Knowing When to STOP, RECOVER, and SEARCH A Modular Framework for GUI Automation

Qijun Han [*1] **Haoqin Tu** [*1] **Zijun Wang** [1] **Haoyue Dai** [2] **Yiyang Zhou** [3] **Nancy Lau** [1] **Alvaro A. Cardenas** [1] **Yuhui Xu** [4] **Ran Xu** [4] **Caiming Xiong** [4] **Zeyu Zheng** [5] **Huaxiu Yao** [3] **Yuyin Zhou** [1] **Cihang Xie** [1]

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. We present **GUI-Pro-Agent**, a modular GUI agentic framework built around three integrated components that guide the system on when to `Stop`, `Recover`, and `Search`. First, a mandatory *Completeness Verifier* enforces UI-observable success criteria and verification at every finish step—with an agent-level verifier that cross-examines completion claims with decision rules, rejecting those lacking direct visual evidence. Second, a mandatory *Loop Breaker* provides multi-tier filtering: switching interaction mode after repeated failures, forcing strategy changes after persistent screen-state recurrence, and binding reflection signals to strategy shifts. Third, an on-demand *Search Agent* searches online for unfamiliar workflows by directly querying a capable LLM with search ability, returning results as plain text. We additionally integrate a Coding Agent for code-intensive actions and a Grounding Agent for precise action grounding, both invoked on demand when required. We evaluate GUI-Pro-Agent across five top-tier backbones, including Opus 4.5, 4.6 and Gemini 3.1 Pro, on two benchmarks with Linux and Windows tasks, achieving top performance on both (**77.5%** on OSWorld and **61.0%** on WindowsAgentArena). Notably, three of the five backbones surpass human performance (i.e., 72.4%) on OSWorld in a single pass. In particular, GUI-Pro-Agent with Sonnet 4.6 at only 15 action steps already surpasses the best published 50-step system. Ablation studies show that all three proposed components consistently improve a strong backbone (e.g., Sonnet 4.6), while a weaker backbone (e.g., Gemini 3 Flash) benefits more from these tools when the step budget is sufficient. Further analysis also shows that the Loop Breaker nearly halves wasted steps for loop-prone models.

1. Introduction

The rapid advancement of multimodal large language models (MLLMs) (Zhang et al., 2024a) has catalyzed a new generation of autonomous Graphical User Interface (GUI) agents (Qin et al., 2025; Agashe et al., 2024; Wang et al., 2025b; Wu et al., 2024; 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., 2024), and related benchmarks (Zhang et al., 2024a; 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. First, agents do not reliably know *when a task is finished*: they routinely declare success prematurely, e.g., after opening a "Save As" dialog before writing the file, or 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., 2025; Cemri et al., 2025). Second, agents fall into *repetitive loops*: cycling through the same failing action without recovery. Moreover, existing anti-looping heuristics operate at a single granularity and cannot escalate across interaction modalities or planning strategies (Yao et al., 2023b; Shinn et al., 2023; Madaan et al., 2023; Li et al., 2023; Kim et al., 2023).

We present **GUI-Pro-Agent**, a modular GUI agent framework that addresses both challenges through three integrated

[*]Equal contribution

Website: https://ucsc-vlaa.github.io/VLAA-GUI
Code: https://github.com/UCSC-VLAA/VLAA-GUI
[1]UC Santa Cruz [2]CMU [3]UNC-Chapel Hill [4]Salesforce [5]UC Berkeley. Correspondence to: Qijun Han <qhan11@ucsc.edu>, Haoqin Tu <htu8@ucsc.edu>.

Published at the Second Workshop on Agents in the Wild: Safety, Security, and Beyond (AIWILD) at ICML 2026. Copyright 2026 by the author(s).

mechanisms—*a Completeness Verifier, a Loop Breaker, and a Search Agent*—to regulate the system in knowing when to `STOP`, `RECOVER` (from repetitive loops), and `SEARCH` online for enhanced performance. To address the early stopping issue, we introduce a mandatory Completeness Verifier: a prompt-level *Completion Gate* that requires the agent to derive UI-observable success criteria and verify them against the current screenshot before every decision, backed by a verifier model that independently cross-examines any completion claim and rejects it under any ambiguity at every finish step. To address repetitive loops, we introduce a mandatory Loop Breaker with three escalation filtering rules: I. switch interaction modality after certain consecutive no-change failures, II. change strategy after a sequence of identical screen states, and III. enforce a mandatory strategy change whenever an external model judge signals a loop. We impose the Loop Breaker after every action step to inform the system when to recover from a potential loop in time.

Beyond the two core challenges, GUI agents frequently stall on unfamiliar application workflows, leading to poor out-of-distribution capabilities. Prior work addresses this via visual browser-based search (Yang et al., 2026), 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, 2025b)), returning plain textual knowledge injected into the Manager's context, which is faster and more reliable. We further integrate a Coding Agent for programmatic edits and a Grounding Agent for agent action grounding following existing works (Gonzalez-Pumariega et al., 2025; Yang et al., 2026; Wang et al., 2024c). We evaluate GUI-Pro-Agent on OSWorld-Verified (Xie et al., 2024) and WindowsAgentArena (WAA) (Bonatti et al., 2024) across five top-tier backbones—Claude Opus 4.6 (Anthropic, 2026), Opus 4.5 (Anthropic, 2025a), Sonnet 4.6 (Anthropic, 2025c), Gemini 3.1 Pro (Google DeepMind, 2026), and Gemini 3 Flash (Google DeepMind, 2025a)—to systematically validate the framework across model families and capability tiers. In Figure 1, our system achieves **77.5%** on OSWorld with Opus 4.6, and three of the five backbones (Opus 4.6 at 77.5%, Opus 4.5 at 74.9%, Gemini 3.1 Pro at 72.5%) surpass human performance (i.e. 72.4%) in a single pass—making GUI-Pro-Agent the first framework to do so. All five backbones outperform prior SOTA (e.g., Agent S3 at 67.5%), and GUI-Pro-Agent w/ Sonnet 4.6 at only 15 action steps already surpasses the best published 50-step system with one-third the budget. On the other hand, GUI-Pro-Agent generalizes well to WAA, reaching 61.0% and outperforming the strongest baselines by over 4%.

Beyond the main results, we conduct three lines of ablation analysis. I. We perform controlled incremental ablations on OSWorld and WAA to isolate each component's marginal contribution, finding that all three components consistently improve a strong backbone (e.g., up to +3.1% each with Sonnet 4.6), while a weaker backbone benefits more at more relaxed budgets, since tool use consumes actions that less efficient models cannot afford under tight budgets. II. We analyze the false completion behavior, revealing that over 86% of failures involve the agent incorrectly believing it has succeeded, and that the Completeness Verifier reduces the overall false completion ratio by up to 3.9%. III. We analyze the repetitive-loop behavior, showing that the Loop Breaker Agent nearly halves wasted steps (4.9%→2.8%) for loop-prone backbones.

2. Related Work

2.1. GUI Agent Benchmarks.

Standardized benchmarks have exposed the gap between agent capability and human performance. OSWorld (Xie et al., 2024) introduced the first large-scale real-computer benchmark with 369 tasks on Ubuntu Linux, where the best agent at launch achieved only 12.24% against 72.36% for humans; WindowsAgentArena (Bonatti et al., 2024) established a complementary Windows-only suite showing a similar gap. More recent benchmarks target specific domains or platforms, such as Spider2-V (Cao et al., 2024) for enterprise data-science workflows, ScreenSpot (Cheng et al., 2024) for visual grounding, and macOSWorld (Yang et al., 2025a) for macOS-specific tasks. Parallel efforts extend evaluation to mobile (Rawles et al., 2024; 2023; Chai et al., 2025; Deka et al., 2017) and web settings (Zhou et al., 2024; Deng et al., 2024; Yao et al., 2022; Lu et al., 2024; Drouin et al., 2024a;b; Singh et al., 2024; Kapoor et al., 2024a; Chezelles et al., 2024; Yoran et al., 2024), building upon classic web-interaction benchmarks (Shi et al., 2017; Nogueira & Cho, 2016; Liu et al., 2018). Beyond task-completion benchmarks, recent work evaluates multimodal model robustness and reliability more broadly, including safety and attribute evaluations under out-of-distribution visual inputs (Tu et al., 2024; Lee et al., 2024; Chen et al., 2026), vision-language reward and reinforce learning (Tu et al., 2025; Chen et al., 2025). Initial results across these benchmarks consistently fall far behind human experts, revealing failure patterns that motivate the design of GUI-Pro-Agent.

2.2. GUI Agents: Models and Frameworks.

End-to-end models trained for GUI interaction—UI-TARS (Qin et al., 2025), AGUVIS (Xu et al., 2024), ShowUI (Lin et al., 2024), CogAgent (Hong et al., 2024), OS-Atlas (Wu et al., 2024), among others (Yang et al., 2024b; Wang et al., 2025b)—achieve strong grounding without HTML or accessibility trees. Screen-based agents (Niu

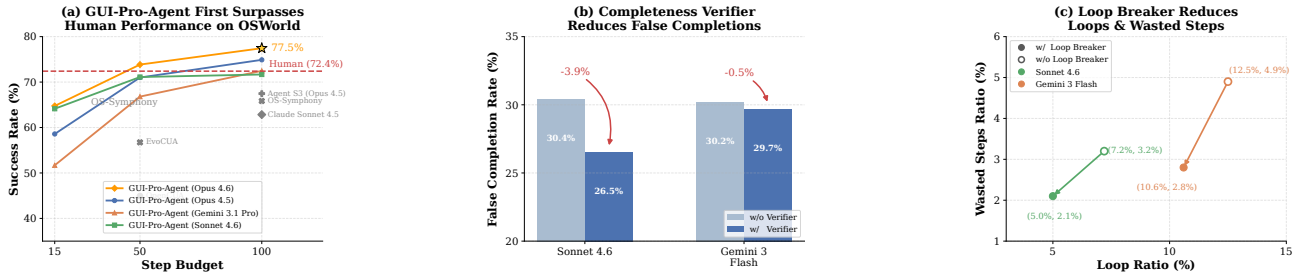

Figure 1. **Advantages of GUI-Pro-Agent.** (a) Our GUI-Pro-Agent (w/ Opus 4.6) achieves the best results (77.5%) on OSWorld-Verified (Xie et al., 2024) and surpasses human performance with one pass. (b) On one hand, by employing Completeness Verifier, GUI-Pro-Agent mitigates the early stopping issue. (c) On the other hand, the proposed Loop Breaker helps ease the repetitive-looping problem and save wasted action steps in GUI tasks.

et al., 2024; Tan et al., 2024; Wang et al., 2024a; Baechler et al., 2024) further explore pixel-space control, and web agents (Zheng et al., 2024; Gur et al., 2024; He et al., 2024) investigate long-horizon decision making in browser environments. Frontier providers have followed with commercial APIs: Claude Computer Use (Anthropic, 2024), OpenAI CUA (OpenAI, 2025), and Seed (ByteDance Seed Team, 2025), the latter serving as our dedicated grounding model.

Complementary modular frameworks compose MLLMs with planning, memory, and tools. The Agent S family (Agashe et al., 2024; 2025) couples hierarchical planning with experience memory; Agent S3 (Gonzalez-Pumariega et al., 2025) adds best-of-N trajectory selection. OS-Symphony (Yang et al., 2026) and GTA1 (Yang et al., 2025b) combine memory and test-time search, while CoAct (Song et al., 2025) and EvoCUA (Xue et al., 2026) emphasize coding-as-actions and synthetic experience. UFO (Zhang et al., 2024b) and AutoGLM (Liu et al., 2024) target Windows and mobile respectively. There are also surveys that discuss broader design trade-offs (Wang et al., 2024b; Kapoor et al., 2024b).

2.3. 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., 2025) 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 an evidence-grounded Completeness Verifier and a multi-level Loop Breaker that escalates recovery strategies.

3. The GUI-Pro-Agent System

3.1. Overview

GUI-Pro-Agent is built around a Manager agent that performs GUI interaction in a perceive–reason–act loop with 2 mandatory action-wise tools and 3 flexible on-demand tools as shown in Figure 2. At each step t, the Manager receives the current belief state b_t, which summarizes the user instruction, trajectory history, and current observation (screenshot, results from other tool agents, etc). Based on the state b_t, the Manager outputs one UI action a_t (e.g., click, scroll through `pyautogui`). The Manager may equally invoke the Coding Agent, Search Agent, or Grounding Agent whenever it decides as needed; these are first-class actions in the same action space, activated according to the runtime situation rather than as subordinate fallbacks. Unlike hierarchical planner–executor systems, the Manager retains full ownership of the task throughout execution and operates end-to-end without explicit subtask decomposition (Gonzalez-Pumariega et al., 2025).

Two mandatory tool agents are invoked after every action to maintain the execution reliability. First, the Loop Breaker Agent analyzes the trajectory and detects repeated behaviors that indicate execution stagnation. Second, a Completeness Verifier cross-examines the Manager's completion claims against UI-observable evidence to ensure that task goals are truly satisfied before the termination. Together, these two modules form mandatory post-action checks that enforce reliable progress and prevent silent failures, while other tools (Searcher, Coder, Grounder) remain flexible and are invoked by the Manager only when needed. Our GUI-Pro-Agent omits explicit planning and memory modules that are widely used in related works (Gonzalez-Pumariega et al.,

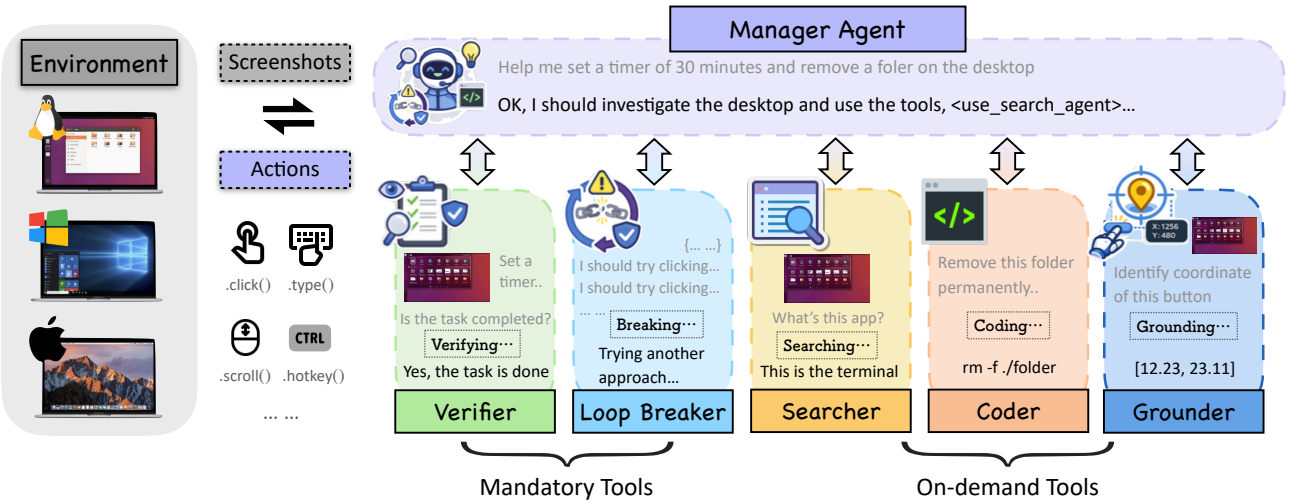

Figure 2. **Overview of GUI-Pro-Agent.** The Manager Agent decides the overall plan and provides concrete actions to the environment. We integrate two mandatory tools Loop Breaker and Completeness Verifier (Verifier) that are called after every action. Three on-demand tools—Search Agent (Searcher), Coding Agent (Coder), and Grounding Agent (Grounder)—are available for the Manager to deploy as needed.

2025; Yang et al., 2026): planners performed poorly in our framework, and memory modules were removed for simplicity.

3.2. Completeness Verifier

A core design principle is that the agent must never declare success without verifiable evidence. We realize this with the *Completeness Verifier*, a two-level mechanism: a *Completion Gate* embedded in the Manager's system prompt enforces mandatory self-verification at every step, and an agent-level *verifier* independently cross-examines any resulting completion claim.

Verifier Gate. At the beginning of a task, the gate derives a set of K UI-observable success criteria $\mathcal{C} = \{c_1, \ldots, c_K\}$, rewriting any hidden-state condition into a clear rule. At each step t, the Manager is asked to self-check on these success criteria and makes a decision based on the belief state b_t:

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

After each action, a rule-based verification is further posed for the expected visual outcome (Table 1) to avoid judgment errors caused by delay.

Completeness Model Judge. Once the Completion Gate outputs DONE, an independent MLLM judge double-checks the verdict. The verifier agent takes the task instruction g, current observation o_t, and recent trajectory b_t as input, and produces a binary accept/reject decision, together with the reason fed to the Manager.

Table 1. Verification rules by action type in the Completeness Verifier.

Action Type	Expected Verification
Click button/menu	New UI element visible (dialog, tab, highlight)
Toggle setting	State label changed ("Enable" → "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

Finally, the verifier is designed to give the acceptance verdict only if every criterion has direct visual evidence, all side-effect actions (saving, exporting, sending) show visible confirmation, and the UI is stable. The final decision of task termination requires both checking modules to agree on task completion. On rejection decisions, the rejection reasons are appended to the trajectory so that subsequent actions are aware of them.

3.3. Loop Breaker

Repetitive action loops are a pervasive GUI agent failure mode. We address this with Loop Breaker, a three-tier mandatory check that takes the t-th step screen observation o_t and UI actions a_t as input.

We define an action-level repetition counter n_t^a and a screen-state repetition counter n_t^o:

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

The Loop Breaker triggers escalation based on these coun-

ters and the reflection signal from an external model judge w_t.

Tier 1: Modality Switch ($n_t^a \geq \tau_a$). If the same action on the same target produces no visible change repeatedly ($n_t^a \geq \tau_a$), the agent must switch the 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 frequently ($n_t^o \geq \tau_o$), the agent must switch its overall strategy (e.g. from menu navigation to programmatic file editing).

Tier 3: Reflection-Driven Judge. Finally, to gain a more comprehensive assessment, we employ an external model judge to inspect the recent trajectory and produce the final decision of "keep" or "switch" the current strategy. When w_t gives the switch signal, a hard directive is injected into the Manager Agent at step $t+1$ that blacklists the repeated action and forces the Manager to select from the remaining actions (e.g., `click` → `type`, GUI actions → `call_coding_agent`).

The three tiers are complementary: Tier 1 handles local action failures, Tier 2 handles navigation dead-ends, and Tier 3 provides an external and overall assessment for patterns the Manager's local checks may miss.

3.4. Search Agent (Searcher)

GUI agents frequently stall on unfamiliar application workflows. A natural remedy is to retrieve step-by-step tutorials on demand. Prior work such as OS-Symphony (Yang et al., 2026) addresses this via visual browser search, whose search quality could be compromised by inaccurate grounding and/or the overall agent ability. We take a more direct and effective approach: the Search Agent $\mathcal{S}$ issues a targeted search query and returns structured results using an LLM's native search capability. (e.g., Gemini 3 Pro (Google DeepMind, 2025b)). Specifically, the Search Agent is exposed to the Manager as a callable tool `search(query)` in the same action space as UI primitives. When the Manager decides external knowledge is needed, it formulates a targeted "How to" question following structured guidelines embedded in the tool description. The Search Agent then issues this query to an LLM with native search grounding, which returns a summarized tutorial as plain text. This knowledge is then injected into the belief state for all subsequent steps as complementary knowledge. Our approach unifies all information in the text domain, avoiding the overhead of browser interaction entirely. Note that, the Search Agent is invoked only when the agent is uncertain about a GUI workflow *and* a tutorial is likely to exist (well-documented application features).

3.5. Coding Agent (Coder) and Grounding Agent (Grounder)

For goals better suited to programmatic execution, e.g., bulk data edits across numerous cells or files, and heavy computations, previous works tend to use the coding tool (Wang et al., 2024c; Yang et al., 2024a). Following previous practice, we integrate a Coding Agent that operates in an independent Python/Bash execution loop with its own step budget. It returns a plain-text execution summary to the Manager and is never called for visual layout tasks or tasks achievable in less than 3 GUI actions.

For common UI elements localization, we utilize an MLLM as the grounding agent as practiced (Agashe et al., 2025; Gonzalez-Pumariega et al., 2025; Yang et al., 2025b). It is designed to integrate both low-level visual cues (e.g., position, appearance) and high-level semantic context (e.g., functionality, relevance) to generate a precise coordinate given the screenshot. We will provide the configuration details about these agents in the supplementary material.

4. Experiments

In this section, we will first give information about benchmarks, implementation details about GUI-Pro-Agent, then we will delve deeper into the main results, ablation studies w.r.t. different components, and most importantly, analysis.

4.1. Experimental Settings

Benchmarks. We evaluate GUI-Pro-Agent on two desktop GUI benchmarks. OSWorld-Verified (OSWorld for short) (Xie et al., 2024) is our primary benchmark, comprising 369 real-world tasks across five application domains (Web, Office, File, Multimedia, and Daily) in an Ubuntu virtual environment. Following common practice (Xie et al., 2024), we exclude the 8 Google Drive tasks, yielding a final evaluation set of 361 tasks. WindowsAgentArena (WAA) (Bonatti et al., 2024) provides 154 tasks on a Windows environment spanning productivity, file management, and system configuration, serving as our cross-platform generalization testbed.

Implementation Details. On OSWorld, we evaluate six configurations. Three use Claude Opus 4.5 (Anthropic, 2025a), Claude Sonnet 4.6 (Anthropic, 2025c), or Gemini 3 Flash (Google DeepMind, 2025a) as the shared backbone for the Manager Agent, Reflection Agent, Completeness Verifier, and Coding Agent, with Gemini 3 Pro (Google DeepMind, 2025b) 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 (Zhou et al., 2025), an Opus 4.6 (Anthropic, 2026) configuration that uses Gemini 3.1 Pro (Google DeepMind, 2026) as the Search Agent, and a Gemini 3.1 Pro configura-

Table 2. Success rates (%) on OSWorld-Verified (Xie et al., 2024). We report results of 15, 50, and 100 steps budgets. Baseline results are available from the OSWorld leaderboard.

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							
EvoCUA (Xue et al., 2026)	50	78.26	59.71	64.55	81.63	27.89	56.73
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., 2025)	50	70.83	60.65	54.09	69.39	42.37	56.39
OS-Symphony w/ GPT-5 (Yang et al., 2026)	50	75.00	64.85	61.19	69.23	54.86	63.61
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							
OpenCUA-72B (Wang et al., 2025b)	100	61.13	44.73	49.95	72.58	22.16	44.91
Seed-1.8 (ByteDance Seed Team, 2025)	100	66.67	68.80	67.05	71.43	42.38	61.87
DeepMiner-Mano-72B (Fu et al., 2025)	100	66.67	63.22	52.51	83.67	24.41	53.91
Claude-Sonnet-4.5 (Anthropic, 2025b)	100	70.83	72.59	61.35	63.27	49.54	62.84
Kimi K2.5 (Team et al., 2026)	100	73.91	69.11	66.42	63.47	46.11	63.34
CoAct-1 w/ GPT-5 (Song et al., 2025)	100	75.00	62.93	57.94	71.43	47.87	59.93
GTA1 w/ GPT-5 (Yang et al., 2025b)	100	79.17	63.91	62.56	79.59	50.91	63.41
OS-Symphony w/ GPT-5 (Yang et al., 2026)	100	79.17	65.73	67.76	69.23	57.98	65.84
Agent S3 w/ GPT-5 (Gonzalez-Pumariega et al., 2025)	100	77.50	66.46	61.23	69.80	51.37	62.63
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., 2025)	100	75.00	76.06	67.51	59.18	59.00	67.46
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**

tion that uses Gemini 3.1 Pro across all components except grounding. On WAA, we evaluate Gemini 3 Flash as the Manager. Seed 1.8 (ByteDance Seed Team, 2025) serves as the default visual grounding model, translating natural-language element descriptions to screen coordinates in the screenshot; the only exception is the Opus 4.5 + MAI-UI variant, which swaps in MAI-UI for grounding. All agents on both benchmarks operate in iterative planning mode with a per-task step budget of maximum 100 actions. The Completeness Verifier is called at temperature $T = 0.2$; all other components use temperature $T = 1.0$. Experiments are run on the officially released Docker environments on Amazon Web Services.

4.2. Main Results

4.2.1. GUI-Pro-Agent Achieves Human-level Performance.

With a 100-step budget, GUI-Pro-Agent w/ Opus 4.5 achieves **74.89%** and w/ Opus 4.6 achieves **77.45%** average success rate, *both* surpassing the reported human-level performance of 72.4% by over 2% and outperforming the strongest prior systems by a clear margin, e.g., Agent S3 w/ Opus 4.5 (67.46%) and most recent HIPPO w/ Opus 4.5 (74.49%). 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. Replacing Seed 1.8 with MAI-UI for the Opus 4.5 setup further improves the 100-step average to 76.26%, with the largest gain in Office tasks (79.38%→84.26%). 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)), both surpassing Agent S3 w/ Opus 4.5 by over 13% (i.e. average 76% vs. 62%). Moreover, even Gemini 3 Flash achieves 68.77%, outperforming the best GPT-5-based frameworks despite using a substantially smaller model, which justifies the superiority of our framework design. Category-wise, Opus 4.6 and Gemini 3 Flash both exceed 91% on the OS domain at 100

Table 3. Results on WindowsAgentArena (WAA) (Bonatti et al., 2024). 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., 2025)	-	-	-	-	-	-	54.1
GUI-Pro-Agent	32.6	73.3	87.5	66.7	52.4	75.0	**60.4**
– Completeness Verifier	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., 2025b)	-	-	-	-	-	-	51.2
Agent S3 w/ GPT-5 (Gonzalez-Pumariega et al., 2025)	-	-	-	-	-	-	56.6
GUI-Pro-Agent	35.0	73.3	87.5	66.7	52.4	83.3	**61.0**
– Completeness Verifier	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

Table 4. Overall success rate under different step budgets on OSWorld. The proposed components prove to be useful for GUI-Pro-Agent.

Method	Success Rate (%)	
	50 Steps	100 Steps
GUI-Pro-Agent w/ Gemini 3 Flash	63.14	**68.77**
– Completeness Verifier	**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**
– Completeness Verifier	68.53	68.81
– Loop Breaker	69.67	71.63
– Search Agent	68.92	70.04

steps, outperforming previous baselines by over 10% (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 Manager matters more in difficult tasks.

On the other hand, GUI-Pro-Agent generalizes well to WindowsAgentArena (Table 3). Our GUI-Pro-Agent reaches 60.4% overall at 50 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 9.2%.

4.2.2. GUI-Pro-Agent Outperforms Published Baselines within only 15 Action Steps.

To probe the limits of our GUI-Pro-Agent system, we conduct experiments given only 15-step configuration on OSWorld. At a 15-step budget, both Opus 4.6 (64.75%) and Sonnet 4.6 (64.13%) already surpass the best reported 50-step system (e.g., OS-Symphony at 63.6%), with one-third the step budget. Furthermore, Opus 4.5 at 15 steps reaches 58.6%, showing a narrow performance gap with the latest systems like UiPath and CoACT-1 in the 50-step budget. At 50 steps, GUI-Pro-Agent w/ Opus 4.6 achieves **73.85%**, already surpassing human-level performance even at half the standard budget. Sonnet 4.6 follows closely at 71.1%, outperforming the best published 50-step system, OS-Symphony w/ GPT-5, by 7.5%. When we take a closer look, the Office and Workflow categories drive this advantage: Sonnet 4.6 scores 79.2% on Office and 66.7% on Workflow at 50 steps, compared to 64.9% and 54.9% for OS-Symphony.

An interesting crossover emerges between Sonnet 4.6 and Opus 4.5: Sonnet leads at both 15 steps and 50 steps (average 67.6% $>$ 64.8%), yet Opus 4.5 overtakes at 100 steps (74.89% vs. 71.67%). This suggests that Sonnet 4.6 is more step-efficient and solves tasks in fewer actions, while Opus 4.5 as a more capable MLLM benefits from the additional budget to recover on harder tasks that require deeper reasoning. The Opus 4.6 combines both strengths, leading at every budget tier. These efficiency results indicate that GUI-Pro-Agent's components—in particular the Completeness Verifier and Loop Breaker—substantially reduce wasted steps, allowing the agent to reach high task success well within tight budgets.

4.3. Ablations and Discussions

4.3.1. Verifying Each Component.

Table 4 isolates each component's contribution on OSWorld. All three modules improve the full system at 100 steps

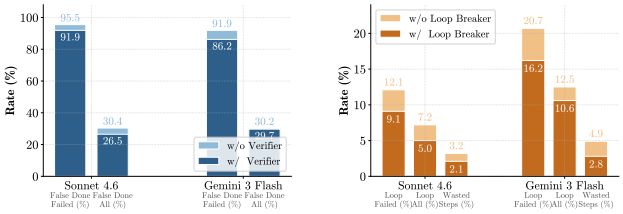

Figure 3. The Completeness Verifier (w/ Verifier) reduces false completion rates (False Done / Failed, False Done / All) and the Loop Breaker reduces loop incidence (Loop / Failed, Loop / All) and the wasted steps ratio on OSWorld.

across both backbones, though their relative importance shifts with model strength. When using Sonnet 4.6, the Completeness Verifier contributes the most (-3.1% when removed at 100 steps), followed by the Search Agent (-1.9%) and Loop Breaker (-1.4% at 50 steps). This aligns with the observation that Sonnet 4.6 is already step-efficient—it loops less often, so the verifier's role in preventing early termination becomes the dominant factor. When switching to Gemini 3 Flash, the pattern shifts: the Loop Breaker matters most at 50 steps (-4.2%) and the Search Agent at 100 steps (-3.0%), suggesting that a weaker backbone relies more on the loop recovery and external knowledge to compensate for its reasoning gap.

On the other hand, ablations on WAA (Table 3) reinforce these findings further. At 50 steps, removing the Search Agent, Completeness Verifier, and Loop Breaker drops overall accuracy by 11.0%, 9.1%, and 7.8% respectively (from 60.4%); at 100 steps the Completeness Verifier becomes the most impactful (-9.7%), followed by Search and Loop Breaker (both decrease by over 5%). For individual categories, removing the Completeness Verifier devastates the Office accuracy by 21.0% (32.6%$\rightarrow$11.6% at 50 steps), suggesting that without explicit completion verification the agent routinely declares Office tasks done prematurely on the Windows platform. Finally, removing the Search Agent also degrades Office (-14.0%) and Media (-14.3%), which underscores the role of external knowledge on tasks involving Windows-specific workflows.

4.3.2. Completeness Verifier Mitigates False Completion Ratio.

To understand *how* the Completeness Verifier helps, we analyze false completion behavior across all OSWorld tasks at 100 steps in Figure 3. We report two metrics: *False Done/Fail* (FDF), the fraction of failed tasks where the agent incorrectly declared completion, and *False Done/All* (FDA), the fraction of all tasks that are false completions. For Sonnet 4.6, the verifier reduces the FDF by 3.4% (95.5% to 91.9%) and FDA by 3.9% (30.4% to 26.5%). For Gemini 3 Flash, FDF drops more sharply from 91.9% to 86.2% (-5.7%). A notable observation is that even with the verifier, FDF remains above 86% for both backbones, again indicating that false completion is by far the dominant failure mode in GUI agents—when the agent fails, it almost always believes it has succeeded. This motivates the Completeness Verifier as a necessary safeguard and proves its utility in computer-use tasks.

4.3.3. 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 obvious for the 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 inclined toward repetitive behavior and benefit more from the loop detection.

4.3.4. Stronger Backbones Leverage Tools More Efficiently.

Figure 4 examines how the Completeness Verifier 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., Verifier +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 somehow hurt under tighter budgets (e.g., Verifier -11.3% and Search -9.7% at 15 steps). This phenomenon can be attributed to the fact that 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.

4.3.5. Case Study.

To illustrate how the system's components interact to recover from failures, we trace a representative task 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 agent's first attempt modifies one master slide and calls `done()`, but the Completeness Verifier rejects it, noting the slide number is still grey and the file has not been saved. This rejection is critical: without verifi-

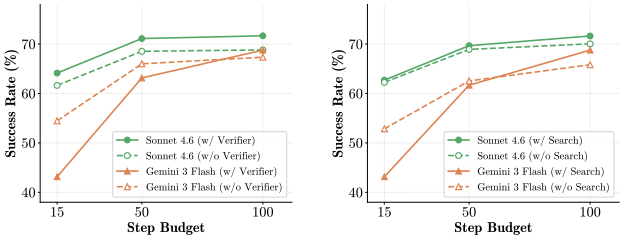

Figure 4. Impact of the Completeness Verifier and Search Agent across step budgets on OSWorld. Sonnet 4.6 benefits consistently at all budgets, while Gemini 3 Flash gains only at relaxed budgets: tool calls consume actions that less efficient models cannot afford under tight constraints.

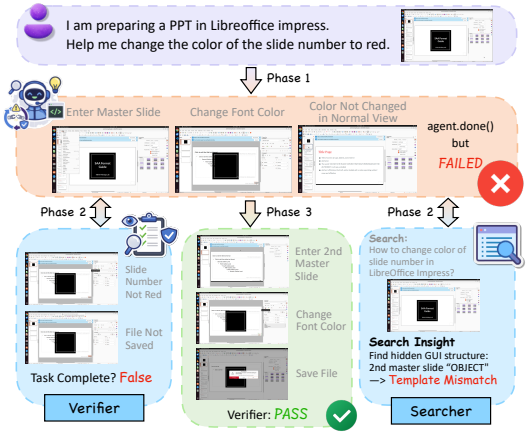

Figure 5. **Case study on OSWorld:** ***Changing slide number color to red in LibreOffice Impress.*** The agent encounters multiple failures before final completion. The Completeness Verifier rejects two premature `done()` calls, and the Search Agent provides critical procedural knowledge about multi-master-slide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

cation, the agent would have terminated before the task was actually completed. Prompted by the rejection, the agent invokes the Search Agent. The retrieved procedural knowledge contains a key insight absent from the agent's prior context: *"If your presentation uses different master slides for different sections, repeat for each master slide."* Acting on this, the agent inspects the slide masters and discovers a second master named `OBJECT` that it had overlooked—explaining why some slides remained unchanged. Armed with this knowledge, the agent applies the color change to both masters and calls `done()` a second time. The Verifier rejects again: the color is correct but the file was not saved. After an explicit `Ctrl+S`, the Verifier accepts the result.

This trajectory highlights the complementary roles of the two components. The Completeness Verifier prevents early termination—each of the rejected `done()` calls would have yielded a score of 0 without it. The Search Agent, in turn, provides task-specific procedural knowledge that the agent could not infer from the visual observation alone.

5. Conclusion

In this paper, we presented GUI-Pro-Agent, a modular GUI agent framework that addresses two fundamental challenges in autonomous desktop agents: premature task completion and repetitive action loops. Our three targeted components—a Completeness Verifier that enforces UI-observable success criteria, a Loop Breaker with multi-tier escalation, and a Search Agent for on-demand procedural knowledge—collectively push OSWorld performance to 77.5% with Opus 4.6, surpassing human-level for the first time, while also performing in the top tier on WindowsAgentArena at 61.0%. Ablation studies confirm that each component contributes meaningfully, though the benefit is modulated by backbone strength and step budget. Analysis further reveals that false completion remains the dominant failure mode across backbones, motivating continued research on completion verification.

Broader Impact and Limitations. 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 and advanced planning strategies (e.g., tree search) 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.

Acknowledgements

This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-24-1-0015.

References

Agashe, S., Han, J., Gan, S., Yang, J., Li, A., and Wang, X. E. Agent S: An open agentic framework that uses computers like a human. *arXiv preprint arXiv:2410.08164*, 2024.

Agashe, S., Wong, K., Tu, V., Yang, J., Li, A., and Wang, X. E. Agent S2: A compositional generalist-specialist framework for computer use agents. *arXiv preprint arXiv:2504.00906*, 2025.

Andrade, T. et al. Self-grounded verification: Detecting hallucinations and bias in MLLM GUI agents. *arXiv preprint arXiv:2507.11662*, 2025.

Anthropic. Developing computer use. https://www.anthropic.com/news/developing-computer-use, 2024.

Anthropic. Claude opus 4.5. https://www.anthropic.com/news/claude-opus-4-5, 2025a.

Anthropic. Introducing claude sonnet 4.5, September 2025b. URL https://www.anthropic.com/news/claude-sonnet-4-5.

Anthropic. Claude sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6, 2025c.

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

Baechler, G., Narayanan, S., et al. ScreenAI: A vision-language model for UI and infographics understanding. *arXiv preprint arXiv:2402.04615*, 2024.

Bai, S., Cai, Y., Chen, R., Chen, K., Chen, X., Cheng, Z., Deng, L., Ding, W., Gao, C., Ge, C., Ge, W., Guo, Z., Huang, Q., Huang, J., Huang, F., Hui, B., Jiang, S., Li, Z., Li, M., Li, M., Li, K., Lin, Z., Lin, J., Liu, X., Liu, J., Liu, C., Liu, Y., Liu, D., Liu, S., Lu, D., Luo, R., Lv, C., Men, R., Meng, L., Ren, X., Ren, X., Song, S., Sun, Y., Tang, J., Tu, J., Wan, J., Wang, P., Wang, P., Wang, Q., Wang, Y., Xie, T., Xu, Y., Xu, H., Xu, J., Yang, Z., Yang, M., Yang, J., Yang, A., Yu, B., Zhang, F., Zhang, H., Zhang, X., Zheng, B., Zhong, H., Zhou, J., Zhou, F., Zhou, J., Zhu, Y., and Zhu, K. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025.

Bonatti, R., Zhao, D., Bonacci, F., Dupont, D., Abdali, S., Li, Y., Lu, Y., Wagle, J., Koishida, K., Bucker, A., Jang, L., and Hui, Z. Windows agent arena: Evaluating multi-modal OS agents at scale. *arXiv preprint arXiv:2409.08264*, 2024.

ByteDance Seed Team. Seed1.8 model card: Towards generalized real-world agency, 2025.

Cao, R., Lei, F., Wu, H., et al. Spider2-V: How far are multimodal agents from automating data science and engineering workflows? *arXiv preprint arXiv:2407.10956*, 2024.

Cemri, M. et al. Why do multi-agent LLM systems fail? a taxonomy of failure patterns in multi-agent LLM systems. *arXiv preprint arXiv:2503.13657*, 2025.

Chai, Y. et al. A3: Android agent arena for mobile GUI agents. *arXiv preprint arXiv:2501.01149*, 2025.

Chen, H., Tu, H., Wang, F., Liu, H., Tang, X., Du, X., Zhou, Y., and Xie, C. Sft or rl? an early investigation into training r1-like reasoning large vision-language models. *Transactions on Machine Learning Research*, 2025.

Chen, R., Sun, S., Guo, X., Zhang, S., Liu, K., Liu, S., Wang, Z., Zhang, Q., Zhang, H., and Cao, X. Where not to learn: Prior-aligned training with subset-based attribution constraints for reliable decision-making. *arXiv preprint arXiv:2602.07008*, 2026.

Cheng, K., Sun, Q., Chu, Y., Xu, F., Li, Y., Zhang, J., and Wu, Z. SeeClick: Harnessing GUI grounding for advanced visual GUI agents. *arXiv preprint arXiv:2401.10935*, 2024.

Chezelles, T. L. S. D., Acero, J., et al. The BrowserGym ecosystem for web agent research. *arXiv preprint arXiv:2412.05467*, 2024.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Deka, B., Huang, Z., Franzen, C., Hibschman, J., Afergan, D., Li, Y., Nichols, J., and Kumar, R. Rico: A mobile app dataset for building data-driven design applications. In *UIST*, pp. 845–854, 2017. doi: 10.1145/3126594.3126651.

Deng, X., Gu, Y., Zheng, B., Chen, S., Stevens, S., Wang, B., Sun, H., and Su, Y. Mind2web: Towards a generalist agent for the web. In *NeurIPS*, 2024.

Drouin, A., Porlier, R., et al. WorkArena: How capable are web agents at solving common knowledge work tasks? *arXiv preprint arXiv:2403.07718*, 2024a.

Drouin, A., Porlier, R., et al. WorkArena++: Benchmarking web agents with agentic and procedural knowledge. *arXiv preprint arXiv:2407.05291*, 2024b.

Fu, T., Su, A., Zhao, C., Wang, H., Wu, M., Yu, Z., Hu, F., Shi, M., Dong, W., Wang, J., Chen, Y., Yu, R., Peng, S., Li, M., Huang, N., Wei, H., Yu, J., Xin, Y., Zhao, X., Gu, K., Jiang, P., Zhou, S., and Wang, S. Mano technical report. *arXiv preprint arXiv:2509.17336*, 2025.

Gonzalez-Pumariega, G., Tu, V., Lee, C.-L., Yang, J., Li, A., and Wang, X. E. The unreasonable effectiveness of scaling agents for computer use. *arXiv preprint arXiv:2510.02250*, 2025.

Google DeepMind. Gemini 3 flash model card. https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf, 2025a.

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

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

Gur, I., Furuta, H., Huang, A., Safdari, M., Matsuo, Y., Eck, D., and Faust, A. A real-world WebAgent with planning, long context understanding, and program synthesis. *arXiv preprint arXiv:2307.12856*, 2024.

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

He, Z., Choi, J., et al. WebVoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.

Hong, W., Wang, W., Lv, Q., Xu, J., Yu, W., Ji, J., Wang, Y., Wang, Z., Dong, Y., Ding, M., and Tang, J. CogAgent: A visual language model for GUI agents. In *CVPR*, 2024.

Kapoor, R., Lu, K., et al. OmniACT: A dataset and benchmark for enabling multimodal generalist autonomous agents for desktop and web. *arXiv preprint arXiv:2402.17553*, 2024a.

Kapoor, S., Stroebl, B., Siegel, Z. S., Rajkumar, N., and Narayanan, A. AI agents that matter. *arXiv preprint arXiv:2407.01502*, 2024b.

Kim, G., Petit, P., Toshev, A., Park, T., Zisserman, A., Mao, C., Ibarz, J., Batra, D., Wu, T., and Tian, Y. Language models can solve computer tasks. *arXiv preprint arXiv:2303.17491*, 2023.

Lee, T., Tu, H., Wong, C. H., Zheng, W., Zhou, Y., Mai, Y., Roberts, J. S., Yasunaga, M., Yao, H., Xie, C., et al. Vhelm: A holistic evaluation of vision language models. *Advances in Neural Information Processing Systems*, 37: 140632–140666, 2024.

Li, T., Azizi, D., Papusha, I., Angelopoulos, A. N., and Yang, D. A zero-shot language agent for computer control with structured reflection. *arXiv preprint arXiv:2310.08740*, 2023.

Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., and Cobbe, K. Let's verify step by step. In *ICLR*, 2024.

Lin, K. Q., Li, L., Gao, D., Yang, Z., Wu, S., Bai, Z., Lei, W., Wang, L., and Shou, M. Z. ShowUI: One vision-language-action model for GUI visual agent. *arXiv preprint arXiv:2411.17465*, 2024.

Liu, E. Z., Guu, K., Pasupat, P., Shi, T., and Liang, P. Reinforcement learning on web interfaces using workflow-guided exploration. *arXiv preprint arXiv:1802.08802*, 2018.

Liu, X., Qin, B., Liang, D., et al. AutoGLM: Autonomous foundation agents for GUIs. *arXiv preprint arXiv:2411.00820*, 2024.

Lu, X., Meng, C., et al. WebLINX: Real-world website navigation with multi-turn dialogue. *arXiv preprint arXiv:2402.05930*, 2024.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., and Clark, P. Self-refine: Iterative refinement with self-feedback. In *NeurIPS*, 2023.

Niu, R., Sharma, Y., Goyal, P., Majumdar, A., et al. ScreenAgent: A vision language model-driven computer control agent. *arXiv preprint arXiv:2402.07945*, 2024.

Nogueira, R. and Cho, K. End-to-end goal-driven web navigation. *arXiv preprint arXiv:1602.02261*, 2016.

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

Pan, J., Zhang, Y., Tomlin, N., Zhou, Y., Levine, S., and Suhr, A. Autonomous evaluation and refinement of digital agents. In *COLM*, 2024.

Qin, Y., Ye, Y., Fang, J., Wang, H., Liang, S., Tian, S., Zhang, J., Li, J., Li, Y., Huang, S., Zhong, W., Li, K., Li, J., Li, C., Niu, M., Zhang, C., Zheng, Y., Qin, S., Lu, Q., Tang, R., Rajmohan, S., Lin, Q., and Zhang, D. UI-TARS: Pioneering automated GUI interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

Rawles, C., Clinckemaillie, S., Chang, Y., Walber, J., and Toyama, D. AndroidWorld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*, 2024.

Rawles, C. et al. Android in the wild: A large-scale dataset for android device control. *arXiv preprint arXiv:2307.10088*, 2023.

Shi, T., Karpathy, A., Fan, L., Hernandez, J., and Liang, P. World of bits: An open-source platform for web-based agents. In *ICML*, pp. 3135–3144, 2017.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

Singh, A., Yang, J., Sriram, A., et al. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. *arXiv preprint arXiv:2401.13649*, 2024.

Song, L., Dai, Y., Prabhu, V., Zhang, J., Shi, T., Li, L., Li, J., Savarese, S., Chen, Z., Zhao, J., Xu, R., and Xiong, C. CoAct-1: Computer-using agents with coding as actions. *arXiv preprint arXiv:2508.03923*, 2025.

Tan, W., Zhang, W., et al. Cradle: Empowering foundation agents towards general computer control. *arXiv preprint arXiv:2403.03186*, 2024.

Team, K., Bai, T., Bai, Y., Bao, Y., Cai, S. H., Cao, Y., Charles, Y., Che, H. S., Chen, C., Chen, G., Chen, H., Chen, J., Chen, J., Chen, J., Chen, J., Chen, K., Chen, L., Chen, R., Chen, Y., Chen, Y., Chen, Y., Chen, Y., Chen, Y., Chen, Y., Chen, Y., Chen, Y., Chen, Z., Chen, Z., Cheng, D., Chu, M., Cui, J., Deng, J., Diao, M., Ding, H., Dong, M., Dong, M., Dong, Y., Dong, Y., Du, A., Du, C., Du, D., Du, L., Du, Y., Fan, Y., Fang, S., Feng, Q., Feng, Y., Fu, G., Fu, K., Gao, H., Gao, T., Ge, Y., Geng, S., Gong, C., Gong, X., Gongque, Z., Gu, Q., Gu, X., Gu, Y., Guan, L., Guo, Y., Hao, X., He, W., He, W., He, Y., Hong, C., Hu, H., Hu, J., Hu, Y., Hu, Z., Huang, K., Huang, R., Huang, W., Huang, Z., Jiang, T., Jiang, Z., Jin, X., Jing, Y., Lai, G., Li, A., Li, C., Li, C., Li, F., Li, G., Li, G., Li, H., Li, H., Li, J., Li, J., Li, J., Li, L., Li, M., Li, W., Li, W., Li, X., Li, X., Li, Y., Li, Y., Li, Y., Li, Y., Li, Z., Li, Z., Liao, W., Lin, J., Lin, X., Lin, Z., Lin, Z., Liu, C., Liu, C., Liu, H., Liu, L., Liu, S., Liu, S., Liu, T., Liu, T., Liu, W., Liu, X., Liu, Y., Liu, Y., Liu, Y., Liu, Y., Liu, Y., Liu, Z., Liu, Z., Lu, E., Lu, H., Lu, Z., Luo, J., Luo, T., Luo, Y., Ma, L., Ma, Y., Mao, S., Mei, Y., Men, X., Meng, F., Meng, Z., Miao, Y., Ni, M., Ouyang, K., Pan, S., Pang, B., Qian, Y., Qin, R., Qin, Z., Qiu, J., Qu, B., Shang, Z., Shao, Y., Shen, T., Shen, Z., Shi, J., Shi, L., Shi, S., Song, F., Song, P., Song, T., Song, X., Su, H., Su, J., Su, Z., Sui, L., Sun, J., Sun, J., Sun, T., Sung, F., Tai, Y., Tang, C., Tang, H., Tang, X., Tang, Z., Tao, J., Teng, S., Tian, C., Tian, P., Wang, A., Wang, B., Wang, C., Wang, C., Wang, C., Wang, D., Wang, D., Wang, D., Wang, F., Wang, H., Wang, H., Wang, H., Wang, H., Wang, H., Wang, J., Wang, J., Wang, J., Wang, K., Wang, L., Wang, Q., Wang, S., Wang, S., Wang, S., Wang, W., Wang, X., Wang, X., Wang, Y., Wang, Y., Wang, Y., Wang, Y., Wang, Y., Wang, Y., Wang, Z., Wang, Z., Wang, Z., Wang, Z., Wang, Z., Wang, Z., Wei, C., Wei, M., Wen, C., Wen, Z., Wu, C., Wu, H., Wu, J., Wu, R., Wu, W., Wu, Y., Wu, Y., Wu, Y., Wu, Z., Xiao, C., Xie, J., Xie, X., Xie, Y., Xin, Y., Xing, B., Xu, B., Xu, J., Xu, J., Xu, J., Xu, L. H., Xu, L., Xu, S., Xu, W., Xu, X., Xu, X., Xu, Y., Xu, Y., Xu, Y., Xu, Z., Xu, Z., Yan, J., Yan, Y., Yang, G., Yang, H., Yang, J., Yang, K., Yang, N., Yang, R., Yang, X., Yang, X., Yang, Y., Yang, Y., Yang, Y., Yang, Z., Yang, Z., Yang, Z., Yao, H., Ye, D., Ye, W., Ye, Z., Yin, B., Yu, C., Yu, L., Yu, T., Yu, T., Yuan, E., Yuan, M., Yuan, X., Yue, Y., Zeng, W., Zha, D., Zhan, H., Zhang, D., Zhang, H., Zhang, J., Zhang, P., Zhang, Q., Zhang, R., Zhang, X., Zhang, Y., Zhang, Y., Zhang, Y., Zhang, Y., Zhang, Y., Zhang, Y., Zhang, Y., Zhang, Y., Zhang, Y., Zhang, Z., Zhao, C., Zhao, F., Zhao, J., Zhao, S., Zhao, X., Zhao, Y., Zhao, Z., Zheng, H., Zheng, R., Zheng, S., Zheng, T., Zhong, J., Zhong, L., Zhong, W., Zhou, M., Zhou, R., Zhou, X., Zhou, Z., Zhu, J., Zhu, L., Zhu, X., Zhu, Y., Zhu, Z., Zhuang, J., Zhuang, W., Zou, Y., and Zu, X. Kimi K2.5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Tu, H., Cui, C., Wang, Z., Zhou, Y., Zhao, B., Han, J., Zhou, W., Yao, H., and Xie, C. How many are in this image a safety evaluation benchmark for vision llms. In *European Conference on Computer Vision*, pp. 37–55. Springer, 2024.

Tu, H., Feng, W., Chen, H., Liu, H., Tang, X., and Xie, C. Vilbench: A suite for vision-language process reward modeling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 6775–6790, 2025.

UiPath. Uipath screen agent. https://github.com/xlang-ai/OSWorld/tree/main/mm_agents/uipath, 2025.

Wang, H., Zou, H., Song, H., Feng, J., Fang, J., Lu, J., Liu, L., Luo, Q., Liang, S., Huang, S., Zhong, W., Ye, Y., Qin, Y., Xiong, Y., Song, Y., Wu, Z., Li, A., Li, B., Dun, C., Liu, C., Zan, D., Leng, F., Wang, H., Yu, H., Chen, H., Guo, H., Su, J., Huang, J., Shen, K., Shi, K., Yan, L., Zhao, P., Liu, P., Ye, Q., Zheng, R., Xin, S., Zhao, W. X., Heng, W., Huang, W., Wang, W., Qin, X., Lin, Y., Wu, Y., Chen, Z., Wang, Z., Zhong, B., Zhang, X., Li, X., Li, Y., Zhao, Z., Jiang, C., Wu, F., Zhou, H., Pang, J., Han, L., Liu, Q., Ma, Q., Liu, S., Cai, S., Fu, W., Liu, X., Wang, Y., Zhang, Z., Zhou, B., Li, G., Shi, J., Yang, J., Tang, J.,

Li, L., Han, Q., Lu, T., Lin, W., Tong, X., Li, X., Zhang, Y., Miao, Y., Jiang, Z., Li, Z., Zhao, Z., Li, C., Ma, D., Lin, F., Zhang, G., Yang, H., Guo, H., Zhu, H., Liu, J., Du, J., Cai, K., Li, K., Yuan, L., Han, M., Wang, M., Guo, S., Cheng, T., Ma, X., Xiao, X., Huang, X., Chen, X., Du, Y., Chen, Y., Wang, Y., Li, Z., Yang, Z., Zeng, Z., Jin, C., Li, C., Chen, H., Chen, H., Chen, J., Zhao, Q., and Shi, G. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*, 2025a.

Wang, J., Mo, H., Ishii, E., et al. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*, 2024a.

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W. X., Wei, Z., and Wen, J.-R. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 2024b.

Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., and Ji, H. Executable code actions elicit better LLM agents. In *ICML*, 2024c.

Wang, X., Wang, B., Lu, D., Yang, J., Xie, T., Wang, J., Deng, J., Guo, X., Xu, Y., Wu, C. H., Shen, Z., Li, Z., Li, R., Li, X., Chen, J., Zheng, B., Li, P., Lei, F., Cao, R., Fu, Y., Shin, D., Shin, M., Hu, J., Wang, Y., Chen, J., Ye, Y., Zhang, D., Du, D., Hu, H., Chen, H., Zhou, Z., Yao, H., Chen, Z., Gu, Q., Wang, Y., Wang, H., Yang, D., Zhong, V., Sung, F., Charles, Y., Yang, Z., and Yu, T. OpenCUA: Open foundations for computer-use agents. *arXiv preprint arXiv:2508.09123*, 2025b.

Wu, Z., Wu, Z., Xu, F., Wang, Y., Sun, Q., Jia, C., Cheng, K., Ding, Z., Chen, L., Liang, P. P., and Qiao, Y. OS-Atlas: A foundation action model for generalist GUI agents. *arXiv preprint arXiv:2410.23218*, 2024.

Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T. J., Cheng, Z., Shin, D., Lei, F., Liu, Y., Xu, Y., Zhou, S., Savarese, S., Xiong, C., Zhong, V., and Yu, T. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *NeurIPS*, 2024.

Xu, Y., Wang, Z., Wang, J., Lu, D., Xie, T., Saha, A., Sahoo, D., Yu, T., and Xiong, C. Aguvis: Unified pure vision agents for autonomous GUI interaction. *arXiv preprint arXiv:2412.04454*, 2024.

Xue, T., Peng, C., Huang, M., Guo, L., Han, T., Wang, H., Wang, J., Zhang, X., Yang, X., Zhao, D., Ding, J., Ma, X., Xie, Y., Pei, P., Cai, X., and Qiu, X. EvoCUA: Evolving computer use agents via learning from scalable synthetic experience. *arXiv preprint arXiv:2601.15876*, 2026.

Yang, B., Jin, K., Wu, Z., Liu, Z., Sun, Q., Li, Z., Xie, J., Liu, Z., Xu, F., Cheng, K., Li, Q., Wang, Y., Qiao, Y., Wang, Z., and Ding, Z. OS-Symphony: A holistic framework for robust and generalist computer-using agent. *arXiv preprint arXiv:2601.07779*, 2026.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024a.

Yang, P., Ci, H., and Shou, M. Z. macOSWorld: A multilingual interactive benchmark for GUI agents. *arXiv preprint arXiv:2506.04135*, 2025a.

Yang, Y., Wang, Y., Li, D., Luo, Z., Chen, B., Huang, C., and Li, J. Aria-UI: Visual grounding for GUI instructions. *arXiv preprint arXiv:2412.16256*, 2024b.

Yang, Y., Li, D., Dai, Y., Yang, Y., Luo, Z., Zhao, Z., Hu, Z., Huang, J., Saha, A., Chen, Z., et al. GTA1: GUI test-time scaling agent. *arXiv preprint arXiv:2507.05791*, 2025b.

Yao, S., Chen, H., Yang, J., et al. WebShop: Towards scalable real-world web interaction with grounded language agents. In *NeurIPS*, 2022.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. In *NeurIPS*, 2023a.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *ICLR*, 2023b.

Yoran, O. et al. AssistantBench: Can web agents solve realistic and time-consuming tasks? *arXiv preprint arXiv:2407.15711*, 2024.

Zhang, C., He, S., Li, L., Li, S., Qian, Y., Wang, Y., Ma, M., Kang, Y., Lin, Q., Rajmohan, S., et al. LLM-brained GUI agents: A survey. *arXiv preprint arXiv:2411.18279*, 2024a.

Zhang, C., Li, L., He, S., Zhang, X., Qiao, B., Qin, S., Ma, M., Kang, Y., Lin, Q., Rajmohan, S., et al. UFO: A UI-focused agent for windows OS interaction. *arXiv preprint arXiv:2402.07939*, 2024b.

Zheng, B., Gou, B., Kil, J., Sun, H., and Su, Y. SeeAct: GPT-4V(ision) is a generalist web agent, if grounded. *arXiv preprint arXiv:2401.01614*, 2024.

Zhou, H., Zhang, X., Tong, P., Zhang, J., Chen, L., Kong, Q., Cai, C., Liu, C., Wang, Y., Zhou, J., and Hoi, S. Mai-ui technical report: Real-world centric foundation gui agents, 2025.

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., Alon, U., and Neubig, G. WebArena: A realistic web environment for building autonomous agents. In *ICLR*, 2024.

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: Implementation details including model configurations and benchmark categories.
- Section A.2: The full action space exposed to the Manager Agent.
- Section A.3: The system prompts for different agent roles.
- Section A.4: Additional ablation results.

A.1. Implementation Details

A.1.1. Configurations.

Table 5 summarizes the model assignments used for each component across our experimental configurations. On OSWorld, we evaluate 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 Verifier, and Coding Agent, with Gemini 3 Pro (Google DeepMind, 2025b) 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, 2026) 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.

A.1.2. 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 Verifier 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 (VLM variant) has a budget of 20 browsing steps; the LLM variant performs a single query.

A.1.3. 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 such environment snapshot on Amazon Web Services virtual machines. Screen-

Table 5. Model assignments for each component across experimental configurations.

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 Verifier	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 / –

shots are captured at 1920×1080 resolution and served as the primary observation modality.

A.1.4. 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.2. Action Space

Table 6 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.3. System Prompts for Agents

A.3.1. Reflection Agent Prompt (Loop Breaker Tier 3)

The Reflection Agent serves as the external model judge in the Loop Breaker's Tier 3 (see Section 3.4 of the main paper). 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
↪  subtask execution by
reflecting on the trajectory of a subtask and
↪  providing feedback for
what the next step should be.
You have access to the Subtask 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:

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
```

Table 6. 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.

```
  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.3.2. Completeness Verifier Prompt

The Completeness Verifier model judge is invoked whenever the Manager's Completion Gate outputs DONE (see Section 3.2 of the main paper). 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.3.3. Manager Agent System Prompt

The Manager Agent's system prompt is generated dynamically by the procedural memory module based on the runtime configuration. Below we report the full prompt for the default full-system Linux configuration: screenshot observation, iterative planning, Coding Agent enabled, Search Agent enabled, Completion Gate enabled, Loop Breaker enabled, and feasibility management enabled. The prompt consists of several sections: (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 either switch strategy completely or
  ↪  call
  `agent.fail()`.
- 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")
```
```

A.4. Additional Ablation Results

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

A.4.1. Completeness Verifier: Failure Breakdown across Step Budgets

Table 7 reports the false completion count and DONE accuracy across step budgets for both backbones, with and without the Completeness Verifier. *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 7. False completion breakdown across step budgets on OSWorld. **w/**: full system with Completeness Verifier; **w/o**: Verifier 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.

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

For Sonnet 4.6, the Completeness Verifier 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.4.2. Loop Breaker: Effectiveness across Step Budgets

Table 8 reports task success rates with and without the Loop Breaker across step budgets for both backbones.

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.

Table 8. 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	61.68%	58.90%	+2.78%
@15	43.15%	49.30%	−6.15%

For Gemini 3 Flash, the Loop Breaker helps substantially at 100 steps (+1.82%) and 50 steps (+2.78%), where Flash loops frequently and the escalation mechanism redirects the agent toward productive actions. However, at 15 steps the Loop Breaker *hurts* (−6.15%). This mirrors the tool-overhead pattern discussed in the main paper: the Loop Breaker's escalation tiers (modality switch, strategy change) themselves consume actions, and at 15 steps 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.