CUA-SPEEDRUN: STANDARDIZED BENCHMARKING OF THE SPEED OF COMPUTER-USE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Computer use agents (CUAs), which use graphical user interfaces (GUIs) to complete tasks on a computer, have recently surpassed human performance on many standard benchmarks, including on difficult long-horizon tasks. Their capabilities are undoubtedly impressive, however, a key barrier to the widespread adoption and deployment of CUAs remains their speed and cost. Progress towards faster yet capable CUAs requires reliable evaluation of their speed, but many CUA benchmarks currently face a reproducibility crisis. Benchmarks rely on complex infrastructure with variable virtual machine and container specifications that confound the evaluation of the execution speed of CUAs. Towards addressing this gap, we propose `cua-speedrun`, which introduces standardized infrastructure and task sets, with a focus on evaluating the speed and efficiency of CUAs. `cua-speedrun` uses a uniform virtual machine setup and execution pipeline, along with a common agent interface that enables single-agent implementations to operate seamlessly across different benchmarks. Across four different CUA benchmarks, we evaluate how reasoning effort, agent harnesses, and environment latency affect performance, speed, and cost. We find no single model family is optimal for all three; none of the open-weight models are on the frontier, and also, unintuitively, for some models increasing the reasoning effort can speed up task completion, while faster environment input-output can slow down overall task completion time. We believe `cua-speedrun` will enable structured progress towards fast, efficient CUAs, unlocking new real-world use cases and applications.

1 INTRODUCTION

Computer use agents (CUAs) are language model agents that interact with graphical user interfaces (GUIs) to achieve a user-specified goal. CUAs operate over the same user interface as human users, enabling them to potentially automate a much wider variety of tasks without software-specific APIs, which many consumer-facing applications and websites do not provide.

When measured in terms of success alone, recent model releases have pushed the capabilities of CUAs past human baselines. The best frontier models today achieve success metrics above human performance on popular CUA benchmarks: Qwen3.8-Max, Claude Mythos 5, GPT-5.5, and Gemini 3.6 Flash report successes ranging from 78.7% to 86.1% on OSWorld-Verified, far above the 72.4% human reference score (Xie et al., 2024). Across long-horizon benchmarks (Aggarwal et al., 2026; Yuan et al., 2026; Jang et al., 2026b), frontier models have also demonstrated the ability to achieve strong performance on complex tasks.

Despite their strong performance, a major bottleneck for widespread adoption of current CUAs in practical settings is their speed and cost of deployment (Abhyankar et al., 2025). In particular, we believe that research on the speed and cost efficiency of CUAs has trailed capabilities research, in part due to the difficulty of fairly benchmarking speed and cost of different models in a consistent manner. Specifically, CUAs require complex infrastructure to run and evaluate. Different benchmarks use varying execution environments, ranging from local virtual machines to cloud-parallelized desktops (Xie et al., 2024; Bonatti et al., 2025). There are drastic differences in deployment setup, API interfaces, hyperparameter settings, agent harnesses, and even benchmark implementations across the community.

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.

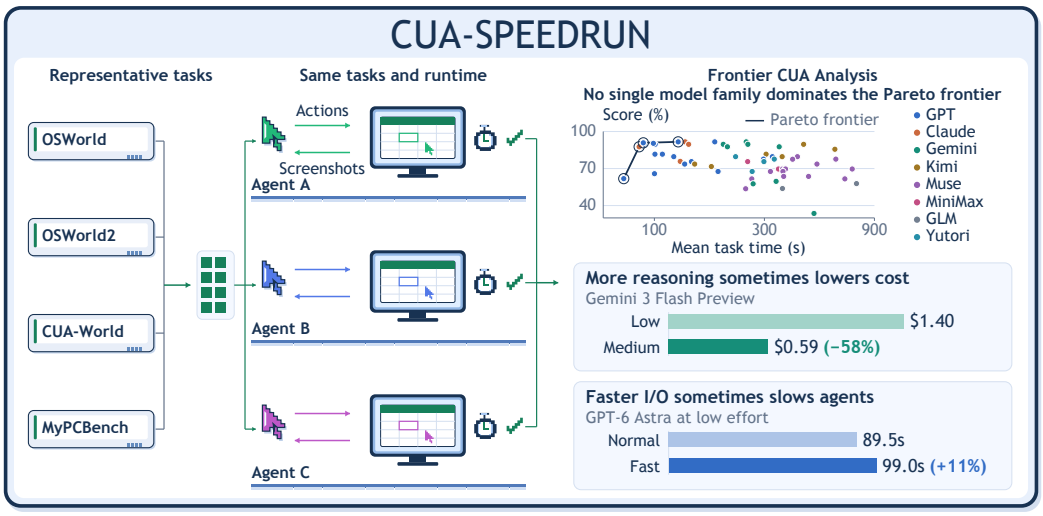

Figure 1: **`cua-speedrun` overview.** `cua-speedrun` is a standardized platform for benchmarking the performance, speed, and cost of computer-use agents. It combines representative task selection with standardized infrastructure on Modal, enabling consistent comparisons across models, agent harnesses, and reasoning settings. Our findings reveal that more reasoning can reduce both task time and model cost for CUAs, while faster environment I/O can make agents slower.

Towards addressing the problem of consistent benchmarking of CUA wall-time, and igniting research on improving the speed and deployment practicality of CUAs, we propose `cua-speedrun`: a standardized platform that can use existing benchmarks to measure the speed and cost efficiency of CUAs (Fig. 1). There are several challenges that `cua-speedrun` addresses, towards standardizing the evaluation of CUAs across different model families (in particular on wall-time metrics):

- **Uniform CUA evaluation infrastructure.** We standardize evaluations through a serverless cloud-based provider[1], where all evaluations run on the exact same execution pipeline and the same virtual machine setup.
- **Common agent interfaces.** Existing CUA implementations are varied and do not always follow the optimal scaffolds. We implement a common agent interface for executing actions and interacting with the environment while allowing for portable agent harnesses and implementations. This allows a single agent implementation to work across different benchmarks.
- **A maximally representative subset for fast and comparable evaluations.** CUA benchmarks take a long time to run. We propose an energy minimization framework that selects a minimal representative subset of tasks within existing CUA benchmarks. This allows us to minimize end-to-end evaluation time while preserving the signal of the results, and maintain the relative ordering of agent performance. Our approach reduces evaluation time on OSWorld by 84.9% while maintaining high correlation with evals run on the full set.

We implement these features on `cua-speedrun`, and benchmark current frontier CUA models on several widely adopted CUA benchmarks, such as OSWorld-Verified (Xie et al., 2024) and OSWorld 2.0 (Yuan et al., 2026). In addition, we also perform evaluations and analysis on CUA-World (Aggarwal et al., 2026) and MyPCBench (Jang et al., 2026a), which test for long-horizon and personalized computer use, respectively. Our analysis on `cua-speedrun` results allows us to maintain a live Pareto ranking of the most cost-effective, fastest, and most capable models. Through extensive analysis of multiple `cua-speedrun` runs, we identify several interesting and applicable insights. Success alone hides large differences in speed: on OSWorld, the open-weight Kimi K3 (max reasoning) matches GPT-6 Astra (high), but takes $4.4\times$ longer per task, and none of the open-weight models we evaluate (Kimi K3, MiniMax M3, GLM-5V Turbo) lie on the performance–time or performance–cost Pareto frontier of any benchmarks. Speed also interacts with reasoning in unintuitive ways: raising reasoning effort can improve benchmark scores while *reducing* mean task

[1] `https://modal.com/`

completion time, since the added reasoning avoids repeated unsuccessful actions. Using a fast I/O system can also slow agents, as the models have not been trained to interact with desktops in different I/O speed configurations.

Our findings highlight the importance of turning our attention beyond performance and success rates. `cua-speedrun` maintains a standardized evaluation of the performance and efficiency of CUAs, two dimensions that we believe are essential future research directions. We hope that the public release of `cua-speedrun` encourages the research community to develop techniques to make CUAs faster, cheaper, and more efficient, paving the way towards more widespread real-world adoption.

2 METHOD

In this section, we describe our approach to address the aforementioned challenges with benchmarking CUAs. We base our techniques to improve CUA benchmarking on standardizing measurements and infrastructure (Sec. 2.1), selecting representative tasks instead of running entire benchmarks (Sec. 2.2), and building infrastructure that executes and times every agent consistently (Sec. 2.3).

2.1 STANDARDIZED COMPUTER-USE EVALUATION

Standardizing computer-use evaluations requires invariance across three different dimensions: (1) the benchmark, which determines what an agent must accomplish; (2) the environment infrastructure, which is responsible for managing VMs, the environment lifecycle, and action/observation contracts; and (3) the agent loop, which determines how the model interacts with the infrastructure. Many computer-use evaluations often make different choices for these three components, making it difficult to study them independently. For example, adopting a new model may introduce a benchmark-specific action loop, or adopting a new benchmark requires changes to the agent or infrastructure. The goal of our setup is to ensure the decoupling of benchmarks from infrastructure from agent design, in order to ensure fair and independent evaluation across these three dimensions.

Problem setup. Following prior computer-use benchmarks, each task specifies an initial desktop state, a natural-language instruction, and a verifier, either a programmatic check or an LLM judge, that scores the agent's trajectory and final state. Agents observe screenshots of the desktop and act through keyboard and mouse actions; Appendix J.1 gives the formal definition. Comparisons between agents hold the benchmark and infrastructure fixed, and similarly comparisons across benchmarks fix agent and infrastructure, so each change behaves as a clean ablation.

Measuring speed. For each task, our infrastructure starts measuring time when the instruction is given to the agent and stops when the agent terminates or reaches the task limit (steps taken, wall-time, or both). Environment provisioning, task setup, agent initialization, and verification occur outside of this interval. The infrastructure records the total task time and additionally separates it into the time spent executing environment operations and the time spent executing agent operations. Thus, the agent time can be fully attributed to the "speed" of the agent, regardless of any time needed to set up CUA evaluations. Additionally, we also log the total cost of model calls. In addition, infra failures are retried, while agent failures end the trajectory, which is then scored.

2.2 SELECTING A REPRESENTATIVE SET

Choosing a representative evaluation set. A full computer-use evaluation is often expensive because of long trajectories adding to API/GPU cost, and often requires overhead for managing virtual machines (VMs), adding substantial evaluation cost. For example, GPT-5.4 costs approximately $4000 on the 200-task CUA-World-Long benchmark (Aggarwal et al., 2026). We therefore ask the question: can we reduce the size of the benchmark needed for eval, while still capturing what would have happened if we evaluated on the full set? Accounting for the benchmarking of multiple agents, we want to choose a subset that keeps the individual agents' scores similar to their full benchmark scores while preserving the ordering of agents' performance.

Selection criteria. Our goal is to automatically select a subset K (where K is a variable) of the benchmark that is representative of the full benchmark. To ensure that the selected subset generalizes beyond the agents used to construct it, we evaluate the selection procedure using leave-one-agent-out validation. For each agent m, we construct a subset using the results of only the other agents (M-1) and then compare the held-out agent's score on the selected subset with its score on the full benchmark.

We found through multiple iterations of leave-one-model-out evaluation that the best method to select tasks was to minimize the energy distance between the distributions of agent scores on the full benchmark and subset (Székely & Rizzo, 2013). Specifically, let $C_{mi} \in [0, 1]$ be the partial score of agent m on task i, and let $B_{mi} = \mathbf{1}[C_{mi} = 1]$ denote exact completion. We represent task i by

$$z_i = (C_{1i}, \ldots, C_{Mi}, B_{1i}, \ldots, B_{Mi}) \in \mathbb{R}^{2M}. \tag{1}$$

Thus, two tasks are considered similar when agents exhibit similar patterns of partial and exact completion on them. Let $D_{ij} = \|z_i - z_j\|_2$ and let $\overline{D}$ be the mean distance between distinct tasks. For a subset S_K of K tasks, we minimize

$$\mathcal{E}(S_K) = \frac{1}{\overline{D}}\left(\frac{2}{KN}\sum_{i\in S_K}\sum_{j=1}^{N} D_{ij} - \frac{1}{K^2}\sum_{i,i'\in S_K} D_{ii'} - \frac{1}{N^2}\sum_{j=1}^{N}\sum_{j'=1}^{N} D_{jj'}\right). \tag{2}$$

This objective favors subsets whose tasks represent the performance patterns present in the full benchmark while avoiding redundant tasks with nearly identical patterns. The final term depends only on the full benchmark and is therefore constant during subset selection. We approximately minimize this objective by starting with 100 task subsets and replacing one selected task with an unselected task whenever this reduces the objective. We thus retain the subset with the lowest objective value across the 100 runs. During leave-one-agent-out evaluation, this procedure is repeated after removing the held-out agent's results. After determining the desired subset size, we construct the final deployed subset once using all available agents.

How many tasks are sufficient? The previous selection criteria can find the representative set for a given value of K. We next ask for the smallest subset of K tasks that consistently preserves the ranking of agents by performance. Let $\rho_q(K)$ be the Spearman correlation between leave-one-agent-out estimates and full benchmark results for evaluation quantity q. We choose

$$K^* = \min\left\{K : \min_q \min_{k\in\{K-1,K,K+1\}} \rho_q(k) \geq 0.95\right\}. \tag{3}$$

We require the correlation threshold to hold for (K-1), (K), and (K+1) to prevent us from selecting a value of (K) that performs well only by chance. This criterion selects 50 of 295 OSWorld tasks, 52 of 63 OSWorld2 tasks, 26 of 143 CUA-World tasks, and 38 of 184 MyPCBench tasks (Table 13), which form the `cua-speedrun` evaluation sets. We also evaluate how well the subsets preserve full-benchmark performance and model comparisons in our experiments (Appendix G).

Removing internet-dependent tasks. Live websites add variance through georestrictions, anti-bot behavior, and changing content (Xue et al., 2025; Jang et al., 2026b), so we remove tasks that require access to the public internet during evaluation (Appendix H).

2.3 Infrastructure

How do we host our infrastructure? We adapt the virtual machine runtime from Gym-Anything (Aggarwal et al., 2026) to run task environments natively in Modal sandboxes. Our infrastructure manages separate sandboxes for agents and environments, ensuring isolation, while Modal sandboxes provide a standardized hosted runtime so that differences in users' infrastructure do not affect evaluation results. We develop a Python library to handle VM scheduling, environment allocation, and parallelization, with further details in Appendix B.

Action and observation modalities. Following standard practice in computer-use agents (Xie et al., 2024; Qin et al., 2025), the action space consists of keyboard actions (e.g., typing text or pressing `Ctrl+C`) and mouse actions (e.g., clicking at coordinates (x, y) or double-clicking). Agent observations are RGB screenshots of the desktop at a resolution of 1920×1080.

To ensure that these interfaces behave consistently across VM runtimes, we developed *CUA-AutoDebug*, an end-to-end test suite in which a controlled application records the keyboard and mouse inputs it receives, compares them with the intended inputs for a large catalog of actions, and checks that screenshots match the application state. These tests uncovered several input errors in widely used CUA tools, which we correct in our infrastructure (Appendix I).

FastCUA: Developing a new fast I/O system for CUAs. Standard computer-use runtimes typically take roughly 2–3 seconds from issuing an action to receiving an observation. This latency is dominated by a variety of elements, such as action execution, waiting for the application to respond, and networking. The delay incidentally gives the application time to process the action (e.g., open a menu), allowing the agent to observe its effect. We therefore follow this design for our main evaluations. However, we develop a fast I/O mode, *FastCUA*, by optimizing networking, action execution, and image processing latencies (see Appendix B). FastCUA reduces action-to-observation latency to 2–28 ms, an improvement of more than an order of magnitude over typical runtimes. At this speed, a screenshot can be captured before the application has updated its UI in response to an action. We evaluate this setting to test whether lower infrastructure latency reduces overall wall time.

Implementing agents on `cua-speedrun`. Since we have standardized the benchmark and infra, new agents can be implemented with a single Python file that contains the agent loop. We also provide standard templates for various popular agents and models, which users can tweak directly. We provide options for both hosted evaluations and local evaluations through a single CLI call. See Appendix C for more details.

3 Results

3.1 Experimental Setup

We evaluate 56 agent configurations on OSWorld and 21 on OSWorld2 using the representative task sets from Sec. 2.2. These configurations cover frontier open-weight and proprietary models, each run through its public reference agent implementation or native computer-use API at selected reasoning-effort settings, with the tasks and infrastructure held fixed (Appendix J). We validate all agent designs for correctness through manual inspection and CUA-AutoDebug on representative tasks. For selected models, we also compare different agent harness designs, standard and fast I/O, and single versus batched tool calls per model response.

Metrics. For each configuration, we report the mean verifier score, which retains partial credit, together with the mean task time (Sec. 2.1) and mean model cost per task. We report the performance–time and performance–cost Pareto frontiers, as well as the joint frontier over all three. Interaction turns, generated tokens, and exact completion are defined in Appendix D.

3.2 Analysis

No single model family dominates the frontier. Fig. 2 shows the performance–time and performance–cost Pareto frontiers on both benchmarks. On OSWorld, the high-performance end of the time frontier extends from Claude Opus 5 (low), with a score of 87.6% in 86 s, through GPT-6 Astra (low), with 90.8% in 90 s, to Astra (xhigh), with 91.6% in 127 s. Gemini 3.8 Flash (low) matches the highest score and takes almost the same time as GPT-6 Astra, but costs \$0.11 rather than \$0.71 per task. At the lower-cost end, GPT-5.6 Luna (low, direct API) costs about \$0.01 per task but scores 61.8%. Thus, `cua-speedrun` enables comparison across various dimensions such as speed, cost and performance, rather than reducing the comparison to a single model ranking.

The Pareto frontier changes across benchmarks. Fig. 2 also shows that, unlike the OSWorld time frontier, the OSWorld2 time frontier is formed entirely by GPT-6 Astra configurations. Astra (high) achieves 75.0% in 914 s, while xhigh reaches 76.9% in 1,237 s, achieving an additional 1.9 percentage points with 35% more time. However, the cost frontier includes other models such as Gemini 3.8 Flash (medium), which achieves 59.9% at \$2.91 per task, compared to Astra (high) at \$7.39. Muse Spark 1.3 (minimal) extends the frontier to lower cost, with a score of 34.0% at an estimated \$0.39. Surprisingly, there is no open model on either the cost or the time frontier for either

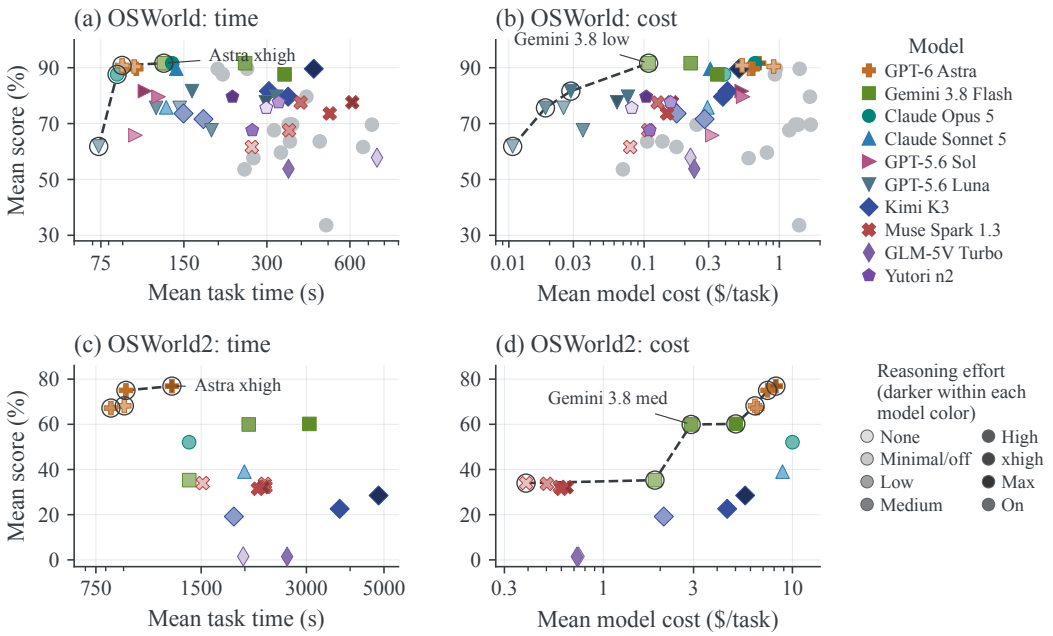

Figure 2: **Different agents define the speed and cost frontiers.** Mean score vs. mean task time and model cost on OSWorld (top) and OSWorld2 (bottom). Dashed lines connect the observed Pareto-frontier points, which are circled. Colors and marker shapes identify models, and darker shades indicate increased reasoning effort. The plots compare complete agent configs. Full configs and plots in Appendix E.

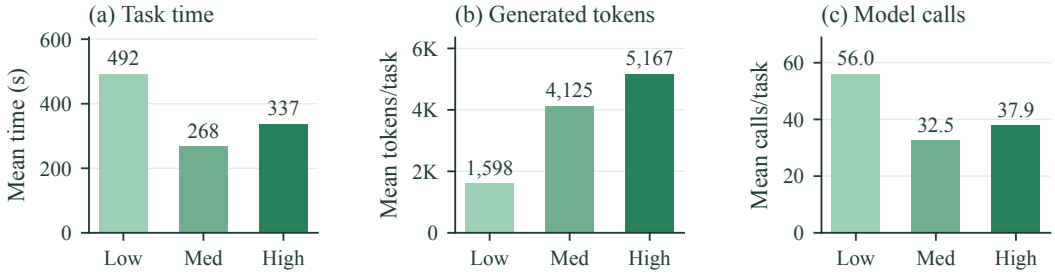

Figure 3: **Less reasoning can make an agent slower.** Gemini 3 Flash Preview at low, medium, and high effort on OSWorld. The panels show mean task time, generated tokens including reasoning, and model calls across all 50 tasks. Scores are 33.6%, 57.6%, and 59.6%, respectively. Medium and high effort generate more tokens than low effort but require fewer calls and complete tasks faster.

OSWorld or OSWorld2. Tracking these frontiers allows us to highlight the trade-off explicitly. The preferred agent depends on both the benchmark and the performance required within a time or cost budget, and there is no universal model family that dominates across different benchmarks.

Less reasoning can sometimes make agents slower. Fig. 3 compares Gemini 3 Flash Preview at different reasoning settings on OSWorld, keeping the model and harness fixed. Moving from low to medium effort improves the score from 33.6% to 57.6% while reducing mean task time from 492 s to 268 s. We diagnose this as follows: at lower reasoning effort, while the model generates fewer tokens (1,598 vs. 4,125 at medium effort), it takes substantially more steps (56.0 vs. 32.5), since most of its steps are suboptimal. For instance, the trajectories show repeated unsuccessful actions at low effort, such as repeatedly trying to enter spreadsheet text through key combinations instead of text entry. Counterintuitively, higher reasoning effort can therefore shorten the interaction enough to reduce total task time, even when it increases the number of generated tokens per step.

More reasoning can increase time and cost without improving performance. Fig. 4 compares reasoning settings within Gemini 3.8 Flash on each benchmark. On OSWorld, low and medium rea-

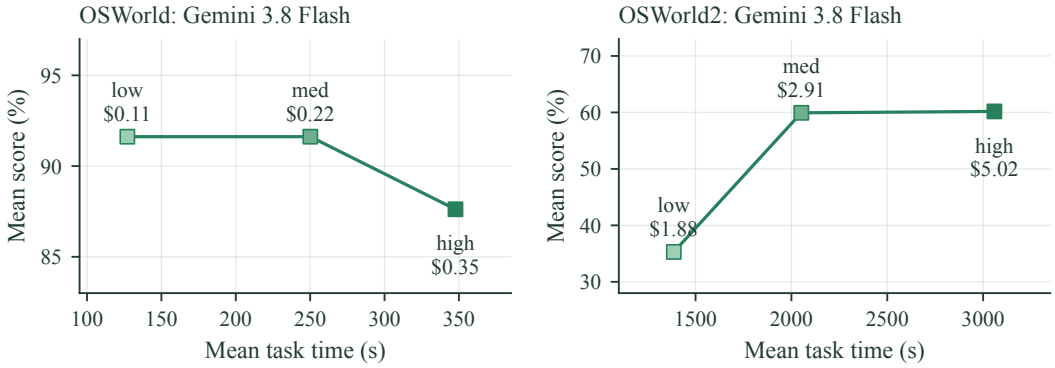

Figure 4: **The useful reasoning setting changes across benchmarks.** Gemini 3.8 Flash at low, medium, and high effort; labels also report mean cost per task. Each curve keeps the model, agent harness, and execution settings fixed. Additional reasoning brings no gain on OSWorld, whereas moving from low to medium substantially improves OSWorld2 performance.

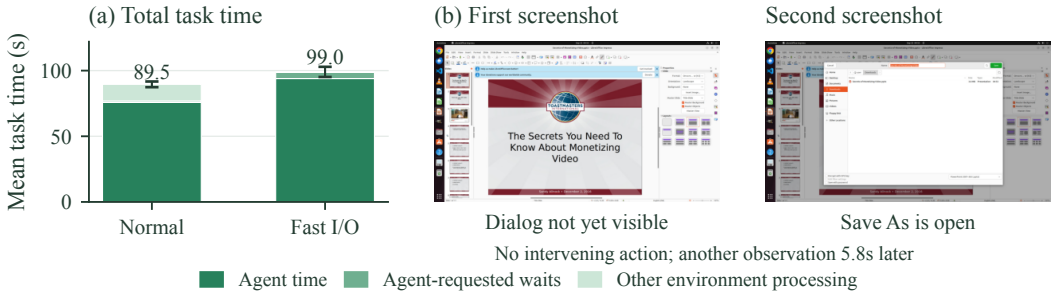

Figure 5: **Faster I/O can be offset by more waiting and agent interaction.** GPT-6 Astra (low), with normal and fast I/O. Panel (a) separates agent time, agent-requested waits, and remaining environment processing, averaged over five seeds per setting; error bars show the standard deviation of mean task time across seeds. Panel (b) shows two consecutive screenshots after opening Save As: the dialog is absent in the first and visible in the second, requested 5.8 s later with no intervening action.

soning both score 91.6%, but medium increases mean task time from 127 s to 250 s and approximately doubles cost. However, the trend changes on OSWorld2: moving from low to medium improves performance from 35.3% to 59.9%. But increasing effort further to high adds only 0.2 percentage points while increasing time by 49% and cost by 72%. GPT-6 Astra shows a similar dependence on the benchmark: on OSWorld, medium effort is within 2 points of xhigh at 29% less time and 15% less cost, whereas on OSWorld2, high effort gains 7.8 points over medium for 10% more time. The useful reasoning setting therefore depends on the tasks as well as the model: additional reasoning can be valuable on one (harder) benchmark and unnecessary on another (simpler) benchmark.

Faster infra can make the agent slower. Fig. 5 compares GPT-6 Astra (low) with the normal input-output mode and our faster input-output mode, FastCUA (Sec. 2.3 and Appendix B.3) across five seeds per setting. Interestingly, while the mean environment processing time falls from 12.68 s to 0.34 s per task, total task time increases from 89.5 s to 99.0 s. We also find that the number of steps per trajectory increases in FastCUA. This is because fast I/O can return screenshots before an application has updated its interface. For example, after opening Save As, the agent receives a screenshot without the dialog and requests another screenshot 5.8 s later, which shows it open. Although theoretically a model could account for such faster I/O by using appropriate wait times (e.g., 100 ms in this case) for the application to update, we find that current models are not able to optimally utilize this, resulting in much higher times.

The agent harness plays an important role in the speed–performance trade-off. Fig. 17 compares GPT-5.6 Luna through the direct-API agent and the Codex harness on OSWorld. At low effort, Codex improves the score from 61.8% to 75.6%, but increases mean task time from 74 s to 145 s. The same model and reasoning setting therefore produce different performance and efficiency when

used through different harnesses. At medium and high effort, the direct-API agent is instead both more accurate and faster (75.6% in 119 s vs. 67.6% in 189 s at medium; 81.6% in 161 s vs. 79.6% in 325 s at high), so neither harness is uniformly better.

Batching actions reduces task time. Each step costs a screenshot, a model call, and environment latency, so agents that issue several actions per response (which we execute sequentially) finish sooner: GPT-6 Astra (low) completes OSWorld tasks in 5.9 action batches on average (Table 11). Restricting GPT-6 Astra (xhigh) to one action per response keeps its score at 91.6% but raises mean task time from 127 s to 183 s. The benefit depends on how well a model plans multi-action sequences, and batching alone does not make an agent fast (Appendix K.1).

Fast agents generate fewer tokens, not tokens faster. Claude Opus 5 (low) and GPT-6 Astra (low) both score 87.6%, but Opus generates $1.5\times$ more tokens: its higher token rate is spent on longer outputs rather than earlier completion, and at a common token rate (Fig. 10) it falls behind the Astra configurations, which remain the fastest high-scoring agents. Similarly, Claude Sonnet 5 (high) takes 141 s for the score that Astra (medium) reaches in 90 s, generating $3.5\times$ more tokens, and Gemini 3.8 Flash (high) generates $5.6\times$ more tokens than at low effort and finishes $2.7\times$ later.

Extending to other benchmarks. We additionally evaluate GPT-6 Astra with Codex at four reasoning settings on MyPCBench (Jang et al., 2026a) and CUA-World (Aggarwal et al., 2026), which test personalized and long-horizon computer use and grade trajectories with vision-language models rather than programmatic checks. Astra exceeds 90% rubric scores on both (Appendix E.5), highlighting the need for more challenging task sets.

3.3 Practical Suggestions for Building CUAs

Based on the analysis above, we make the following suggestions for building efficient CUAs:

- **Choose a frontier configuration for the required performance and budget.** No single configuration is best on every metric, and the frontier shifts across benchmarks (Figs. 2 and 6), so candidates should be measured on tasks that resemble the target workload.
- **Tune reasoning effort rather than setting it to an extreme.** Too little reasoning can lengthen trajectories, while too much can add time and cost without improving performance (Figs. 3 and 4). Start with low or medium effort and raise it only for a measured gain.
- **Batch actions when intermediate screenshots are unnecessary.** Fewer steps mean fewer screenshots, model calls, and environment round trips.
- **Reduce generated tokens and steps rather than maximizing token rate or I/O speed.** Faster generation or I/O helps only if the agent does not spend it on longer outputs or additional steps (Fig. 5).
- **Optimize the harness alongside the model.** The same model and reasoning setting can land at different points on the frontier depending on its harness (Fig. 17).

4 Related Work

Computer-use agent benchmarks. The first computer-use agent benchmarks used synthetic interfaces (Liu et al., 2018; Yao et al., 2022)). Follow-up evaluations moved to self-hosted realistic website analogs ((Zhou et al., 2024), (Koh et al., 2024), (Drouin et al., 2024)), static datasets of real live website tasks ((Deng et al., 2023)), and live-Internet evaluations ((He et al., 2024), (Xue et al., 2025)). Live, dynamic desktop benchmarks such as OSWorld (Xie et al., 2024) evaluate agents on real Linux desktops using programmatic verifiers, with WindowsAgentArena (Bonatti et al., 2025), macOSWorld (Yang et al., 2025), iOSWorld (Jang et al., 2026c), and AndroidWorld (Rawles et al., 2025) extending coverage to other platforms and professional workflows. Recent benchmarks stress long horizons and scale, such as OSWorld 2.0 (Yuan et al., 2026), CUA-World from Gym-Anything (Aggarwal et al., 2026), and Odysseys (Jang et al., 2026b). Another emphasis has become personal assistant evaluations, including OpenClaw-style works such as ClawBench (Zhang et al., 2026b), WildClawBench (Ding et al., 2026), and MyPCBench (Jang et al., 2026a), which evaluate

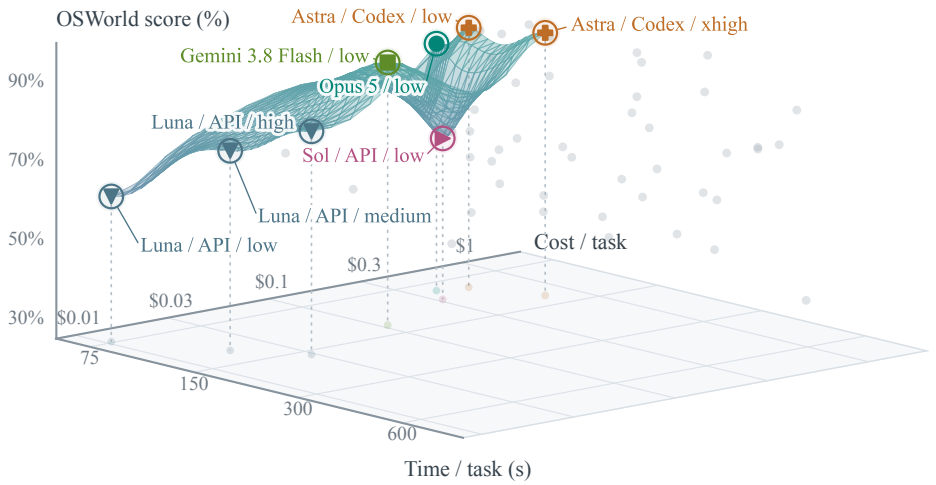

Figure 6: **Choosing an agent by performance, time, and cost.** Joint Pareto frontier on OSWorld; the OSWorld2 frontier is in Appendix E, Fig. 13. Circled points are the configurations on the Pareto frontier. Time and cost axes are logarithmic.

personalized computer use and digital assistants. OSWorld-Human (Abhyankar et al., 2025) measures agent latency on OSWorld and compares step counts against human reference trajectories, finding that agents take longer than humans.

Computer-use agent models. Frontier model releases have increasingly prioritized computer use (OpenAI, 2026; Anthropic, 2026; Google, 2026; Qwen Team, 2026), with several reporting above-human scores on OSWorld-Verified. Models such as Kimi K3 (Kimi Team, 2026), GLM-5 (GLM-5 Team, 2026), and Muse Spark (Menghini et al., 2026) have also included agentic workflows in their tech releases. Alongside these large general models, a line of work trains specialized computer-use models, including UI-TARS (Qin et al., 2025), MolmoWeb (Gupta et al., 2026), Fara-1.5 (Awadallah et al., 2026), ScaleCUA (Lv et al., 2026), and Qwen-CUA (Lu et al., 2026), often in verifiable environments generated at scale (Wang et al., 2026; Aggarwal et al., 2026).

Reproducible and efficient evaluation. Agent evaluations are noisy (Kapoor et al., 2024; 2026), as Xue et al. (2025) showed that reported web-agent success is inflated by evaluation issues, and Sahu & Pandey (2026) show that single-run claims are unreliable because outcomes vary substantially across data draws and run non-determinism. A separate line of work reduces the evaluation cost by selecting a small subset of a benchmark that predicts full-benchmark scores, via item response theory ((Maia Polo et al., 2024), (Kipnis et al., 2025), Zhou et al. (2026)), sparse optimization ((Zhang et al., 2026a)), or human validation (SWE-bench Verified (Chowdhury et al., 2024)). PACE (Song et al., 2026) predicts agentic benchmark scores from cheap, non-agentic evaluations.

5 Conclusion

In this work, we introduced `cua-speedrun`, a framework for evaluating the performance, speed, and cost of computer-use agents under standardized infrastructure. Our evaluations on OSWorld and OSWorld2 identify the performance–time and performance–cost Pareto frontiers and show how these change across benchmarks. We further find that more reasoning can reduce task time by avoiding repeated unsuccessful actions, while faster I/O can increase it through additional interaction. These results highlight the need to study reasoning and interaction together when developing faster agents. We hope `cua-speedrun` enables the community to improve both capability and efficiency, making computer-use agents more practical to deploy.

AI use statement

We used AI assistance for general software coding, making figures, polishing writing across the manuscript, and verifying related work.

REFERENCES

Reyna Abhyankar, Qi Qi, and Yiying Zhang. OSWorld-Human: Benchmarking the efficiency of computer-use agents. *arXiv preprint arXiv:2506.16042*, 2025.

Pranjal Aggarwal, Graham Neubig, and Sean Welleck. Gym-Anything: Turn any software into an agent environment. *arXiv preprint arXiv:2604.06126*, 2026.

Anthropic. Claude Fable 5 and Claude Mythos 5. https://www.anthropic.com/news/claude-fable-5-mythos-5, 2026.

Ahmed Awadallah, Sahil Gupta, Yash Lara, Yadong Lu, Hussein Mozannar, Akshay Nambi, Zach Nussbaum, Yash Pandya, Aravind Rajeswaran, Corby Rosset, Alexey Taymanov, Luiz do Valle, Vibhav Vineet, Spencer Whitehead, and Andrew Zhao. Fara-1.5: Scalable learning environments for computer use agents. *arXiv preprint arXiv:2606.20785*, 2026.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Jang, and Zack Hui. Windows Agent Arena: Evaluating multi-modal OS agents at scale. In *ICML*, 2025.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/, 2024.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. In *NeurIPS*, 2023.

Shuangrui Ding, Xuanlang Dai, Long Xing, Shengyuan Ding, Ziyu Liu, Yang JingYi, Penghui Yang, Zhixiong Zhang, Xilin Wei, Xinyu Fang, Yubo Ma, Haodong Duan, Jing Shao, Jiaqi Wang, Dahua Lin, Kai Chen, and Yuhang Zang. WildClawBench: A benchmark for real-world, long-horizon agent evaluation. *arXiv preprint arXiv:2605.10912*, 2026.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How capable are web agents at solving common knowledge work tasks? In *ICML*, 2024.

GLM-5 Team. GLM-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Google. Introducing Gemini 3.6 Flash, 3.5 Flash-Lite, and 3.5 Flash Cyber. https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-6-flash-3-5-flash-lite-3-5-flash-cyber/, 2026.

Tanmay Gupta, Piper Wolters, Zixian Ma, Peter Sushko, Rock Yuren Pang, Diego Llanes, Yue Yang, Taira Anderson, Boyuan Zheng, Zhongzheng Ren, Harsh Trivedi, Taylor Blanton, Caleb Ouellette, Winson Han, Ali Farhadi, and Ranjay Krishna. MolmoWeb: Open visual web agent and open data for the open web. *arXiv preprint arXiv:2604.08516*, 2026.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an end-to-end web agent with large multimodal models. In *ACL*, 2024.

Lawrence Keunho Jang, Andrew Keunwoo Jang, Jing Yu Koh, and Ruslan Salakhutdinov. MyPCBench: A benchmark for personally intelligent computer-use agents. *arXiv preprint arXiv:2606.16748*, 2026a.

Lawrence Keunho Jang, Jing Yu Koh, Daniel Fried, and Ruslan Salakhutdinov. Odysseys: Benchmarking web agents on realistic long horizon tasks. *arXiv preprint arXiv:2604.24964*, 2026b.

Lawrence Keunho Jang, Mareks Woodside, Geronimo Carom, Andrew Keunwoo Jang, Jing Yu Koh, and Ruslan Salakhutdinov. iosworld: A benchmark for personally intelligent phone agents, 2026c. URL https://arxiv.org/abs/2606.09764.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter, 2024. URL https://arxiv.org/abs/2407.01502.

Sayash Kapoor, Benedikt Stroebl, Peter Kirgis, Nitya Nadgir, Zachary S Siegel, Boyi Wei, Tianci Xue, Ziru Chen, Felix Chen, Saiteja Utpala, Franck Ndzomga, Dheeraj Oruganty, Sophie Luskin, Kangheng Liu, Botao Yu, Amit Arora, Dongyoon Hahm, Harsh Trivedi, Huan Sun, Juyong Lee, Tengjun Jin, Yifan Mai, Yifei Zhou, Yuxuan Zhu, Rishi Bommasani, Daniel Kang, Dawn Song, Peter Henderson, Yu Su, Percy Liang, and Arvind Narayanan. Holistic agent leaderboard: The missing infrastructure for AI agent evaluation. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/pdf/2510.11977.

Kimi Team. Kimi K3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

Alex Kipnis, Konstantinos Voudouris, Luca M. Schulze Buschoff, and Eric Schulz. metabench – a sparse benchmark of reasoning and knowledge in large language models. In *ICLR*, 2025.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. In *ACL*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *SOSP*, 2023.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *ICLR*, 2018.

Dunjie Lu, Shuai Bai, Tianyi Bai, et al. Qwen-CUA: Native computer use for (almost) everything. *arXiv preprint arXiv:2608.02352*, 2026.

Bowen Lv, Xiao Liu, Yanyu Ren, Hanyu Lai, Bohao Jing, Hanchen Zhang, Yanxiao Zhao, Shuntian Yao, Jie Tang, and Yuxiao Dong. ScaleCUA: Scaling computer use agents with verifiable task synthesis and efficient online RL. *arXiv preprint arXiv:2607.11185*, 2026.

Felipe Maia Polo, Lucas Weber, Leshem Choshen, Yuekai Sun, Gongjun Xu, and Mikhail Yurochkin. tinyBenchmarks: evaluating LLMs with fewer examples. In *ICML*, 2024.

Cristina Menghini, Peter Ney, Hamza Kwisaba, Zifan Wang, Miles Turpin, et al. Muse Spark safety & preparedness report. *arXiv preprint arXiv:2606.12429*, 2026.

OpenAI. GPT-6 Astra: A new generation of intelligence. https://openai.com/index/gpt-6-astra/, 2026.

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

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork. https://qwen.ai/blog?id=qwen3.8, 2026.

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

Barada Sahu and Shivesh Pandey. Teach it to stop, not just to click. *arXiv preprint arXiv:2607.17136*, 2026.

Yueqi Song, Lintang Sutawika, Jiarui Liu, Lindia Tjuatja, Jiayi Geng, Yunze Xiao, Daniel Lee, Aditya Bharat Soni, Vincent Lo, Xiang Yue, and Graham Neubig. PACE: A proxy for agentic capability evaluation. *arXiv preprint arXiv:2607.02032*, 2026.

Gábor J. Székely and Maria L. Rizzo. Energy statistics: A class of statistics based on distances. *Journal of Statistical Planning and Inference*, 143(8):1249–1272, 2013.

Bowen Wang, Dunjie Lu, Junli Wang, Tianyi Bai, Shixuan Liu, Zhipeng Zhang, Haiquan Wang, Hao Hu, Tianbao Xie, Shuai Bai, Dayiheng Liu, Que Shen, Junyang Lin, and Tao Yu. CUA-Gym: Scaling verifiable training environments and tasks for computer-use agents. *arXiv preprint arXiv:2605.25624*, 2026.

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

Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. An illusion of progress? Assessing the current state of web agents. In *COLM*, 2025.

Pei Yang, Hai Ci, and Mike Zheng Shou. macOSWorld: A multilingual interactive benchmark for GUI agents. In *NeurIPS*, 2025.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. In *NeurIPS*, 2022.

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. *arXiv preprint arXiv:2606.29537*, 2026.

Taolin Zhang, Hang Guo, Wang Lu, Tao Dai, Shu-Tao Xia, and Jindong Wang. SparseEval: Efficient evaluation of large language models by sparse optimization. In *ICLR*, 2026a.

Yuxuan Zhang, Yubo Wang, Yipeng Zhu, Penghui Du, Junwen Miao, Xuan Lu, Zhuofeng Li, Xingwei Qu, Zhengkang Guo, Yuanzhe Shen, Dingjie Song, Han Zhou, Tuney Zheng, Xian Wu, Hao Yu, Songcheng Cai, Yi Lu, Yunzhuo Hao, Minyi Lei, Liang Chen, Kai Zou, Huifeng Yin, Wendong Xu, Dongfu Jiang, Ping Nie, Jiaheng Liu, Wenhu Chen, and Kelsey R. Allen. ClawBench: Can AI agents complete everyday online tasks? *arXiv preprint arXiv:2604.08523*, 2026b.

Hongli Zhou, Hui Huang, Ziqing Zhao, Lvyuan Han, Huicheng Wang, Kehai Chen, Muyun Yang, Wei Bao, Jian Dong, Bing Xu, Conghui Zhu, Hailong Cao, and Tiejun Zhao. Lost in benchmarks? Rethinking large language model benchmarking with item response theory. In *AAAI*, 2026.

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

APPENDIX TABLE OF CONTENTS

A Limitations . . . 15

B Infrastructure: Technical Details . . . 15

B.1 Agent and Environment Sandboxes . . . 15

B.2 Model Serving, Resource Allocation, and Parallel Evaluation . . . 15

B.3 Fast I/O Implementation . . . 15

B.4 Action Latency: Fast I/O, Gym-Anything, and OSWorld . . . 16

B.5 Latency Breakdown . . . 16

B.6 Fast I/O Consistency Across Five Seeds . . . 17

C Implementing and Evaluating an Agent . . . 17

C.1 Agent Templates . . . 18

C.2 Minimal Agent Example . . . 18

C.3 Hosted and Local Evaluation . . . 19

D Interaction and Token Measurements . . . 19

D.1 Model Calls, Action Batches, and Individual Actions . . . 19

D.2 Token Accounting . . . 20

D.3 Full Interaction and Token Comparisons . . . 20

D.4 Performance–Interaction and Performance–Token Frontiers . . . 22

E Full Evaluation Results . . . 24

E.1 Model, Harness, and Reasoning-Effort Configurations . . . 24

E.2 OSWorld: Performance–Time and Performance–Cost Frontiers . . . 26

E.3 OSWorld2: Performance–Time and Performance–Cost Frontiers . . . 27

E.4 OSWorld2: Joint Performance–Time–Cost Frontier . . . 28

E.5 Reasoning Effort on MyPCBench . . . 29

E.6 Reasoning Effort on CUA-World . . . 29

F Run-to-Run Variation . . . 30

F.1 Five-Seed Evaluation Setup . . . 30

F.2 Scores and Times Across Seeds . . . 30

F.3 Task-Level Score Agreement . . . 31

F.4 Variation in Action Sequences . . . 31

G Representative Task Selection . . . 32

G.1 Models and Tasks Used for Selection . . . 32

G.2 Initialization, Optimization, and Task Count . . . 32

G.3 Held-Out Score and Ranking Accuracy . . . 34

G.4 Preserving the Speed–Performance Frontier . . . 35

G.5 Comparing Task-Selection Methods . . . 36

G.6 Selected Task Identities . . . 36

H Internet-Dependence Audit . . . 38

H.1 Inclusion Rule . . . 38

H.2 Reviewer Agreement . . . 38

I Validating Actions with CUA-AutoDebug . . . 39

I.1 Separating Model, Harness, and Runtime Errors . . . 39

I.2 Input Failures Detected by the Runtime Tests . . . 39

I.3 Failures in Agent Harnesses . . . 39

J Experimental Details . . . 40

J.1 Task Definition . . . 40

J.2 Observations, Actions, and Interaction History . . . 40

J.3 Task Limits and Execution . . . 40

J.4 Benchmark Verification . . . 41

J.5 Model Cost . . . 41

K Additional Agent Analyses . . . 41

K.1 Controlled Batching Comparisons . . . 41

K.2 Harness Comparisons Across Reasoning Efforts . . . 42

K.3 Comparing Reasoning Efforts on the Same Successful Tasks . . . 43

L Automated Search for Task-Selection Methods . . . 44

L.1 Search and Evaluation . . . 44

L.2 Validation Gains Do Not Generalize to the Held-Out Model . . . 44

A LIMITATIONS

Reported times and costs reflect API inference conditions at the time of evaluation (Appendix J). Providers control subsequent changes to inference speed and pricing. We validate representative task subsets on held-out agents (Appendix G); subset scores approximate full-benchmark performance. Changes in agent capabilities may require revalidation of these subsets.

Our evaluations cover a broad range of agents across four benchmarks (Appendix E). Evaluation cost limits coverage of every combination of model, harness, reasoning effort, and benchmark.

B INFRASTRUCTURE: TECHNICAL DETAILS

B.1 AGENT AND ENVIRONMENT SANDBOXES

We run the agent and task environment in separate sandboxes. The environment contains the desktop, applications, and initial task state; the agent interacts with it through screenshot and action requests. Task initialization and verification remain independent of the agent, allowing the same agent implementation to run across benchmarks. When the agent terminates or reaches its limit, the benchmark's verifier scores the trajectory or final environment state.

We use Modal to host the sandboxes and adapt the environment runtime from Gym-Anything (Aggarwal et al., 2026). The environment measures task time and the time spent executing actions and returning observations, independently of the agent implementation. As in Sec. 2.1, task preparation, agent initialization, and verification are excluded from task time.

B.2 MODEL SERVING, RESOURCE ALLOCATION, AND PARALLEL EVALUATION

Agents access models through API endpoints or a self-hosted vLLM inference server (Kwon et al., 2023). We use L40S GPUs for self-hosted inference and a Python library for GPU scheduling. The evaluation configuration specifies the compute resources, task limits, and number of concurrent agents. Model initialization and weight loading finish before task timing begins.

We assign tasks to available agents and environments. Each task starts from a fresh environment state. Its clock starts after the agent and environment are ready and stops when the agent finishes or reaches its limit, excluding time waiting for an execution slot. Model servers reuse loaded weights across tasks. We record a trajectory for each task.

B.3 FAST I/O IMPLEMENTATION

FastCUA, our Fast I/O implementation, reduces the time between issuing an action and receiving a screenshot. We encode screenshots in the background and combine action execution and screenshot delivery in a single request.

Screenshot capture and encoding. We use QEMU's D-Bus display interface to keep the current screen in memory and encode updated frames in the background. We use JPEG at quality 95 with no chroma subsampling and retain the full 1920×1080 resolution.

Action execution and communication. We execute the action and return its screenshot in one request, reusing the network connection across requests. A lightweight command client reduces per-action startup overhead. We also place agent and environment sandboxes in the same Modal region to reduce network latency. Fast I/O returns the screenshot as soon as action execution and image preparation finish. It can therefore return a screenshot before the application has updated its interface, as illustrated in Figure 5.

B.4 Action Latency: Fast I/O, Gym-Anything, and OSWorld

Infrastructure	Click	Escape	Ctrl+U	Type 100 characters
Gym-Anything	2758.70	2865.00	2858.29	3521.72
OSWorld	2515.41	2707.74	2708.41	2756.47
Fast I/O (ours)	**27.58**	**2.43**	**2.91**	**14.98**

Table 1: **Fast I/O reduces action-to-observation latency to milliseconds.** Median latency in milliseconds over 20 calls per action. For each system, the timer starts immediately before `env.step(action)` and stops when it returns the observation. All default action waits are included.

Fast I/O reduces latency by one to three orders of magnitude. Table 1 compares Fast I/O with Gym-Anything (Aggarwal et al., 2026) and OSWorld (Xie et al., 2024) for mouse clicks, key presses, and text entry. Fast I/O takes 2.4–27.6 ms per call, compared to 2.5–3.5 s for the existing implementations. For example, a click takes 27.6 ms compared with 2.8 s with Gym-Anything, while typing 100 characters takes 15.0 ms compared with 3.5 s.

Measurement setup. We time each runtime's native `env.step(action)` call, from issuing the action until the observation is returned. We test four actions on a 1920×1080 Ubuntu desktop: clicking the Activities button, pressing Escape, pressing Ctrl+U, and typing 100 ASCII characters into a terminal. The caller and VM run in the same Modal sandbox. OSWorld uses its Docker provider with screenshot observations; both baselines use their default waits and action implementations. Table 1 reports the median of 20 calls per action, without profiling. Setup, pauses between calls, and saving returned screenshots occur outside the timed interval.

B.5 Latency Breakdown

Component (ms)	Click	Escape	Ctrl+U	Type 100 characters
Gym-Anything				
Fixed waits	2100.2	2100.2	2100.2	2100.2
Action request	153.5	153.7	154.0	805.5
Screenshot capture	258.9	354.0	354.3	354.1
Screenshot transfer	238.1	212.4	220.6	211.3
Cleanup and other	53.1	53.1	53.1	53.1
Total	**2803.9**	**2873.4**	**2882.2**	**3524.2**
Total minus 2 s	803.9	873.4	882.2	1524.2
OSWorld				
Fixed wait	2000.1	2000.1	2000.1	2000.1
Action request	198.6	191.7	192.8	245.1
Screenshot request	324.7	521.2	521.8	523.2
Other	0.2	0.2	0.2	0.2
Total	**2523.6**	**2713.3**	**2714.9**	**2768.7**
Total minus 2 s	523.6	713.3	714.9	768.7

Table 2: **Fixed waits account for most baseline latency.** Mean component times from 10 profiled calls per action for Gym-Anything and 20 for OSWorld. The final row for each system is the profiled total minus its two-second post-action wait.

Where do the baselines spend time? Table 2 shows that fixed sleeps account for most of the baseline latency. Both implementations sleep for two seconds after an action, and Gym-Anything adds another 100 ms per step. Subtracting the two-second sleep leaves 804–1524 ms for Gym-Anything and 524–769 ms for OSWorld. Gym-Anything captures a PNG with FFmpeg, transfers it, and deletes the temporary file; typing also adds a 6 ms interval between characters. In OSWorld, most of the remaining time is spent in the action and screenshot requests, including processing inside the VM and communication.

Component (ms)	Click	Escape	Ctrl+U	Type 100 characters
Input execution	27.30	1.85	2.55	18.39
Frame capture	1.03	0.77	0.77	0.80
Foreground image preparation	0.27	0.02	0.02	0.18
Other environment processing	0.93	0.94	1.00	0.90
Transport	4.67	4.99	4.76	4.62
Screenshot-file delivery	0.43	0.45	0.43	0.44
Command overhead	4.41	4.52	4.38	4.38
Total	**39.06**	**13.55**	**13.90**	**29.71**

Table 3: **Fast I/O completes the four tested actions in under 50 ms.** Mean times over 20 commands per action, from the agent issuing a command to receiving the screenshot file. This includes command overhead, communication between sandboxes, and screenshot-file delivery in addition to the `env.step` call.

Full command-to-screenshot latency. Table 3 measures latency from the agent issuing a command to receiving the screenshot file, including communication between sandboxes. We place both sandboxes in New York and test the four actions on two OSWorld desktops, with 0.5 s and 8 s gaps between requests. All 80 timed commands complete in under 50 ms. Communication contributes 4.6–5.0 ms on average. Image preparation after the request takes 0.02–0.27 ms; encoding runs in the background. Clicks take longer because input execution includes a 25 ms interval between pressing and releasing the mouse button.

B.6 Fast I/O Consistency Across Five Seeds

Run	Score (%)	Time (s)	Agent (s)	Steps	Env. (s)	Waits (s)	Env.−waits (s)
1	91.62	103.96	98.07	8.68	5.891	5.553	0.338
2	91.62	93.98	89.02	8.04	4.955	4.628	0.327
3	91.62	96.48	91.67	7.96	4.812	4.488	0.324
4	87.62	100.22	95.28	8.20	4.934	4.604	0.330
5	89.62	100.46	95.68	8.12	4.785	4.424	0.361
Mean	90.42	99.02	93.94	8.20	5.075	4.739	0.336
SD	1.79	3.87	3.58	0.28	0.462	0.462	0.015

Table 4: **Fast I/O remains consistent across five evaluations.** Each row averages all 50 OSWorld tasks for GPT-6 Astra at low reasoning effort. SD is the sample standard deviation across run-level means; score SD is in percentage points. Steps count action batches, and waits are explicitly requested by the agent.

Environment processing time remains low across runs. Table 4 reports five evaluations of GPT-6 Astra at low effort on the same 50 OSWorld tasks with Fast I/O. We keep the agent prompt fixed and use fresh task seeds, a 500-step limit, and agent and environment sandboxes in New York. Environment time excluding agent-requested waits is 0.336 ± 0.015 s per task (mean $\pm$ standard deviation across runs). The mean task time is 99.0 ± 3.9 s, and the score is $90.4 \pm 1.8\%$.

Most environment time is spent on agent-requested waits. Table 4 also separates the waits requested by the agent from other environment processing. These waits account for 4.739 s of the 5.075 s of mean environment time per task, leaving only 0.336 s for the remaining processing. Appendix D defines these timing components.

C Implementing and Evaluating an Agent

To evaluate a new agent with `cua-speedrun`, users implement its model calls, prompts, and interaction history. The infrastructure handles task setup, desktop interaction, and verification through the common interface in Sec. 2.3. The same agent implementation and environment interface can then be used across benchmarks.

C.1 AGENT TEMPLATES

We provide templates for direct model-API agents, Codex, Claude Code, and open-weight agents served through vLLM. Each template contains two files: `init.py` prepares dependencies and model serving, and `agent.py` implements the agent loop. Initialization finishes before task timing begins. Users can retain `init.py` and modify only the agent loop to test a new model, prompt, or interaction strategy.

C.2 MINIMAL AGENT EXAMPLE

Listing 1 shows a minimal agent. The script receives the environment URL and task instruction as command-line arguments and uses the `Computer` client to request screenshots, execute actions, and end the task. In each iteration, `choose_actions` receives the instruction, screenshot, and interaction history. This user-defined function calls the model and returns an action list, or `None` when the agent considers the task complete. Users determine how to present this information to the model.

Listing 1: Minimal agent loop.

```
import os
import sys
from cua_speedrun.client import Computer

def run(env_url, task):
    computer = Computer(env_url)
    history = []
    limit = int(os.environ.get("CS_MAX_STEPS", "500"))
    for _ in range(limit):
        observation = computer.observe()
        actions = choose_actions(task, observation["png"], history)
        if actions is None:
            break
        computer.step(actions)
        history.append((observation["png"], actions))
    computer.done()

if __name__ == "__main__":
    run(sys.argv[1], sys.argv[2])
```

For example, an action list can contain `{"mouse": {"left_click": [640, 400]}}` followed by `{"keyboard": {"text": "hello"}}`. These two actions are executed in order within one step. Convenience methods such as `click(640, 400)`, `type_text("hello")`, and `keys(["ctrl", "c"])` each submit a single action. The agent can execute several actions before requesting another screenshot. Calling `done()` ends the interaction; the benchmark's verifier determines the score.

C.3 Hosted and Local Evaluation

Listing 2 runs the Codex template on the representative OSWorld set using Modal. The command specifies the submission directory, benchmark, and number of parallel evaluations. Omitting `--remote` runs both the agent and environment VMs on the evaluator host. For local desktop evaluation, this must be a Linux host with the VM runtime and hardware required by the benchmark, as well as model-serving resources when using open-weight models.

Listing 2: Evaluating a supplied agent template.

```
cua-speedrun run --remote \
    --submission templates/codex_cli \
    --benchmark benchmarks/osworld-energy50-representative \
    --agents-per-evaluation 1 --parallel-evaluations 8
```

Users can also submit agents through the dashboard. For each evaluation, we record the benchmark, hardware, evaluation settings, and task seeds, along with task scores and timings. Action logs and screenshots allow users to inspect how the agent completed or failed each task.

D Interaction and Token Measurements

Alongside performance, time, and cost, we measure how often an agent calls its model, how many actions it executes, and how many tokens it generates. These measurements support the analysis of agent speed in Sec. 3. The full configuration tables average over all tasks, including unsuccessful attempts.

Aggregate metrics. For an agent π evaluated on K tasks, the mean verifier score P, mean task time T, and mean model cost C are

$$P(\pi) = \frac{1}{K}\sum_{i=1}^{K} r_i, \qquad T(\pi) = \frac{1}{K}\sum_{i=1}^{K} t_i, \qquad C(\pi) = \frac{1}{K}\sum_{i=1}^{K} c_i, \tag{4}$$

where r_i is the verifier score defined in Appendix J.1, and t_i and c_i are the execution time and model cost of task i. The mean score retains partial credit; we also report the fraction of exactly completed tasks and the median time on those tasks.

Agent time, environment time, and requested waits. We separate total task time into agent time and environment time. For task i, environment time e_i sums the intervals recorded by the environment server for executing actions and returning observations. This includes explicit waits requested by the agent; we report their requested durations separately and subtract them when reporting environment time excluding agent waits. Agent time is the remainder, $t_i - e_i$, which includes model requests and agent-side processing.

D.1 Model Calls, Action Batches, and Individual Actions

Model responses. We count each completed model response, including responses without a computer action. A response can contain several tool calls, each of which may execute several actions. We therefore count model responses and desktop actions separately.

Action batches and individual actions. An agent can execute several actions in one request. For example, clicking a field, typing text, and pressing Enter in one request counts as one batch and three actions. Issuing these actions separately counts as three batches and three actions. A *step* denotes one request containing keyboard, mouse, or wait actions. For task i with S_i batches, let $\mathcal{A}_{ij}$ be the action list in request j. The total number of individual actions is $A_i = \sum_{j=1}^{S_i} |\mathcal{A}_{ij}|$.

We count actions as submitted by the agent: typing a string, pressing a key combination, or requesting a wait each counts as one action.

D.2 Token Accounting

We count every generated token once per response, including reasoning and other output, using the provider's reported usage. We also report provider-supplied reasoning-token counts separately.

To compare token rates, we divide total generated tokens by total agent time: $\sum_i G_i / \sum_i (t_i - e_i)$, where G_i is the generated-token count for task i. This *effective token rate* includes the time spent on model requests and agent-side processing.

D.3 Full Interaction and Token Comparisons

Tables 5 and 6 report mean batches, individual actions, model responses, and generated tokens for the configurations in Appendix E. Configuration IDs match the frontier plots and result tables.

Table 5: **Interaction and token counts on OSWorld.** Means per task, including unsuccessful tasks. IDs match the OSWorld configuration tables in Appendix E. Tokens include reasoning and other output.

ID	Configuration	Batches	Actions	Responses	Tokens
1	GPT-6 Astra / xhigh	7.14	19.90	–	2,060
2	Gemini 3.8 Flash / low	16.60	20.02	18.52	1,564
3	Claude Opus 5 / high	19.92	26.92	20.62	3,735
4	Gemini 3.8 Flash / medium	27.16	34.70	29.10	5,365
5	GPT-6 Astra / medium	6.30	19.26	–	1,366
6	GPT-6 Astra / high	6.52	19.76	–	1,593
7	GPT-6 Astra / low / fast	8.20	27.10	–	1,328
8	Claude Sonnet 5 / high	20.24	29.20	22.28	4,840
9	Gemini 3.7 Flash / medium	19.44	23.34	21.36	3,594
10	Gemini 3.7 Flash / high	24.26	28.30	26.20	5,731
11	Kimi K3 / max / batched	11.70	53.58	11.68	10,435
12	Claude Opus 5 / low	14.04	23.80	16.08	1,795
13	GPT-6 Astra / low	5.86	19.10	–	1,579
14	Gemini 3.7 Flash / low	18.24	20.88	20.14	2,944
15	Gemini 3.8 Flash / high	37.18	51.46	39.08	8,789
16	GPT-5.6 Luna / medium / API	14.42	48.88	15.34	1,751
17	GPT-5.6 Sol / xhigh	12.66	36.78	13.62	2,402
18	Kimi K3 / high / single	8.92	113.60	10.70	7,567
19	GPT-5.6 Sol / medium	12.72	42.64	13.68	2,717
20	GPT-5.6 Luna / high / Codex	15.58	39.80	–	8,637
21	Kimi K3 / high / batched	8.76	60.20	9.20	7,125
22	Muse Spark 1.1 / xhigh	69.82	145.56	26.02	10,701
23	GPT-5.6 Luna / xhigh / Codex	13.88	33.10	–	8,160
24	Muse Spark 1.3 / xhigh	56.58	86.72	30.62	11,505
25	Muse Spark 1.3 / medium	41.54	69.32	26.10	8,952
26	Claude Sonnet 5 / low	21.24	29.50	24.16	3,468
27	GPT-5.6 Luna / low / Codex	7.22	26.44	–	3,191
28	MiniMax M3 / thinking_off	34.32	51.10	35.14	–
29	GPT-5.6 Luna / high / API	18.28	53.88	19.24	3,358
30	Kimi K3 / low / batched	5.74	30.00	6.90	1,865
31	Muse Spark 1.3 / high	51.30	84.78	28.98	10,538
32	Kimi K3 / low / single	8.12	55.10	9.62	3,171
33	MiniMax M3 / thinking_on	35.00	55.30	35.94	–
34	Muse Spark 1.1 / medium	54.36	114.00	22.82	7,352
35	Muse Spark 1.1 / high	69.90	126.88	25.66	10,132
36	Muse Spark 1.1 / low	58.48	149.86	22.12	6,526
37	Muse Spark 1.2 / xhigh	66.54	200.52	43.08	17,198
38	GPT-5.6 Luna / medium / Codex	9.50	25.64	–	4,524
39	Muse Spark 1.1 / minimal	53.44	131.68	20.62	5,957
40	Muse Spark 1.3 / low	40.50	71.82	23.54	6,747
41	GPT-5.6 Sol / low	11.20	45.34	12.16	1,236
42	Muse Spark 1.2 / low	31.12	137.82	23.94	4,723
43	Muse Spark 1.2 / medium	44.20	152.84	28.34	7,789

Continued on next page

Table 5: OSWorld interaction counts, continued.

ID	Configuration	Batches	Actions	Responses	Tokens
44	GPT-5.6 Luna / low / API	10.12	37.66	11.06	1,063
45	Muse Spark 1.3 / minimal	32.04	59.52	19.76	3,792
46	Muse Spark 1.2 / high	45.72	145.58	33.74	9,551
47	Gemini 3 Flash Preview / high	33.28	34.56	37.86	5,167
48	GLM-5V Turbo / thinking_off	16.86	21.52	17.54	8,150
49	Gemini 3 Flash Preview / medium	28.26	30.68	32.52	4,125
50	GLM-5V Turbo / thinking_on	17.86	19.24	18.62	8,563
51	Muse Spark 1.2 / minimal	23.20	87.38	18.72	2,379
52	Gemini 3 Flash Preview / low	39.52	40.26	56.02	1,598
53	Yutori n2 / none	18.74	56.54	20.90	3,652
54	Yutori n2 / low	22.96	58.72	25.32	12,188
55	Yutori n2 / medium	19.06	55.14	21.12	7,562
56	Yutori n2 / xhigh	17.56	48.16	19.72	7,499

Table 6: **Interaction and token counts on OSWorld2.** Means per task, including unsuccessful tasks. IDs match the OSWorld2 configuration tables in Appendix E. Tokens include reasoning and other output.

ID	Configuration	Batches	Actions	Responses	Tokens
1	GPT-6 Astra / xhigh	52.81	194.10	–	18,332
2	GPT-6 Astra / high	51.29	149.02	–	14,146
3	GPT-6 Astra / low	42.58	151.40	–	10,190
4	GPT-6 Astra / medium	45.46	151.08	–	11,382
5	Gemini 3.8 Flash / high	216.54	331.88	218.48	72,149
6	Gemini 3.8 Flash / medium	167.79	252.65	169.65	49,532
7	Claude Opus 5 / low	146.52	316.17	158.29	30,820
8	Claude Sonnet 5 / low	237.25	378.54	253.50	57,018
9	Gemini 3.8 Flash / low	127.04	200.75	128.83	19,840
10	Muse Spark 1.3 / minimal	115.35	488.06	66.02	16,150
11	Muse Spark 1.3 / low	130.50	655.08	80.73	25,131
12	Muse Spark 1.3 / medium	149.81	832.44	89.75	41,149
13	Muse Spark 1.3 / xhigh	147.29	832.75	93.96	52,628
14	Muse Spark 1.3 / high	151.21	754.98	89.83	44,241
15	Kimi K3 / low	47.25	313.75	49.17	21,852
16	MiniMax M3 / thinking_on	97.56	407.33	98.62	–
17	MiniMax M3 / thinking_off	98.54	368.62	98.65	–
18	GLM-5V Turbo / thinking_off	49.15	79.35	48.38	28,558
19	GLM-5V Turbo / thinking_on	47.75	91.75	47.12	28,385
20	Kimi K3 / high	58.77	350.06	59.04	81,494
21	Kimi K3 / max	66.38	386.56	60.79	123,603

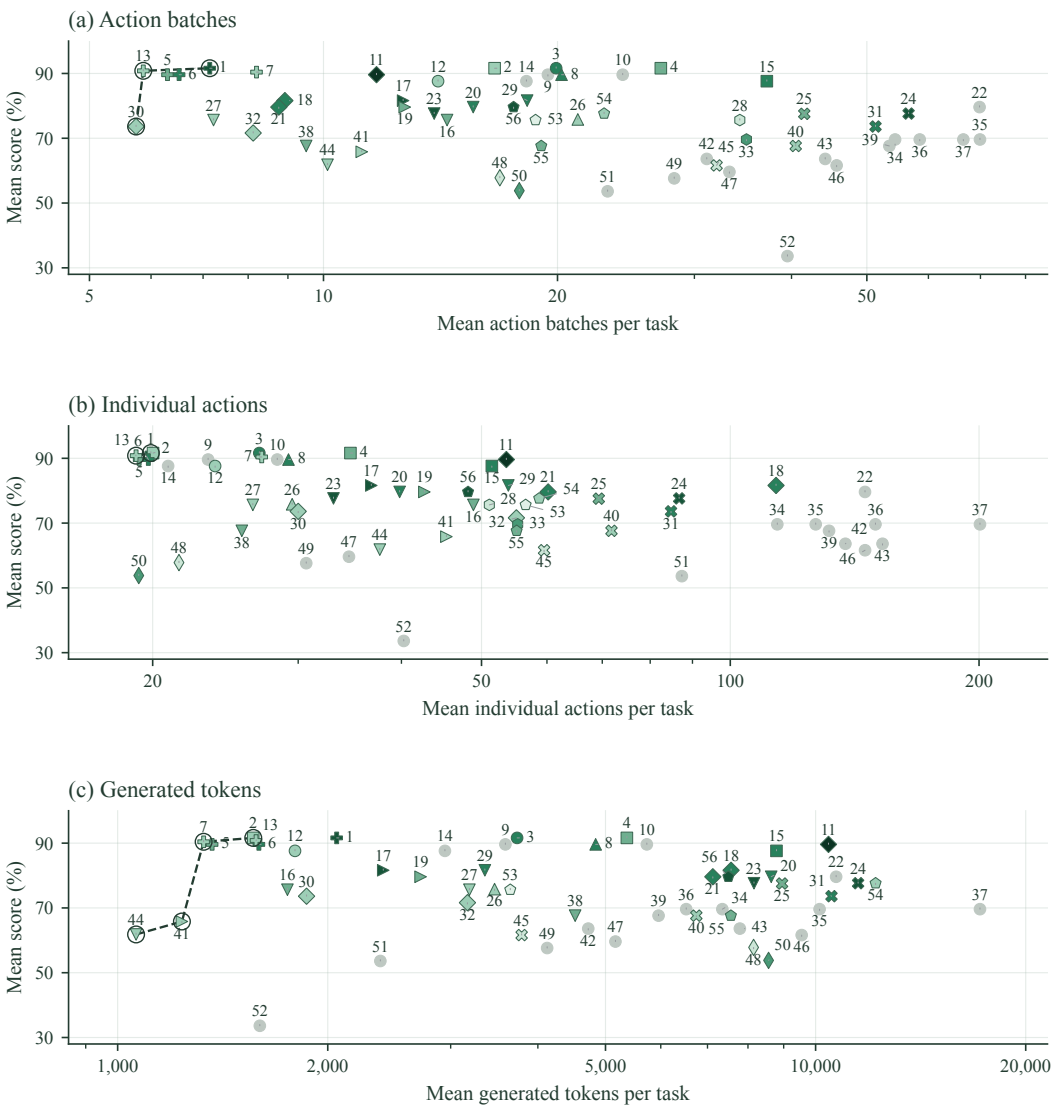

Figure 7: **Performance–interaction and performance–token frontiers on OSWorld.** Mean score against mean action batches, individual actions, and generated tokens per task. Tokens include reasoning and other output. Panels (a)–(b) show all 56 configurations; panel (c) shows the 54 with token counts. Dashed lines connect the circled frontier points. IDs match the configuration tables in Appendix E; darker green indicates greater reasoning effort.

D.4 Performance–Interaction and Performance–Token Frontiers

Figures 7 and 8 compare performance with the number of action batches, individual actions, and generated tokens on OSWorld and OSWorld2. Circled points mark the Pareto frontier for each metric. We use the same trajectories as the time and cost comparisons and average over all tasks.

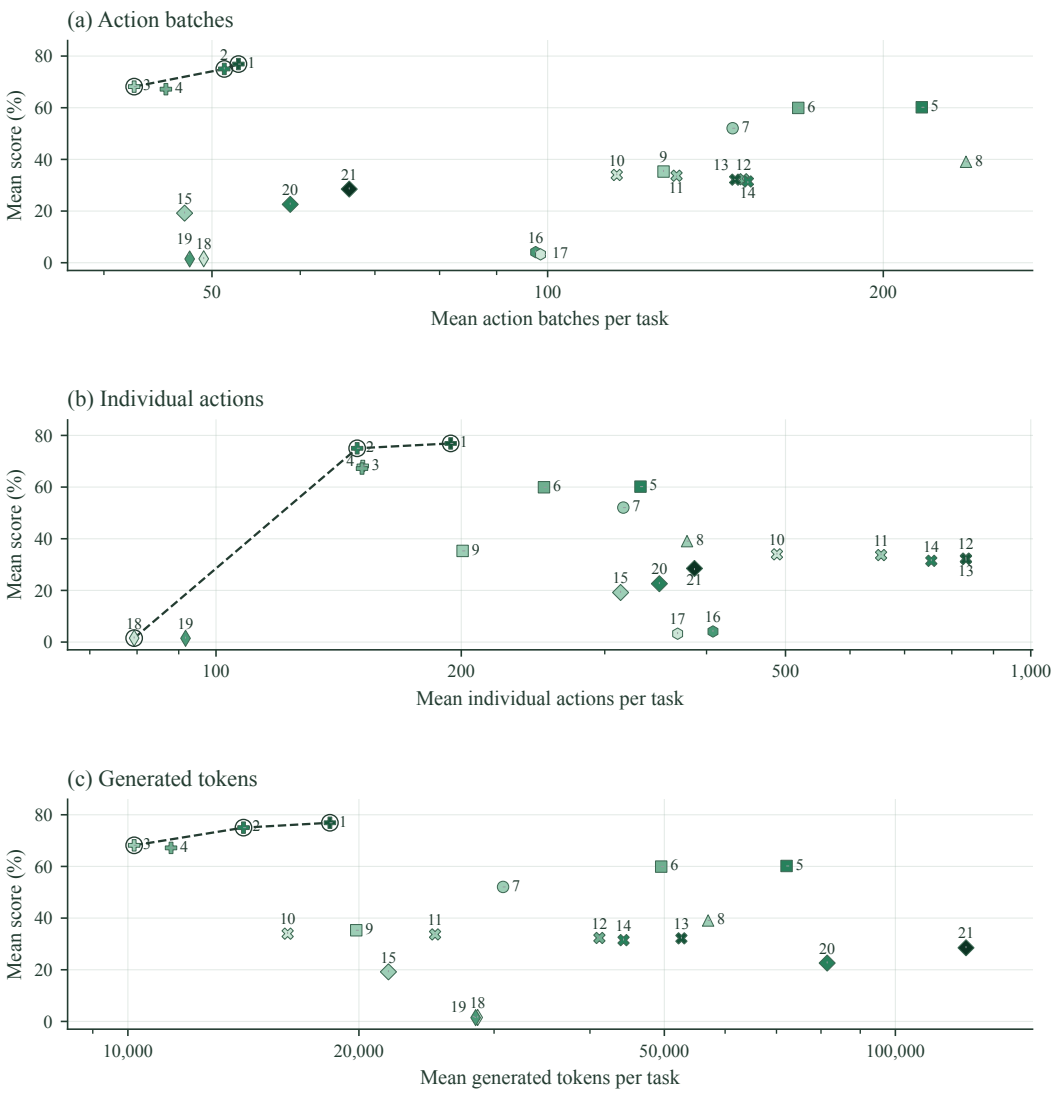

Figure 8: **Performance–interaction and performance–token frontiers on OSWorld2.** Mean score against mean action batches, individual actions, and generated tokens per task. Panels (a)–(b) show all 21 configurations; panel (c) shows the 19 with token counts. IDs match the OSWorld2 configuration table in Appendix E. Other plotting conventions follow Figure 7.

E Full Evaluation Results

E.1 Model, Harness, and Reasoning-Effort Configurations

Tables 7–9 report mean score, time, cost, and token use for each model, harness, and reasoning setting. The configuration IDs identify the corresponding points in the figures.

We evaluate Luna through both the direct API and Codex, and Kimi with single-call and batched-call harnesses. We use the Yutori cookbook.

ID	Configuration	Score (%)	Time (s)	Cost ($)	Tokens
1	GPT-6 Astra / xhigh	91.6	126.8	0.708	2,060
2	Gemini 3.8 Flash / low	91.6	127.1	0.108	1,564
3	Claude Opus 5 / high	91.6	136.0	0.661	3,735
4	Gemini 3.8 Flash / medium	91.6	250.1	0.221	5,365
5	GPT-6 Astra / medium	89.6	90.2	0.600	1,366
6	GPT-6 Astra / high	89.6	100.8	0.636	1,593
7	GPT-6 Astra / low / fast	90.4	99.0	0.908	1,328
8	Claude Sonnet 5 / high	89.6	140.8	0.309	4,840
9	Gemini 3.7 Flash / medium	89.6	198.9	0.944	3,594
10	Gemini 3.7 Flash / high	89.6	254.3	1.412	5,731
11	Kimi K3 / max / batched	89.6	443.7	0.503	10,435
12	Claude Opus 5 / low	87.6	85.9	0.390	1,795
13	GPT-6 Astra / low	90.8	89.5	0.531	1,579
14	Gemini 3.7 Flash / low	87.6	207.3	0.926	2,944
15	Gemini 3.8 Flash / high	87.6	347.7	0.347	8,789
16	GPT-5.6 Luna / medium / API	75.6	118.9	0.019	1,751
17	GPT-5.6 Sol / xhigh	81.6	108.3	0.527	2,402
18	Kimi K3 / high / single	81.6	305.1	0.414	7,567
19	GPT-5.6 Sol / medium	79.6	121.3	0.539	2,717
20	GPT-5.6 Luna / high / Codex	79.6	324.8	0.076	8,637
21	Kimi K3 / high / batched	79.6	358.1	0.382	7,125
22	Muse Spark 1.1 / xhigh	79.6	417.2	1.685	10,701
23	GPT-5.6 Luna / xhigh / Codex	77.6	297.2	0.063	8,160
24	Muse Spark 1.3 / xhigh	77.6	612.1	0.161	11,505
25	Muse Spark 1.3 / medium	77.5	397.6	0.126	8,952
26	Claude Sonnet 5 / low	75.8	129.4	0.294	3,468

Table 7: OSWorld configurations. Bold IDs mark the joint performance–time–cost frontier. Tokens include reasoning and other generated output.

ID	Configuration	Score (%)	Time (s)	Cost ($)	Tokens
27	GPT-5.6 Luna / low / Codex	75.6	144.8	0.025	3,191
28	MiniMax M3 / thinking_off	75.6	253.8	–	–
29	GPT-5.6 Luna / high / API	81.6	160.5	0.029	3,358
30	Kimi K3 / low / batched	73.6	149.5	0.174	1,865
31	Muse Spark 1.3 / high	73.6	506.9	0.148	10,538
32	Kimi K3 / low / single	71.6	176.6	0.280	3,171
33	MiniMax M3 / thinking_on	69.6	345.1	–	–
34	Muse Spark 1.1 / medium	69.6	359.8	1.379	7,352
35	Muse Spark 1.1 / high	69.6	368.5	1.706	10,132
36	Muse Spark 1.1 / low	69.6	370.4	1.312	6,526
37	Muse Spark 1.2 / xhigh	69.6	720.9	0.244	17,198
38	GPT-5.6 Luna / medium / Codex	67.6	188.8	0.035	4,524
39	Muse Spark 1.1 / minimal	67.6	318.5	1.184	5,957
40	Muse Spark 1.3 / low	67.6	361.0	0.107	6,747
41	GPT-5.6 Sol / low	65.8	100.1	0.319	1,236
42	Muse Spark 1.2 / low	63.6	363.6	0.105	4,723
43	Muse Spark 1.2 / medium	63.6	466.2	0.136	7,789
44	GPT-5.6 Luna / low / API	61.8	73.6	0.011	1,063
45	Muse Spark 1.3 / minimal	61.6	264.4	0.079	3,792
46	Muse Spark 1.2 / high	61.6	669.8	0.174	9,551
47	Gemini 3 Flash Preview / high	59.6	337.2	0.809	5,167
48	GLM-5V Turbo / thinking_off	57.8	752.4	0.220	8,150
49	Gemini 3 Flash Preview / medium	57.6	268.0	0.592	4,125
50	GLM-5V Turbo / thinking_on	53.8	359.2	0.235	8,563
51	Muse Spark 1.2 / minimal	53.6	248.8	0.070	2,379
52	Gemini 3 Flash Preview / low	33.6	492.2	1.401	1,598
53	Yutori n2 / none	75.6	298.5	0.081	3,652
54	Yutori n2 / low	77.6	331.0	0.157	12,188
55	Yutori n2 / medium	67.6	265.4	0.110	7,562
56	Yutori n2 / xhigh	79.6	225.0	0.104	7,499

Table 8: OSWorld configurations, continued. Bold IDs mark the joint performance–time–cost frontier. Tokens include reasoning and other generated output.

ID	Configuration	Score (%)	Time (s)	Cost ($)	Tokens
1	GPT-6 Astra / xhigh	76.9	1237.3	8.175	18,332
2	GPT-6 Astra / high	75.0	914.0	7.391	14,146
3	GPT-6 Astra / low	68.2	904.2	6.336	10,190
4	GPT-6 Astra / medium	67.2	828.1	6.519	11,382
5	Gemini 3.8 Flash / high	60.2	3058.8	5.019	72,149
6	Gemini 3.8 Flash / medium	59.9	2051.8	2.915	49,532
7	Claude Opus 5 / low	52.1	1385.7	10.006	30,820
8	Claude Sonnet 5 / low	39.1	1993.6	8.863	57,018
9	Gemini 3.8 Flash / low	35.3	1386.7	1.875	19,840
10	Muse Spark 1.3 / minimal	34.0	1516.0	0.390	16,150
11	Muse Spark 1.3 / low	33.7	2280.8	0.503	25,131
12	Muse Spark 1.3 / medium	32.3	2292.1	0.588	41,149
13	Muse Spark 1.3 / xhigh	32.2	2261.0	0.640	52,628
14	Muse Spark 1.3 / high	31.5	2179.3	0.599	44,241
15	Kimi K3 / low	19.2	1860.9	2.088	21,852
16	MiniMax M3 / thinking_on	4.1	1673.2	–	–
17	MiniMax M3 / thinking_off	3.2	1446.9	–	–
18	GLM-5V Turbo / thinking_off	1.5	1979.4	0.739	28,558
19	GLM-5V Turbo / thinking_on	1.5	2641.9	0.725	28,385
20	Kimi K3 / high	22.6	3733.9	4.517	81,494
21	Kimi K3 / max	28.5	4825.1	5.625	123,603

Table 9: OSWorld2 configurations. Bold IDs mark the joint performance–time–cost frontier. Tokens include reasoning and other generated output.

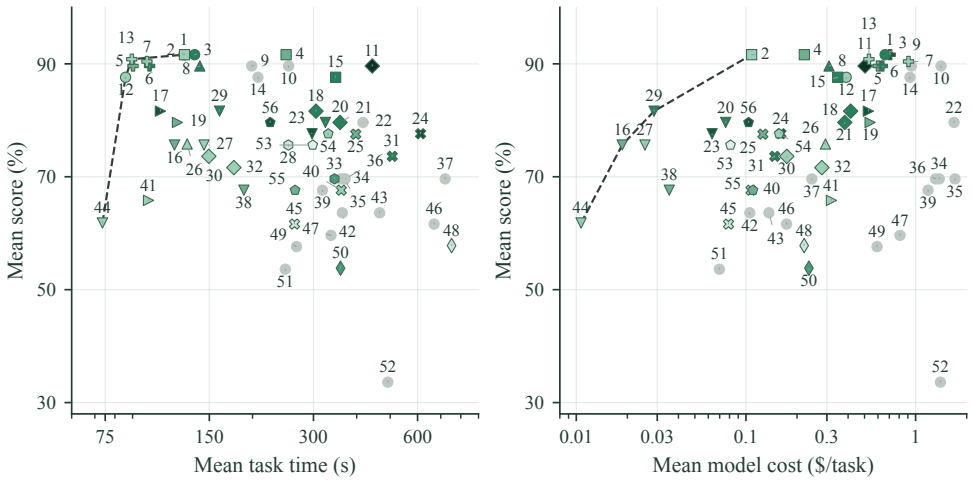

Figure 9: **Full OSWorld performance–time and performance–cost comparisons.** All 56 configurations appear in the time comparison. IDs identify the model, reasoning effort, and harness in Tables 7–8. Dashed lines connect nondominated configurations for each pair of axes. Scores, times, and costs are means across the same 50 tasks, including unsuccessful tasks.

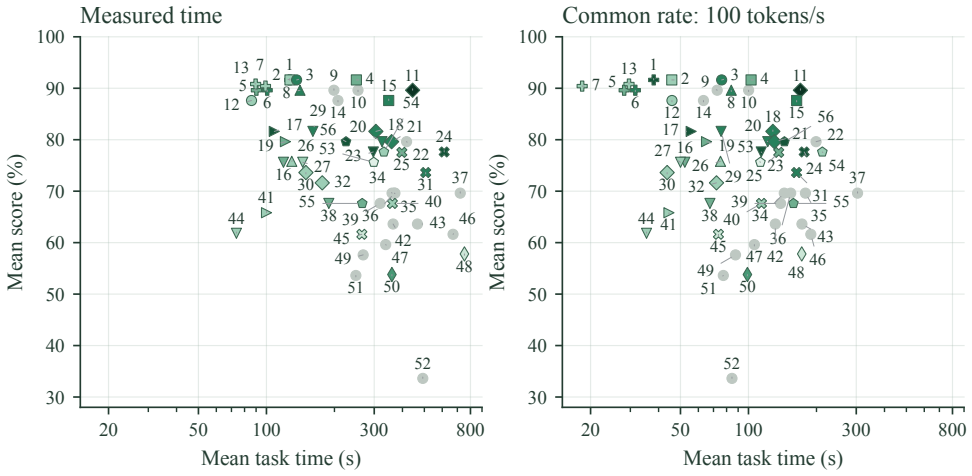

Figure 10: **Full OSWorld comparison at a common token rate.** Both plots use the same 54 configurations with token counts. The right plot replaces measured mean task time with mean recorded environment time plus generated tokens divided by 100 tokens/s. Tokens include reasoning and other output; scores and trajectories are unchanged. Point IDs refer to the OSWorld configuration tables.

E.2 OSWorld: Performance–Time and Performance–Cost Frontiers

Figure 9 shows the full OSWorld comparison from Figure 2, with configuration IDs for all 56 points.

A common token-generation rate. Figure 10 compares measured task time with an estimate that assigns every model the same generation rate. We compute this estimate as recorded environment time plus generated tokens divided by 100 tokens/s, using the same trajectories and scores. Opus 5 (low) scores 87.6% in 85.9 s, while Astra (low) scores 90.8% in 89.5 s. Opus generates 1,795 tokens per task compared with 1,579 for Astra. With generation time assigned at the common rate, Astra is faster. Environment time includes agent-requested waits (Appendix D).

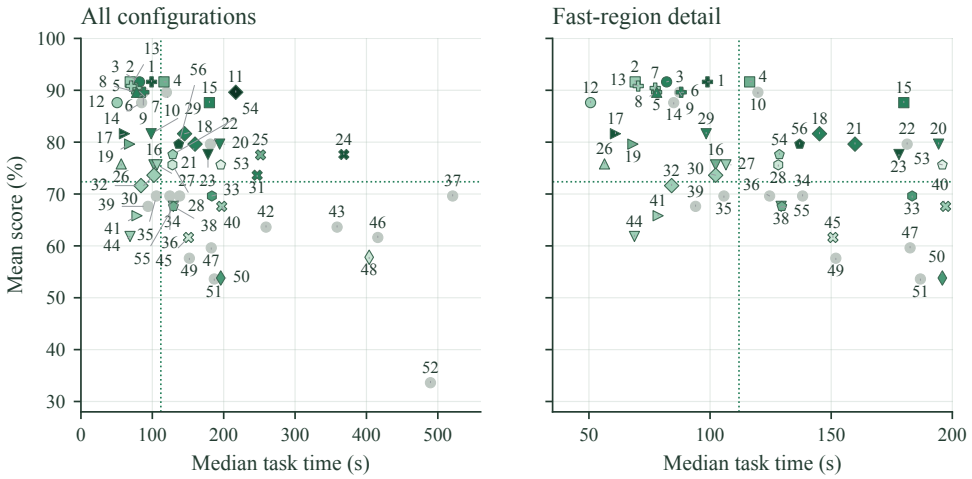

Figure 11: **Full OSWorld median-time view and fast-region detail.** Mean verifier score versus median task time for all 56 configurations. The right plot enlarges the region containing the fastest configurations; point IDs match the full OSWorld tables. Green dotted lines show the published human references of 72.36% and 111.94 s from the original 369-task OSWorld benchmark (Xie et al., 2024).

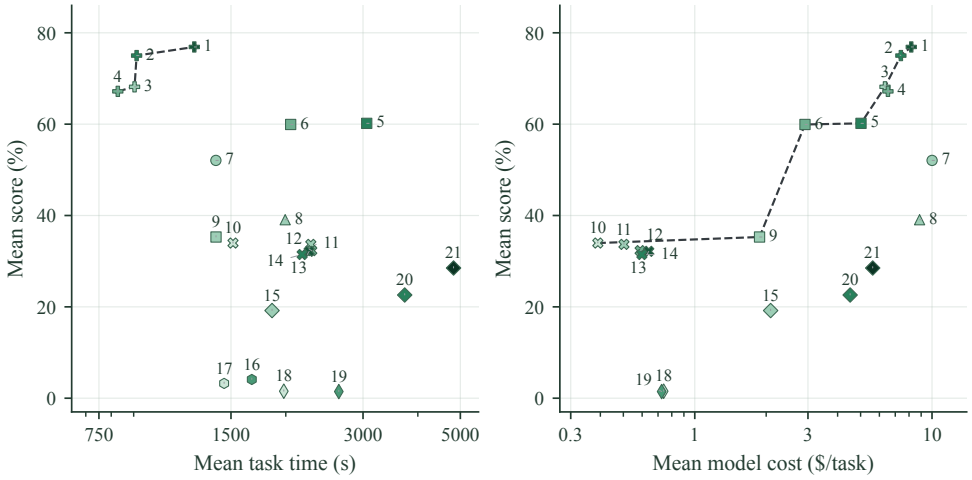

Figure 12: **Full OSWorld2 performance–time and performance–cost comparisons.** All 21 configurations are evaluated on the same 52 tasks. Point IDs refer to the OSWorld2 configuration table. Marker shapes follow Figure 2; darker green indicates greater reasoning effort.

Median task time. Figure 11 reports median task time, which summarizes the time for a typical task.

E.3 OSWorld2: Performance–Time and Performance–Cost Frontiers

Figure 12 compares all 21 OSWorld2 configurations. Astra defines the entire time frontier, while Gemini 3.8 Flash and Muse Spark 1.3 offer lower-cost choices on the cost frontier.

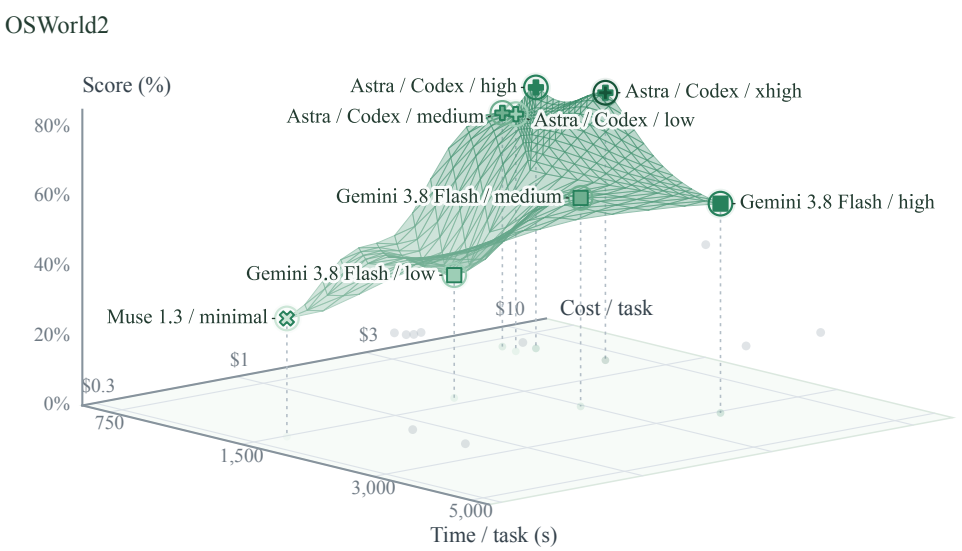

Figure 13: **Joint performance–time–cost frontier on OSWorld2.** Circled points are the evaluated frontier configurations. Other configurations appear in gray. Time and cost axes are logarithmic. Marker shapes follow Figure 2; darker green indicates greater reasoning effort.

E.4 OSWorld2: Joint Performance–Time–Cost Frontier

Figure 13 extends the joint frontier in Figure 6 to OSWorld2. A configuration is on this frontier if no other evaluated configuration improves one of the three metrics without worsening another. It includes Astra at all four reasoning settings, Gemini 3.8 Flash at all three settings, and Muse Spark 1.3 at minimal effort, allowing a choice based on the desired score and the available time and cost budgets.

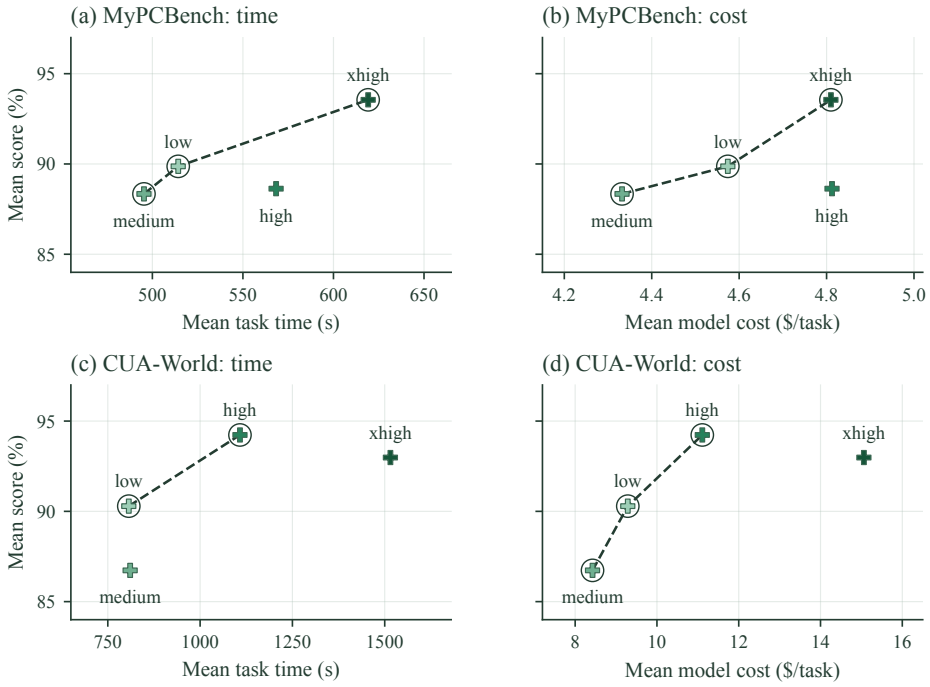

Figure 14: **Performance–time and performance–cost frontiers on MyPCBench and CUA-World.** GPT-6 Astra through Codex at four reasoning settings. Each point averages the same 38 MyPCBench tasks (top) or 26 CUA-World tasks (bottom). Darker green indicates greater reasoning effort. Dashed lines connect the observed Pareto-frontier points, which are circled.

	MyPCBench (38 tasks)			CUA-World (26 tasks)		
Effort	Score (%)	Time (s)	Cost ($)	Score (%)	Time (s)	Cost ($)
Low	89.9	514	4.57	90.3	**807**	9.29
Medium	88.3	**495**	**4.33**	86.7	810	**8.42**
High	88.6	568	4.81	**94.2**	1,108	11.11
Xhigh	**93.6**	619	4.81	93.0	1,515	15.06

Table 10: **Reasoning effort on MyPCBench and CUA-World.** GPT-6 Astra through Codex, reporting mean partial-credit score, time, and model cost per task. Bold indicates the highest score, lowest time, or lowest cost within each benchmark.

E.5 Reasoning Effort on MyPCBench

Figure 14 and Table 10 compare GPT-6 Astra through Codex at four reasoning settings on MyPCBench. Xhigh achieves the highest score, 93.6%, while medium has the lowest mean time and cost, 495 s and $4.33 per task. We evaluate the 38-task representative set from Sec. 2.2, with a limit of 100 action batches. All reported means include unsuccessful tasks.

E.6 Reasoning Effort on CUA-World

Figure 14 and Table 10 show a different outcome on CUA-World. High effort scores 94.2%, compared with 93.0% at xhigh, while taking less time (1,108 vs. 1,515 s) and costing less ($11.11 vs. $15.06 per task). All four settings use Astra through Codex on the same 26 long-horizon tasks, with a limit of 500 action batches.

Run	Score (%)	Time (s)	Agent (s)	Steps	Env. (s)	Waits (s)	Env.−waits (s)
1	87.62	86.11	72.90	5.90	13.209	1.328	11.881
2	91.62	89.56	75.31	5.82	14.248	1.082	13.166
3	91.62	90.13	76.52	5.88	13.606	1.040	12.566
4	91.62	89.81	75.78	5.78	14.033	1.152	12.881
5	91.62	92.10	78.19	5.94	13.914	1.012	12.902
Mean	90.82	89.54	75.74	5.86	13.802	1.123	12.679
SD	1.79	2.16	1.93	0.06	0.405	0.126	0.494

Table 11: **Variation across five seeds for GPT-6 Astra at low reasoning effort.** Each row averages all 50 OSWorld tasks. SD is the sample standard deviation of these means across the five runs; score SD is in percentage points. Steps count action batches, and waits are explicitly requested by the agent.

F Run-to-Run Variation

We repeat the evaluation across five seeds to measure variation in scores, task times, and action sequences. We also compare scores on individual tasks to determine whether the same tasks succeed across runs. Appendix B.6 reports the corresponding timing results for Fast I/O.

F.1 Five-Seed Evaluation Setup

We evaluate GPT-6 Astra at low reasoning effort using Codex on the same 50 OSWorld tasks. Each run uses the same agent prompt and a 500-step limit. Only the task seed changes. The full-result tables and figures use five-run averages for each I/O setting. Median task times are computed within each run and then averaged across runs.

F.2 Scores and Times Across Seeds

Table 11 shows that both score and task time vary little across seeds. Mean task time is 89.54 ± 2.16 s, and the score is $90.82 \pm 1.79\%$ (mean $\pm$ standard deviation). The mean number of action batches is also stable at 5.86 ± 0.06 per task. Timing components are defined in Appendix D.

F.3 TASK-LEVEL SCORE AGREEMENT

I/O	k	Pass@k (%)	Best-of-k partial score (%)
Standard I/O	1	87.20	90.82
	2	88.00	91.62
	3	88.00	91.62
	4	88.00	91.62
	5	88.00	91.62
Fast I/O	1	86.80	90.42
	2	88.80	92.42
	3	89.20	92.82
	4	89.60	93.22
	5	90.00	93.62

Table 12: **Task-level repeatability across five seeds of Astra at low effort.** Values average uniformly over all size-k subsets of the five runs, then over the 50 tasks. Pass@k measures whether any of the k attempts succeeds; best-of-k uses the highest partial score.

Table 12 shows that additional attempts improve performance only slightly. With standard I/O, 42 tasks receive full credit in all five runs, two in four runs, and six in none. Partial-credit scores are identical across all five runs on 48 of the 50 tasks. The low variation in mean score therefore reflects consistent performance on individual tasks: almost every task receives the same score on every attempt. With Fast I/O, 40 tasks receive full credit in all five runs, four in four runs, one in one run, and five in none. Partial-credit scores are identical on 45 tasks.

Pass@k and partial credit. For a task with c fully successful runs among the five, the probability that a uniformly selected set of k runs contains a success is $1 - \binom{5-c}{k} / \binom{5}{k}$. Pass@$k$ averages this quantity over tasks. For partial credit, the table averages the maximum task score within each size-k set. Standard-I/O pass@1 is 87.2% and pass@5 is 88.0%; its best-of-five partial score is 91.62%.

F.4 VARIATION IN ACTION SEQUENCES

We next compare the number of action batches used for the same task across seeds. For each task, we compute the standard deviation and range across its five trajectories. With standard I/O, the median of these task-level standard deviations is 0.55 batches, and the median range is one batch. For Fast I/O, the corresponding values are 0.99 and two batches.

Two ways to complete the same spreadsheet task. For example, on the Calc task that cleans movie titles, all five standard-I/O runs receive full credit while taking 7, 6, 5, 6, and 2 batches. The seven-batch trajectory enters separate `PROPER(TRIM(...))` formulas for successive rows and revisits the paste-special dialog. The two-batch trajectory selects `C2:C29`, enters one formula with Alt+Enter to fill the selection, then saves. The two trajectories take 97.4 and 40.3 s, respectively. Here, the agent reduces the number of batches by applying the formula to all rows at once.

G REPRESENTATIVE TASK SELECTION

We select a small set of tasks from each benchmark to reduce evaluation time while preserving the scores and relative performance of agents. This section gives the selection procedure from Sec. 2.2 and tests its score estimates, rankings, and speed–performance frontier on held-out models. Table 13 lists the resulting task counts.

Benchmark	Selection pool	Selected subset
OSWorld (Xie et al., 2024)	295	50
OSWorld2 (Yuan et al., 2026)	63	52
CUA-World (Aggarwal et al., 2026)	143	26
MyPCBench (Jang et al., 2026a)	184	38

Table 13: Task counts for the representative subsets selected for `cua-speedrun`.

G.1 MODELS AND TASKS USED FOR SELECTION

OSWorld. We use nine configurations evaluated on the same 295 tasks: Claude Sonnet 5 xhigh, Gemini 3.6 Flash, Gemini 3 Flash Preview, GLM-5V Turbo, GPT-5.6 Luna xhigh, Kimi K3, Meta Muse Spark, MiniMax M3, and Qwen3.5-9B-Thinking. Each task is represented by its nine partial scores and nine exact-completion indicators. The internet-dependence audit defining this task population is described in Appendix H.

OSWorld2. Selection uses seven configurations evaluated on 63 tasks: MiniMax-M3, Claude Opus 4.7, Claude Sonnet 4.6 at medium and maximum effort, GPT-5.5, GPT-5.6, and Qwen3.7. The resulting evaluation set contains 52 tasks.

CUA-World. Selection uses seven configurations with scores on more than 30 tasks each: Qwen3-VL-2B-Thinking, GPT-5.4 through Azure, GPT-5.4, Claude Opus 4.7, Claude Sonnet 4.6, Gemini 3 Flash Preview, and Kimi K2.5. Selection uses each model's distribution of observed partial scores. Repeated runs remain separate observations. The selected set contains 26 tasks.

MyPCBench. The evaluation uses 38 of the 184 canonical tasks, selected with the unweighted energy method and 100 initializations. The task identities for all four benchmarks are listed in Appendix G.6.

G.2 INITIALIZATION, OPTIMIZATION, AND TASK COUNT

We seek a subset that represents the full benchmark's distribution of score patterns. For OSWorld and OSWorld2, let z_i concatenate the partial scores and exact-completion indicators for task i. Distances are $d_{ij} = \|z_i - z_j\|_2$. For a subset S of size K from N tasks, the normalized energy objective is

$$E(S) = \frac{1}{\bar{d}} \left[\frac{2}{KN} \sum_{i \in S} \sum_{j=1}^{N} d_{ij} - \frac{1}{K^2} \sum_{i,j \in S} d_{ij} - \frac{1}{N^2} \sum_{i,j=1}^{N} d_{ij} \right], \qquad \bar{d} = \frac{\sum_{i \neq j} d_{ij}}{N(N-1)}. \tag{5}$$

We estimate a model's benchmark score by its unweighted mean score on the selected tasks.

Balanced initializations. To cover a range of score profiles in each initial subset, we sort tasks by the leading principal coordinate of their centered score profiles and choose evenly spaced tasks from this ordering. Specifically, for a uniformly sampled phase $\phi \in [0, 1)$, the initial positions are $\lfloor (\phi + t)N/K \rfloor$ for $t = 0, \ldots, K-1$. We use 100 initializations with seeds $7000, \ldots, 7099$, optimizing each distinct initial subset once.

One-swap optimization. At each iteration, we evaluate every exchange of a selected task with an unselected task and apply the exchange that reduces energy the most. We stop when no exchange improves the objective beyond numerical tolerance and retain the lowest-energy subset across the initializations.

Listing 3: Representative-task selection and validation.

```
for each candidate task count K:
    for each model h:
        remove model h from the calibration models
        form task profiles from the remaining models
        optimize 100 balanced initial subsets by one-task swaps
        select the subset with the lowest calibration energy
        predict h's partial and exact scores by subset means
    compare predictions with each model's full-set means

choose the smallest K for which both rank correlations
are at least 0.95 at K-1, K, and K+1
refit the K-task subset using all models
```

To choose the number of tasks, we repeat the full selection procedure with each model held out in turn (Listing 3). We construct task profiles and select tasks using the remaining models, then compare the held-out model's subset score with its full-set score. We choose the smallest K for which both partial-score and exact-completion rank correlations are at least 0.95 at $K-1$, K, and $K+1$. Checking the neighboring task counts tests the stability of the ranking criterion. Each task count is optimized independently. After choosing K, we select the final evaluation set using all models.

Selection with missing scores. Some CUA-World models have scores on only part of the benchmark. We therefore compare each model's distribution of observed partial scores on selected tasks with its distribution over all observed tasks, using one-dimensional energy distance. We first ensure that the subset contains an observation for every model used for selection, then minimize the mean of these distances. The task-count rule uses partial-score rank correlation at $K-1, K, K+1$. At $K = 25, 26, 27$, the seven-model held-out correlations are 0.964, 1.000, and 1.000, respectively; the corresponding score errors are 7.43, 5.12, and 2.10 percentage points. These errors compare unweighted means of observed run scores.

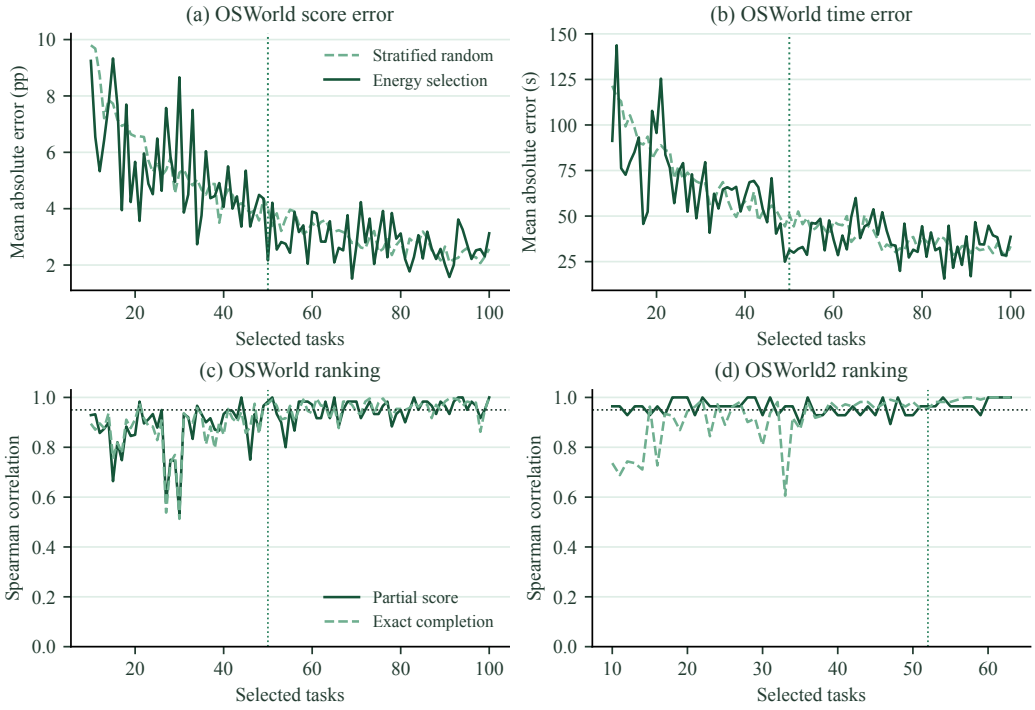

Figure 15: **Score error and rank correlation on held-out models.** OSWorld uses nine leave-one-agent-out folds; OSWorld2 uses seven. The randomized baseline averages 300 difficulty-stratified selections per task count. Dotted vertical lines mark 50 and 52 tasks; horizontal lines mark rank correlation 0.95. Each task count is optimized independently.

G.3 Held-Out Score and Ranking Accuracy

Figure 15 compares energy-based selection with a randomized baseline that samples tasks across difficulty levels. We measure error as the mean absolute difference between each held-out model's subset estimate and its full-set result. At 50 OSWorld tasks, energy-based selection reduces partial-score error from 4.02 to 2.16 percentage points. It also reduces mean-task-time error from 51.0 to 31.4 s, even though selection uses only scores. Thus, selecting tasks with representative score patterns also improves estimates of execution time.

Benchmark	K	Partial ρ	Exact ρ	Partial MAE	Exact MAE
OSWorld	49	0.967	0.967	4.36	4.57
	50	0.983	0.975	2.16	2.17
	51	1.000	0.992	4.21	3.95
OSWorld2	51	0.964	0.955	1.70	2.28
	52	0.964	0.955	1.33	2.03
	53	0.964	0.982	0.74	1.04

Table 14: **Held-out-model validation around the selected task count.** MAE is in percentage points. Each model is excluded from task selection in its fold. Both correlations exceed 0.95 at the selected task count and its two neighbors.

Table 14 shows that the selected task sets closely preserve model rankings on both benchmarks. At 50 tasks, OSWorld achieves partial-score and exact-completion rank correlations of 0.983 and 0.975; at 52 tasks, OSWorld2 achieves 0.964 and 0.955. Both correlations remain above 0.95 at the neighboring task counts.

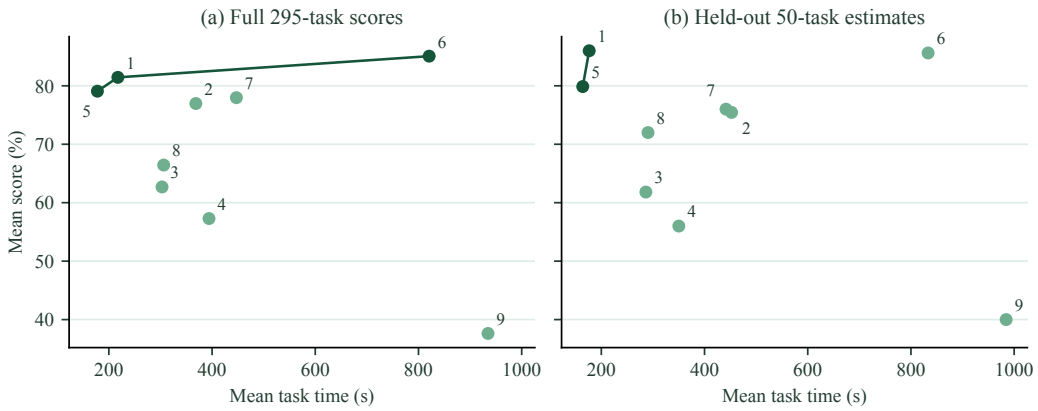

Figure 16: **Speed–performance frontiers from full-set scores and held-out subset estimates.** Dark points and connecting lines mark nondominated configurations. Model IDs are: 1, Sonnet 5; 2, Gemini 3.6 Flash; 3, Gemini 3 Flash Preview; 4, GLM-5V Turbo; 5, Luna; 6, Kimi K3; 7, Muse Spark; 8, MiniMax M3; 9, Qwen3.5-9B. Each point in (b) uses a 50-task subset selected with that model held out.

G.4 Preserving the Speed–Performance Frontier

Figure 16 compares the full-set frontier with the frontier estimated from 50-task subsets for held-out models. The estimated frontier recovers two of the three full-set frontier configurations and adds no others, giving 100% precision and 66.7% recall. Pairwise dominance agrees for 91.7% of model pairs. This analysis uses the models' scores and times; selection itself uses only scores.

Evaluation savings. Selecting 50 of the 295 OSWorld tasks reduces the number of tasks by $5.9\times$. Across the nine models, these tasks account for 15.1% of the total recorded task time, a $6.61\times$ reduction. We compute this saving by summing trajectory times, excluding setup and verification.

Method	Partial MAE	Exact MAE	Mean MAE
Uniform random	5.88	6.01	5.95
Difficulty-stratified random	5.05	5.11	5.08
IRT-inspired selection	7.94	8.21	8.08
Energy-based selection	**4.50**	**4.34**	**4.42**

Table 15: **Task-selection methods on OSWorld with 32 tasks.** Errors are averaged across nine leave-one-agent-out folds and reported in percentage points. Mean MAE averages partial-score and exact-completion MAE. Random baselines average 300 selections per fold.

G.5 Comparing Task-Selection Methods

Table 15 compares task-selection methods at a budget of 32 OSWorld tasks, using the same nine leave-one-agent-out folds. Energy-based selection achieves a mean error of 4.42 percentage points, compared to 5.08 for difficulty-stratified random selection and 8.08 for the IRT-inspired method.

We use energy-based selection for its simple scoring rule: each agent's benchmark score is the average of its scores on the selected tasks. The IRT-inspired method fits a one-parameter ability model using the eight agents available in each fold and predicts full-benchmark performance from the selected-task scores.

G.6 Selected Task Identities

The following lists give the final task sets used in our evaluations. OSWorld IDs use the original task UUIDs. OSWorld2 uses its three-digit identifiers; MyPCBench IDs combine the task family and suffix; CUA-World IDs combine the environment and task name.

G.6.1 OSWorld

Table 16: Selected OSWorld task identities.

Application	Task UUID
chrome	`06fe7178-4491-4589-810f-2e2bc9502122`
chrome	`2ae9ba84-3a0d-4d4c-8338-3a1478dc5fe3`
chrome	`3720f614-37fd-4d04-8a6b-76f54f8c222d`
chrome	`44ee5668-ecd5-4366-a6ce-c1c9b8d4e938`
chrome	`af630914-714e-4a24-a7bb-f9af687d3b91`
gimp	`045bf3ff-9077-4b86-b483-a1040a949cff`
gimp	`554785e9-4523-4e7a-b8e1-8016f565f56a`
gimp	`734d6579-c07d-47a8-9ae2-13339795476b`
gimp	`a746add2-cab0-4740-ac36-c3769d9bfb46`
libreoffice_calc	`0326d92d-d218-48a8-9ca1-981cd6d064c7`
libreoffice_calc	`1de60575-bb6e-4c3d-9e6a-2fa699f9f197`
libreoffice_calc	`1e8df695-bd1b-45b3-b557-e7d599cf7597`
libreoffice_calc	`3a7c8185-25c1-4941-bd7b-96e823c9f21f`
libreoffice_calc	`4de54231-e4b5-49e3-b2ba-61a0bec721c0`
libreoffice_calc	`535364ea-05bd-46ea-9937-9f55c68507e8`
libreoffice_calc	`7efeb4b1-3d19-4762-b163-63328d66303b`
libreoffice_calc	`8b1ce5f2-59d2-4dcc-b0b0-666a714b9a14`
libreoffice_calc	`a9f325aa-8c05-4e4f-8341-9e4358565f4f`
libreoffice_calc	`abed40dc-063f-4598-8ba5-9fe749c0615d`
libreoffice_impress	`39be0d19-634d-4475-8768-09c130f5425d`
libreoffice_impress	`455d3c66-7dc6-4537-a39a-36d3e9119df7`
libreoffice_impress	`986fc832-6af2-417c-8845-9272b3a1528b`
libreoffice_impress	`a097acff-6266-4291-9fbd-137af7ecd439`
libreoffice_impress	`a434992a-89df-4577-925c-0c58b747f0f4`
libreoffice_impress	`af2d657a-e6b3-4c6a-9f67-9e3ed015974c`
libreoffice_impress	`b8adbc24-cef2-4b15-99d5-ecbe7ff445eb`
libreoffice_writer	`0810415c-bde4-4443-9047-d5f70165a697`
libreoffice_writer	`0e47de2a-32e0-456c-a366-8c607ef7a9d2`
libreoffice_writer	`72b810ef-4156-4d09-8f08-a0cf57e7cefe`
libreoffice_writer	`adf5e2c3-64c7-4644-b7b6-d2f0167927e7`
libreoffice_writer	`ecc2413d-8a48-416e-a3a2-d30106ca36cb`

Application	Task UUID
multi_apps	42f4d1c7-4521-4161-b646-0a8934e36081
multi_apps	48d05431-6cd5-4e76-82eb-12b60d823f7d
multi_apps	6d72aad6-187a-4392-a4c4-ed87269c51cf
multi_apps	81c425f5-78f3-4771-afd6-3d2973825947
multi_apps	d9b7c649-c975-4f53-88f5-940b29c47247
multi_apps	e135df7c-7687-4ac0-a5f0-76b74438b53e
multi_apps	f5c13cdd-205c-4719-a562-348ae5cd1d91
os	b6781586-6346-41cd-935a-a6b1487918fc
thunderbird	7b1e1ff9-bb85-49be-b01d-d6424be18cd0
thunderbird	9b7bc335-06b5-4cd3-9119-1a649c478509
thunderbird	a1af9f1c-50d5-4bc3-a51e-4d9b425ff638
thunderbird	dfac9ee8-9bc4-4cdc-b465-4a4bfcd2f397
thunderbird	f201fbc3-44e6-46fc-bcaa-432f9815454c
vlc	cb130f0d-d36f-4302-9838-b3baf46139b6
vlc	d06f0d4d-2cd5-4ede-8de9-598629438c6e
vlc	fba2c100-79e8-42df-ae74-b592418d54f4
vs_code	0ed39f63-6049-43d4-ba4d-5fa2fe04a951
vs_code	5e2d93d8-8ad0-4435-b150-1692aacaa994
vs_code	ec71221e-ac43-46f9-89b8-ee7d80f7e1c5

G.6.2 OSWORLD2

The selected task IDs are 001, 002, 010, 012, 013, 015, 018, 020, 021, 022, 028, 029, 030, 033, 038, 040, 042, 043, 044, 046, 047, 049, 051, 053, 054, 057, 058, 059, 061, 063, 065, 066, 068, 070, 071, 072, 076, 080, 085, 086, 088, 091, 094, 096, 100, 101, 102, 103, 104, 106, 107, 108.

G.6.3 MYPCBENCH

Task family	Selected IDs
aggregation	f002, f005, f029, f033
contradiction	f015, f023
counterfactual	f001, f004, f008
cua_only	f004, f006, f018, f021, f023
hard_app	f011, f019, f022
long_horizon	f028, f043, f045, f047, f050, f060, f074
preference_inference	f004, f014, f021, f023, f025
retrieval	f005, f029
situated_action	f001, f003, f006, f017, f026, f035, f039

G.6.4 CUA-WORLD

Table 17: Selected CUA-World task identities.

Environment	Task name
ardour_env	broadcast_podcast_stem_delivery
dhis2_env	rmncah_scorecard_dashboard
docker_desktop_env	diagnose_broken_microservices_stack
gpredict_env	poes_downlink_schedule_setup
gvsig_desktop_env	vulnerability_map_remote_communities
jstock_env	quarterly_portfolio_rebalance
librehealth_ehr_env	implement_lab_workflow_and_process_patient
moodle_env	configure_tiered_assessment_pathway
nextgen_connect_integration_engine_env	adt_census_lab_validation_pipeline
nosh_env	care_quality_remediation
odoo_inventory_env	pharma_lot_recall_quarantine
openclinic_ga_env	insured_consultation_billing
oracle_database_env	claims_pipeline_reconciliation

Environment	Task name
project_libre_env	schedule_recovery_rebaseline
pymol_env	kinase_selectivity_comparison
redmine_env	q1_milestone_reconciliation
rocket_chat_env	compliance_audit_remediation
snap_env	multicriteria_suitability_mapping
splunk_env	threat_intel_enrichment_pipeline
sumo_env	optimize_network_signal_timing
thunderbird_env	litigation_email_triage
wireshark_env	web_app_breach_investigation
wondershare_edrawmax_env	healthcare_it_architecture_review
woo_commerce_env	launch_coffee_product_line
wordpress_env	launch_woocommerce_coffee_roastery
wps_presentation_env	rebrand_restructure_pitch_deck

H Internet-Dependence Audit

Live public services can change their content, become unavailable, or restrict access between evaluations. We therefore audit whether each task requires the agent to use such services. Tasks that use a local application or a benchmark-controlled website remain eligible, as do tasks that only require downloading fixed files during setup.

H.1 Inclusion Rule

We first use a coding agent to assess internet dependence for the 369 OSWorld tasks. Three human reviewers then independently assess whether each task requires interaction with a live public service. We include a task only when all three reviewers agree to include it. This gives 295 tasks.

Reviewer or outcome	Included	Excluded
Reviewer 1	297	72
Reviewer 2	302	67
Reviewer 3	295	74
Unanimous decision	295	67
Disagreement	7	

Table 18: **Independent inclusion decisions for 369 OSWorld tasks.** The seven disagreements are excluded by the unanimous-inclusion rule.

H.2 Reviewer Agreement

Table 18 shows that 362 of 369 tasks receive a unanimous decision. Pairwise agreement is 98.64%, 99.46%, and 98.10% for reviewer pairs 1–2, 1–3, and 2–3, respectively. Their Cohen's κ values are 0.956, 0.983, and 0.939; Fleiss' κ across all three reviewers is 0.959.

Applying the criterion across benchmarks. We apply the same criterion to the other benchmarks. OSWorld2 provides benchmark-controlled web applications, and MyPCBench provides local, pre-authenticated services, so these tasks remain eligible. For CUA-World, we exclude tasks requiring internet access during agent interaction while retaining those that only use the network for setup.

I VALIDATING ACTIONS WITH CUA-AUTODEBUG

A task can fail because the model chooses an incorrect action or because the infrastructure executes a correct action incorrectly. We developed CUA-AutoDebug to distinguish these errors. A controlled desktop application records the keyboard and mouse inputs it receives, including modifier keys and typed text. We also check the resulting application state: a click should reach the intended target, a drag should move the object, and typed text should match the requested string. For shortcuts handled by the operating system, we record input events before they reach the application.

I.1 SEPARATING MODEL, HARNESS, AND RUNTIME ERRORS

We first submit predefined actions directly to the VM runtime. We then test agent harnesses by giving the model a short instruction, such as clicking a colored square, and comparing its response with the executed action and received input. An incorrect model action is a model error; incorrect translation of a correct action is a harness error.

I.2 INPUT FAILURES DETECTED BY THE RUNTIME TESTS

Requested input	Received input	Cause
Type <	>	Incorrect X11 key/modifier mapping
Type accented text	Accented characters omitted	Characters absent from the key map
Type a literal shell variable	Variable expanded into a path	Text interpreted by the command shell
Keypad Enter or Menu	No corresponding key event	Missing key-name translation
Toggle Caps Lock, then type	Lowercase text	Caps Lock press silently omitted

Table 19: **Input errors detected by CUA-AutoDebug when executing actions through SSH and PyAutoGUI.** The tests compare requested input with the keyboard and mouse events received.

Table 19 shows examples of input errors detected by these tests, such as typing > when < was requested or silently dropping accented characters. We test 94 supported cases covering clicks, drags, scrolls, key presses, key combinations, text entry, and sequences of these actions. We repeat each case five times; 83 pass and 11 fail when executed through SSH and PyAutoGUI. The failures include one shell-expansion case, one shifted-symbol case, five Unicode cases, and four named-key cases.

The corrected runtime passes all 470 tests. We correct these errors in CUA-Speedrun's QEMU runtime by preserving literal text when sending commands, explicitly mapping key names and modifiers, and supporting characters absent from the default keyboard mapping. After these corrections, all 94 supported cases pass in all five repetitions.

I.3 FAILURES IN AGENT HARNESSES

Harness	Model response	Executed behavior
Gemini	Scroll down by five wheel clicks	Scroll up by 600 ticks
Gemini	Press F5 or Page Down	Type the key name as text
Qwen3.5	Middle-click the target	No action
Qwen3.5	Ctrl-click the target	Release Ctrl before clicking
Qwen3.5	Move, then scroll or drag	Execute only the first tool call

Table 20: **Harness errors can change a correct model action.** Examples detected by comparing model responses with the actions executed and the input events received.

Table 20 shows that harness errors can prevent a correct model response from reaching the application. For example, the Gemini harness reversed scroll direction and interpreted a fallback scroll distance in pixels as a number of wheel ticks. All six scroll-direction test repetitions failed before correction and passed afterward. The Qwen3.5 harness released the modifier key before a click and silently discarded tool calls after the first.

J EXPERIMENTAL DETAILS

Appendix E.1 lists the model, harness, and reasoning setting for each configuration.

J.1 TASK DEFINITION

We define a computer-use task as $x_i = (E_i, s_i^0, p_i, V_i)$, where E_i is an interactive environment with initial state s_i^0, p_i is a natural-language instruction, and V_i is a verification function. An agent π receives p_i and a sequence of observations and produces computer actions until it terminates or reaches the task limit. This interaction produces a trajectory τ_i and final state s_i^T, with score $r_i = V_i(\tau_i, s_i^T)$. A benchmark $B = \{x_i\}_{i=1}^N$ is a collection of such tasks.

Agent implementations. We connect each model's public reference agent or native computer-use API to the common desktop interface.

J.2 OBSERVATIONS, ACTIONS, AND INTERACTION HISTORY

The environment provides 1920×1080 screenshots of the desktop. Mouse actions use pixel coordinates in these images. Harnesses that resize screenshots or use normalized coordinates convert the model's coordinates to desktop pixels before executing an action.

Action family	Operations
Mouse clicks	Left, right, middle, double, and triple click at a coordinate
Pointer motion	Move, drag along coordinates, hold or release a mouse button
Scrolling	Signed vertical scroll at the current pointer position
Keyboard	Type text, press a key or chord, hold and release modifiers
Waiting	Wait for an agent-specified duration
Observation	Request a screenshot without changing the desktop
Completion	Stop interaction and invoke the benchmark verifier

Table 21: **Desktop actions supported by the infrastructure.** A batch contains one or more actions executed in order. The model-facing tool schema can differ between harnesses.

Table 21 lists the actions available to agents. Direct-API agents translate the model's computer-use responses into these actions and return screenshots in the provider's tool-result format. Codex interacts with the task desktop through the supplied interface from a separate agent sandbox. Each harness determines which screenshots and previous responses remain in the model's context.

Reasoning settings. We vary reasoning effort within each model using the settings exposed by its provider, such as low and high.

J.3 TASK LIMITS AND EXECUTION

Each evaluation fixes the task set, agent implementation, reasoning setting, environment resources, and number of parallel tasks. Every task starts from its benchmark-specified initial state. We measure task time from when the instruction is given to the agent until it terminates or reaches the task limit, excluding setup, initialization, and verification (Sec. 2.1).

Time and action limits. The OSWorld and OSWorld2 configurations in Appendix E.1 have an environment deadline of 39,600 s per task, while the MyPCBench Astra sweep uses 7,200 s. These deadlines apply separately from the harness's action or model-turn limit and the timeout on an individual model request.

We allow 500 action batches per task in the Astra batching comparison, five-seed experiments, and CUA-World sweep, and 100 in the MyPCBench sweep. A batch can contain several actions before the next observation; the single-action Astra ablation requires a screenshot after every action (Appendix K.1). In the Fast I/O comparison, we keep the model and reasoning setting fixed and change the action and screenshot implementation (Appendix B.3).

J.4 Benchmark Verification

We use each benchmark's verification procedure. OSWorld and OSWorld2 score task completion with task-specific programmatic checks. CUA-World uses Gemini 3 Flash Preview to assess the recorded trajectory against its visual checklist. MyPCBench uses its visual trajectory judge and averages the rubric scores with equal weight; a task receives full credit only when every rubric passes. We also retain each benchmark's task instructions and context, including MyPCBench's persona and pre-authenticated local applications, alongside the agent's system prompt.

J.5 Model Cost

We compute model cost by summing the cost of each task's model requests. Each request's cost comes from the provider's recorded charge or its recorded token usage at the applicable API prices. Generated tokens follow the accounting in Appendix D.2.

Model cost excludes environment hosting, model-server rental, setup, and separate verification calls.

K Additional Agent Analyses

We examine how action batching, agent harnesses, and reasoning effort affect task time through comparisons within the same model. Action batches, individual actions, and generated tokens are defined in Appendix D.

K.1 Controlled Batching Comparisons

Model	Effort	Interaction rule	Score	Time	Batches	Actions	Tokens
Astra	xhigh	Batched	91.62	126.82	7.14	19.90	2,060
Astra	xhigh	One action + image	91.62	182.73	16.50	16.50	2,559
Kimi K3	low	Single tool	71.62	176.61	8.12	55.10	3,171
Kimi K3	low	Batched tools	73.62	149.50	5.74	30.00	1,865
Kimi K3	high	Single tool	81.62	305.07	8.92	113.60	7,567
Kimi K3	high	Batched tools	79.62	358.13	8.76	60.20	7,125
Kimi K3	max	Single tool	85.62	606.27	12.18	61.24	14,359
Kimi K3	max	Batched tools	89.62	443.74	11.70	53.58	10,435

Table 22: **Batching comparisons within the same model and reasoning effort on OSWorld.** Score is a percentage; time is in seconds. All counts and times are means over the same 50 tasks. Tokens include reasoning and other output.

Batching makes Astra faster at the same score. Table 22 compares Astra xhigh with its standard batched actions and a restriction to one action between screenshots. Both settings score 91.62%, but the single-action setting takes 182.7 s per task compared with 126.8 s for batching. It executes fewer individual actions (16.50 versus 19.90), yet requires more action batches (16.50 versus 7.14) and generates more tokens (2,559 versus 2,060). We keep the model, reasoning effort, 50-task set, and configured limits fixed. To enforce the single-action restriction, we reject multi-action requests and prevent further actions until the agent retrieves a new screenshot, including actions issued in separate back-to-back requests.

Kimi benefits from batched tools at some reasoning settings. Table 22 also compares Kimi at matched low, high, and maximum effort. Kimi's single-tool setting disables parallel tool calls and allows multiple GUI actions within one code block. At maximum effort, batched tools reduce mean task time from 606.3 to 443.7 s and improve the score from 85.62% to 89.62%. At high effort, batching increases time from 305.1 to 358.1 s while reducing the score from 81.62% to 79.62%.

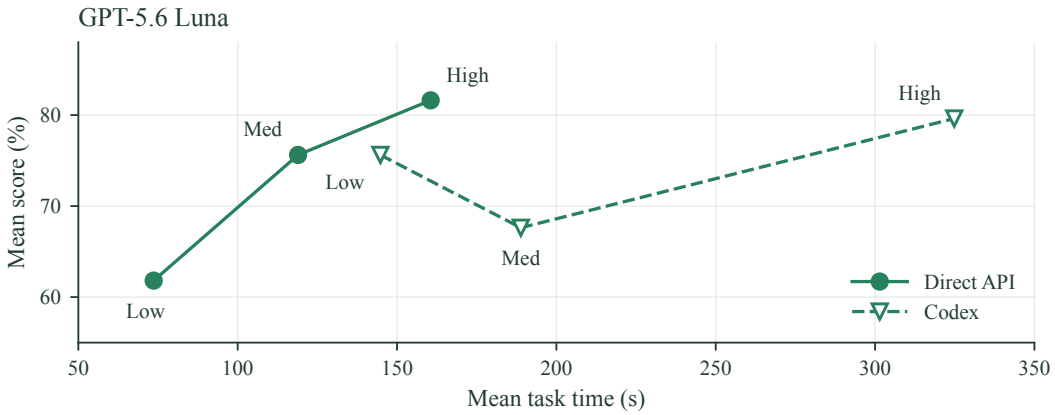

Figure 17: **Harness performance depends on reasoning effort.** GPT-5.6 Luna at low, medium, and high reasoning effort, using the direct-API agent or Codex on OSWorld. Each curve connects reasoning settings within one harness. All six evaluations use the same 50-task set.

K.2 Harness Comparisons Across Reasoning Efforts

Figure 17 compares GPT-5.6 Luna with Codex and the direct-API agent at low, medium, and high effort. At low effort, Codex improves the score from 61.8% to 75.6% but increases mean task time from 74 to 145 s. At medium and high effort, the direct-API agent is both more accurate and faster: it scores 75.6% in 119 s compared with 67.6% in 189 s at medium, and 81.6% in 161 s compared with 79.6% in 325 s at high.

K.3 Comparing Reasoning Efforts on the Same Successful Tasks

Model	Tasks	Effort	Time	Batches	Actions	Tokens	Tok/s
Gemini 3.8 Flash	5	low	1291.0	116.40	171.60	29,596	47.57
		medium	1363.9	112.60	158.40	34,446	43.10
		high	3024.5	190.60	246.60	73,578	42.18
GPT-6 Astra	13	low	791.8	47.08	146.69	7,878	15.18
		medium	752.6	47.15	138.85	8,376	17.07
		high	817.3	53.15	171.23	11,001	21.31
		xhigh	1131.6	55.38	223.46	14,359	18.04
Muse Spark 1.3	2	minimal	833.1	82.00	195.00	7,521	15.04
		low	1183.3	99.50	291.00	12,997	16.79
		medium	1008.8	78.50	235.00	14,621	21.55
		high	907.3	68.50	246.00	14,095	22.87
		xhigh	1546.2	109.50	400.00	23,362	21.77

Table 23: **Reasoning effort on tasks completed fully at every compared effort on OSWorld2.** Every reported score is 100%. Time is in seconds per task; token rate divides total generated tokens by total agent time. Each model uses its own common task set.

Higher effort can generate longer outputs and more actions on the same successful tasks. Table 23 restricts each OSWorld2 comparison to tasks that receive full credit at every evaluated effort for that model. On the five tasks that Gemini 3.8 Flash completes at all three efforts, high effort takes 3,024 s per task compared with 1,291 s at low effort. It generates more than twice as many tokens (73,578 versus 29,596) and executes more actions and batches. In comparison, its effective token rate falls from 47.57 to 42.18 tokens/s.

We observe a similar increase at Astra's highest effort. On the 13 tasks it completes at all four settings, xhigh takes 1,132 s, compared with 792 s at low and 753 s at medium. It generates 14,359 tokens and executes 223 actions, compared with 7,878 tokens and 147 actions at low effort. Muse Spark 1.3 completes two tasks at all five efforts, while Kimi K3 has no task completed at all three efforts.

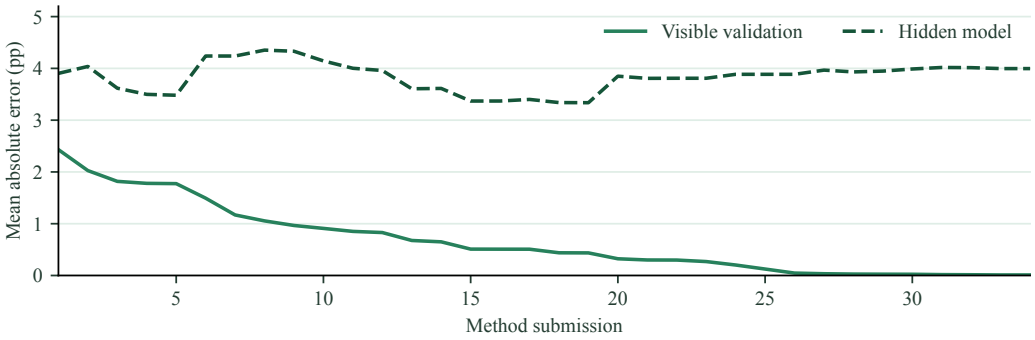

Figure 18: **Validation gains do not generalize to the held-out model.** Across 34 candidate methods, the coding agent receives only validation errors. We measure test error separately on the held-out model. Each method selects 32 tasks. Error averages the mean absolute errors in partial score and exact completion, in percentage points.

L Automated Search for Task-Selection Methods

We investigate whether a coding agent can find task-selection methods that better estimate a new model's score on the full task set. The agent develops these methods using results from six models; we evaluate them on a seventh model whose results are withheld throughout the search.

L.1 Search and Evaluation

We provide the coding agent with results from six models on 107 OSWorld2 tasks. This task set includes all 63 tasks used for representative-set selection (Appendix G). The agent can modify both how the tasks are selected and how scores on the selected tasks are used to predict the full-set score. Each candidate method selects exactly 32 tasks. We evaluate it using six-fold leave-one-agent-out validation: five models are used to fit the method, and the sixth to measure prediction error. The agent receives only this validation error and can revise its method. We separately test each candidate on the seventh model after fitting to all six available models.

L.2 Validation Gains Do Not Generalize to the Held-Out Model

Figure 18 compares validation and test error across 34 candidate methods. Validation error falls from 2.43 to 0.01 percentage points, while error on the held-out model changes from 3.90 to 4.00 percentage points. The search nearly eliminates validation error without improving the estimate for the new model. Validation errors guide repeated method selection, allowing the search to overfit these six models.