Hanyu Lai, Hanchen Zhang, Hanlin Zhao, Iat Long Iong, Jiadai Sun, Jiaqi Wang, Junjie Gao, Junjun Shan, Kangning Liu, Shudan Zhang, Shuntian Yao, Siyi Cheng, Wentao Yao, Wenyi Zhao, Xinghan Liu, Xinyi Liu, Xinying Chen, Xinyue Yang, Yang Yang, Yifan Xu, Yu Yang, Yujia Wang, Yulin Xu, Zehan Qi, Yuxiao Dong, and Jie Tang. AutoGLM: Autonomous foundation agents for GUIs. *arXiv preprint arXiv:2411.00820*, 2024. URL https://arxiv.org/abs/2411.00820.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *NeurIPS*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

OpenAI. Introducing Operator. https://openai.com/index/introducing-operator, 2025.

Jiayi Pan, Yichi Zhang, Nicholas Tomlin, Yifei Zhou, Sergey Levine, and Alane Suhr. Autonomous evaluation and refinement of digital agents. In *COLM*, 2024. URL https://openreview.net/forum?id=NPAQ6FKSmK.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, Xiaojun Xiao, Kai Cai, Chuang Li, Yaowei Zheng, Chaolin Jin, Chen Li, Xiao Zhou, Minchao Wang, Haoli Chen, Zhaojian Li, Haihua Yang, Haifeng Liu, Feng Lin, Tao Peng, Xin Liu, and Guang Shi. UI-TARS: Pioneering automated GUI interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025. URL https://arxiv.org/abs/2501.12326.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, Ran Xu, and Caiming Xiong. CoAct-1: Computer-using multi-agent system with coding actions. In *ICLR*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/ce65173b994cf7c925c71b482ee14a8d-Abstract-Conference.html.

UiPath. UiPath Screen Agent. https://github.com/xlang-ai/OSWorld/tree/main/mm_agents/uipath, 2025.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, Wanjun Zhong, Yining Ye, Yujia Qin, Yuwen Xiong, Yuxin Song, Zhiyong Wu, Aoyan Li, Bo Li, Chen Dun, Chong Liu, Daoguang Zan, Fuxing Leng, Hanbin Wang, Hao Yu, Haobin Chen, Hongyi Guo, Jing Su, Jingjia Huang, Kai Shen, Kaiyu Shi, Lin Yan, Peiyao Zhao, Pengfei Liu, Qinghao Ye, Renjie Zheng, Shulin Xin, Wayne Xin Zhao, Wen Heng, Wenhao Huang, Wenqian Wang, Xiaobo Qin, Yi Lin, Youbin Wu, Zehui Chen, Zihao Wang, Baoquan Zhong, Xinchun Zhang, Xujing Li, Yuanfan Li, Zhongkai Zhao, Chengquan Jiang, Faming Wu, Haotian Zhou, Jinlin Pang, Li Han, Qi Liu, Qianli Ma, Siyao Liu, Songhua Cai, Wenqi Fu, Xin Liu, Yaohui Wang, Zhi Zhang, Bo Zhou, Guoliang Li, Jiajun Shi, Jiale Yang, Jie Tang, Li Li, Qihua Han, Taoran Lu, Woyu Lin, Xiaokang Tong, Xinyao Li, Yichi Zhang, Yu Miao, Zhengxuan Jiang, Zili Li, Ziyuan Zhao, Chenxin Li, Dehua Ma, Feng Lin, Ge Zhang, Haihua Yang, Hangyu Guo, Hongda Zhu, Jiaheng Liu, Junda Du, Kai Cai, Kuanye Li, Lichen Yuan, Meilan Han, Minchao Wang, Shuyue Guo, Tianhao Cheng, Xiaobo Ma, Xiaojun Xiao, Xiaolong Huang, Xinjie Chen, Yidi Du, Yilin Chen, Yiwen Wang, Zhaojian Li, Zhenzhu Yang, Zhiyuan Zeng, Chaolin Jin, Chen Li, Hao Chen, Haoli Chen, Jian Chen, Qinghao Zhao, and Guang Shi. UI-TARS-2 technical report: Advancing GUI agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*, 2025a. URL https://arxiv.org/abs/2509.02544.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *ICML*, 2024. URL https://proceedings.mlr.press/v235/wang24h.html.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, Zhennan Shen, Zhuokai Li, Ryan Li, Xiaochuan Li, Junda Chen, Boyuan Zheng, Peihang Li, Fangyu Lei, Ruisheng Cao, Yeqiao Fu, Dongchan Shin, Martin Shin, Jiarui Hu, Yuyan Wang, Jixuan Chen, Yuxiao Ye, Danyang Zhang, Yipu Wang, Heng Wang, Diyi Yang, Victor Zhong, Y. Charles, Zhilin Yang, and Tao Yu. OpenCUA: Open foundations for computer-use agents. In *NeurIPS*, 2025b. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/cc7ae529e945226b0d52ea4ac478c4f3-Abstract-Conference.html.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, and Yu Qiao. OS-ATLAS: Foundation action model for generalist GUI agents. In *ICLR*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/0faa4bc5f522076947a030273629d4fe-Abstract-Conference.html.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *NeurIPS*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-Datasets_and_Benchmarks_Track.html.

Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous GUI interaction. In *ICML*, pp. 69772–69805, 2025. URL https://proceedings.mlr.press/v267/xu25ae.html.

Taofeng Xue, Chong Peng, Mianqiu Huang, Linsen Guo, Tiancheng Han, Haozhe Wang, Jianing Wang, Xiaocheng Zhang, Xin Yang, Dengchang Zhao, Jinrui Ding, Xiandi Ma, Yuchen Xie, Peng Pei, Xunliang Cai, and Xipeng Qiu. EvoCUA: Evolving computer use agents via learning from scalable synthetic experience. *arXiv preprint arXiv:2601.15876*, 2026. URL https://arxiv.org/abs/2601.15876.

Bowen Yang, Kaiming Jin, Zhenyu Wu, Zhaoyang Liu, Qiushi Sun, Zehao Li, Jingjing Xie, Zhoumianze Liu, Fangzhi Xu, Kanzhi Cheng, Yian Wang, Qingyun Li, Yu Qiao, Zun Wang, and Zichen Ding. OS-Symphony: A holistic framework for robust and generalist computer-using agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22300–22330, 2026a. doi: 10.18653/v1/2026.acl-long.1021. URL https://aclanthology.org/2026.acl-long.1021/.

Pei Yang, Hai Ci, and Mike Zheng Shou. macOSWorld: A multilingual interactive benchmark for GUI agents. In *NeurIPS*, 2025a. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/c276c3303c0723c83a43b95a44a1fcbf-Abstract-Conference.html.

Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, Ran Xu, Liyuan Pan, Caiming Xiong, and Junnan Li. GTA1: GUI test-time scaling agent. In *ICLR*, 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/4fb7d5059cdbe2db2b62dcfe14caa658-Abstract-Conference.html.

Yuhao Yang, Yue Wang, Dongxu Li, Ziyang Luo, Bei Chen, Chao Huang, and Junnan Li. Aria-UI: Visual grounding for GUI instructions. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 22418–22433, 2025b. doi: 10.18653/v1/2025.findings-acl.1152. URL https://aclanthology.org/2025.findings-acl.1152/.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *NeurIPS*, 2023a. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *ICLR*, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Guyue Liu, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. Large language model-brained GUI agents: A survey. *TMLR*, 2025a. URL https://openreview.net/forum?id=xChvYjvXTp.

Chaoyun Zhang, Liqun Li, Shilin He, Xu Zhang, Bo Qiao, Si Qin, Minghua Ma, Yu Kang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. UFO: A UI-focused agent for windows OS interaction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 597–622, 2025b. doi: 10.18653/v1/2025.naacl-long.26. URL https://aclanthology.org/2025.naacl-long.26/.

Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, and Steven Hoi. MAI-UI technical report: Real-world centric foundation GUI agents. *arXiv preprint arXiv:2512.22047*, 2025. URL https://arxiv.org/abs/2512.22047.

A APPENDIX

This supplementary material provides more implementation details, prompt templates, the full action space, and additional ablation results for GUI-Pro Agent. We organize the content as follows:

- Section A.1: Controlled component and transfer studies, including the complete 60-task manifest in Section A.1.1.
- Section A.2: Composition of the fixed 60-task subset.
- Section A.3: Inference cost and tool attribution.
- Section A.4: Broader impact and limitations.
- Section A.5: Implementation details including model configurations and benchmark categories.
- Section A.6: The full action space exposed to the Manager Agent.
- Section A.7: The system prompts for different agent roles.
- Section A.8: Additional ablation results.

A.1 CONTROLLED COMPONENT AND TRANSFER STUDIES

Setup. We supplement the main evaluation with component, transfer, and usage studies on a fixed, non-random 60-task subset selected from the same 361-task OSWorld-Verified evaluation set used for the main experiments. The subset was fixed before the controlled experiments and selected to approximately preserve application-level coverage, without reference to the outcomes of the evaluated module configurations. Its composition is given in Table 5, and the complete task manifest is provided in Appendix A.1.1. All subset configurations use the same task IDs, Gemini 3 Flash for the Manager and Reflection backbone, Seed 1.8 grounding, the same GUI action space, and a 100-step Manager budget. We evaluate this fixed subset only with the 100-step configuration. Every subset score in the CoAct-1 transfer experiment (Figure 4), and usage analysis (Table 6) is obtained under this budget, rather than averaged across step budgets. No 15- or 50-step configurations are evaluated for these subset studies. In these additional studies, the Coder and Searcher use Gemini 3.1 Pro when present, which differs from the main-suite Gemini 3 Flash configuration in Table 7. The base scaffold (w/ coder + grounder) retains standard coding and grounding access; the three experimental factors are completion verification, loop control, and search. We report aggregate scores as subset point estimates, without confidence intervals or significance claims. All differences below are calculated by subtracting the displayed percentage scores and are denoted by %.

Recovery thresholds and scope of generalization. The same two-attempt action trigger and three-occurrence screen trigger are used across tasks; they are not tuned to individual OSWorld scenarios. On WAA, removing the same Loop Breaker design reduces scores by 7.8% at 50 steps (60.4% to 52.6%) and by 5.2% at 100 steps (61.0% to 55.8%). Together with the CoAct-1 transfer, this supports the usefulness of the recovery design across the tested platforms and scaffolds. It does not establish threshold optimality or robustness to arbitrary distributions: we have not performed a threshold-sensitivity sweep, and the negligible 0.04% Sonnet 4.6 gain at 100 steps illustrates that explicit recovery is most useful when the agent is prone to loops.

A.1.1 TASK IDS AND DESCRIPTIONS

Table 4 lists all 60 task IDs used in the fixed subset, grouped by application domain, with concise English descriptions of the tasks. The domain counts are Chrome (8), GIMP (4), LibreOffice Calc (8), LibreOffice Impress (8), LibreOffice Writer (4), Multi-app (15), OS (4), Thunderbird (3), VLC (3), and VS Code (3). The coverage percentages are reported in Table 5. A dagger (†) preserves the five tasks flagged in the subset specification for nonexistent or unsupported capabilities, or restrictive tool constraints. This annotation describes the task specification, not an observed success or failure of the evaluated agent.

Table 4: Task manifest for the fixed 60-task OSWorld-Verified subset. Descriptions summarize the task specifications; † marks the five flagged cases described above.

Task ID	Task description
Chrome (8)	
bb5e4c0d-f964-439c-97b6-bdb9747de3f4	Set Bing as the default search engine.
af630914-714e-4a24-a7bb-f9af687d3b91	Increase font size in accessibility settings.
99146c54-4f37-4ab8-9327-5f3291665e1e	Automatically clear browsing data on exit.
6766f2b8-8a72-417f-a9e5-56fcaa735837	Extract and locally install an extension.
3720f614-37fd-4d04-8a6b-76f54f8c222d	† Change to a nonexistent language.
fc6d8143-9452-4171-9459-7f515143419a	Look up flights through live web interaction.
7f52cab9-535c-4835-ac8c-391ee64dc930	Search and filter results in Google Shopping.
f0b971a1-6831-4b9b-a50e-22a6e47f45ba	Retrieve a Super Bowl score from the web.
GIMP (4)	
7a4deb26-d57d-4ea9-9a73-630f66a7b568	Reduce brightness using a filter.
2a729ded-3296-423d-aec4-7dd55ed5fbb3	Make the background transparent using a selection.
d16c99dc-2a1e-46f2-b350-d97c86c85c15	Resize a layer.
045bf3ff-9077-4b86-b483-a1040a949cff	† Convert to CMYK mode, which GIMP does not support in the task setting.
LibreOffice Calc (8)	
357ef137-7eeb-4c80-a3bb-0951f26a8aff	Compute work hours times hourly wages with formulas.
1954cced-e748-45c4-9c26-9855b97fbc5e	Create a pivot table.
12382c62-0cd1-4bf2-bdc8-1d20bf9b2371	Create a column chart.
37608790-6147-45d0-9f20-1137bb35703d	Split a name column for data cleaning.
3aaa4e37-dc91-482e-99af-132a612d40f3	Export a CSV file.
0bf05a7d-b28b-44d2-955a-50b41e24012a	Pad IDs with leading zeros.
4188d3a4-077d-46b7-9c86-23e1a036f6c1	Freeze the first row.
8b1ce5f2-59d2-4dcc-b0b0-666a714b9a14	Highlight weekends with conditional formatting.
LibreOffice Impress (8)	
5d901039-a89c-4bfb-967b-bf66f4df075e	Turn an image into a cover slide (complex editing).
3b27600c-3668-4abd-8f84-7bcdebbccbdb	Set a blue background on all slides.
39be0d19-634d-4475-8768-09c130f5425d	Insert a table.
9ec204e4-f0a3-42f8-8458-b772a6797cab	Duplicate slides and interleave them (difficult editing).
986fc832-6af2-417c-8845-9272b3a1528b	Underline and change the font of body text only.
0a211154-fda0-48d0-9274-eaac4ce5486d	Set yellow backgrounds only on slides containing photos of people.
455d3c66-7dc6-4537-a39a-36d3e9119df7	Export slides as PNG images.
0f84bef9-9790-432e-92b7-eece357603fb	Disable dual-screen presentation mode.
LibreOffice Writer (4)	
0e47de2a-32e0-456c-a366-8c607ef7a9d2	Add a page number on the left side of the footer.
0b17a146-2934-46c7-8727-73ff6b6483e8	Format the numeral in H_2O as a subscript.
66399b0d-8fda-4618-95c4-bfc6191617e9	Insert a 7×5 table.
4bcb1253-a636-4df4-8cb0-a35c04dfef31	Export a PDF file.
Multi-app (15)	
2c9fc0de-3ee7-45e1-a5df-c86206ad78b5	Run `git push` in a terminal.
f918266a-b3e0-4914-865d-4faa564f1aef	Run a Python script and obtain its output.
f7dfbef3-7697-431c-883a-db8583a4e4f9	Batch-convert DOC files to PDF from the command line.
81c425f5-78f3-4771-afd6-3d2973825947	Transfer data from Calc to Impress.
58565672-7bfe-48ab-b828-db349231de6b	Open an AWS billing link from an email.

Continued on next page

Table 4 (continued)

Task ID	Task description
c2751594-0cd5-4088-be1b-b5f2f9ec97c4	Export an image from an email attachment and edit it in GIMP.
47f7c0ce-a5fb-4100-a5e6-65cd0e7429e5	Extract a video frame and set it as the wallpaper.
2fe4b718-3bd7-46ec-bdce-b184f5653624	Create an animation using VLC and GIMP.
67890eb6-6ce5-4c00-9e3d-fb4972699b06	Find ACL best papers and enter them in a spreadsheet.
6f4073b8-d8ea-4ade-8a18-c5d1d5d5aa9a	Compile statistics on conference host cities.
bc2b57f3-686d-4ec9-87ce-edf850b7e442	Complete a data-analysis assignment following `reminder.docx`.
869de13e-bef9-4b91-ba51-f6708c40b096	Sort and organize desktop files.
26150609-0da3-4a7d-8868-0faf9c5f01bb	Complete a Snake game coding task.
2373b66a-092d-44cb-bfd7-82e86e7a3b4d	Monitor system resources with `sar`.
6d72aad6-187a-4392-a4c4-ed87269c51cf	† Convert a presentation to video using only Impress.
OS (4)	
94d95f96-9699-4208-98ba-3c3119edf9c2	Install Spotify.
f9be0997-4b7c-45c5-b05c-4612b44a6118	Turn off Do Not Disturb in notification settings.
5812b315-e7bd-4265-b51f-863c02174c28	Create a restricted SSH user (advanced administration).
c288e301-e626-4b98-a1ab-159dcb162af5	† Set the default interpreter to nonexistent Python 4.
Thunderbird (3)	
15c3b339-88f7-4a86-ab16-e71c58dcb01e	Configure an Outlook account.
5203d847-2572-4150-912a-03f062254390	Create a local folder and automatic filtering rules.
d38192b0-17dc-4e1d-99c3-786d0117de77	Compose an email and add an attachment.
VLC (3)	
8f080098-ddb1-424c-b438-4e96e5e4786e	Convert a video to MP3.
aa4b5023-aef6-4ed9-bdc9-705f59ab9ad6	Rotate a video and save it.
5ac2891a-eacd-4954-b339-98abba077adb	Disable automatic exit after playback.
VS Code (3)	
eabc805a-bfcf-4460-b250-ac92135819f6	Install the Python extension.
930fdb3b-11a8-46fe-9bac-577332e2640e	Customize keyboard shortcuts (advanced configuration).
7c4cc09e-7a92-40dd-8338-b2286535c4ed	† Switch the interface to Arabic without extensions.

A.2 Composition of the Controlled Subset

Table 5 reports the fixed, non-random subset used in Appendices A.1 and A.3. The subset is selected from the same 361-task evaluation set used for the main results and the same-backbone Sonnet 4.6 comparison. Coverage is calculated against this set, which excludes the eight Google Drive tasks. The ten groups below are application-level groups, rather than the five aggregated reporting categories in Table 1.

Table 5: Application-domain coverage of the fixed 60-task subset within our 361-task OSWorld evaluation set.

Domain group	Subset tasks	Evaluation-set tasks	Coverage (%)
Chrome	8	46	17.4
GIMP	4	26	15.4
LibreOffice Calc	8	47	17.0
LibreOffice Impress	8	47	17.0
LibreOffice Writer	4	23	17.4
Multi-app	15	93	16.1
OS	4	24	16.7
Thunderbird	3	15	20.0
VLC	3	17	17.6
VS Code	3	23	13.0
Total	60	361	16.6

A.3 Inference Cost and Tool Attribution

Table 6 reports mean environment actions and tracked usage per task on the same 60-task subset, using only the 100-step Manager-budget configuration. The action counts are observed means, not separate evaluation budgets. The model-call accounting includes executor, Reflection, Coder, Gate, and Judge records where tracked; these are usage-log categories and do not imply that every Gate check is an independent model call. The $\geq$ markers are retained because token/cost coverage, and call coverage for some variants, is incomplete. Tracked cost is therefore a lower bound on API expenditure rather than a complete end-to-end cost estimate. Separate input/output-token totals, complete provider/tool charges, and infrastructure costs are not available in these measurements.

Among the standalone additions, Completeness-Aware Planner has the largest tracked overhead and increases mean environment actions from 16.80 to 27.32. The full configuration achieves a higher score with fewer tracked calls than the CAP-only configuration, indicating that overhead is not additive across modules because the modules change the execution trajectory. The reported cost lower bounds do not support a ranking of complete API expenditure. Accordingly, performance under a fixed Manager-action budget should not be interpreted as equivalent model-call, latency, or monetary efficiency.

A.4 Broader Impact and Limitations

Models, tools, and inference overhead All evaluated Manager backbones are proprietary. The additional Gemini 3 Flash studies also use proprietary Gemini 3.1 Pro Coder and Searcher models

Table 6: Mean usage per task on the fixed 60-task subset with a 100-step Manager budget. All scores use this budget; calls and observed actions are different quantities. Dollar values are tracked API cost in USD; $\geq$ denotes a lower bound from incomplete usage accounting.

Configuration	Score (%)	Actions	Model calls	Tokens	Cost ($)
Base scaffold	65.05	16.80	28.40	$\geq$ 257,618	$\geq$ 0.137
+ Search Agent	68.86	12.70	$\geq$ 32.55	$\geq$ 343,640	$\geq$ 0.227
+ Completeness-Aware Planner	67.19	27.32	$\geq$ 89.13	$\geq$ 901,936	$\geq$ 0.717
+ Loop Breaker	69.00	14.28	$\geq$ 41.52	$\geq$ 422,074	$\geq$ 0.323
Full	**77.15**	19.67	47.39	$\geq$ 579,719	$\geq$ 0.316

when present, so they do not establish generality to open models. Model-version changes, API availability, and changing external search results limit exact reproducibility. Coding and grounding are adopted tools, and their invocation counts do not replace independent access-matched ablations.

Evaluation and remaining verification failures The fixed 60-task subset is non-random; it supplies controlled point estimates, not population-level or statistically significant claims. Loop-control thresholds are hand-designed and have not been evaluated in a sensitivity sweep. The same-backbone and transfer experiments reduce confounding by Manager capability, but cover only the reported models and scaffolds. False completion remains dominant among failures, and visually plausible states can still conceal incorrect file contents, pending writes, or evaluator-relevant state.

Deployment safeguards Our results come from benchmark virtual machines, not unrestricted user desktops. Incorrect or overconfident actions can modify, delete, send, or expose information before a stopping error is detected. Practical deployment requires sandboxing, scoped permissions, confirmation before irreversible or sensitive operations, audit logs, command allowlists, and rollback mechanisms. Completion verification is a reliability mechanism and does not itself provide these security or authorization controls.

Limitations and Future directions Our current system adopts a relatively simple memory and planning architecture without long-horizon task decomposition or cross-task knowledge transfer. More sophisticated memory mechanisms may further improve performance on complex multi-step tasks. On the other hand, GUI-Pro Agent generates high-quality, verified execution trajectories as a byproduct of its verification and loop recovery. A promising direction is leveraging these trajectories to train unified end-to-end multimodal models, distilling agentic reasoning into a single model that acts directly from pixels—bridging the reliability of agentic frameworks with the efficiency of end-to-end approaches.

A.5 Implementation Details

A.5.1 Configurations.

Table 7 summarizes the model assignments used for each component across our experimental configurations. For the main OSWorld evaluation, we use six configurations. Three use Claude Opus 4.5, Claude Sonnet 4.6, or Gemini 3 Flash as the shared backbone for the Manager Agent, Reflection Agent, Completeness Model Judge, and Coding Agent, with Gemini 3 Pro (Google DeepMind, 2025) as the Search Agent. We additionally evaluate an Opus 4.5 variant that keeps the same backbone and Search Agent but replaces the grounding model with MAI-UI, an Opus 4.6 (Anthropic, 2026a) configuration that uses Gemini 3.1 Pro (Google DeepMind, 2026) as the Search Agent, and a Gemini 3.1 Pro configuration that uses Gemini 3.1 Pro across all components except grounding. On WindowsAgentArena, we evaluate Gemini 3 Flash as the Manager backbone.

Table 7: Model assignments for the main-suite configurations. The additional 60-task studies instead use Gemini 3.1 Pro for the Coder and Searcher when present.

Component	Model	Provider
Manager Agent	Claude Opus 4.5 / Claude Opus 4.6 / Sonnet 4.6 / Gemini 3 Flash / Gemini 3.1 Pro	Anthropic / Google
Reflection Agent (Loop Breaker)	Same as the Manager Agent	Anthropic / Google
Completeness Model Judge	Same as Manager Agent	Anthropic / Google
Search Agent	Gemini 3 Pro for Opus 4.5 / Opus 4.5 + MAI-UI / Sonnet 4.6 / Gemini 3 Flash; Gemini 3.1 Pro for Opus 4.6 / Gemini 3.1 Pro (w/ search grounding)	Google
Coding Agent	Same as the Manager Agent	Anthropic / Google
Grounding Agent	Seed 1.8 for all configurations except MAI-UI for Opus 4.5 + MAI-UI	ByteDance / –

A.5.2 Hyperparameters.

All agents operate in iterative planning mode (no hierarchical subtask decomposition). The per-task step budget is set to 15, 50, or 100 actions depending on the experiment. The Completeness Model

Judge within Completeness-Aware Planner uses temperature $T{=}0.2$ for conservative judgment; all other components use $T{=}1.0$. The Coding Agent has an independent step budget of 20 code execution steps. The Search Agent performs a single query.

A.5.3 Action-Outcome Verification Rules.

Table 8 lists the rule-based checks Completeness-Aware Planner applies to the expected visual outcome of each action type.

Table 8: Verification rules by action type in Completeness-Aware Planner.

Action Type	**Expected Verification**
Click button/menu	New UI element visible (dialog, tab, highlight)
Toggle setting	State label changed ("Enable" $\rightarrow$ "Disable")
Type text	Field contains typed text; cursor moved
Export/save	New file in folder, success toast, or title bar change
No visible change	`wait(1)` before re-check; do not repeat immediately

A.5.4 Infrastructure.

For OSWorld, experiments run on the officially released Ubuntu Docker containers hosted on Amazon Web Services. For WindowsAgentArena, we use the official Windows environments infrastructure and deploy the environment snapshot on Amazon Web Services virtual machines. Screenshots are captured at $1920{\times}1080$ resolution and served as the primary observation modality.

A.5.5 Reported Categories.

On OSWorld, we follow the official evaluation setting, where tasks are grouped into the following categories: OS, Office (LibreOffice Calc, Impress, Writer), Daily (Chrome, VLC Player, Thunderbird), Professional (VS Code and GIMP), and Workflow (tasks involving interactions across multiple applications). On WindowsAgentArena, tasks are organized into several functional categories following OS-Symphony. Office includes LibreOffice Writer and LibreOffice Calc tasks; Web (Web Browsing) includes Edge and Chrome tasks; Sys. (Windows System) includes File Explorer and Settings tasks; Code includes VSCode tasks; Media (Media & Video) includes VLC tasks; and Util. (Windows Utilities) includes Notepad, Clock, Paint, and WindowsCalc tasks.

A.6 Action Space

Table 9 lists the executable actions exposed to the Manager Agent in the full-system configuration. The table shows the Linux/OSWorld setting with the Coding Agent and Search Agent enabled. The `save_to_knowledge` action (for memory) is excluded in this configuration.

A.7 System Prompts for Agents

A.7.1 Reflection Agent Prompt (Loop Breaker Tier 3)

The Reflection Agent serves as the external model judge in the Loop Breaker's Tier 3 (Section 3.3). It is invoked after every Manager action to analyze the trajectory and produce structured signals. When the Strategy signal is SWITCH, the Loop Breaker injects a hard directive forcing the Manager to change its approach. The full prompt is shown below.

```
You are a reflection agent designed to assist in task execution by
reflecting on the trajectory of a task and providing feedback for
what the next step should be.
You have access to the Task Description and the Current Trajectory
of another computer agent. The Current Trajectory is a sequence of a
desktop image, chain-of-thought reasoning, and a desktop action for
each time step. The last image is the screen's display after the last
action.
Your task is to generate a reflection. Your generated reflection must
fall under one of the two cases listed below:
```

Table 9: Action space exposed to the Manager Agent in the full-system configuration.

Action	Type	Description
`agent.click`	GUI	Clicks a grounded UI element with optional repeated clicks, mouse-button choice, and modifier keys.
`agent.double_click`	GUI	Double-clicks a grounded UI element.
`agent.type`	GUI	Focuses an element if needed, types text, optionally overwrites existing text, and can submit with Enter.
`agent.drag_and_drop`	GUI	Drags from one grounded element to another grounded element.
`agent.highlight_text_span`	GUI	Selects text between a starting phrase and an ending phrase.
`agent.scroll`	GUI	Scrolls vertically or horizontally over a grounded target region.
`agent.open`	Navigation	Opens an application or file.
`agent.switch_applications`	Navigation	Switches to an already opened application.
`agent.hotkey`	Keyboard	Sends a key chord such as `Ctrl+S`.
`agent.hold_and_press`	Keyboard	Holds one set of keys while pressing another key sequence.
`agent.call_code_agent`	Tool	Delegates a task or subtask to the Coding Agent, which runs Python/Bash and returns a structured report.
`agent.call_search_agent`	Tool	Calls the Search Agent to retrieve a step-by-step tutorial for a specific GUI procedure.
`agent.set_cell_values`	Tool	Performs spreadsheet cell edits programmatically (Linux-specific).
`agent.wait`	Terminal	Waits for the UI to update before the next decision.
`agent.done`	Terminal	Terminates the task successfully.
`agent.fail`	Terminal	Terminates the task as infeasible or blocked.

```
Case 1. The trajectory is not going according to plan. This is often
due to the latest action not being executed correctly, or a cycle of
actions being continually repeated with no progress being made. In this
case, explicitly highlight why the current trajectory is incorrect, and
encourage the computer agent to try a new action. However, DO NOT
encourage a specific action in particular.

Case 2. The trajectory is going according to plan. In this case, simply
tell the agent to continue proceeding as planned. DO NOT encourage a
specific action in particular.

To be successful, you must follow the rules below:
- DO NOT suggest any specific future plans or actions. Your only goal
  is to provide a reflection, not an actual plan or action.
- Your reflection MUST be evidence-based and outcome-focused, not
  procedural.
- Any response that falls under Case 1 should explain why the
  trajectory is not going according to plan. You should especially look
  out for cycles of actions that are continually repeated with no
  progress.
- Any response that falls under Case 2 should be concise, since you
  just need to affirm the agent to continue with the current
  trajectory.

Required fields (write them as labeled lines):
- Progress signal: What visibly changed since the prior step?
  (or "no visible change")
- Outcome signal: Did we achieve the intended subgoal (not just click
  something)? What is the evidence?
- Loop signal: Are we repeating the same interaction/attempt 2+ times
  or seeing the same screen 3 times? (yes/no + evidence)
- Feasibility signal: Is the task still feasible?
  (feasible / uncertain / impossible + evidence)
- Termination signal: DONE / FAIL / CONTINUE (evidence-based). Use
  DONE only when ALL task success criteria have SPECIFIC VISIBLE
  EVIDENCE. Use FAIL when the task is demonstrably infeasible. Use
  CONTINUE otherwise.
- Strategy signal: KEEP / SWITCH (and brief reason if SWITCH). If
  Loop signal = yes OR Feasibility signal = uncertain/impossible,
  Strategy signal should usually be SWITCH.
- Verdict: Case 1 or Case 2
```

A.7.2 Completeness-Aware Planner: Model Judge Prompt

The Completeness Model Judge within Completeness-Aware Planner is invoked whenever the Manager's Completion Gate outputs DONE (Section 3.2). It receives the task instruction, current screenshot, and recent trajectory, and produces a binary accept/reject decision. The prompt enforces conservative verification: it requires direct visual evidence for every criterion and prefers false negatives over false positives. The full prompt is shown below.

```
You are an extremely strict task completion verifier for desktop
automation. Your job is to decide whether the task is ALREADY fully
completed right now.

Decision rules (be conservative):
- Mark completion ONLY if every requirement in the instruction is
  satisfied AND there is direct, unambiguous evidence in the current
  screenshot and/or provided recent history.
- Do NOT assume hidden state or intent. If something could be
  incomplete, mark incomplete.
- For side effects (sending/submitting/saving/downloading/creating/
  deleting/installing/moving), require visible confirmation (success
  toast, item present in the right place, sent email visible, saved
  file visible, etc.).
- Require a stable UI state when possible (no open dropdowns/menus,
  no blocking modal, no loading indicator suggesting the action is
  still in progress).
- Prefer false negatives over false positives.

Semantic evidence rules (prevent false positives from workarounds):
- If the task involves a file format requirement (e.g., "export as
  SVG", "save as PDF"), require visible evidence of the correct file
  extension AND an app dialog/status/file-browser entry confirming
  that format. If not visible, mark incomplete.
- If the task involves a mode or state (e.g., "CMYK mode", "dark
  theme enabled"), require explicit UI indicators (mode label in
  title/status bar, toggle state). Do NOT infer from visual appearance
  alone.
- If a workaround was used instead of the exact requested method,
  verify it produces a semantically equivalent result. If equivalence
  is uncertain, mark incomplete.

Exact-value verification (CRITICAL):
- If the task specifies an EXACT value (hex color, font size, config
  key, URL parameter, file path, specific text string, numeric value),
  you MUST find that EXACT value readable on screen or confirmed in
  the trajectory text.
- "Looks like the right color" does NOT verify "#00FF00". "Looks like
  a small font" does NOT verify "11pt".
- For values set via code agent: check the trajectory for the code
  agent's output confirming the value was written/set correctly.
- For values set via GUI: the value must be readable in an input
  field, status bar, dialog, or properties panel on the current
  screenshot.
- If the exact value cannot be confirmed, mark INCOMPLETE.

Trajectory cross-check (MANDATORY):
- Review the trajectory text for actual actions taken. Verify that the
  actions match what the task requested.
- If the code agent ran commands, check the command output in the
  trajectory for success/failure indicators and correct values.
- If the trajectory shows errors, failed commands, or values different
  from what was requested, mark INCOMPLETE regardless of how the
  screenshot looks.

Output:
- Return ONLY a JSON object with:
  {"complete": true/false, "reason": "<short>",
   "missing_steps": "<what is still needed, or empty if complete>"}
```

The verifier agent also applies conservative post-processing: if the model claims completion but lists missing steps or uses uncertainty phrases (*e.g.*, "not sure", "unclear", "cannot verify"), the verdict is automatically overridden to incomplete.

A.7.3 MANAGER AGENT SYSTEM PROMPT

Below we report the full prompt for the Manager agent under the default Linux configuration: (1) a termination contract that mandates explicit terminal actions, (2) the Completion Gate protocol enforcing per-step self-verification, (3) micro-verification rules for post-action checks, (4) the Loop Breaker rules (Tiers 1–2), (5) feasibility management guidelines, (6) agent usage guidelines describing when to invoke the Coding and Search Agents, and (7) the output format specification.

```
You are an expert in graphical user interfaces and Python code. You are
responsible for executing the task: `TASK_DESCRIPTION`.
You are working in linux.
```

```
# GUIDELINES

## Termination Contract (NON-NEGOTIABLE)
Every trajectory MUST terminate with exactly one terminal API call:
- `agent.done()` when ALL success criteria are visibly satisfied.
- `agent.fail()` when the task is infeasible or permanently blocked.
NEVER "run out" the episode without a terminal call. If uncertain,
perform a verification action, then decide.
If success criteria are satisfied, the very next grounded action MUST
be `agent.done()` -- no extra cleanup steps.

## Completion Gate (run FIRST every step)
Before choosing any action, make a binary decision using only visible
evidence from the current observation and recent history:
1. (Step 1 only, or when the task changes): Write 1-3 success criteria
   as UI-observable statements (button labels, file names in lists,
   toggle states, presence/absence of UI elements, dialog titles). If
   a criterion is not directly observable, rewrite it into an
   observable proxy. If a criterion requires an exact value (hex color,
   font size, config key, file content), tag it as [EXACT CHECK].
2. (Every step): For each criterion, update its status using ONLY
   visible evidence. For [EXACT CHECK] criteria: cite the literal
   value readable on screen or confirmed by code agent output.
3. If ALL criteria are satisfied AND the UI is stable, the next
   grounded action MUST be `agent.done()`.
4. If the task is impossible based on evidence, the next grounded
   action MUST be `agent.fail()`.
5. Otherwise, proceed with one concrete next action.

## Micro-Verification (MANDATORY after every UI action)
When verifying the previous action, apply the appropriate check:
- Clicked a button/menu: Verify the expected new UI element is visible.
- Toggled a setting: Verify the toggle state actually changed.
- Typed text: Verify the field contains the typed text.
- Exported/saved: Verify the new file appears or success toast shows.
- No visible change: Do NOT repeat. Wait, then re-check or switch
  strategy.

## Hard Loop Breaker (MANDATORY)
- If the same action type + same target produces no visible change
  twice, the next action MUST be a different modality.
- If the same screen state appears three times in a row with no
  progress, you MUST switch strategy completely. Call `agent.fail()`
  only when separate feasibility evidence establishes infeasibility.
- When the reflection agent returns Strategy signal = SWITCH, you MUST
  change modality/approach on the very next action.

## Feasibility Management
If the instruction depends on uncertain prerequisites:
1. Identify the 1-3 most likely prerequisites.
2. Run a short feasibility probe (first 1-3 steps).
3. Maintain a small evidence trail of what you checked.
4. If after ~6-8 actions there is no positive evidence, call
   `agent.fail()`.

## Agent Usage Guidelines
You have access to: GUI, Code, Search. Choose the correct one.

### GUI Agent
- Use for: All direct UI interactions (clicking, typing, dragging).

### Code Agent
Use code agent ONLY when:
- The task requires bulk edits (>= 20 cells/lines), OR
- The task requires non-trivial computation, OR
- The GUI route is blocked.
Do NOT use code agent when the change is achievable in <= 3 GUI
actions or for charts/graphs/pivot tables/visual layout.

### Search Agent
- Use when you are unsure how to perform a GUI-based task.
- Call with a clear "How to" query for a single specific action.

# Output Format
You are provided with: (1) A screenshot, (2) History of previous
interactions, (3) A text reflection from the Reflection Agent,
(4) Tutorials from the Search Agent.

Your response must include:
(Completion Gate) - success criteria, evidence check, feasibility
status, and decision (DONE / FAIL / CONTINUE).
(Previous action verification) - micro-verification of the last action.
(Screenshot Analysis) - description of current desktop state.
(Next Action) - natural language description of the next action.
(Grounded Action) - a single Python API call, e.g.:
```python
agent.click("The menu button at the top right", 1, "left")
```
```

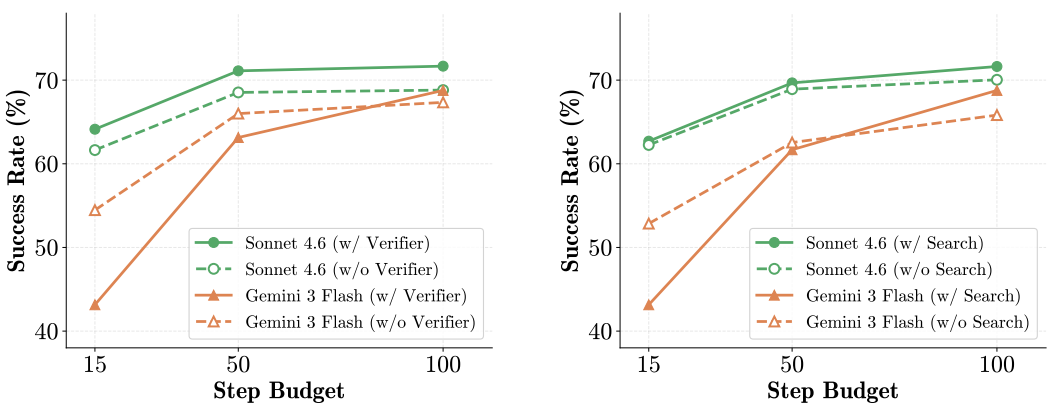

Figure 6: Impact of Completeness-Aware Planner and Search Agent across step budgets on OSWorld. Sonnet 4.6 benefits consistently at all budgets, while Gemini 3 Flash gains only at relaxed budgets: tool calls consume actions that less efficient models cannot afford under tight constraints.

A.8 Additional Ablation Results

This section provides detailed per-budget breakdowns for Completeness-Aware Planner and Loop Breaker, supplementing the aggregate analysis in the main paper.

A.8.1 Stronger Backbones Leverage Tools More Efficiently.

Figure 6 examines how Completeness-Aware Planner and Search Agent interact with the step budget across both Claude Sonnet 4.6 and Gemini 3 Flash. For Sonnet 4.6, both components deliver consistent gains at every budget (*e.g.*, CAP +2.5% at 15 steps, +2.9% at 100 steps), as Sonnet is step-efficient enough to absorb the action overhead of tool invocations. When switching to Gemini 3 Flash, both components help at 100 steps (*e.g.*, Search +3.0%) but hurt under tighter budgets (*e.g.*, CAP −11.3% and Search −9.7% at 15 steps). This occurs because invoking tools inherently consumes action steps. Gemini 3 Flash already needs more raw actions to complete tasks, so the tool overhead crowds out task execution under constrained budgets.

A.8.2 Completeness-Aware Planner: Failure Breakdown across Step Budgets

Table 10 reports the false completion count and DONE accuracy across step budgets for both backbones, with and without Completeness-Aware Planner. *False DONE* is the number (and percentage over all failed cases) of tasks where the agent incorrectly declared success. *DONE Accuracy* is the percentage of all DONE signals that are correct.

Table 10: False completion breakdown across step budgets on OSWorld. **w/**: full system with Completeness-Aware Planner; **w/o**: CAP removed. False DONE reports the count and fraction of failed tasks with false completion. DONE Accuracy reports the fraction of all completion signals that are correct.

		@100 steps		@50 steps		@15 steps	
Backbone	**Metric**	w/	w/o	w/	w/o	w/	w/o
Sonnet 4.6	False DONE	91 (91.9%)	105 (95.5%)	86 (85.1%)	98 (88.3%)	63 (50.0%)	73 (53.7%)
	DONE Acc.	73.5%	69.6%	74.4%	70.9%	78.2%	74.7%
Gemini 3 Flash	False DONE	100 (86.2%)	102 (91.9%)	72 (52.6%)	97 (80.2%)	28 (13.7%)	63 (38.7%)
	DONE Acc.	70.3%	69.8%	74.6%	70.6%	84.3%	75.5%

For Sonnet 4.6, Completeness-Aware Planner consistently reduces the false DONE rate across all budgets, with DONE accuracy improving by 3–4% at every tier (*e.g.*, 69.6%→73.5% at 100 steps). The effect is moderate because Sonnet already produces relatively calibrated completion signals.

For Gemini 3 Flash, the verifier has a much larger effect under tighter budgets. At 50 steps, the false DONE fraction drops sharply from 80.2% to 52.6% (−27.6%), and DONE accuracy rises from 70.6% to 74.6%. At 15 steps the gap is even starker: 38.7%→13.7% (−25.0%). This is because without the verifier, Flash tends to declare completion early when it cannot make progress within the tight budget; the verifier catches these premature claims. At 100 steps, where Flash has enough budget to actually complete more tasks, the gap narrows (91.9%→86.2%).

A.8.3 Loop Breaker: Effectiveness across Step Budgets

Table 11 reports task success rates with and without the Loop Breaker across step budgets for both backbones.

Table 11: Loop Breaker ablation across step budgets on OSWorld. Δ is the difference when adding the Loop Breaker.

Budget	w/ Loop Breaker	w/o Loop Breaker	Δ
Sonnet 4.6			
@100	71.67%	71.63%	+0.04%
@50	71.11%	69.67%	+1.44%
@15	64.13%	62.70%	+1.43%
Gemini 3 Flash			
@100	68.77%	66.95%	+1.82%
@50	63.14%	58.90%	+4.24%
@15	43.15%	49.30%	−6.15%

For Sonnet 4.6, the Loop Breaker provides consistent gains at tighter budgets (+1.4% at both 15 and 50 steps), while the effect at 100 steps is negligible (+0.04%). This aligns with the main paper's finding that Sonnet loops infrequently—at 100 steps it has enough budget to self-recover, so explicit loop breaking adds little; under tighter budgets, even occasional loops waste precious steps that the Loop Breaker reclaims. For Gemini 3 Flash, the Loop Breaker helps substantially at 100 steps (+1.82%) and 50 steps (+4.24%), where it loops frequently and the escalation mechanism redirects the agent toward productive actions. However, at 15 steps the Loop Breaker *hurts* (−6.15%). The Loop Breaker's escalation tiers (modality switch, strategy change) themselves consume actions, and at 15 steps Gemini 3 Flash cannot afford this overhead. When budget is sufficient, the cost of escalation is recouped through recovered trajectories; when budget is tight, the escalation actions crowd out task execution.