REWIND AND RETRY: ON THE TEST-TIME SCALING AND HARNESS OF LONG HORIZONAL COMPUTER USE AGENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Computer-use agents (CUAs) interact with computers through GUIs and CLIs to complete complex tasks. As task horizons grow longer, the bottleneck shifts from the policy itself to the execution harness and test-time scaling strategy. We study this harness axis rather than proposing a new policy. Simple test-time scaling methods, such as *best-of-N*, resemble depth-first search and are inefficient in exploring the solution space. We study three mechanisms for test-time scaling in computer-use agents: **pruned tree search**, **evidence-based evaluation**, and **hybrid GUI–CLI execution**. We instantiate these ideas in a practical harness, **Checkpoint Tree Search (CTS)**, which decomposes a task into milestones, branches from VM checkpoints, selects among candidate continuations using evidence-based verification, and resumes execution from the selected state. Against a single rollout in the same harness, CTS improves mean graded score by *+9%* relative on OSWorld and *+48%* relative on the longer OSWorld-V2. On the multi-milestone subset of OSWorld-V2, three-branch CTS exceeds the judge-free pass@3 upper bound of independent full-episode sampling best-of-3, suggesting that enabling mid-episode rewinds breaches the theoretical ceiling of independent trajectory sampling. This work also presents extensive and novel empirical studies on the harness and evaluation for CUAs. We will release the complete reproduceable codebase on GitHub.

1 INTRODUCTION

Computer-use agents (CUAs) interact with computers through graphical user interfaces (GUIs) and command-line interfaces (CLIs) to assist humans in completing tasks, offering substantial potential to augment the societal workforce. With recent advances in general-purpose CUAs (Hu et al., 2024; OpenAI, 2026a;b), today's systems perform well on tasks requiring relatively few interaction steps, such as those in OSWorld V1 (Xie et al., 2024). However, their performance on long-horizon tasks, such as those in OSWorld V2 (Yuan et al., 2026), remains far below the level required for commercial deployment. In addition to requiring sufficiently long model context windows, long-horizon tasks pose significant challenges for test-time scaling, including agent memory, task decomposition, and error recovery (Wang et al., 2024a; Shinn et al., 2023). In coding agents, these challenges are often addressed through improved agent harnesses (Jimenez et al., 2024; Wang et al., 2024b; 2025). This paper presents an in-depth investigation of harness design and test-time scaling for CUAs.

Simple test-time scaling methods on CUA, such as GRPO rollouts (Shao et al., 2024) or best-of-N sampling (Cobbe et al., 2021; Brown et al., 2024; Snell et al., 2024), can be viewed as depth-first search (DFS) over the solution space: a CUA performs a complete rollout until task termination and then evaluates the resulting trajectory. Such methods are highly search-inefficient, as every rollout remains subject to the full range of challenges associated with long-horizon tasks, including limited memory. Many effective pathfinding algorithms, such as A* (Hart et al., 1968), combine breadth-first tree search (BFS) with heuristic-guided pruning. This approach provides valuable insights into how the search efficiency of CUAs can be substantially improved (Yao et al., 2023a; Hao et al., 2023; Zhou et al., 2024a).

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.

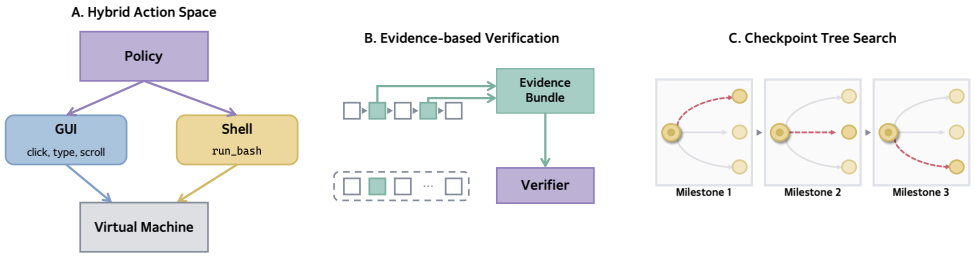

Figure 1: The three components studied in this work: hybrid execution, evidence-based verification, and checkpointed search. **(A)** One policy is given both a GUI channel and a shell and chooses per step, neither is forced. **(B)** During a rollout the CUA files proof artifacts and the evaluation model reads that proof instead of the last k frames. **(C)** Search branches at milestone boundaries, evaluates the branches, and restores the winner before continuing.

We investigate how test-time compute should be allocated in long-horizon computer-use agents. As one practical instantiation, we introduce **Checkpoint Tree Search (CTS)**, a CUA harness that replaces the all-or-nothing episode with search over resumable VM states. A planner first decomposes the task into milestones. At each milestone boundary the agent snapshots the machine, runs N attempts from that snapshot, has an evaluation model pick the best attempt and prune the rest, then restores the winner and continues from it (§3). Turning that idea into a working harness takes three things (Figure 1), and we study each of them as an object in its own right (Wang et al., 2025).

Evidence-based evaluation. The quality of evaluation and reward signals is a critical component of both test-time scaling and reinforcement learning (Cobbe et al., 2021; Lightman et al., 2024). If we cannot reliably determine which trajectory is more successful, the search process becomes meaningless. Prior work has shown that using an LLM as a judge to inspect the final few frames can improve best-of-N selection (Zheng et al., 2023; Pan et al., 2024; Lù et al., 2025; Sun et al., 2026). In this work, we propose evidence-based evaluation, in which the CUA prepares an evidence document while completing the task. The judge LLM then selects the best trajectory solely on the basis of this document, yielding more reliable evaluations than methods that inspect only the final few frames (§5.1; Appendix H).

VM checkpointing. This technique enables us to fully capture the computer's execution state, including the cursor position and active applications. Creating and restoring a checkpoint take only about 14.3 seconds and 11 seconds, respectively (Appendix B). This capability allows the agent to rewind and replay challenging stages, or milestones, much like reattempting a difficult level in a video game.

Hybrid GUI–CLI interaction. We provide the CUA with access to both GUI and CLI interaction channels (Song et al., 2026; Fan et al., 2026), allowing the model to select the most appropriate channel based on the task type. Our experiments demonstrate that this strategy improves both efficiency and task success rates (§5.1; Appendix E).

Results. Against a single rollout in the same harness, CTS improves mean graded score by 9% relative on OSWorld and 48% relative on the longer OSWorld-V2 (§5, Figure 3; Appendix D). On the multi-milestone subset of OSWorld V2, our harness even surpasses pass@N, the theoretical upper bound of best-of-N, demonstrating the strategic advantages of tree search and pruning. We further conduct ablation experiments and empirical studies on the proposed components. Compared with the GUI-only setting, our hybrid GUI–CLI approach uses 23.6% fewer steps while achieving an 18.0% relative improvement in performance. It also outperforms the CLI-only setting by 10.9%, indicating that different tasks may benefit from different interaction modalities (Figure 5; Table 11). In terms of evaluation, our evidence-based approach yields a significant 23.7% relative improvement over conventional LLM-as-a-judge evaluation (Table 2; Appendix H).

Contributions and Impact. Ultimately, our core contribution lies not in proposing a novel search algorithm, but in delivering a comprehensive empirical framework for CUA test-time scaling,

evidence-based evaluation, and harness engineering. By design, this modular harness serves as a model-agnostic capability multiplier for diverse foundation policies. Furthermore, our rigorous analysis of mid-episode rollouts and verifiable reward signals establishes a highly transferable blueprint for future RL pipelines in long-horizon digital environments.

2 Related Work

Computer-use benchmarks and agents. OSWorld (Xie et al., 2024) grades agents on real desktop tasks with execution-based checks, and has been followed by environments for Windows (Bonatti et al., 2025), Android (Rawles et al., 2025) and the browser (Zhou et al., 2024b; Koh et al., 2024; Deng et al., 2023). Agent-side work has gone into perception and grounding (Yang et al., 2023; Hong et al., 2024; Cheng et al., 2024; Qin et al., 2025; Xu et al., 2025; Zhang et al., 2025) and into modular frameworks adding memory and subtask routing (Agashe et al., 2025a;b; Wu et al., 2024; Zheng et al., 2024). The frontier is reached either by training the policy harder (Wang et al., 2026; Xue et al., 2026; Lu et al., 2026) or by restructuring what it is asked to track (Yang et al., 2026b; Hu et al., 2024), in every case one episode per task. We instead hold the policy fixed and change what the harness lets it do, the axis Wang et al. (2025) argue is undermeasured. Table 3 places this work against those lines along all three and highlights a recurring pattern: most prior systems that enlarge the harness change several axes at once. We deliberately vary one axis at a time while freezing the others.

Acting through code. Outside the GUI setting it is standard for agents to act by writing programs (Yao et al., 2023b; Wang et al., 2024b; Gao et al., 2023; Chen et al., 2023; Schick et al., 2023; Wang et al., 2024a), and software-engineering agents work almost entirely through a shell (Jimenez et al., 2024; Wang et al., 2025). Recent desktop systems mix the two, delegating to a programmer agent (Song et al., 2026), folding a coding agent into the action space (Gonzalez-Pumariega et al., 2026), or driving the machine through code and keeping the screen for what code cannot reach (Yang et al., 2026b); each beats its own pixel-only configuration. What those reports cannot settle is how much of the gain is the channel, since the mixed system also changes the orchestration, the context handling and often the backbone — and Fan et al. (2026) find the sign of the effect flipping between models under one harness. We measure the channel alone, paired, with everything else frozen (§5.1), and re-run the delegating system under our own executor so that the comparison between dispatching and choosing is not also a comparison between backbones.

Test-time compute. Sampling many solutions and selecting with a verifier reliably converts compute into accuracy (Cobbe et al., 2021; Brown et al., 2024; Snell et al., 2024), and search over partial solutions extends it (Yao et al., 2023a; Hao et al., 2023; Zhou et al., 2024a). All of it assumes a partial solution can be extended in several directions cheaply — free for text, not free for an agent embedded in a stateful operating system, which is why the computer-use literature has stayed with independent full-episode sampling. The strongest such result narrates each candidate episode and has a judge compare the narratives (Gonzalez-Pumariega et al., 2026); the closest to resuming anything is GTA1 (Yang et al., 2026c), which re-ranks candidates with a vision-language reward model, but at whole-trajectory granularity. In both, the unit of selection is a finished episode, so the compute spent on a branch that goes wrong early is spent in full.

Verifying agent trajectories. LLM judges are standard for open-ended evaluation (Zheng et al., 2023; Bai et al., 2022) and for GUI agent trajectories (Pan et al., 2024; Lù et al., 2025), where the input is a sequence of screenshots. Process supervision (Lightman et al., 2024; Setlur et al., 2025) shows that grading intermediate steps beats grading only the outcome, and models critique their own work poorly without external grounding (Huang et al., 2024) while tool-grounded critique fares much better (Gou et al., 2024; Shinn et al., 2023; Madaan et al., 2023). Our evidence mechanism applies that to the desktop (§5.1).

3 Method

Given a task instruction x, a computer-use agent completes the task by acting on a desktop computer. Let s denote its full state, which determines how the computer responds to every subsequent action.

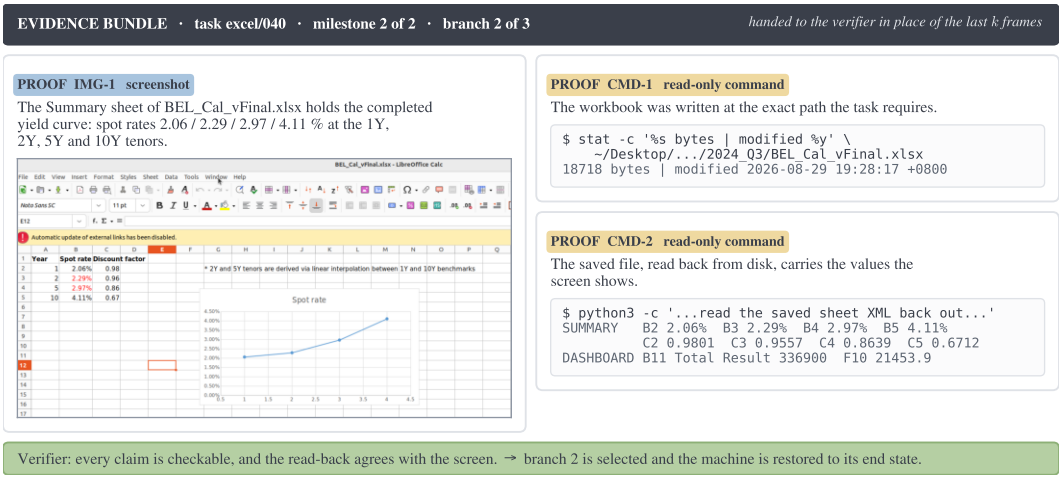

Figure 2: An evidence bundle, abridged for width: the document a branch hands to the verifier instead of its last few frames. The screenshot is a real frame from an OSWorld-V2 run. Each proof pairs a checkable claim with the artifact that settles it, and here two of the three artifacts are read-backs from the saved file rather than pictures of it.

The policy π does not observe s directly: at each step it receives a screenshot, a rendering of s, and outputs an action, until it declares completion or exhausts its step budget. Checkpoint Tree Search (CTS) runs this policy inside a search over saved states: a planner decomposes x into milestones, the harness checkpoints s at each milestone boundary, samples N branches from that checkpoint, retains the branch that the verifier selects, and restores its end state before the next milestone (Figure 1). Section 3.1 describes the checkpoint, Section 3.2 the channels through which the policy acts, Section 3.3 the evidence a branch records and the verifier that reads it, and Section 3.4 the search that combines them.

3.1 VM Checkpointing

Since s resides in the operating system rather than in the agent's context, an intermediate state can be reached again only by re-executing the trajectory from s_0, unless a copy of s has been saved. At a milestone boundary the applications involved in the task are still running and hold state that is not on disk, such as an open dialog or unsaved edits, so restoring the disk alone does not recover the state from which the next milestone continues. Benchmark tasks run inside a virtual machine (VM), which we run under QEMU and checkpoint with its internal snapshots: a snapshot pauses the virtual CPUs and stores the guest memory, the device state and the disk as a unit, and restoring it returns the VM to the identical execution state. The settings on which the snapshot cost depends and the measured times are given in Appendix B; a checkpoint or a restore takes far less time than the execution of a branch, and §3.4 uses this primitive for both branching and resumption.

3.2 Hybrid action space

The policy acts through up to two channels: the provider's native computer-use tool, which takes a screenshot as input and outputs mouse and keyboard actions, and a shell tool, `run_bash`, which executes one command inside the VM and returns its output without opening a terminal window (Prompt 1). In the hybrid configuration both channels are available and the policy selects one at each step, so file and text manipulation can be completed with single commands while sub-goals whose correctness depends on the rendered output are carried out through the GUI. To measure the effect of the action space alone, we vary only the exposed tool set and keep the model, the prompt, the step budget and the observation identical across the GUI-only, shell-only and hybrid configurations (§5.1; Appendix E). Appendix E also describes the safeguards that keep the two channels distinct: a shell-only policy receives a screenshot after every command, and commands that synthesize keyboard or mouse input are rejected.

3.3 EVIDENCE-BASED EVALUATION

The search selects branches solely on the output of the verifier V, so its performance is bounded by the accuracy of V at each boundary. Existing verifiers for GUI agents use an LLM as a judge over the final few frames of a trajectory (Pan et al., 2024; Lù et al., 2025), but a screenshot is a rendering of s and does not determine it: whether a file was written to the required path, for example, cannot be read from the screen, and at a milestone boundary, where the task is incomplete, the screen shows even less. Execution-based checkers read s directly but require task-specific knowledge that is available only when the benchmark is constructed. We therefore make evidence collection a secondary objective of the policy: besides the action channels, it has `capture_evidence(claim)`, which stores the current screenshot as a numbered proof image, and `run_readonly_bash(command)`, which executes a whitelisted read-only command and preserves its output (Prompt 4). When a branch completes its milestone, the policy organizes the collected artifacts into an evidence bundle E, in which each entry pairs a verifiable claim with the artifact that supports it (Figure 2), and the verifier reads E in place of the last k frames and checks each claim against its artifact (Prompts 5 and 6). At a milestone boundary the bundles are evaluated against the end state of the assigned milestone, and at the end of an episode against x (Prompts 9 and 8).

3.4 CHECKPOINT TREE SEARCH

Milestone decomposition. A planner is called once per task and returns an ordered list of milestones $g_1, \ldots, g_m$, each a verifiable state rather than an operation, so that the verifier can evaluate whether a branch has reached it (Prompt 7). The search branches only at milestones, so the plan determines the depth of the tree, and a plan with a single milestone reduces to best-of-N. Because a branch inherits s but not the context of the preceding segment, each milestone description is self-contained, and information that a later milestone requires must be left on screen or stated in the final message of the branch.

Branching, selection and resumption. Let s_0 denote the initial state and s_{i-1} the state retained after milestone $i-1$. At milestone i, CTS checkpoints s_{i-1} and samples N branches from it,

$$b_{i,1}, \ldots, b_{i,N} \;\sim\; \pi(\cdot \mid s_{i-1},\, g_i\,),$$

each an independent continuation with a fresh context that attempts only g_i and ends in a state $s_{i,j}$ with an evidence bundle $E_{i,j}$. The verifier returns the index of the branch to retain, $j_i^\star = V(E_{i,1}, \ldots, E_{i,N};\, g_i)$; the harness restores $s_i = s_{i,j_i^\star}$, which becomes the checkpoint for milestone $i+1$, and discards the other branches with their checkpoints. The state returned to the user is s_m, produced by the concatenation $(b_{1,j_1^\star}, \ldots, b_{m,j_m^\star})$ of one segment per milestone.

Comparison with best-of-N. CTS differs from best-of-N with the same branching factor in two respects. The trajectory before a branch point is executed once and shared by the N branches, so N branches at each of m milestones require approximately the model time of N full episodes, and retaining a single branch per boundary keeps this cost linear in m. The returned trajectory is one of N^m combinations of per-milestone segments, whereas independent sampling returns only the N combinations in which every segment belongs to the same episode, so a concatenated trajectory can complete a task that none of the N full episodes completes (§5).

4 EXPERIMENTAL SETUP

Benchmarks and metrics. We evaluate on OSWorld (Xie et al., 2024), 361 desktop tasks graded by execution-based checkers, and on its successor OSWorld-V2 (Yuan et al., 2026), whose 108 tasks have a substantially longer horizon (a human requires a median of roughly 1.6 hours per task). A small number of OSWorld-V2 tasks cannot be run on our host and are excluded (Appendix K). All settings within a comparison are evaluated on the same tasks, and the number of tasks is reported with each result. The checker of each task returns a single score in $[0, 1]$ and emits no separate pass flag; on OSWorld roughly 96% of runs score exactly 0 or 1, and fractional values arise only when a task's checker is a conjunction of sub-checks. Our primary metric is the mean of that score over tasks, which we call *graded score*; it is the quantity the online leaderboard reports as success rate, and the one on which every published number we quote was obtained. Because a mean over

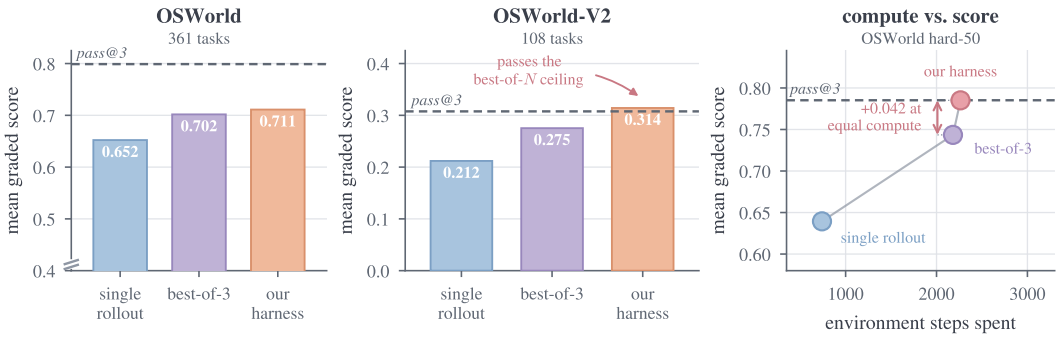

Figure 3: **Left, centre:** our harness against its ablations. The dashed rule is `pass@3`, the score a perfect verifier would obtain over the same three rollouts; it bounds every selection rule at $N = 3$ and needs no verifier to define. On OSWorld-V2 our harness is above it. Both bar panels span the same range of graded score so heights are comparable; the left axis is truncated and marked as such. **Right:** the same arms against the environment steps they consumed on `hard-50`. The two search arms spend within a few percent of each other; what differs is where the compute goes.

partially credited tasks does not say where the credit came from, some tables add two binarizations of the same per-task scores: *soft success*, the fraction of tasks scoring at least 0.5, and *strict success*, the fraction scoring exactly 1. Ablations and prompt development use `hard-50`, the 50 OSWorld tasks with the deepest milestone plans (§3.4).

Models. Unless stated otherwise the policy, the planner and the verifier are `gpt-5.4` through the provider's computer-use tool. Verifier prompts were developed on the labeled `hard-50` pool and frozen before any evaluation reported here was run; that iteration is reported in Figure 4 (top) and Appendix C.

Settings and budgets. We compare three settings, a *single rollout*, i.e., one episode from the initial state, *best-of-3*, and *CTS* with $N = 3$, which reduces to best-of-3 on a one-milestone plan and is run as such (§3.4), and report *`pass@3`*, the maximum checker score over the three episodes of best-of-3, which is not a system but an upper bound on any selection rule at $N = 3$. All settings within a comparison receive the same per-episode step budget, which no OSWorld episode reached. Cost is measured in environment steps, the number of actions executed in the VM over all episodes a setting runs, including the branches that CTS discards. All experiments ran on a single 32-vCPU host with twelve concurrent VMs (Appendix B), Appendix K lists every run.

5 Results and Analysis

5.1 Main Results

Selection and search. As shown in Figure 3, best-of-3 achieves 0.702 on OSWorld and 0.275 on OSWorld-V2, compared with 0.652 and 0.212 for a single rollout, and CTS achieves 0.711 and 0.314 (Table 6 in Appendix D). The selection-based improvement, from a single rollout to best-of-3, is +0.050 on OSWorld (Figure 6) and of similar magnitude on OSWorld-V2, whereas the search-based improvement, from best-of-3 to CTS, is +0.009 on OSWorld as shown in Figure 6 and considerably larger on OSWorld-V2.

The search-based improvement is small on OSWorld because most OSWorld plans contain a single milestone, on which CTS reduces to best-of-3 (§3.4), whereas most OSWorld-V2 plan includes multiple milestones plans; the distribution of plan depth is given in Table 10 (Appendix D). On OSWorld-V2, CTS exceeds `pass@3` (0.308), the maximum checker score over the three episodes of best-of-3 and hence an upper bound on any selection rule over them; we return to this result in §5.2. The cost of search is bounded because the trajectory preceding a branch point is executed only once: on `hard-50`, CTS and best-of-3 consume nearly the same number of environment steps, and CTS performs 0.042 better (Figure 3, right). The environment steps of every setting on both benchmarks, including the branches that CTS discards, are reported in Table 6 (Appendix D).

Table 1: Context and one controlled comparison. The upper block quotes each system's own published numbers, on the metric its source reports. The lower two blocks hold the model configuration fixed within a column, so every number in a column was produced by the same executor and differs only in the harness around it.

System	model config	OSWorld	OSWorld V2
as published			
MiniMax M3 (MiniMax, 2026)	own	0.752†	0.223
Qwen3.7-Plus (Qwen Team, 2026)	own	0.733†	0.215
Kimi K2.6 (Moonshot AI, 2026)	own	0.731†	0.221
our model config, their harness			
CoAct-1 (Song et al., 2026)	ours	0.579	0.234
best-of-3 (prompt from Gonzalez-Pumariega et al., 2026)	ours	0.716	0.435
best-of-3 (prompt from Sun et al., 2026)	ours	0.678	0.432
our model config, our harness			
single rollout	ours	0.652	0.403
best-of-3 (our prompt)	ours	0.702	0.438
this harness	ours	**0.711**	**0.457**

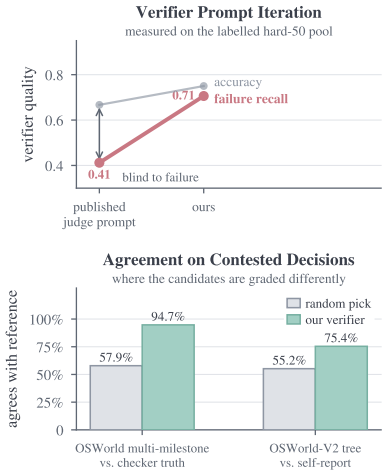

Figure 4: **Top:** what iterating the verifier prompt buys. **Bottom:** the verifier against a random pick on contested decisions.

Comparison with published systems. The upper block of Table 1 lists the published performance of frontier general-purpose models on their own harnesses, taken from the OSWorld-Verified leaderboard for OSWorld and from Yuan et al. (2026) for OSWorld-V2, both at a larger step budget than ours. These systems score above CTS on OSWorld and far below it on OSWorld-V2. Because each row pairs a different policy model with a different harness, these entries serve as reference points rather than as controlled comparisons; the contrast between the two benchmarks indicates that OSWorld does not distinguish between harnesses at this level of policy capability, which motivates our use of the longer benchmark. The lower two blocks hold the policy model fixed within each column. The two selection baselines apply the verifier prompts of Gonzalez-Pumariega et al. (2026) and Sun et al. (2026) to our harness. They show comparable results to our prompt, indicating that the gain of CTS is not attributable to prompt engineering. CoAct-1 (Song et al., 2026), whose grounding is provided by the provider's computer-use tool, is the only published system that can be re-run with our policy model while retaining its own orchestration, prompts and step budgets (Appendices F and G); under this setting it shows 0.579 on OSWorld and 0.234 on OSWorld-V2, below a single rollout, and we interpret this result together with the action space in §5.1.

The harness consists of three components: the hybrid action space (§3.2), the evidence-based verifier input (§3.3) and checkpoint tree search (§3.4). We measure the effect of each component with a controlled experiment that varies that component and keeps the other two unchanged. Figure 6 summarizes these comparisons on a single axis, reporting for each the paired difference in mean graded score and its bootstrap confidence. It shows that the selection-based improvement persists under a stronger policy model: with `gpt-5.6-sol`, best-of-3 outperforms a single rollout by 0.032 on OSWorld-V2.

The action space. Figure 5 varies only the tool set exposed to a single rollout on OSWorld-V2, fixing the policy model (`gpt-5.6-sol`), the prompt, the step budget and the observation (§3.2; Appendix E). The hybrid configuration outperforms both single-channel configurations, reaching 0.446 against 0.378 for GUI-only and 0.402 for shell-only, whereas the two single-channel configurations are statistically indistinguishable from each other. The single-channel configurations differ in cost rather than in mean score: the shell-only configuration requires 60 environment steps per task on average, compared with 123 for GUI-only, because a task whose difficulty lies in file or text manipulation is completed by a single command, whereas a task whose success criterion depends on the rendered output does not become easier through the shell. The hybrid configuration lies between the two in cost (94 steps) and exceeds both in score because the policy selects the channel at each step: the benchmark contains tasks suited to either channel, and a channel contributes only on the tasks for which it is appropriate. This observation is consistent with Fan et al. (2026), who report that the effect of a shell channel varies in sign across models.

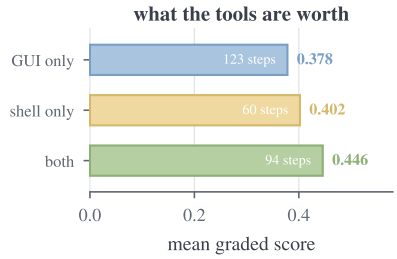

Figure 5: Action space, OSWorld-V2, single rollout, varying only the exposed tools. The annotation inside each bar is mean environment steps, which separates the arms far more sharply than the score does.

Table 2: The evidence mechanism, OSWorld-V2, `gpt-5.4`. The two tree rows differ in one bundle: the executor's evidence tools, the document it files, and the judge rubric that audits it. Best-of-3 is the reference the tree has to clear, and it changes sides.

System	graded	SR	exact
best-of-3	0.2750	24.2%	6.1%
tree, frame window	0.2444	21.2%	7.1%
tree, evidence	**0.3024**	**29.3%**	**9.1%**
Δ paired	+0.0581	[+0.009, +0.109], 99%	

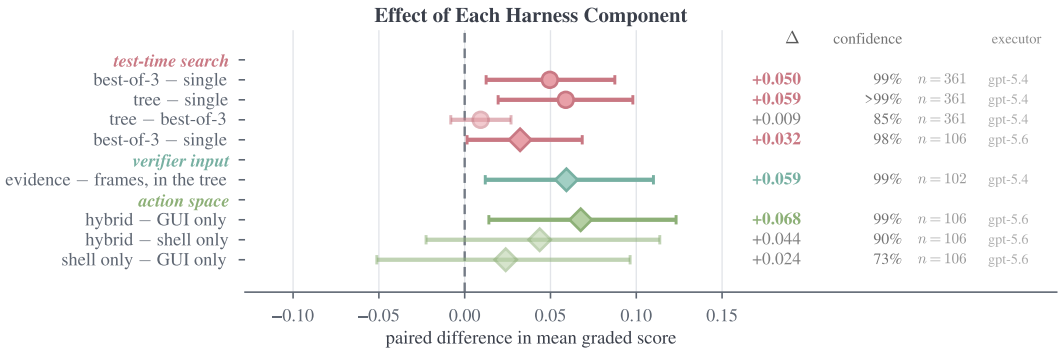

Figure 6: Every controlled comparison in the paper on one axis, grouped by the component under test. Circles = OSWorld, diamonds = OSWorld-V2; intervals are paired bootstrap on per-task graded score. The components do not overlap, they compose, and none alone carries the result.

CoAct-1 (Song et al., 2026) also provides both channels, but assigns them to separate agents and delegates the choice between them to an orchestrator. Under the same policy model, CoAct-1 underperforms our harness on both benchmarks and falls below a single rollout (Table 1). The two harnesses differ in where the channel is chosen: our policy chooses at each step after observing the current state, whereas the orchestrator of CoAct-1 commits to a channel before the consequence of that choice is observed.

What the verifier reads. The verifier input is the only component that can be varied with the policy, the plans, the tasks and the verifier model unchanged. As shown in Table 2, on OSWorld-V2 the tree whose verifier reads evidence bundles outperforms the tree whose verifier reads the last k frames. The evidence-based tree also outperforms best-of-3 (0.275) in graded score, success rate and strict success, whereas the frame-based tree underperforms best-of-3 in graded score and success rate. Because the search selects branches solely on the verifier output (§3.3), the verifier input determines whether the same search performs better or worse than best-of-3.

Figure 4 (top) presents the accuracy and failure recall of each version of the verifier prompt on `hard-50`, where each prompt is run as a pass/fail judge of single episodes and compared with checker outcomes (Appendix C). The baseline prompt, binary judge from OSReward Sun et al. (2026), attains a failure recall of 0.41, i.e., it recognizes fewer than half of the failed episodes; revising the prompt using a log of its errors raises failure recall to 0.71 and also improves accuracy (performance in Table 5, Appendix C). Failure recall is the relevant quantity for selection, since selection benefits from rejecting failed candidates rather than from confirming successful ones. On contested decisions, in which the candidates receive different scores so that the choice affects the outcome, the verifier agrees with the checker outcome in 94.7% of decisions on the 72 multi-milestone OSWorld tasks and with the branch's own best claim in 75.4% of decisions on OSWorld-V2 tree branches as shown in Figure 4, bottom and Appendix H.

5.2 THE BOTTLENECK IS EXPLORATION, NOT SELECTION

`pass@3` is the maximum checker score over the three episodes of best-of-3, so no rule that selects among these episodes can exceed it, and an improved verifier can raise best-of-3 at most to this bound. On OSWorld-V2, CTS exceeds `pass@3` (0.314 against 0.308; Figure 3). The trajectory that CTS returns is a concatenation of the branches retained at each boundary and is therefore not one of the three full episodes. Exceeding `pass@3` thus shows that the concatenated trajectory completes tasks that none of the three full episodes completes: the improvement of CTS over best-of-3 is a change in which trajectories are reached, not in how well the verifier selects among the same trajectories. On `hard-50`, where plans are deep by construction, CTS reaches `pass@3` whereas best-of-3 remains below it (Table 7, Appendix D).

Appendix D (Table 9) examines the 72 OSWorld tasks that a single rollout fails. We call a task *selection-bound* if some episode succeeded but the verifier failed to select it, and *exploration-bound* if no episode succeeded, in which case no selection rule can recover it. On this subset, CTS attains nearly the maximum score available among its own branches, whereas best-of-3 remains below `pass@3`. The tasks that remain unsolved after three episodes therefore fail because no successful trajectory exists in the pool, not because the verifier fails to select one, and the remaining gap of CTS is likewise determined by the trajectories that its branches reach.

These results indicate where additional compute is best spent. A better verifier cannot solve an exploration-bound task, because no candidate in the pool is correct. Sampling additional full episodes can add correct candidates, but each new episode repeats the part of the task before the branch point. In contrast, branching from a checkpoint reuses that part and spends the compute on new continuations from an intermediate state; the resulting trajectories are not in the original pool, and the improvement of CTS appears on the tasks that selection could not improve. In this setting, additional test-time compute is therefore most effective when it produces new trajectories rather than when it improves selection among existing ones.

6 LIMITATIONS

Our evaluation focuses on desktop computer-use tasks represented by OSWorld and OSWorld-V2, using the VM-based interaction and checkpointing interface developed in this work. The framework is therefore evaluated primarily in settings where the environment can provide reproducible intermediate states and where tasks admit meaningful milestone-based decomposition. Broader validation across additional environments, task structures, and executors would further clarify how widely the same harness design transfers. We leave these extensions, as well as richer forms of state abstraction and checkpoint management, to future work.

7 CONCLUSION

We presented a practical harness for scaling long-horizon computer-use agents at test time through checkpointed search over desktop VM states. The central contribution is a unified mechanism that combines milestone-based decomposition, hybrid GUI–CLI interaction, evidence-based verification, and checkpoint-and-restore branching, allowing an agent to spend additional computation not only on generating more complete trajectories, but also on revisiting intermediate states and exploring alternative continuations. This perspective distinguishes the approach from flat best-of-N sampling: instead of treating each rollout as an independent attempt from the initial state, the harness can reuse progress already made and branch from intermediate points in the task. Across OSWorld and OSWorld-V2, the results show that selection improves over a single rollout, while checkpointed search provides larger gains on the longer-horizon, multi-milestone setting. In particular, on multi-milestone OSWorld-V2 tasks, three-branch checkpointed search exceeds the judge-free pass@3 reference of independent full-episode sampling, showing that mid-episode rewinds can access trajectories that independent full-episode sampling cannot produce. Taken together, these results support a broader view of test-time scaling for computer-use agents: additional inference compute can be used not only to generate or rank complete attempts, but to expand the set of reachable states and allocate computation around the intermediate decisions that shape long-horizon execution.

AI USE STATEMENT

Generative AI is both the object of study and a tool used in preparing this work, so we separate the two. We used AI assistance in two parts of the writing process. We used a language model for copy-editing of prose we had already drafted, and we reviewed every resulting sentence. We used model assistance to help locate related work, but we read each cited paper and verified every citation against its published record rather than relying on model-supplied references. We did not use generative AI to originate the research question, the experimental design, or the analysis of results, and we did not use it to generate or impute any data point. The authors take responsibility for the full content of this paper, including all text, claims and artifacts produced with AI assistance.

ETHICS STATEMENT

This work involves no human subjects and no personal data. All experiments run inside disposable virtual machines on public benchmarks (Xie et al., 2024; Yuan et al., 2026), using synthetic task files distributed with those benchmarks; no private user files, credentials or network services were touched. The capability we study — letting an agent act through a shell as well as a GUI, and letting it retry from saved machine state — makes desktop agents more effective at unattended work, which cuts both ways: the same mechanism that recovers from a mis-click also lets a misaligned or prompt-injected agent do damage faster, and a shell channel raises the ceiling on that damage relative to pixel-only control. We therefore confine every run to an isolated guest with no access to the host or to user accounts, and we would regard sandboxing of this kind as a precondition for deploying the harness rather than an artifact of benchmarking.

REPRODUCIBILITY STATEMENT

Appendix K lists every run behind the reported numbers, together with the executor, the benchmark split and the task count for each, and describes the artifacts each run writes: a full action trace, all screenshots, the verifier transcript and the benchmark checker's verdict, per task and per rollout. The harness is specified in Section 3; the exact verifier, planner and executor prompts are reproduced verbatim in Appendix J, including the two external prompts we port for the baselines in Table 1. Appendix B gives the host configuration and the snapshot settings needed to reach the reported checkpoint latencies, which are the one part of the setup where hardware matters. Source code for the harness and the scoring scripts will be released.

REFERENCES

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Wang. Agent s: An open agentic framework that uses computers like a human. In *International Conference on Learning Representations*, volume 2025, pp. 22924–22946, 2025a.

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, 2025b. URL https://arxiv.org/abs/2504.00906.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional AI: Harmlessness from ai feedback, 2022. URL https://arxiv.org/abs/2212.08073.

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 Aarti Singh, Maryam Fazel,

Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 4874–4910. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/bonatti25a.html.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=YfZ4ZPt8zd.

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

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, 2021. URL https://arxiv.org/abs/2110.14168.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Siqi Fan, Minghao Li, Xiaoqian Ma, Wenhui Tan, Xiusheng Huang, Juntong Wu, Liujie Zhang, Shuo Shang, and Weihang Chen. Screenshots or tools? eliciting tool use and managing multimodal context in hybrid GUI-mcp computer-use agents, 2026. URL https://arxiv.org/abs/2608.03327.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International conference on machine learning*, pp. 10764–10799. PMLR, 2023.

Gonzalo Gonzalez-Pumariega, Vincent Tu, Chih-Lun Lee, Jiachen Yang, Ang Li, and Xin Eric Wang. Scaling agents for computer use, 2026. URL https://arxiv.org/abs/2510.02250.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Nan Duan, Weizhu Chen, et al. Critic: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, volume 2024, pp. 57734–57811, 2024.

Qijun Han, Haoqin Tu, Zijun Wang, Haoyue Dai, Yiyang Zhou, Nancy Lau, Alvaro A. Cardenas, Yuhui Xu, Ran Xu, Caiming Xiong, Zeyu Zheng, Huaxiu Yao, Yuyin Zhou, and Cihang Xie. Vlaa-GUI: Knowing when to stop, recover, and search, a modular framework for gui automation, 2026. URL https://arxiv.org/abs/2604.21375.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 8154–8173, 2023.

Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for gui agents. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 14281–14290. IEEE, 2024.

Siyuan Hu, Mingyu Ouyang, Difei Gao, and Mike Zheng Shou. The dawn of GUI agent: A preliminary case study with claude 3.5 computer use, 2024. URL https://arxiv.org/abs/2411.10323.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International conference on learning representations*, volume 2024, pp. 32808–32824, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

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

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 *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024.

Dunjie Lu, Shuai Bai, Tianyi Bai, Sicheng Fan, Chang Gao, Jian Guan, Feng Hu, Mianqiu Huang, Xingyang Huang, Yizhen Jiang, Yuheng Jing, Dehui Kong, Ning Li, Dayiheng Liu, Shixuan Liu, Zheng Liu, Que Shen, Bowen Wang, Junli Wang, Chencan Wu, Rui Xie, Tianbao Xie, Zhihui Xie, Haiyang Xu, An Yang, Tao Yu, Wenzhen Yuan, Xi Zhang, Zhenru Zhang, Mingkang Zhu, Zhaoqing Zhu, Yizhong Cao, Kai Dang, Binyuan Hui, Kaixin Li, Junyang Lin, Haiquan Wang, Zekun Wang, Yiheng Xu, Fan Yan, Mengqi Yuan, Danyang Zhang, Jiajun Zhang, Zhipeng Zhang, Fan Zhou, and Fan Zhou. Qwen-cua: Native computer use for (almost) everything, 2026. URL https://arxiv.org/abs/2608.02352.

Xing Han Lù, Amirhossein Kazemnejad, Nicholas Meade, Arkil Patel, Dongchan Shin, Alejandra Zambrano, Karolina Stańczak, Peter Shaw, Christopher J. Pal, and Siva Reddy. Agentrewardbench: Evaluating automatic evaluations of web agent trajectories, 2025. URL https://arxiv.org/abs/2504.08942.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

MiniMax. MiniMax M3. Release note, 2026. https://www.minimaxi.com/blog/minimax-m3.

Moonshot AI. Kimi K2.6. Release note, 2026. https://www.kimi.com/blog/kimi-k2-6.

OpenAI. GPT-5.4. Model card, snapshot gpt-5.4-2026-03-05, 2026a. https://developers.openai.com/api/docs/models/gpt-5.4.

OpenAI. GPT-5.6 Sol. Model card, snapshot gpt-5.6-sol, 2026b. https://developers.openai.com/api/docs/models/gpt-5.6-sol.

Jiayi Pan, Yichi Zhang, Nicholas Tomlin, Yifei Zhou, Sergey Levine, and Alane Suhr. Autonomous evaluation and refinement of digital agents, 2024. URL https://arxiv.org/abs/2404.06474.

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, 2025. URL https://arxiv.org/abs/2501.12326.

Qwen Team. Qwen3.7-Plus. Release note, 2026. https://qwen.ai/blog?id=qwen3.7.

Chris Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Androidworld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, volume 2025, pp. 406–441, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Amrith Setlur, Chirag Nagpal, Adam Fisch, Xinyang Geng, Jacob Eisenstein, Rishabh Agarwal, Alekh Agarwal, Jonathan Berant, and Aviral Kumar. Rewarding progress: Scaling automated process verifiers for llm reasoning. In *International Conference on Learning Representations*, volume 2025, pp. 60808–60838, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

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

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Zeyuan Chen, Jieyu Zhao, Ran Xu, et al. Coact-1: Computer-using multi-agent system with coding actions. In *International Conference on Learning Representations*, volume 2026, pp. 127091–127107, 2026.

Qiushi Sun, Kanzhi Cheng, Yian Wang, Bowen Yang, Hang Yan, Liheng Chen, Fangzhi Xu, Zichen Ding, Nuo Chen, Jialin Cao, Xingdong Gong, Zehao Li, Kaiming Jin, Xinfeng Yuan, Zhoumianze Liu, Jingyang Gong, Zhangyue Yin, Jiahui Gao, Zhiyong Wu, Tianbao Xie, Jianbing Zhang, Ben Kao, and Lingpeng Kong. OSreward: Instituting standardized evaluation for cross-platform computer-use reward models, 2026. URL https://arxiv.org/abs/2607.28609.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024a. ISSN 2835-8856. URL https://openreview.net/forum?id=ehfRiF0R3a.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232. PMLR, 21–27 Jul 2024b. URL https://proceedings.mlr.press/v235/wang24h.html.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Wu, et al. Opencua: Open foundations for computer-use agents. *Advances in Neural Information Processing Systems*, 38:139756–139806, 2026.

Wenyi Wu, Sibo Zhu, Kun Zhou, Aayush Salvi, Zixuan Song, and Biwei Huang. Structagent: Harness long-horizon digital agents with unified causal structure, 2026. URL https://arxiv.org/abs/2607.11388.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. OS-Copilot: Towards generalist computer agents with self-improvement, 2024. URL https://arxiv.org/abs/2402.07456.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Yiheng Xu, 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 Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 69772–69805. PMLR, 13–19 Jul 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, 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, et al. 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.

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. Set-of-Mark prompting unleashes extraordinary visual grounding in GPT-4V, 2023. URL https://arxiv.org/abs/2310.11441.

Yan Yang, Xiangru Jian, Ziyang Luo, Zirui Zhao, Yutong Dai, Ziji Shi, Hanshu Yan, Jun Hao Liew, Silvio Savarese, and Junnan Li. Stateact: Program state, before pixels, for long-horizon computer-use agents, 2026b. URL https://arxiv.org/abs/2607.22798.

Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, et al. Gta1: Gui test-time scaling agent. In *International Conference on Learning Representations*, volume 2026, pp. 48295–48323, 2026c.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

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

Miaosen Zhang, Ziqiang Xu, Jialiang Zhu, Qi Dai, Kai Qiu, Yifan Yang, Chong Luo, Tianyi Chen, Justin Wagle, Tim Franklin, and Baining Guo. Phi-ground tech report: Advancing perception in GUI grounding, 2025. URL https://arxiv.org/abs/2507.23779.

Miaosen Zhang, Xiaohan Zhao, Zhihong Tan, Zhou Huoshen, Yijia Fan, Yifan Yang, Kai Qiu, Bei Liu, Justin Wagle, Chenzhong Yin, Mingxi Cheng, Ji Li, Qi Dai, Chong Luo, Xu Yang, Xin Geng, and Baining Guo. Covering human action space for computer use: Data synthesis and benchmark, 2026. URL https://arxiv.org/abs/2605.12501.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. GPT-4V(ision) is a generalist web agent, if grounded. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 61349–61385. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/zheng24e.html.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160. PMLR, 21–27 Jul 2024a. URL https://proceedings.mlr.press/v235/zhou24r.html.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024b.

A Positioning

Table 3 places this work against the lines it borrows from, on the three axes it varies. The pattern worth noting is that the cells are almost never contested one at a time. Desktop agents have concentrated on the policy and left the harness at its default — one channel, an environment-side signal, one episode per task — while the systems that do enlarge the harness enlarge several parts of it at once. Tree search over saved state, meanwhile, is routine wherever a state is cheap to copy, and absent wherever it is not. The bottom row is the combination this paper measures, and the reason the appendices below are organised one axis at a time.

Table 3: Where this paper sits. Most computer-use work varies the policy and keeps the harness fixed: one channel, a frame-window or environment-side signal, and one episode per task. A recent line adds a code channel and selects among finished episodes, but each of those systems changes several things at once, so no single axis is isolated. Tree search over program state exists for text and the browser DOM, where a state is cheap to copy. We vary the three axes one at a time on a full desktop.

	action channel	**verification signal**	**test-time search**
Desktop GUI agents (Xie et al., 2024; Hong et al., 2024; Qin et al., 2025)	pixels	environment checker	single episode
Modular GUI frameworks (Agashe et al., 2025b; Wu et al., 2024)	pixels	environment checker	single episode
Code-augmented desktop agents (Song et al., 2026; Yang et al., 2026b; Gonzalez-Pumariega et al., 2026)	GUI + code	re-read environment	best-of-N
Code / SWE agents (Yao et al., 2023b; Jimenez et al., 2024; Wang et al., 2025)	code or shell	unit tests	best-of-N
Trajectory judges (Pan et al., 2024; Lù et al., 2025)	—	last-k frames	re-ranking
LM tree search (Yao et al., 2023a; Zhou et al., 2024a)	text / DOM	self-evaluation	tree over copies
This work	**GUI + shell**	**filed evidence**	**tree over VM state**

B INFRASTRUCTURE MEASUREMENTS

Whether mid-episode search is worth doing is partly an engineering question: a branch is only worth taking if returning to a saved state costs less than the work it saves. Figure 7 and Table 4 report where that cost ended up. Two changes account for nearly all of it, and neither is exotic — the snapshot path and the observation path were both configured for durability by default, and a search workload wants neither.

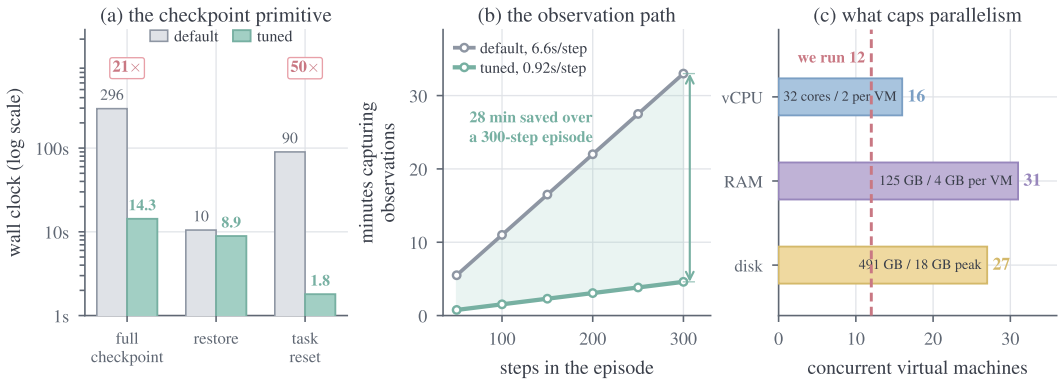

Figure 7: The engineering that makes mid-episode search affordable. **(a)** Checkpoint, restore and inter-task reset, before and after disk-cache and golden-image tuning; log axis. **(b)** The observation path dominated per-step latency until it was cut to under a second, which is worth roughly half an hour of wall clock on a single long episode. **(c)** With that in place the binding constraint on throughput is host vCPU, not checkpoint storage.

Experiments run on a single host with 32 vCPUs, 125 GB of RAM and 491 GB of disk, hosting 12 concurrent guests at 2 vCPUs and 4 GB each. Guest CPU model must be pinned (`-cpu Westmere`); the host-passthrough and modern named models hang on restore because the snapshot was taken under a different feature set. Checkpoint storage is not what binds at this scale: a worker holds a mean of 12.5 GB and peaks at 18 GB of snapshots, so the disk would allow more concurrency than the host's vCPUs do (Figure 7c). It becomes the binding resource once fan-out grows, at which point snapshots need pruning.

Screenshot capture was initially 6.6s per step, of which 5.9s was PNG compression at `optimize=True`; at `compress_level=1` the same compression costs 0.17s and end-to-end capture drops to 0.92s. At 300 steps that single line cost roughly half an hour of wall clock per episode.

Table 4: Checkpoint and restore cost. The disk cache mode is the dominant factor; `cache=unsafe` is safe for this use because a snapshot interrupted by a crash is discarded and retaken rather than trusted.

Operation	OSWorld	OSWorld-V2
full checkpoint, `cache=writeback`	295.7s	–
full checkpoint, `cache=unsafe`	14.3s	12.5s
restore	10.5s	8.9s
per-task reset from golden snapshot	1.8s	–
cold boot (reference)	28.7s	–
cold boot, 12 concurrent	113–131s	–
screenshot capture, before optimisation	6.6s	
screenshot capture, after	0.92s	

Read against the operations it enables, the tuned checkpoint is cheap in the only sense that matters. A branch on these benchmarks runs for tens of model calls and many minutes; paying about ten seconds to save the state it starts from, and about the same to return to it, is a rounding error against the segment it avoids re-running. The per-task reset is the other half of the story and is what makes

repeated sampling affordable at all: restoring a golden snapshot is roughly fifty times faster than booting a guest, and the saving is paid on every rollout of every arm, not only on the search arms.

C VERIFIER ITERATION

Every round followed the same protocol: score the current prompt on the labelled pool, read each misjudgement it made, and add the narrowest clause that would have fixed the largest cluster of them. We report the trajectory as stages rather than as a prompt-by-prompt changelog, since the individual edits are not the contribution. Two features of it are worth noting. Almost all of the movement is in recall on the failure class — overall accuracy rises by eight points while fRec rises by thirty — which is the asymmetry §5.1 builds on. And strictness is not monotonically good: the strictest stage has the best fRec of any, but pays for it in recall on the success class, so in selection it would have discarded correct rollouts. One round is kept in the table as an explicit negative: a broadly worded "non-visual evidence is admissible" clause re-introduced a false positive, and the narrow rule that replaced it is what shipped. The final prompt was frozen before any number reported in this paper was produced, and never re-tuned against OSWorld-V2, whose tasks it had not seen at freeze time. Prompts 5 and 6 give the endpoints verbatim.

Table 5: Verifier stages on the labelled `hard-50` pool (the 50 OSWorld tasks with the deepest milestone plans, ground truth the execution checker). *fRec* is recall on the failure class — the fraction of genuinely failed rollouts the verifier calls failed, which is the quantity best-of-N selection consumes. Rows are cumulative except the one marked *(neg.)*, which was reverted.

Stage	what it adds	accuracy	precision	recall	F1	fRec
baseline	published binary judge	0.667	0.714	0.806	0.758	0.412
+ anti-leniency	structured sub-requirements	0.708	0.815	0.710	0.759	0.706
+ frame citation	plus an infeasibility rule	0.688	0.786	0.710	0.746	0.647
+ mechanism / quantifier	the strictest stage	0.750	**0.880**	0.710	0.786	**0.824**
+ devil's advocate	self-check before verdict	0.750	0.852	0.742	0.793	0.765
(neg.) broad off-screen	reverted	0.729	0.821	0.742	0.780	0.706
ours (frozen)	narrow off-screen artifact rule	**0.750**	0.828	0.774	**0.800**	0.706

D FULL PER-ARM RESULTS

This appendix lists every arm at full precision, including the reference rows the main text only quotes: the worst of the three rollouts in a pool, their mean, and the `pass@3` ceiling. Together they separate how much of a result comes from the pool of candidates and how much from the rule that reads it.

Table 6: Main results, mean graded score; Figure 3 plots the graded columns. The single-rollout row is the executor on its own, and best-of-3 is the standard selection baseline that recent computer-use systems scale with (Brown et al., 2024; Gonzalez-Pumariega et al., 2026). Headroom is the fraction of the `pass@3` gap recovered; relative cost counts environment steps against a single rollout (OSWorld then OSWorld-V2) and charges the tree for every branch it ran, not only the one it kept.

	OSWorld		OSWorld-V2		
Arm	graded	headroom	graded	headroom	rel. cost
single rollout (OpenAI, 2026a)	0.6522	0.0%	0.2119	0.0%	1.0×
best-of-3 (Gonzalez-Pumariega et al., 2026)	0.7018	34.0%	0.2752	66.1%	3.2/3.1×
our harness	**0.7111**	**40.2%**	**0.3142**	**106.9%**	3.6/4.5×
pass@3	0.7989	100%	0.3076	100%	3.2/3.1×

Table 6 adds the two columns Figure 3 has no room for. The headroom column is the one to read. It normalises each arm by the gap between a single rollout and `pass@3`, which makes the two benchmarks comparable even though they differ by more than a factor of two in absolute score. Selection converts about a third of that gap on OSWorld and two thirds on OSWorld-V2; our harness

converts more than all of it on OSWorld-V2. Headroom above 100% is arithmetically unavailable to any rule that merely picks among the three rollouts, so that entry is the same fact as clearing the judge-free bound, stated as a fraction. The relative-cost column charges the tree for every branch it ran rather than for the one it kept, so it is an upper bound on what search costs: read that way the tree still spends less than half again what best-of-3 spends.

Table 7: OSWorld, `hard-50`. All arms use `gpt-5.4` at 50 steps.

System	graded	soft success	environment steps
worst of 3 rollouts	0.5413	–	2181
single rollout	0.6394	64.6%	740
best-of-3	0.7435	75.0%	2181
tree search	**0.7851**	**79.2%**	2264
pass@3	0.7852	–	2181

Table 7 is the `hard-50` pool on OSWorld, where the harness was developed. Two rows carry more information than the headline. The worst of three rollouts falls a tenth of a point below a single rollout, which is the variance a selector exists to exploit. And the tree lands on the `pass@3` ceiling to within a thousandth, while best-of-3 stays four points under it. On this pool mid-episode resumption is already doing what no rule selecting among the same three episodes could do, at a step cost within a few percent of best-of-3.

Table 8: OSWorld-V2 with `gpt-5.4`, both experiment stages. The $n = 106$ stage samples three flat rollouts per task; the $n = 102$ stage adds the search arms and excludes four tasks whose environment setup fails deterministically on our host. *Tree, forced* is the variant of runnning the tree machinery even on one-milestone plans.

System	graded	soft success	n
worst of 3 rollouts	0.0930	–	106
mean of 3 rollouts	0.1874	–	106
single rollout	0.2039	19.8%	106
best-of-3	0.2648	23.6%	106
pass@3	0.2960	28.3%	106
single rollout	0.2119	–	102
best-of-3	0.2752	–	102
tree, forced	0.2965	–	102
tree search (ours)	**0.3142**	–	102
pass@3	0.3076	–	102

Table 8 gives both OSWorld-V2 stages. The spread inside a single pool is what makes the benchmark worth searching: the worst of three rollouts scores less than half what the best of the same three scores, so the candidates genuinely differ and there is something for a verifier to find. Our harness ends above the `pass@3` row of its own stage.

Table 9: OSWorld multi-milestones subset: the 72 tasks whose plan has two or more milestones. *Tree pass@k* is the maximum over the branches the tree actually explored, i.e. the ceiling of the tree's own selection step.

System	graded
single rollout	0.5278
best-of-3	0.5694
tree search	**0.6111**
tree pass@k	0.6250
flat `pass@3`	0.6667

Table 9 restricts attention to the OSWorld hard tasks a single rollout fails, which are the only tasks where test-time compute can change anything. It separates two ceilings that are easy to conflate.

Flat `pass@3` is what a perfect verifier would obtain over three independent episodes; *tree pass@k* is the best branch the tree actually produced, that is, the ceiling of its own selection step. The tree realises almost all of its own ceiling. What limits it on this subset is therefore the set of trajectories its branches reach, not the verifier choosing among them: the per-table form of the argument in §5.2.

Table 10: Plan depth distribution. OSWorld-V2 plans are deeper in every bucket.

Milestones per task	0	1	2	3	4	≥ 2 share
OSWorld-V2 ($n = 102$ planned)	5	6	41	40	10	89.2%
OSWorld ($n = 361$)	76% single-milestone					24%

Table 10 is why the two benchmarks respond differently to the same harness. Nearly nine in ten OSWorld-V2 plans have at least two milestones, against roughly one in four on OSWorld. A tree cannot branch where there is no boundary to branch at, so the newer benchmark offers several times the opportunity at identical settings, and the ordering in Table 6 follows from the task distribution rather than from anything tuned per benchmark.

E ACTION SPACE

Table 11: Action space, OSWorld-V2, single rollout, everything but the exposed tool set held fixed ($n = 106$). Deltas are against GUI-only with a 10,000-sample paired bootstrap. Strict success is identical in all three arms, which is itself the finding: the extra channel moves partial credit, not the count of perfectly solved tasks.

Action space	tools exposed	graded	strict	mean steps	Δ vs. GUI
GUI only	`computer`	0.3782	13.2%	122.6	–
shell only	`run_bash`	0.4021	15.1%	59.8	+0.024 (73% conf.)
hybrid	`computer` + `run_bash`	**0.4458**	14.1%	93.5	**+0.068** (99% conf.)

Reading the table. Three things separate cleanly. The hybrid arm is the only one whose interval against GUI-only excludes zero, so it is the only arm the data supports calling better. The shell-only arm is nominally ahead of GUI-only but not distinguishable from it; nothing here licenses the claim that a terminal is simply the better channel, which is the claim the mixed-system literature is often read as making. Steps separate far more sharply than score does: the shell arm finishes in roughly half the steps of the GUI arm, and the hybrid arm sits between the two, having spent its budget where it chose to rather than where it was forced to. Strict success, finally, is identical in all three arms. The extra channel moves partial credit rather than the count of perfectly solved tasks, which is what one expects if what it buys is getting further into a task rather than finishing more of them outright.

Keeping the observation fixed. A shell-only agent that never receives a screenshot after acting would differ from a GUI agent in what it can *see* as well as in what it can *do*, and the two effects would be inseparable. We therefore re-attach the current screen after each shell command in the shell-only condition, which is the same information the hybrid agent obtains by taking a screenshot action.

Keeping the channels separate. The comparison in §5.1 is only meaningful if the shell cannot be used as a second GUI channel. Nothing prevents a model from writing `python3 -c "import pyautogui; pyautogui.click(...)"`, which would silently turn the shell-only arm back into a GUI arm and make the hybrid arm's routing vacuous. The prompts in Appendix J ask for the right behaviour, but we do not rely on them: every command is screened before it reaches the guest. Synthetic-input libraries have no legitimate non-GUI use and are refused outright; `xdotool` is refused only for its input verbs, since its query verbs are an ordinary way to inspect window state. The guard runs per command rather than per tool call, so a compound line cannot smuggle a click past it, and heredoc bodies are recursed into rather than tokenised as shell — the body of `python3 - «'PY'` is Python, and reading it as shell turns any string literal that happens to name a program into a spurious refusal. A refusal is returned to the model as tool output, so it can recover by taking the GUI action instead; refusals are also logged, and across the runs reported here they are rare enough that no arm's step budget is meaningfully affected by them.

F WHICH PUBLISHED SYSTEMS CAN BE RE-RUN, AND WHICH CANNOT

A row that differs from ours in the architecture *and* the policy measures their sum. To put a published system in Table 1 below the rule we need to replace its policy with ours and leave everything else as its authors wrote it, which requires runnable code and requires that the policy be the only learned component in the loop. Table 12 applies that test to every system we are aware of that reports on either benchmark. Exactly one passes.

Table 12: Why each published system is quoted rather than re-run. Scores are as published, on whichever benchmark and metric the source reports.

System	reported	why not re-run
the policy is the system		
UI-TARS-72B (Qin et al., 2025)	0.246	Grounding and decision share one set of weights, so there is no policy to replace and no architecture left over.
OpenCUA-72B (Wang et al., 2026)	0.450	As above.
EvoCUA (Xue et al., 2026)	0.567	As above.
Qwen-CUA (Lu et al., 2026)	0.484	As above.
MiniMax M3, Kimi K2.6, Qwen3.7-Plus (MiniMax, 2026; Moonshot AI, 2026; Qwen Team, 2026)	0.73–0.75	As above; these are the leaderboard rows quoted in Table 1.
a second trained component sits in the loop		
Agent S3 (Gonzalez-Pumariega et al., 2026)	0.726	Coordinates come from a separately trained grounder, whose accuracy is its own research problem (Zhang et al., 2025); swapping the planner would leave that grounder in place and change two things at once.
GTA1 (Yang et al., 2026c)	—	Re-ranks with a trained vision-language reward model, so the selector is not ours either.
VLAA-GUI (Han et al., 2026)	0.775	Modular recovery and search around a grounding model in the same position.
no runnable release		
StructAgent (Wu et al., 2026)	0.789	No code at the time of writing.
StateAct (Yang et al., 2026b)	0.616	No code at the time of writing.
incompatible with our execution substrate		
OS-Symphony (Yang et al., 2026a)	0.658	Orchestration runs on a second machine alongside the task VM, which the checkpoint substrate of Appendix B does not snapshot as one unit.
re-run		
CoAct-1 (Song et al., 2026)	0.608	Grounding comes from the provider's computer-use tool, so the policy is the only learned component; code is released, and the OSWorld-V2 maintainers ship a port of it.

The middle block is the one worth dwelling on, because it is a statement about the field rather than about those papers. Grounding a click is a trained capability with its own ceiling — the strongest dedicated models still sit far below saturation on hard grounding benchmarks (Zhang et al., 2025), and the gap widens on the long-tail interactions that click-centric evaluations under-represent (Zhang et al., 2026). A harness built around a grounder therefore carries a component whose quality is independent of the policy, which is exactly what makes the substitution we need impossible. It is also why our hybrid action space (§5.1) is worth measuring separately: the shell channel reaches state without grounding anything.

G RE-RUNNING COACT-1 UNDER OUR EXECUTOR

CoAct-1 (Song et al., 2026) is the one published desktop system we could re-run without confounding the comparison; Appendix F works through the alternatives one at a time. CoAct-1 takes its grounding from the provider's computer-use tool, so replacing its policy with ours leaves the orchestration, the prompts and the task set exactly as its authors wrote them.

The released harness targets an earlier computer-use deployment, and six things had to change before it ran at all. None touches its method; all six are interface debt between a 2025 client and a 2026 endpoint, and we list them because a reader is entitled to know how much of the reported gap could be ours rather than theirs. **(i)** The current tool returns an ordered `actions` list and leaves the singular `action` field null, so a client written against the preview tool executes nothing; we replay the list in order. **(ii)** Screenshots are sent with `detail: "original"`, without which the endpoint rescales them and returns coordinates in the rescaled frame, so every click misses. **(iii)** The preview tool echoed a safety-check field the current one rejects; it is stripped. **(iv)** Arrow keys arrive under their browser names and were lowercased rather than translated, which `pyautogui` refuses. **(v)** The guest image exposes a generic command endpoint rather than the dedicated script route the client expects, so the programmer's code is shipped over the former — the same channel our own shell tool uses, with the timeout the client asks for. **(vi)** The harness constructs its environment with an HTTP proxy enabled; our runs do not use one, and leaving it on failed every web task with a connection error before the agent could act. The first four would drive the score to near zero and the last would remove a domain, so a re-run that skipped them would not be evidence about anything.

What remains fixed is everything the comparison is about: the same 361 tasks, the same executor at the same reasoning effort, the same virtual machine image, and CoAct-1's own orchestrator, prompts, sub-agent decomposition and step budgets. Its score under our executor is close to the number its authors report with theirs, which is the outcome that makes the port credible: the substitution moved the policy and little else.

H EVIDENCE MECHANISM DIAGNOSTICS

The mechanism only works if agents file evidence unprompted and if what they file can be read back. Table 13 reports both, over every OSWorld-V2 tree branch we ran. Branches use the channel heavily and asymmetrically — around a dozen read-only commands against a handful of proof screenshots each — which is the behaviour the design is for: a command answers a question about machine state that a picture can only suggest. Almost all bundles parse without repair, so the verifier is reading structured evidence rather than salvaged text. The last row is the one that matters for selection: on contested decisions, where the branches disagree about which of them did better, the verifier sides with the branch's own best claim far more often than a random pick would.

Table 13: What the agents actually filed, over 679 OSWorld-V2 tree branches.

Quantity	value
bundles parsing without repair	91.9%
proof screenshots filed	1859 (2.7 per branch)
read-only commands run	8225 (12.1 per branch)
contested selections	61
verifier agrees with the branch's own best claim	75.4% (random: 55.2%)

I WHERE THE GAIN LIVES, DOMAIN BY DOMAIN

The aggregate numbers hide that most domains are untouched by any of this. On `hard-50` five of ten domains score identically under all three arms — they are either already solved (`os`, `vs_code`, `lo-writer`) or systematically unsolved (`gimp`). Four are lifted by plain best-of-3, and one, `lo-impress`, is lifted only by the tree: at the domain mean, best-of-3 leaves it exactly where a single rollout does, and only resuming from a mid-episode checkpoint moves it. This is the per-domain form of the exploration-bound argument in §5.2.



Figure 8: Per-domain mean graded score on `hard-50`. **Left:** one dumbbell per domain; markers coincide where the arms tie, which is what *no change* marks. **Right:** the same information as a count of domains. The single domain only the tree lifts (`lo-impress`, 0.484 → 0.817) is the one an earlier error analysis had singled out as unlikely to yield to resampling.

Table 14: The numbers behind Figure 8. Arms are a single rollout, best-of-3 with the frame verifier, and tree search, all `gpt-5.4` on `hard-50`.

Domain	n	single rollout	best-of-3	tree search
`chrome`	6	0.656	**0.823**	**0.823**
`gimp`	4	0.500	0.500	0.500
`libreoffice_calc`	7	0.571	**0.714**	**0.714**
`libreoffice_impress`	6	0.484	0.484	**0.817**
`libreoffice_writer`	3	0.999	0.999	0.999
`multi_apps`	12	0.500	**0.667**	0.666
`os`	3	1.000	1.000	1.000
`thunderbird`	2	0.500	**1.000**	**1.000**
`vlc`	2	0.926	0.926	0.926
`vs_code`	3	1.000	1.000	1.000

J PROMPTS

Every listing below is generated directly from the source that produced the runs (`scripts/export_prompts.py`), so what is printed here is byte-identical to what the models were shown, modulo re-wrapping to the column width.

J.1 ACTION SPACE

The shell is exposed as an ordinary function tool rather than as an instruction to open a terminal, which matters for the comparison in §5.1: a GUI agent told to open a terminal is still a GUI agent, and its shell use is entangled with its ability to find and drive the terminal application. Here the command executes out of band through the environment controller and the screen does not change. Prompt 1 is the tool description all shell-enabled arms see.

Prompt 1: `run_bash` tool description, used by the CMD and HYBRID arms.

```
Run ONE shell command on the machine and return its stdout/stderr. The command runs out of
band: no terminal window opens and the screen does not change by itself, though the
command's effects are of course real. Use it to DO work as well as to check it -- create,
move, rename, edit, convert or inspect files, install packages, query databases, launch
applications. It is a NON-GUI channel: commands that synthesise keyboard or mouse input
(pyautogui, pynput, xdotool key/type/click, xte, ydotool) are refused.
```

The two arms then differ only in the guidance appended to the system prompt. Prompt 2 removes the mouse and keyboard entirely; Prompt 3 keeps both and asks the model to choose per sub-goal. Both spell out that the shell is a non-GUI channel, because the failure mode we had to design against is the model reaching the GUI *through* the shell — scripting `pyautogui` or `xdotool key` would silently turn the CMD arm back into a GUI arm and void the comparison. The prompt is only the polite half of that guard; the enforcing half is a static check on every command before it runs (Appendix E).

Prompt 2: CMD-only guidance.

```
- **You have no mouse and no keyboard.** The ONLY way you can act on this machine is
  run_bash(command, purpose), which runs one shell command. You still receive a screenshot
  every turn so you can see the state of the desktop, but you cannot click, type, scroll or
  drag. Anything you want to change must be changed by command.
- Work with the tools the box provides: standard Unix utilities, python3, and the
  command-line interfaces of the installed applications. For file formats without a CLI,
  python3 -c with the appropriate library is usually the shortest route.
- **Do not try to reach the GUI through the shell.** Scripting synthetic input -- pyautogui,
  pynput, xdotool key/type/click, xte, ydotool -- is refused. Change the machine's state
  directly instead: edit the file, run the converter, call the CLI.
- If a task genuinely requires the GUI, get as close as you can by command and say clearly
  in your final message what remained impossible and why.
```

Prompt 3: HYBRID guidance. The routing decision is left to the model, per sub-goal, with the trade-off stated in terms of what the success criterion depends on rather than as a list of task types.

```
- Besides the mouse and keyboard you also have run_bash(command, purpose), which runs one
  shell command out of band. No terminal window opens and the screen does not change by
  itself, but the command's effects are real.
- **Choose the cheaper correct tool for each sub-goal.** The shell is far better at bulk and
  exact work: renaming or moving many files, checking that a file exists at an exact path,
  reading a value out of a file that is not on screen, editing text precisely, converting
  formats, installing something. The GUI is better -- and sometimes the only option -- when
  the task is about what is rendered: reading a chart or an image, setting a control in an
  app with no scriptable interface, anything whose correctness is visual.
- The two mix freely. A common good pattern is to do the bulk edit by command and then open
  the result in the GUI to confirm it looks right.
- Prefer whichever gets the task DONE correctly. Doing office-document work by command is
  allowed and often better, as long as the resulting file is what was asked for.
- **The two channels stay separate.** run_bash is for non-GUI work; it will refuse to
  synthesise keyboard or mouse input (pyautogui, pynput, xdotool key/type/click, xte,
  ydotool). When you want to click or type, use the mouse and keyboard you already have.
```

J.2 EVIDENCE PROTOCOL

Prompt 4 is what turns a rollout into an evidence bundle. The two tools it introduces are deliberately asymmetric: `capture_evidence` pins a frame the agent believes is probative, while `run_readonly_bash` may read the machine but never change it, so that the bundle cannot become a second, hidden action channel. The read-only restriction is enforced by a command whitelist, not by the prompt alone.

Prompt 4: Evidence-collection protocol appended to the system prompt in all evidence arms.

```
Besides operating the computer you must COLLECT EVIDENCE that your work is correct. Two
extra tools are available for this and they never touch the screen:
- capture_evidence(claim): pins the CURRENT screen as a numbered proof image and returns a
  proof_id such as "IMG-3". Call it at the exact moment something is visible that proves
  part of your work -- a value you just read, a dialog showing the right setting, the
  finished document. The screen is not changed and no window is opened.
- run_readonly_bash(command, purpose): runs one shell command on the machine OUT OF BAND and
  returns its stdout/stderr. **No terminal window appears and the screen does not change.**
**run_readonly_bash IS FOR READING ONLY. It must NEVER be used to perform the task.** Do not
  write, move, delete, rename or chmod anything, do not install anything, do not launch GUI
  programs, and do not use it to take a shortcut around work you were asked to do with the
  mouse and keyboard. Reading is allowed and encouraged: ls, cat, head, stat, grep, find,
  sha256sum, python3 -c with a read-only script, unzip -l, sqlite3 with a SELECT, and so on.
  Any command that mutates the machine is a violation and invalidates your evidence.
```

J.3 VERIFIER

§5.1 compares a baseline judge prompt against ours. Prompt 5 is the baseline: a binary trajectory judge written to the consensus recipe that published GUI-agent reward models use — the first frame plus the last seven, click points circled, the full thought-and-action history, the agent's final message, and rules covering explicit answers, grounding, blocked tasks and long trajectories. It is a reasonable

prompt, and its failure mode is the one this literature already reports: LLM judges of agent trajectories are systematically lenient (Lù et al., 2025; Pan et al., 2024). Prompt 6 lists only the clauses our version adds; everything else is unchanged, which is what makes the comparison a prompt ablation rather than a rewrite. The additions cluster around one theme, which is the one Figure 4 (top) shows the baseline is weakest on: they make it harder to return SUCCESS on the strength of a plausible narrative, and easier to accept an off-screen artifact as proof when the requirement genuinely is a file. Table 5 gives the intermediate rounds.

Prompt 5: Baseline binary judge prompt.

```
[ROLE]
You are a reward model evaluating a GUI agent operating across diverse platforms (e.g.,
Desktop, Web, Mobile). Your job is to determine whether the agent's trajectory successfully
completes the user's task.

[INPUTS]
You will receive varying combinations of the following evidence:
1. User Instruction: The task to be completed.
2. Visual States: Screenshots from selected steps of the trajectory (e.g., the final few
states, or a mix of initial and final states). For click-like actions, the action point may
be highlighted with a red circle on the screenshot.
3. Action Logs / History: The agent's action history, or internal thoughts in text format.

[EVALUATION GOAL]
Your task is to synthesize all the evidence above and determine whether the agent reasonably
completed the task according to the user's instruction. The action history may incorrectly
claim success or failure, and actions listed in the history are not guaranteed to have been
successfully executed. You must judge task completion based on all available evidence.

[NOTES / CONSENSUS RULES]
1. Explicit answer requirement:
- For general tasks (e.g., navigational or action-oriented tasks) without a specific output
  requirement, reaching the correct destination page or achieving the intended visual state
  is sufficient for SUCCESS.
- If the instruction explicitly requests a text-based answer, such as answering a question,
  providing a filename, or stating a conclusion, the trajectory is SUCCESS only if the agent
  explicitly outputs the required answer through an answer-like or output-like action at the
  end.
- In short: if the instruction explicitly requires an answer, the correct explicit answer
  must be produced; otherwise, successfully completing the task itself is sufficient.

2. Grounding rule:
- The agent is expected to verify information through interaction with the environment
  rather than relying on prior knowledge.
- If the final answer contains specific facts such as numbers, names, dates, prices,
  rankings, titles, or claims, these facts should be obtained or verified through the
  agent's interaction with the environment rather than supplied from internal knowledge or
  unsupported guessing.
- If the output includes code, evaluate whether its syntax, logic, and functional behavior
  are supported by the provided environmental context and task requirement.
- Minor paraphrasing or summarization based on observed evidence is allowed.
- Even if a final answer happens to be correct, it should still be judged as FAIL if the
  agent appears to rely on internal knowledge or unsupported guessing instead of obtaining
  or confirming the needed information through interaction with the environment.

3. Blocked / impossible rule:
- If the task fails because it is persistently blocked by external constraints, the final
  judgment must be FAIL, even if the agent behaved logically.
- Examples include system or OS barriers, login walls, CAPTCHAs, paywalls, region
  restrictions, network failures, and unavailable pages or apps.

4. Long trajectory rule:
- A trajectory may be long and may contain intermediate mistakes, detours, retries, or
  loops.
- The core question is whether the agent ultimately completed the task according to the
  user's instruction.
- Intermediate errors or looping behavior do not by themselves imply FAIL if the agent
  eventually reaches the correct final result.

[OUTPUT FORMAT]
Thought: Explain briefly why the task succeeded or failed.
Judge: SUCCESS or FAIL

Output exactly in this order: Thought, Judge.
```

Prompt 6: Clauses our verifier prompt adds to the baseline. Rule numbering continues from the baseline's four.

```
5. Calibration / anti-leniency rule:
- LLM judges are systematically over-optimistic about GUI task completion. Most judge errors
  are false positives: FAIL trajectories labelled SUCCESS.
- Do not accept the agent's own claim of success at face value. The agent's final message is
  an unreliable, self-serving report.
```

```
- A plausible-looking sequence of actions is NOT evidence of success. Only observable state
  in the screenshots is evidence.
- If the decisive final state is not visible in any provided screenshot, the correct verdict
  is FAIL, not SUCCESS.

6. Evidence citation rule:
- Screenshots are labelled "FRAME k". Every [CONFIRMED] item must cite the frame index whose
  pixels show the requirement being met, e.g. "[CONFIRMED] FRAME 4 shows the cell B2
  containing 42". A claim you cannot attach to a frame is not [CONFIRMED].

7. Infeasible-task rule (HIGH FREQUENCY - read carefully):
- This benchmark deliberately includes tasks that CANNOT be accomplished with a mouse, a
  keyboard and the software installed on the machine: the application genuinely lacks the
  requested feature, the requested object does not exist, the request is self-contradictory,
  or it requires acting outside the machine.
- For such tasks, the CORRECT agent behaviour is to refuse and state that the task is
  infeasible. Therefore:
        * task genuinely infeasible AND agent explicitly refused / declared it infeasible ->
      SUCCESS
        * task genuinely infeasible BUT agent produced a workaround or pretended to finish
      -> FAIL
        * task genuinely feasible BUT agent gave up or declared it infeasible -> FAIL
- A workaround is NOT a completion. If the user asked for something the named application
  cannot do and the agent achieved a similar effect by another route, that is FAIL, not
  SUCCESS.
- Before you decide, you MUST explicitly answer this question in your Checklist as item 0:
  "Is this task actually achievable with the installed software using only mouse and
  keyboard, in the exact manner the user asked for? YES / NO, and why."
- Be careful in BOTH directions. Merely difficult, obscure, or many-step tasks are feasible.
  "The agent could not find the menu item" is not the same as "the feature does not exist".

8. Persistence rule (desktop-specific, high frequency):
- Many desktop tasks require the result to be PERSISTED, not merely displayed.
- If the task edits a document/spreadsheet/presentation/source file, SUCCESS requires
  evidence that the file was saved: no unsaved-changes marker in the title bar, an explicit
  Ctrl+S, or a save dialog that was confirmed. A visible "Keep current format" / "Use ...
  Format!" dialog that was never confirmed means the save did NOT happen.
- If the task changes a setting or preference, SUCCESS requires evidence the change was
  applied (OK / Apply pressed, dialog closed, new value visible), not merely that the dialog
  was opened.
- If the task requires creating a file at a path, SUCCESS requires evidence the file exists
  at that exact path with that exact name.

9. Specified-mechanism rule:
- If the instruction names the tool, feature, application or channel to use ("using only the
  command line", "use Impress's Summary Slide feature", "inside the app, not the launcher",
  "by installing the X extension", "within GIMP"), then achieving the same end state by any
  OTHER mechanism is FAIL, no matter how good the result looks.
- Watch specifically for: doing GUI work via a shell script when the task asked for the GUI;
  doing shell work via the GUI when the task said "command line"; faking a formatting change
  by drawing a shape, covering something up, or hard-coding a value instead of using the
  feature that was named.

10. Correct-target rule:
- Verify the agent acted on the object the USER referred to, not on a convenient look-alike.
- If the instruction says "my photo on the desktop", "the current file", "this
  presentation", "all .doc files in the current directory", confirm from the frames that the
  agent located and operated on that exact object, and on ALL of them when the instruction
  is plural or says "all".
- An agent that could not find the intended target and silently substituted another one is
  FAIL.

11. Quantifier rule:
- "all", "every", "each", and explicit counts are mandatory. If the instruction says "all
  .doc files" and the evidence shows one file processed, you must confirm from the frames
  that one file is genuinely all of them; otherwise FAIL.

12. Off-screen artifact rule (narrow, re-balancing):
- Some goal states are files, and files are not visible on screen. Refusing to accept any
  evidence except a picture of the goal object creates false negatives.
- The following DO count as [CONFIRMED] evidence that a file was created or modified:
    (a) a Save / Export / "Save As" dialog frame showing the correct filename and the
    correct target directory, followed by a frame in which that dialog is gone and no error
    is shown;
    (b) a shell command in a terminal frame whose printed output demonstrates the result
    (`ls -l`, `cat`, `file x.png`, a script's output);
    (c) the application's title bar or recent-files list showing the file at the required
    path;
    (d) an application reloading the file and displaying the new content.
- The following still do NOT count: the agent asserting it did something; an action in the
  log with no subsequent frame consistent with its effect; a save dialog that is still open
  in the final frame; a format-warning dialog ("Use ... Format!") that was never confirmed.

13. Scope-of-instruction rule (re-balancing):
- Judge ONLY the requirements the instruction actually states. Do not invent extra criteria.
- Cosmetic deviations the instruction never mentioned -- capitalisation of text the user did
  not specify, extra whitespace, styling, column widths, which window has focus at the end
```

```
  -- are NOT grounds for FAIL.
- Anything the instruction DID state (a named mechanism, an exact path, a quantifier, an
  explicit answer) remains mandatory.
- If you are about to fail a trajectory for a defect the user never mentioned, that is a
  false negative: re-read the instruction and check whether it is truly required.

[OUTPUT FORMAT]
Checklist: Decompose the user instruction into every atomic, mandatory sub-requirement,
numbered. For each one write exactly one of:
    [CONFIRMED] <what you literally see in which screenshot proves it>
    [NOT CONFIRMED] <what evidence is missing>
    [AMBIGUOUS] <why it cannot be decided>
Devil's advocate: Before you commit, argue the OPPOSITE of your leaning verdict in 1-3
sentences. If you are leaning SUCCESS, state the strongest concrete reason this trajectory
might actually have failed. If you are leaning FAIL, state the strongest concrete reason it
might actually have succeeded. Then say whether that counter-argument is refuted by the
screenshots.
Thought: Explain briefly why the task succeeded or failed. The LAST frame is the
authoritative final state; an intermediate frame showing the goal does not count if the last
frame shows it was undone or never committed.
Judge: SUCCESS or FAIL

A verdict of SUCCESS is only allowed if EVERY mandatory sub-requirement is [CONFIRMED]. Any
[NOT CONFIRMED] or [AMBIGUOUS] mandatory sub-requirement forces FAIL.
Output exactly in this order: Checklist, Devil's advocate, Thought, Judge.
```

J.4 MILESTONE DECOMPOSITION

The planner is called once per task, on the initial screenshot, and its output fixes the tree's branch points for the whole episode. Two of its instructions carry most of the weight. The first is that a milestone names a *state* rather than an operation, which is what makes a boundary something a verifier can score at all. The second is the standalone requirement: because a resumed branch inherits the machine but not the conversation, anything one milestone discovers and the next one needs has to be written into the screen or stated aloud before the boundary.

Prompt 7: Milestone planner. Called once per task on the initial screenshot; its milestone list defines where the tree may branch.

```
You are the planning module of a computer-use agent operating a real Ubuntu 22.04 desktop at
1920x1080. A separate execution agent will carry out your plan by controlling the mouse and
keyboard.

You are given the user's task and a screenshot of the machine's initial state. Produce a
plan that decomposes the task into an ordered list of MILESTONES.

[WHAT A MILESTONE IS]
A milestone is a verifiable STATE of the machine, not an operation.
    GOOD: "LibreOffice Calc is open with /home/user/report.xlsx loaded and the Sheet1 tab
  active."
    GOOD: "The Chrome tab shows the search results page for 'python asyncio', and the title
  of the first non-ad result has been read."
    BAD:  "Click the File menu."            (an operation, not a state)
    BAD:  "Open the file and then edit it and then save it and then close it."  (too coarse)
Each milestone should be a meaningful chunk of work -- roughly 5 to 15 GUI actions -- not a
single interaction. Prefer 2-4 milestones in total; a genuinely simple task may need only
ONE milestone, and that is fine and often correct. Never emit more than 6.
Over-decomposition is actively harmful here: milestones do not share memory (see below), so
every extra boundary is another place for context to be lost, and the selector that picks a
winner at each boundary is far less accurate mid-task than it is at a finished end state.
Split a task only where it genuinely has separate phases; never manufacture a boundary to
create one. When in doubt, merge two adjacent milestones rather than splitting one.

[HARD RULES]
1. Every milestone MUST require at least one mouse/keyboard action. Never emit a milestone
that is purely cognitive, such as "report the answer to the user", "decide which value is
largest", or "summarise the findings". If the task needs an answer stated to the user, fold
that into the same milestone that gathers the information, e.g. "... and the agent has
stated the total revenue value it read from cell D12."
2. Milestones DO NOT share memory. The executing agent starts each milestone with a blank
context and sees only your milestone text plus the current screen. Therefore:
      - Each milestone must restate enough context to be executed standalone: what state the
     machine is in when it begins, what has to be done, and what state it must end in.
      - If milestone i discovers information that milestone i+1 needs (a filename, a number,
     a row index, a URL), milestone i's description MUST instruct the agent to state that
     information in plain text before it stops, and milestone i+1's description must say
     that this information will be supplied to it.
      - Never write "the file from the previous step" or "the value you found". Either
     describe how to re-identify it, or make milestone i state it explicitly.
3. "stop_position" is where the agent must LEAVE the machine, because the next milestone
starts from exactly that screen with no memory of how it got there. Name the application,
the window, the file and the view that must be on screen when the agent stops, and any
saving or closing that must happen first -- for example "the spreadsheet has been saved with
```

```
Ctrl+S and Sheet1 of budget.xlsx is the active tab in LibreOffice Calc". Choose the position
that makes the NEXT milestone easiest to begin. For the last milestone, state the position
that best displays the finished work.
4. Do not plan verification-only milestones ("check that the file was saved"). Fold the
verification into the milestone that performs the work.
5. Assume the sudo password is "password" and the home directory is /home/user.
6. NEVER reinterpret, tighten, loosen or "clean up" the user's requirements. Copy every
literal detail of the request through into your milestone text EXACTLY as the user wrote it:
strings to type or search for, numbers and thresholds, file names and paths, the named
application, how many items are affected, and the requested output format. Do not turn "all
the places that say X" into "the exact word X"; do not turn "modified 30 days ago" into
"older than 30 days"; do not turn "in VS Code" into "in a text editor". You are deciding
WHICH PART of the work each milestone covers -- you are not restating WHAT the work is. If
you cannot express a milestone without paraphrasing a requirement, quote the user's own
words for that requirement instead.

[INFEASIBLE TASKS]
This benchmark deliberately contains tasks that CANNOT be completed with a mouse, a keyboard
and the software installed on the machine. Examples: requesting a feature the application
does not have, asking to contact a person, asking for information that exists only outside
the machine, or asking for something self-contradictory.
If, and only if, you are confident the task is impossible in this environment, set
"feasible": false and explain why in "infeasibility_reason", and return an empty milestone
list.
Be conservative. A task that is merely difficult, obscure, or requires many steps IS
feasible. Web browsing, terminal commands, and installing nothing new are all available.
Default to "feasible": true.

[OUTPUT FORMAT]
Return a single JSON object and nothing else. Angle brackets below describe what to put in
each field; replace them with real values, and never emit the angle brackets themselves.
{
    "feasible": <bool, true unless you are confident the task is impossible on this
  machine>,
    "infeasibility_reason": <str, empty string unless "feasible" is false>,
    "analysis": <str, one or two sentences about the current screen and the overall
  approach>,
    "milestones": [
        {
            "id": <int, 1-based position of this milestone in the list>,
            "name": <str, a short label>,
            "description": <str, self-contained instruction for the execution agent. State
      the starting situation, the work to do, and the exact end state. Written as an
      imperative task, as if it were the whole task the agent was given. Reproduce every
      literal requirement from the user's request word for word; never paraphrase a
      parameter.>,
            "stop_position": <str, the exact place the agent must leave the machine in when
      this milestone is finished -- which application, window, file and view is on screen,
      plus any saving or closing to do first, e.g. "the file has been saved and the Sheet1
      tab of budget.xlsx is showing in LibreOffice Calc">
        },
        ...
    ]
}
Emit as many milestone objects as the task needs, following the guidance above.
```

J.5 SELECTION

Flat best-of-N and the tree use different selection prompts because they answer different questions. At the end of an episode the question is which final state best satisfies the request (Prompt 8). At a milestone boundary it is which state is the best place to *continue* from, which is not the same thing: no candidate has finished the task and none is supposed to have, so a prompt that rewards apparent completion rewards overreach, and a prompt that rewards a declared impossibility rewards giving up early. Prompt 9 scores against the assigned milestone's own end state instead, and treats unrecoverable states and unrequested side effects as defects rather than as progress. We did not run the substitution as a controlled ablation; the separation is a design decision, and both prompts are printed below so it can be judged.

Prompt 8: End-of-episode comparative judge, used for flat best-of-N selection.

```
[ROLE]
You are a reward model selecting the single best trajectory produced by a GUI agent that
attempted the same task several times on identical starting machines.

[INPUTS]
1. User Instruction: the task all candidates attempted.
2. For each candidate: its action history, its final message, and a sequence of screenshots
labelled "CANDIDATE i FRAME k". For click-like actions the click point is marked with a red
circle.

[EVALUATION GOAL]
```

```
Choose the candidate whose FINAL machine state best satisfies the user instruction.

[RULES]
1. Judge observable final state, not effort, not narration, not trajectory length. The
agent's own claim of success is unreliable and self-serving.
2. A candidate that fully and verifiably completed the task always beats one that partially
completed it, no matter how much more work the latter appears to have done.
3. Persistence counts: an edit that was never saved, a dialog that was never confirmed, and
a setting that was never applied are all incomplete.
4. Exactness counts: filenames, paths, cell references and literal strings must match the
instruction exactly.
5. If the task is genuinely infeasible, the best candidate is the one that explicitly
refused and declared it infeasible.
6. If no candidate completed the task, choose the one that got furthest along without
leaving the machine in a broken or misleading state (e.g. prefer a candidate that opened the
right file over one that corrupted a different file).
7. Ties: prefer the candidate with the cleaner final state and fewer destructive side
effects.

[OUTPUT FORMAT]
Analysis: For each candidate, one or two sentences on what its final state actually shows
and which mandatory sub-requirements it satisfies.
Best: <index>

<index> must be a single integer, the 1-based index of the chosen candidate.
Output exactly in this order: Analysis, Best.
```

Prompt 9: Milestone-boundary comparative judge, used inside the tree.

```
[ROLE]
You are a reward model inside a planning agent. A long task has been split into steps.
Several agents each attempted THE SAME SINGLE STEP starting from an identical machine state.
Exactly one of their end states will be kept, and the rest of the task will continue from
it.

[INPUTS]
1. The user's overall request, for context on literal details such as filenames, strings and
numbers.
2. THE STEP THAT WAS ASSIGNED, and the end state that step was supposed to reach.
3. For each candidate: its action history, its final message, and screenshots labelled
"CANDIDATE i FRAME k". For click-like actions the click point is marked with a red circle.

[EVALUATION GOAL]
Choose the candidate whose FINAL machine state is the best place to continue the rest of the
task from. You are not judging whether the overall task is finished -- it is not supposed to
be.

[RULES]
1. Score against THE ASSIGNED STEP's end state, not against the overall request. A candidate
that achieved the assigned end state is correct even though the overall task is unfinished.
2. Judge the observable screen, not effort, not narration, not trajectory length. The
agent's own claim of success is unreliable and self-serving.
3. Persistence counts: an edit that was never saved, a dialog left open over the document, a
setting that was never applied, and a modal still awaiting input are all incomplete, and
they also block whatever comes next.
4. Exactness counts: filenames, paths, cell references and literal strings must match the
user's own wording exactly. A step done neatly on the wrong file or the wrong cell is worse
than a step left unfinished on the right one, because the next step will build on the error.
5. If the step required specific information to be reported, the candidate must actually
state that information in its final message. A candidate that found it but did not say it
has lost it -- the next step runs with a blank memory and cannot see this trajectory.
6. Going beyond the assigned step is not a bonus. Extra work that is consistent with the
user's request is acceptable, but extra work that changes files, settings or windows the
step did not call for is a defect: it was never checked and the following steps do not
expect it.
7. Prefer recoverable states. Between two candidates that both failed the step, choose the
one that left the machine cleaner and closer to the starting point -- an unchanged desktop
beats a half-edited file, a corrupted document or an unsaved dialog.
8. A candidate that declared the step impossible only wins if every other candidate left the
machine in a worse state than it started. Giving up is not progress.

[OUTPUT FORMAT]
Analysis: For each candidate, one or two sentences on what its final state actually shows,
whether the assigned end state was reached, and whether it is safe to continue from.
Best: <index>

<index> must be a single integer, the 1-based index of the chosen candidate.
Output exactly in this order: Analysis, Best.
```

K REPRODUCIBILITY

Every number in this paper is produced by one of the runs in Table 15 and is recorded, with the command that produced it, in a single machine-readable file; the figure scripts read that file directly,

and the scoring scripts regenerate every reported value from it. Each run writes one directory per task per rollout containing the full action trace, every screenshot, the verifier's transcript, and the benchmark checker's verdict, so any individual number can be re-derived without re-running the agent.

Table 15: Runs behind the reported results. Executors are Azure-hosted; the `computer` tool is the provider's GA computer-use tool in both cases. All OSWorld-V2 runs share one plan cache so that flat and search arms differ only at execution time, which is also the limitation.

Run	benchmark	executor	what it provides	tasks
flat pool	OSWorld	`gpt-5.4`	three rollouts/task, all flat arms	361
tree	OSWorld	`gpt-5.4`	checkpoint tree search	361
verifier sweep	OSWorld `hard-50`	`gpt-5.4`	Table 5	50
diversity sweep	OSWorld `hard-50`	`gpt-5.4`	-	50
flat pool	OSWorld-V2	`gpt-5.4`	three rollouts/task	106
tree, frames	OSWorld-V2	`gpt-5.4`	evidence ablation, control	102
tree, evidence	OSWorld-V2	`gpt-5.4`	evidence ablation, treatment	102
flat pool	OSWorld-V2	`gpt-5.6-sol`	stronger-executor arms	106
tree, transferred plans	OSWorld-V2	`gpt-5.6-sol`	-	97
shell only	OSWorld-V2	`gpt-5.6-sol`	action-space arm	106
hybrid	OSWorld-V2	`gpt-5.6-sol`	action-space arm	106