COBRA: CO-TRAINING BI-LEVEL OBSERVATION ROUTING AND ACTION POLICIES FOR GUI AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

The information a GUI agent needs changes from one action to the next, but existing agents typically construct the actor's context with a fixed rule. This can omit useful interface structure or introduce irrelevant information. We model observation selection and action generation as a *bi-level partially observable Markov decision process* and introduce COBRA (Co-trained Bi-level Routing and Action) to train both policies jointly online. At each step, a lightweight router, *GUIRouter*, chooses between the screenshot and accessibility (a11y) information and selects the relevant nodes. To learn routing from sparse terminal rewards, we introduce *CMA* (Counterfactual Modality Advantage), which compares terminal outcomes across routing choices at comparable information states without a learned value network. We analyze when a shared adapter update fails to reduce both modality losses and use two modality-specific LoRA adapters on a frozen shared 8B backbone. With the same initialization, training tasks, and rollout budget, COBRA outperforms the strongest of three fixed-observation online-RL controls by 5.5 percentage points on OSWorld-Verified. It also reduces average per-step a11y tokens by 86% compared with fusing visual input and the a11y tree. Compared with its Qwen3-VL-8B-Thinking base actor, COBRA improves OSWorld-Verified success from 33.9% to 46.3%, with consistent gains on AndroidWorld and Windows Agent Arena.

1 INTRODUCTION

Autonomous GUI agents can observe desktop, mobile, or web interfaces through screenshots and accessibility (a11y) trees at each decision step. Screenshots depict the current state at the pixel level but provide limited semantic structure, whereas a11y trees provide precise node labels, roles, and bounding boxes. The useful modality depends on the interface state. A screenshot may suffice for a visually distinctive icon, whereas a deeply nested control among similar-looking rows may require a small subset of the a11y tree. Raw a11y trees from production applications, however, can contain tens of thousands of tokens. Such trees exceed typical context budgets and bury the task-relevant elements. Agents whose pipelines are publicly documented instead use these modalities statically, committing to a single modality for the entire deployment (Qin et al., 2025; Wang et al., 2026c) or concatenating both unconditionally (Agashe et al., 2025a;b). The latter exposes the actor to the full a11y tree at every step, regardless of its relevance. Neither approach determines how much structured evidence to include in the actor's context or which evidence is relevant at a given step.

The cost of a static policy extends beyond context usage. When the two channels conflict, agents that receive both systematically favor structured text over pixels: they follow outdated structure on up to 88% of unedited stale snapshots from live pages and recover from an incorrect initial belief in at most 3% of cases (Zhang et al., 2026a). Even without an explicit conflict, irrelevant or stale nodes compete with task-relevant elements for the actor's limited attention. Thus, neither a fixed preference nor unconditional fusion reliably selects the evidence needed at each step. Since both channels are available before each action, the problem is to select and compress the evidence passed to the actor based on the current observation. We therefore learn a lightweight compression policy from interaction rather than use a hand-written fusion rule.

The router determines which evidence reaches the actor, yet GUI benchmarks supervise this decision only through sparse terminal rewards, and treating routing as an actor hyperparameter provides no

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.

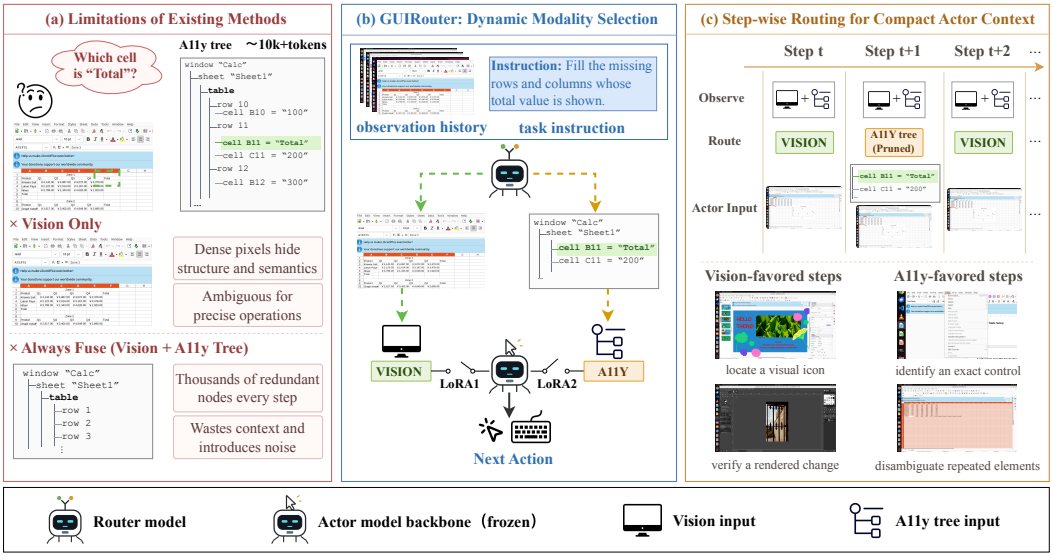


Figure 1: Motivation and architecture of COBRA. **(a)** Vision alone hides actionable structure, whereas always fusing the a11y tree adds thousands of redundant nodes (∼10k+ tokens) per step. **(b)** Given the observation history and the instruction, *GUIRouter* scores the two candidate observations and selects one, which activates the matching LoRA adapter on the frozen shared actor backbone. **(c)** The selected modality changes across steps: the screenshot alone suffices to locate a visual icon or verify a rendered change, while a pruned a11y tree is added to identify an exact control or disambiguate repeated elements.

objective for learning it directly. We therefore model the router–actor interaction as a *bi-level partially observable Markov decision process* (Section 2), with observation compression as the upper-level policy and action generation as the lower-level one. Two difficulties follow. First, a trajectory outcome assigns the same reward to every routing decision and does not reveal which steps benefited from a11y evidence. To recover local credit, we group rollout steps whose router information states are similar in representation space. We then contrast the terminal outcomes of the screenshot-only and a11y-augmented branches within each group. Second, the branch choice changes the actor's input format: one branch contains only the screenshot, whereas the other adds selected a11y nodes. A single adapter updated on both formats need not reduce both branch losses, especially when their gradients differ in direction or magnitude. We therefore keep the actor backbone shared and frozen, and train a separate adapter per branch.

We instantiate this design as COBRA (**Co**-trained **B**i-level **R**outing and **A**ction), which trains router and actor together online. On OSWorld-Verified, COBRA outperforms the strongest of three fixed-observation online-RL controls by 5.5 points under the same initialization, the same tasks, and the same rollout budget, and improves its Qwen3-VL-8B-Thinking base actor by 12.4 points. It improves over the same base actor on AndroidWorld and Windows Agent Arena as well, while the average number of a11y tokens per step drops by 86% relative to always-fused input.

Our main contributions are as follows: **(1)** We formalize step-wise observation routing as the upper level of a bi-level POMDP and realize it as *GUIRouter*, a lightweight 2B router that admits structured context only when needed and prunes it to a task-relevant subset. **(2)** We propose COBRA, which co-trains the two levels in one online loop. Its routing signal is *CMA*, a critic-free advantage estimator that is asymptotically unbiased under exact state matching and standard unconfoundedness. Its actor is trained with one LoRA adapter per observation branch, which provably removes the direct cross-branch term incurred by a shared update. **(3)** Under matched training conditions, learned routing improves task success over three fixed-observation online-RL controls on OSWorld-Verified and over the base actor across mobile and desktop GUI environments, while substantially shortening the actor's context. This shows that what an agent observes is itself worth learning.

2 PROBLEM FORMULATION

Both views are available at every step, so our agent makes two distinct decisions: one selects a subset of the current observation before acting, while the other selects a GUI operation that advances the environment. They are coupled: the admitted evidence shapes the current action, which determines future states and available evidence. We therefore use two levels of one process, sharing a latent state and task reward, rather than a flat action space with one undifferentiated signal.

We model the GUI interaction task as a *bi-level partially observable Markov decision process* (Kaelbling et al., 1998; Sutton et al., 1999) $\mathcal{M} = (\mathcal{S}, \mathcal{A}_{\text{rt}}, \mathcal{A}_{\text{gui}}, P, \mathcal{R})$, where $\mathcal{S}$ is the set of latent environment states, $\mathcal{A}_{\text{rt}}$ is the upper-level router action space (detailed below), $\mathcal{A}_{\text{gui}}$ is the lower-level space of GUI actions (click, type, scroll, etc.), P is the transition function, and $\mathcal{R}$ is the undiscounted binary terminal reward specified below. We write `vis` for a screenshot-only actor input and `a11y` for the screenshot augmented with a pruned a11y tree; choosing between them is *modality selection*.

At step t, the environment exposes two partial observations of the true state $s_t \in \mathcal{S}$: a screenshot o_t^{vis} and a raw a11y tree $o_t^{\text{a11y,raw}}$. To make the a11y branch tractable for a lightweight router, we apply a fixed rule-based pre-processing step that removes duplicated and information-less nodes, producing a rule-compressed tree $\bar{o}_t^{\text{a11y}}$; this step is shared by all controls across all three environments. Denote by $\bar{h}_t = \left(o_{t-n:t-1}^{\text{vis}},\, a_{t-n:t-1}^{\text{gui}}\right)$ the compact interaction history composed of the last n screenshots and the last n GUI actions (n=3 in our experiments), and let g denote the task instruction. The router state is then $s_t^{\text{rt}} = \left(g,\, o_t^{\text{vis}},\, \bar{o}_t^{\text{a11y}},\, \bar{h}_t\right)$. The system then operates as follows:

- **Upper-level (routing) policy** $\pi_{\theta_{\text{rt}}}^{\text{rt}}$: given s_t^{rt}, outputs a composite router action $a_t^{\text{rt}} = (m_t, \mathbf{w}_t) \in \mathcal{A}_{\text{rt}} = \mathcal{M} \times \mathcal{W}$ with $\mathcal{M} = \{\texttt{vis}, \texttt{a11y}\}$, where m_t is the selected modality and $\mathbf{w}_t \in \mathcal{W}$ is a keyword-weight dictionary. The dictionary is empty when $m_t = \texttt{vis}$, and when $m_t = \texttt{a11y}$ it further compresses $\bar{o}_t^{\text{a11y}}$ into a task-conditioned pruned tree $\tilde{o}_t^{\text{a11y}}(\mathbf{w}_t)$. For brevity we write $a_t^{\text{rt}} = \texttt{a11y}$ (resp. `vis`) whenever only the modality component matters.
- **Lower-level (actor) policy** $\pi_{\theta_{\text{gui}}}^{\text{gui}}$: the actor always receives the current screenshot; in the `a11y` branch, the pruned a11y tree is additionally concatenated:

$$\hat{o}_t = o_t^{\text{vis}} \;\text{ if } a_t^{\text{rt}} = \texttt{vis}, \qquad \hat{o}_t = \left(o_t^{\text{vis}},\, \tilde{o}_t^{\text{a11y}}(\mathbf{w}_t)\right) \;\text{ if } a_t^{\text{rt}} = \texttt{a11y}. \tag{1}$$

  Together with $\bar{h}_t$, the actor produces the GUI action $a_t^{\text{gui}} \in \mathcal{A}_{\text{gui}}$.

The task terminates at step T with binary return $G \in \{0, 1\}$, and the joint objective is

$$\max_{\theta_{\text{rt}}, \theta_{\text{gui}}} \mathbb{E}_{\pi_{\theta_{\text{rt}}}^{\text{rt}}, \pi_{\theta_{\text{gui}}}^{\text{gui}}}\left[G\right].$$

3 METHOD

This section instantiates the bi-level POMDP of Section 2 as COBRA, a single online loop in which the router and the actor are trained together from task-outcome reward alone: each iteration rolls out the routed policy, then updates both levels from their corresponding trajectories. The three subsections follow that loop, covering the routing interface (Section 3.1), the per-step routing credit recovered from a single binary return (Section 3.2), and the actor update under two structurally different observation formats (Section 3.3), which Figure 2 summarizes as rollout, router-update, and actor-update stages.

3.1 GUIROUTER: ROUTER OBSERVATION PARADIGM

The upper-level policy is a lightweight router model $\pi_{\theta_{\text{rt}}}^{\text{rt}}$ (Qwen3-VL-2B-Instruct (Bai et al., 2025)). At each decision step t it reads the router state s_t^{rt} of Section 2 and emits a JSON object with two types of fields: `modality`, which takes the value `"screenshot"` or `"a11y_tree"`, and `keyword_weights`, which is empty in the *visual-only* branch and in the *a11y-augmented* branch holds a dictionary $\mathbf{w}_t = \{(k_{t,j}, w_{t,j})\}_{j=1}^{J_t}$ whose weight $w_{t,j}$ ranks the task relevance of keyword $k_{t,j}$. The admissible value ranges, a concrete schema, and examples are given in Appendix A.

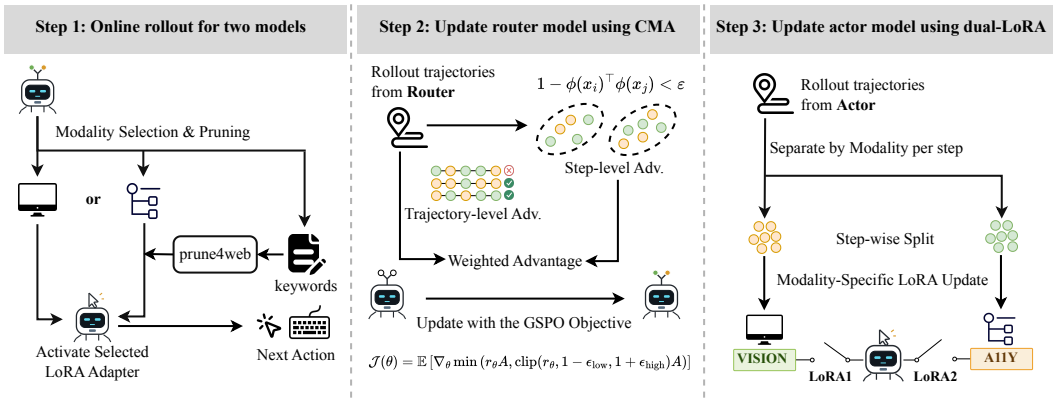


Figure 2: Three-stage online training pipeline of COBRA. **(1) Rollout:** the router selects the modality and prunes the a11y tree when needed, after which the actor activates the corresponding LoRA adapter. **(2) Router update:** router trajectories provide trajectory-level credit, while representation-matched neighborhoods provide *CMA* step-level credit; the two signals are combined for the router policy update. **(3) Actor update:** actor records are partitioned by their per-step routed modality and used to update the `A11Y` and `VISION` LoRA adapters separately.

Given the router output, the actor input $\hat{o}_t$ of Eq. 1 is constructed deterministically. In the a11y-augmented branch, we reuse the keyword-based filtering mechanism of Prune4Web (Zhang et al., 2026b) as a fixed post-processing step over $\bar{o}_t^{\text{a11y}}$: each remaining tree element e receives score $\text{score}_t(e) = \sum_{j=1}^{J_t} w_{t,j} f(k_{t,j}, e)$, and the top-$K_{\text{node}}$ elements form the task-conditioned pruned tree $\tilde{o}_t^{\text{a11y}}$. The two types of router fields therefore act as a two-stage compressor of the raw dual-channel observation: `modality` gates whether any structured evidence is admitted, and `keyword_weights` sets the compression level within the admitted tree by determining which nodes survive pruning.

3.2 COUNTERFACTUAL MODALITY ADVANTAGE (CMA)

Training the router under sparse terminal reward is challenging because one outcome is shared by every routing decision in a trajectory. *CMA* assigns per-step credit by casting a11y augmentation as a binary treatment on the actor's information channel within the potential-outcome framework (Rubin, 1974). The treatment is the *modality* component $m \in \mathcal{M} = \{\texttt{vis}, \texttt{a11y}\}$ of the composite router action $a^{\text{rt}} = (m, \mathbf{w})$; the keyword weights $\mathbf{w}$ are marginalized out under the current router policy rather than intervened on. The router observes both candidate channels before acting, so s_t^{rt} is a shared pre-treatment state. We index routing-step records by i, where record i from task τ_i, rollout ℓ_i and step t_i carries $(s_i^{\text{rt}}, m_i, \mathbf{w}_i, G_i)$. The estimand is the modality-level policy value and its policy-centered advantage,

$$Q_M(s,m) = \mathop{\mathbb{E}}_{\substack{\mathbf{w}\sim\pi^{\text{rt}}_{\theta_{\text{rt}}}(\mathbf{w}|s,m)\\ \tau\sim\pi^{\text{rt}}_{\theta_{\text{rt}}},\,\pi^{\text{gui}}_{\theta_{\text{gui}}}}} \left[Y(\tau) \mid \text{do}(M{=}m),\, s^{\text{rt}}{=}s\right], \qquad A_M(s,m) = Q_M(s,m) - V_M(s), \tag{2}$$

where $Y(\tau) \in \{0,1\}$ is the terminal return of trajectory τ and $V_M(s) = \sum_{m'\in\mathcal{M}} \pi^{\text{rt}}_{\theta_{\text{rt}}}(m' \mid s) Q_M(s,m')$. So the estimand is a modality-level policy effect: it is defined after marginalizing over the keyword-weight policy, which is instead learned from the episode return and the format reward (Eq. 4). This is a policy-dependent conditional treatment effect of the modality gate, not a learned Q/V critic. Because only one potential outcome is observed per record, *CMA* estimates A_M by contrasting rollouts that reach near-identical router states but select different modalities.

Anchor neighborhoods and CMA estimator. An anchor neighborhood $\mathcal{N}$ clusters routing records by cosine distance between mean-centered, ℓ_2-normalized last-layer features $\phi_{\theta_{\text{rt}}}(s_i^{\text{rt}})$. Records i and j from the same task are grouped when $1 - \phi_{\theta_{\text{rt}}}(s_i^{\text{rt}})^\top \phi_{\theta_{\text{rt}}}(s_j^{\text{rt}}) < \varepsilon$. The features are

recomputed each iteration, making the grouping track the current policy. Within each neighborhood, *CMA* contrasts modality-conditional and pooled terminal returns:

$$\hat{A}_M(\mathcal{N}, m) = \underbrace{\frac{1}{|\mathcal{N}_m|} \sum_{i \in \mathcal{N}_m} G_i}_{\hat{Q}_M(\mathcal{N}, m)} - \underbrace{\frac{1}{|\mathcal{N}|} \sum_{i \in \mathcal{N}} G_i}_{\hat{V}_M(\mathcal{N})}, \tag{3}$$

where $\mathcal{N}_m = \{i \in \mathcal{N} : m_i = m\}$ and $\mathcal{N}_i$ denotes the neighborhood containing record i. Since records in $\mathcal{N}_m$ draw $\mathbf{w}$ from the router policy, the group mean targets the marginalized value of Eq. 2. Under conditional modality unconfoundedness and exact state matching, this estimator is asymptotically unbiased (Proposition B.1); the proof and approximation details are deferred to Appendix B.

Anchor construction in practice. For each record, we run the *current* router on s_i^{rt} and take its last-layer hidden state at the final prompt token; features are mean-centered and ℓ_2-normalized before grouping, and recomputed after every policy update. At $t_i{=}1$, all rollouts of the same task share the same router state and are grouped exactly by task ID. For $t_i{>}1$, we perform single-pass greedy cosine clustering (Wang et al., 2026b; Feng et al., 2026) with $\varepsilon = 0.1$. If $\mathcal{N}$ contains only one modality, its unsupported counterfactual is not extrapolated and we set $\hat{A}_M(\mathcal{N}, \cdot) = 0$. The matching assumptions and this variance–bias trade-off are analyzed in Appendix B.

Router objective and optimization. We combine episode-level, step-level, and format credit:

$$\hat{A}_i = w_{\mathrm{ep}} A_i^{\mathrm{ep}} + w_{\mathrm{cma}} \hat{A}_i^M + w_{\mathrm{fmt}} r_i^{\mathrm{fmt}}, \tag{4}$$

where $A_i^{\mathrm{ep}} = G_i - \mathrm{LOO}_{\tau_i}(i)$ (Ahmadian et al., 2024), with $\mathrm{LOO}_{\tau_i}(i)$ denoting the mean return of the other rollouts for task τ_i, and $\hat{A}_i^M = \hat{A}_M(\mathcal{N}_i, m_i)$ is given by Eq. 3, and $r_i^{\mathrm{fmt}} \in \{0, 1\}$ indicates whether the router output parses under the router JSON schema. The first two terms apply to the full router action, so the keyword weights are learned from them, while $\hat{A}_i^M$ sharpens only the modality gate. Following Liu et al. (2025), we subtract the batch mean without standard-deviation scaling. The router is updated with the clipped GSPO objective (Zheng et al., 2025)

$$\mathcal{J}(\theta_{\mathrm{rt}}) = \mathbb{E}_{i \sim \mathcal{B}^{\mathrm{rt}}} \Big[\min\Big(\rho_i \hat{A}_i,\ \mathrm{clip}(\rho_i,\ 1 - \epsilon_{\mathrm{low}},\ 1 + \epsilon_{\mathrm{high}})\, \hat{A}_i \Big) \Big], \qquad \rho_i = \exp\left(\frac{1}{L_i} \sum_{u=1}^{L_i} \Delta_{i,u} \right), \tag{5}$$

where $\hat{A}_i$ is given by Eq. 4, $\Delta_{i,u} = \log \pi^{\mathrm{rt}}_{\theta_{\mathrm{rt}}}(a^{\mathrm{rt}}_{i,u}) - \log \pi^{\mathrm{rt}}_{\theta^{\mathrm{old}}_{\mathrm{rt}}}(a^{\mathrm{rt}}_{i,u})$ is the log-probability difference of the u-th router output token between the current and the sampling policy, and L_i is the response length. Taking the ratio at the sequence level prevents the longer `a11y` responses, whose keyword dictionary is non-empty, from receiving systematically larger updates than `vis` responses. Algorithm 1 in Appendix B states the complete router-update procedure.

3.3 Dual-LoRA Actor Training

The routing policy induces two branch-conditioned actor objectives: `vis` records contain screenshot-only observations, whereas `a11y` records additionally contain selected structural tokens. We share the pretrained actor across these observation formats but decouple their trainable updates. Specifically, the Qwen3-VL-8B-Thinking backbone (Bai et al., 2025), denoted by θ_0, is frozen and equipped with two LoRA adapters (Hu et al., 2021), ψ_{vis} and ψ_{a11y}. The routed actor is

$$\pi^{\mathrm{gui}}(a_t^{\mathrm{gui}} \mid \hat{o}_t, \bar{h}_t, a_t^{\mathrm{rt}}) = \pi^{\mathrm{gui}}_{\theta_0, \psi_{a_t^{\mathrm{rt}}}}(a_t^{\mathrm{gui}} \mid \hat{o}_t, \bar{h}_t), \qquad \psi_{a_t^{\mathrm{rt}}} = \begin{cases} \psi_{\mathrm{vis}}, & a_t^{\mathrm{rt}} = \texttt{vis}, \\ \psi_{\mathrm{a11y}}, & a_t^{\mathrm{rt}} = \texttt{a11y}. \end{cases} \tag{6}$$

Only the routed adapter participates in the forward and backward passes at step t. The frozen backbone therefore supplies a common visual and action prior, while the low-rank parameters specialize to the corresponding input format.

Because the base actor has not been trained on our screenshot–a11y serialization, we cold-start the routed actor with 500 demonstrations collected from GPT-5.5 on the training-pool tasks[1] before

[1] Accessed through the OpenAI API: `https://platform.openai.com/docs/models`

online RL. Each online rollout is then serialized into step-aligned router records for Algorithm 1 and actor records $(\tau_i, \ell_i, t_i, \hat{o}_i, \bar{h}_i, a_i^{\text{gui}}, a_i^{\text{rt}}, G_i)$. For the actor, we compute $A_i^{\text{gui}} = G_i - \text{LOO}_{\tau_i}(i)$ on the full task group before partitioning the batch as $\mathcal{B}_b^{\text{gui}} = \{i : a_i^{\text{rt}} = b\}$ for $b \in \{\texttt{vis}, \texttt{a11y}\}$. We then apply the objective of Eq. 5 separately to each partition, with $\hat{A}_i$ replaced by A_i^{gui} and ρ_i computed from the actor response under the routed adapter, updating ψ_b only with $\mathcal{B}_b^{\text{gui}}$ while keeping θ_0 fixed. Advantages are not re-normalized within either partition, so both updates retain the same task-level baseline.

To characterize the optimization difference from a shared adapter, consider a fixed rollout batch and let $\mathcal{L}_{\text{vis}}(\theta)$ and $\mathcal{L}_{\text{a11y}}(\theta)$ be its mixture-weighted branch losses under a single adapter θ. Denote their gradients by $\mathbf{g}_{\text{vis}}$ and $\mathbf{g}_{\text{a11y}}$, respectively. The mixed-batch update follows $-\mathbf{g} = -(\mathbf{g}_{\text{vis}} + \mathbf{g}_{\text{a11y}})$.

Proposition 3.1 (Shared-update compatibility and adapter decoupling)**.** *For nonzero branch gradients, $-\mathbf{g}$ is a strict first-order descent direction for both branch losses if and only if*

$$\langle \mathbf{g}_{\text{vis}}, \mathbf{g}_{\text{a11y}} \rangle > -\min\{\|\mathbf{g}_{\text{vis}}\|^2, \|\mathbf{g}_{\text{a11y}}\|^2\}. \tag{7}$$

For $\kappa = \|\mathbf{g}_{\text{vis}}\|/\|\mathbf{g}_{\text{a11y}}\|$, this is equivalent to $\cos(\mathbf{g}_{\text{vis}}, \mathbf{g}_{\text{a11y}}) > -\min\{\kappa, \kappa^{-1}\}$. With a frozen backbone and disjoint adapters, the lifted branch gradients instead occupy separate parameter blocks and are orthogonal. If each branch loss is β_b-smooth, its independent update is a descent step for $0 < \eta_b < 2/\beta_b$. The proof and finite-step statements are given in Appendix C.

Equation 7 shows that shared-update compatibility depends on both gradient alignment and relative magnitude; under norm imbalance, even modest anti-alignment can prevent descent on the smaller-gradient branch. This is the same conflicting-gradient phenomenon studied in multi-task optimization (Yu et al., 2020). The proposition does not assume that the two formats necessarily conflict. Rather, it shows that disjoint adapters remove the direct cross-branch term when such conflict occurs. Section 4.3 evaluates this design against a rank- and compute-matched shared-adapter control.

4 Experiments

4.1 Setup

We apply our method to three GUI benchmarks: OSWorld-Verified (Xie et al., 2024) (Linux desktop), AndroidWorld (Rawles et al., 2025) (mobile), and Windows Agent Arena (Bonatti et al., 2024) (Windows desktop). The controlled comparisons against fixed observation strategies and the component-level ablations, which both require repeated retraining, are conducted on OSWorld-Verified, which offers the most mature tooling and the largest set of prior open-source results for comparison. Following the standard evaluation protocol, we use the task-specific rule-based evaluation functions defined by each benchmark to compute rewards and report task success rate (%) as the primary metric. The base actor is Qwen3-VL-8B-Thinking; the router is Qwen3-VL-2B-Instruct (Bai et al., 2025). To keep the comparison focused on the training method, online updates use a fixed subset of each benchmark's released task pool rather than an additional task-generation pipeline. For OSWorld-Verified, following the task selection principle of ARPO (Lu et al., 2025), we identify 184 valuable tasks from the 361 released tasks for online training, while the remaining tasks never receive updates and are re-evaluated separately in Appendix E. All online-RL variants use the same task instances, initialization, rollout budget, and interaction limit.[2]

We train three online RL controls from the same cold-start SFT actor, one for each way of fixing the observation strategy: *visual-only* receives the screenshot alone at every step, *a11y-only* receives the rule-compressed a11y tree alone, and *always-fused* concatenates the screenshot with the rule-compressed a11y tree at every step. None of the three uses learned routing or modality-specific adapter switching. All three are optimized through full-parameter fine-tuning, with all other experimental settings identical to those of COBRA, and appear in Table 2 as *Visual-only*, *A11y-only*, and *Always-fused Online RL*. We refer to them collectively as the three fixed-observation controls.

[2]Our experiments compare observation strategies, rather than establish a benchmark ranking. Online-RL variants use the same training subset of the evaluation pool under matched budgets; Appendix E separately evaluates transfer to tasks excluded from updates.

Table 1: Task success rate (%) across three GUI benchmarks. The Cost column reports the deployment cost of one trajectory in US dollars ($/traj.) for closed-source systems accessed through a paid API, and the total parameter count for open-source models. † marks the base model.

Model	Cost	OS	Office	Daily	Prof.	Workflow	OSWorld	AndroidWorld	WAA
Closed-Source Models									
OpenAI CUA o3 (2025a)	$1.1	62.5	14.6	22.6	38.7	16.5	23.0	52.5	–
OpenAI CUA (2025b)	$1.3	70.8	24.6	36.8	45.5	15.8	31.3	57.6	–
Seed1.5-VL (2025)	$0.2	39.1	46.3	54.7	52.0	15.4	40.2	62.1	39.6
UI-TARS-2 (2025)	$0.2	41.7	60.1	61.9	57.5	34.1	53.1	73.3	50.6
Claude-3-7-Sonnet (2025a)	$1.0	50.0	36.4	45.1	44.1	17.7	35.8	60.8	38.9
Claude-4-Sonnet (2025b)	$0.9	45.8	45.2	55.0	53.4	28.5	43.9	–	–
Claude-4.5-Sonnet (2025c)	$0.9	70.8	72.6	61.4	63.3	49.5	62.8	69.0	–
Open-Source Models									
Holo2-8B (2025)	8B	–	–	–	–	–	39.9	60.4	–
RLAnything-OS-8B (2026e)	8B	71.4	33.4	47.3	**77.1**	15.4	40.2	–	–
Qwen3.5-9B (2026)	9B	–	–	–	–	–	41.8	57.8	–
DART-GUI-7B (2025)	7B	62.5	**42.6**	50.5	74.6	16.7	42.1	–	–
ToolCUA-8B (2026)	8B	–	–	–	–	–	–	–	33.8
UI-TARS-1.5-7B (2025)	7B	25.5	30.7	33.6	53.3	8.6	27.3	–	18.1
Qwen3-VL-8B† (base model) (2025)	8B	47.8	27.9	50.0	48.9	16.3	33.9	47.4	24.1
Cobra (ours)	8B + 2B	**75.0**	40.3	**53.1**	75.5	**25.5**	**46.3** (+12.4)	**61.6** (+14.2)	**36.6** (+12.5)

4.2 Main Results

Table 1 situates Cobra among reported systems, while the controlled comparison against fixed observation strategies is carried out on OSWorld-Verified (Table 2). Since Cobra is a post-training method rather than a foundation-model effort, we compare against open-source models whose training pipelines are publicly documented, and list closed-source systems as reference points. Against the three fixed-observation controls trained from the same cold-start actor on the same tasks under an identical budget, Cobra improves by 5.5 points over the strongest of them, visual-only input, and by 6.7 and 15.9 points over always-fused and a11y-only input. Relative to its own backbone, it raises OSWorld-Verified success from 33.9% to 46.3%. Appending the tree unconditionally is itself 1.2 points below visual-only input, and substituting it for the screenshot costs a further 9.2 points, so neither fixed use of structured evidence pays for what it displaces. The corresponding AndroidWorld and Windows Agent Arena results in Table 1 extend the evaluation to mobile and Windows environments. The online training curves behind these three final numbers are reported in Appendix D.

4.3 Observation Paradigms and Component Analysis

Table 2 reports per-domain and aggregate OSWorld-Verified success rates for all variants, with each variant specified by its component columns: a ✓ marks an enabled observation channel or training component, and the lower block removes one component of Cobra at a time. Every online-RL variant starts from the same cold-start SFT actor and uses the same training budget, so all differences below are attributable to the compared design choice alone. Section 4.2 already established that no *fixed* observation strategy suffices on its own, so the comparisons here instead dissect Cobra itself, asking three questions in turn: whether adapter specialization is necessary, what the router's training objective contributes, and whether the learned routing behavior generalizes to unseen tasks while paying off in context cost.

Router training and CMA credit. Using the same dual-LoRA actor configuration and training budget, an untrained router reaches 42.4% success. Training the router with episode-level RL raises this to 44.2%; adding *CMA* raises it to 46.3% (+2.1 further, +3.9 over the untrained router). Episode-level RL gives every decision in a rollout the same terminal-return advantage, whereas *CMA* adds a modality-specific contrast when similar states are observed under both modalities. Figure 4 (right) shows that the share of dual-modality anchor groups gradually decreases from 53.6% in the first 12 iterations to 43.5% in the last 12, which indicates more consistent routing at similar states. Such groups still account for 50.1% on average, and the mean absolute modality advantage stays near 0.05 throughout training, so this additional step-level signal remains available.

Table 2: Observation paradigms and COBRA components under a matched training protocol (OSWorld-Verified, success rate %, 30 steps, all 361 released tasks, identically for every variant in this table). **Vision/Tree**: the actor input contains the screenshot / the rule-compressed a11y tree; **Route**: a per-step router decides tree admission and pruning instead of a fixed pipeline; **Dual-LoRA**: modality-specialized adapters instead of one shared adapter; **CMA**: step-level counterfactual modality credit on top of episode-level credit, which is only defined on top of router RL.

	Observation			Training				OSWorld-Verified success rate (%)					
Method	**Vision**	**Tree**	**Route**	**Actor RL**	**Dual LoRA**	**Router RL**	**CMA**	**OS**	**Office**	**Daily**	**Prof.**	**Workflow**	**Avg**
Fixed-observation controls													
Qwen3-VL-8B (base model)	✓	×	×	×	×	×	×	47.8	27.9	50.0	48.9	16.3	33.9
Visual-only Online RL	✓	×	×	✓	×	×	×	69.6	36.3	48.7	65.1	19.6	40.8
A11y-only Online RL	×	✓	×	✓	×	×	×	54.8	25.7	42.1	45.7	12.3	30.4
Always-fused Online RL	✓	✓	×	✓	×	×	×	66.9	33.2	50.1	58.6	21.8	39.6
Cobra component ablation													
w/o online RL *(cold-start SFT)*	✓	✓	✓	×	×	×	×	60.4	29.8	46.6	60.2	18.2	36.6
w/o dual-LoRA *(shared adapter)*	✓	✓	✓	✓	×	✓	✓	73.9	32.6	52.9	69.3	21.7	41.9
w/o router RL *(untrained)*	✓	✓	✓	✓	✓	×	×	**77.1**	33.1	53.4	71.5	20.7	42.4
w/o CMA *(episode credit)*	✓	✓	✓	✓	✓	✓	×	76.4	35.1	**56.3**	73.1	22.1	44.2
Cobra (full)	✓	✓	✓	✓	✓	✓	✓	75.0	**40.3**	53.1	**75.5**	**25.5**	**46.3**

Dual-LoRA vs. single-LoRA. To evaluate the benefit of modality-specialized adapters, we train a shared-adapter control on the same mixed batches and GSPO objective with rank 128, matching the active per-step adapter rank and compute of the dual-LoRA actor. Under the same training budget, the single-LoRA multimodal control reaches 41.9% OSWorld-Verified success, whereas the full COBRA reaches 46.3%, an improvement of 4.4 points. Figure 3 provides a complementary view of the optimization dynamics. We log the mean pairwise cosine similarity between the micro-batch gradients accumulated within each actor update, separately for the shared adapter and each modality-specific adapter. Averaged over training, this statistic is 0.143 for the shared adapter, compared with 0.179 and 0.215 for the `vis` and `a11y` adapters, respectively; the `a11y` branch is above the shared adapter in most iterations. These results align with the decoupling rationale in Proposition 3.1: separate adapter parameters remove direct cross-branch coupling, and this design exhibits higher within-adapter gradient agreement and task success in our experiments.

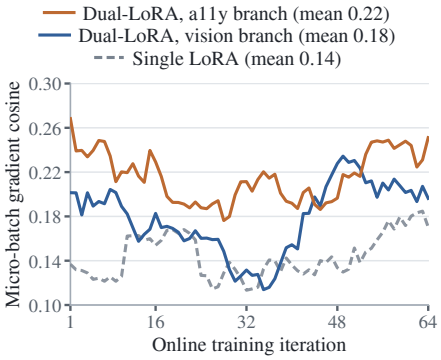


Figure 3: Mean pairwise cosine similarity between micro-batch gradients within each actor update, measured separately for the shared adapter and each modality-specific adapter.

Tasks outside the online-training pool. Online updates touch 184 of the 361 OSWorld-Verified tasks, so the rows above are in-domain online-RL results and we re-evaluate the same variants on the tasks that are never trained on (Table 3, Appendix E). This is not a random hold-out: by construction it retains the tasks that the base actor either always or never solved in the 32 filtering rollouts, so the tasks on which outcome-based RL has the most headroom are precisely the ones removed from it. Visual-only online RL ends up below its own untrained backbone ($25.6 \rightarrow 23.9$), whereas admitting structured observations transfers positively, $+1.6$ points for always-on fusion and $+2.7$ for COBRA.

Token efficiency of adaptive observation routing. Figure 4 (left) compares the per-step context budgets of COBRA and the fixed-observation controls across the three benchmarks. By admitting the a11y tree selectively and pruning it to task-relevant nodes, COBRA reduces a11y-tree tokens by 86% and total actor input by approximately 37% per step relative to the always-fused control. Its context budget also remains close to that of visual-only input: averaged across the three benchmarks, total actor-input tokens per step increase by only about 7% over the visual-only control. Alongside this modest input overhead, COBRA improves OSWorld-Verified success from 40.8% to 46.3%, a gain of 5.5 points (Table 2).

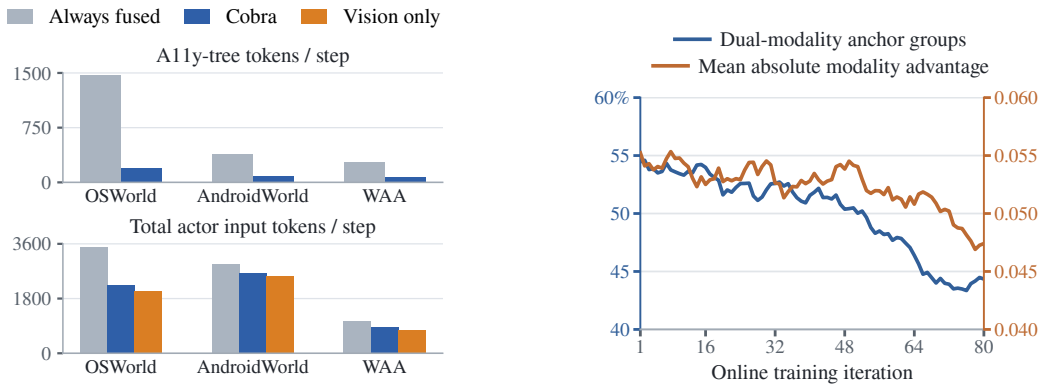


Figure 4: **Left:** per-step a11y-tree and total actor-input tokens on the three benchmarks; unlike always-on fusion, which appends the rule-compressed tree at every step, COBRA admits it selectively and prunes it to task-relevant nodes, cutting a11y-tree tokens by 86% and total actor input by about 37% per step. **Right:** *CMA* routing credit over the RL stage: the dual-modality anchor-group fraction falls from 53.6% to 43.5% as routing grows more consistent, still averaging 50.1%, and the mean absolute modality advantage stays near 0.05, keeping the step-level contrast available.

5 RELATED WORK

GUI Agents and Multi-Step RL. Executable benchmarks (Xie et al., 2024; Rawles et al., 2025; Bonatti et al., 2024) support open GUI agents (Qin et al., 2025; Wang et al., 2026c; Yang et al., 2025; Xu et al., 2026) and online training at scale (Wang et al., 2026e;d;a; Li et al., 2025). Their optimizers are predominantly critic-free, replacing the value network of PPO (Schulman et al., 2017) with group-relative baselines (Shao et al., 2024; Zheng et al., 2025; Liu et al., 2025). For credit within an episode, GiGPO and HiMAC use hierarchical step-level advantages (Feng et al., 2026; Jin et al., 2026) and BiPACE clusters bisimilar states for counterfactual action estimation (Wang et al., 2026b; Buesing et al., 2018). These estimators score *environment* actions; *CMA* scores the observation channel that precedes them.

Context Compression. Language-model context compression uses token-level prompt pruning (Jiang et al., 2023) or learned soft summaries (Mu et al., 2023) to preserve task-relevant content within a length budget, largely independently of the downstream policy. COBRA instead selects the actor's evidence at every environment step and optimizes compression using only task outcomes, rather than a reconstruction or perplexity proxy.

Multimodal Observations for GUI Agents. GUI observation pipelines are fixed by design. Visual agents act from screenshots alone (Cheng et al., 2024; Qin et al., 2025), Agent S and Agent S2 pair every screenshot with an image-augmented a11y tree (Agashe et al., 2025a;b), and UFO drives Windows applications from UIA metadata (Zhang et al., 2025). Others fix a compression level or a channel priority (Ma et al., 2026; Takeshita et al., 2026; Yang et al., 2026), including Prune4Web, whose keyword-scoring programs filter DOM elements (Zhang et al., 2026b). COBRA learns from task outcomes whether to admit structured context and how aggressively to prune it at each step.

6 CONCLUSION

We introduced COBRA, which treats multimodal observation use in GUI agents as learned, stepwise compression. A lightweight router decides when to admit a11y context and which nodes to retain; *CMA* provides critic-free, per-step routing credit from sparse terminal rewards; and modality-specialized LoRA adapters support the two heterogeneous observation formats. On OSWorld-Verified, COBRA outperforms the strongest fixed-observation online-RL control by 5.5 points under matched training conditions and improves its base actor by 12.4 points, while the average number of a11y tokens per step drops by 86% relative to always-fused input. Gains on AndroidWorld and Windows Agent Arena further support co-training observation selection and action generation.

AI USE STATEMENT

We used GPT-5.5 through the OpenAI API to obtain 500 demonstrations for the cold-start supervised fine-tuning stage described in Section 3.3. We also used generative AI assistants to improve the clarity of the manuscript, suggest and revise research code, and identify and summarize relevant literature. We reviewed AI-assisted text and code before incorporating them and checked cited works against the original sources. We take responsibility for the final content of this paper, including its code, data, citations, results, and conclusions.

REPRODUCIBILITY STATEMENT

Experimental results are reported on the public OSWorld-Verified, AndroidWorld, and Windows Agent Arena benchmarks. Model configurations and training details are described in Section 3 and Appendix F; complete system-prompt templates are provided in Appendix I. The released artifacts will include the exact task lists and protocol, the cold-start SFT checkpoint, complete training code, and final router and actor model checkpoints.

REFERENCES

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

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

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting reinforce-style optimization for learning from human feedback in llms. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12248–12267, 2024.

Anthropic. Claude 3.7 Sonnet and Claude Code, February 2025a. URL https://www.anthropic.com/news/claude-3-7-sonnet.

Anthropic. Introducing Claude 4, May 2025b. URL https://www.anthropic.com/news/claude-4.

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

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

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, et al. Windows agent arena: Evaluating multi-modal os agents at scale. *arXiv preprint arXiv:2409.08264*, 2024.

Lars Buesing, Theophane Weber, Yori Zwols, Sebastien Racaniere, Arthur Guez, Jean-Baptiste Lespiau, and Nicolas Heess. Woulda, coulda, shoulda: Counterfactually-guided policy search. *arXiv preprint arXiv:1811.06272*, 2018.

ByteDance Seed. UI-TARS-1.5, 2025. URL https://seed.bytedance.com/en/blog/bytedance-seed-agent-model-ui-tars-1-5-open-source-achieving-sota-performance-in-various-benchmarks.

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

H Company. Holo2 - open foundation models for navigation and computer use agents, 2025. URL https://huggingface.co/collections/Hcompany/holo2.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for llm agent training. *Advances in Neural Information Processing Systems*, 38:46375–46408, 2026.

Dong Guo, Faming Wu, Feida Zhu, Fuxing Leng, Guang Shi, Haobin Chen, Haoqi Fan, Jian Wang, Jianyu Jiang, Jiawei Wang, et al. Seed1. 5-vl technical report. *arXiv preprint arXiv:2505.07062*, 2025.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Xuhao Hu, Xi Zhang, Haiyang Xu, Kyle Qiao, Jingyi Yang, Xuanjing Huang, Jing Shao, Ming Yan, and Jieping Ye. Toolcua: Towards optimal gui-tool path orchestration for computer use agents. *arXiv preprint arXiv:2605.12481*, 2026.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 13358–13376, 2023.

Hongbo Jin, Rongpeng Zhu, Jiayu Ding, Guibo Luo, and Ge Li. Himac: Hierarchical macro-micro learning for long-horizon llm agents. *arXiv preprint arXiv:2603.00977*, 2026.

Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

Pengxiang Li, Zechen Hu, Zirui Shang, Jingrong Wu, Yang Liu, Hui Liu, Zhi Gao, Chenrui Shi, Bofei Zhang, Zihao Zhang, et al. Efficient multi-turn rl for gui agents via decoupled training and adaptive data curation. *arXiv preprint arXiv:2509.23866*, 2025.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

Fanbin Lu, Zhisheng Zhong, Shu Liu, Chi-Wing Fu, and Jiaya Jia. Arpo: End-to-end policy optimization for gui agents with experience replay. *arXiv preprint arXiv:2505.16282*, 2025.

Longhui Ma, Di Zhao, Siwei Wang, Zhao Lv, and Miao Wang. Trifuse: Enhancing attention-based gui grounding via multimodal fusion. *arXiv preprint arXiv:2602.06351*, 2026.

Jesse Mu, Xiang Li, and Noah Goodman. Learning to compress prompts with gist tokens. *Advances in Neural Information Processing Systems*, 36:19327–19352, 2023.

OpenAI. Introducing OpenAI o3 and o4-mini, April 2025a. URL https://openai.com/index/introducing-o3-and-o4-mini/.

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

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

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

Donald B Rubin. Estimating causal effects of treatments in randomized and nonrandomized studies. *Journal of educational Psychology*, 66(5):688, 1974.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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

Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

Michito Takeshita, Takuro Kawada, Takumi Ohashi, Shunsuke Kitada, and Hitoshi Iyatomi. A11y-compressor: A framework for enhancing the efficiency of gui agent observations through visual context reconstruction and redundancy reduction. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (ACL 2026)*, pp. 563–580, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

Bowen Wang, Dunjie Lu, Junli Wang, Tianyi Bai, Shixuan Liu, Zhipeng Zhang, Haiquan Wang, Hao Hu, Tianbao Xie, Shuai Bai, et al. Cua-gym: Scaling verifiable training environments and tasks for computer-use agents. *arXiv preprint arXiv:2605.25624*, 2026a.

Hanyang Wang, Weijieying Ren, Yuxiang Zhang, Ding Cao, Zhizhao Zeng, Ke Zeng, and Tianxiang Zhao. Bipace: Bisimulation-guided policy optimization with action counterfactual estimation for llm agents. *arXiv preprint arXiv:2606.25556*, 2026b.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, et al. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*, 2025.

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

Yinjie Wang, Xuyang Chen, Xiaolong Jin, Mengdi Wang, and Ling Yang. Openclaw-rl: Train any agent simply by talking. *arXiv preprint arXiv:2603.10165*, 2026d.

Yinjie Wang, Tianbao Xie, Ke Shen, Mengdi Wang, and Ling Yang. Rlanything: Forge environment, policy, and reward model in completely dynamic rl system. *arXiv preprint arXiv:2602.02488*, 2026e.

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

Haiyang Xu, Xi Zhang, Haowei Liu, Junyang Wang, Zhaozai Zhu, Shengjie Zhou, Xuhao Hu, Feiyu Gao, Junjie Cao, Zihua Wang, et al. Mobile-agent-v3. 5: Multi-platform fundamental gui agents. *arXiv preprint arXiv:2602.16855*, 2026.

Chenyu Yang, Shiqian Su, Shi Liu, Xuan Dong, Yue Yu, Weijie Su, Xuehui Wang, Zhaoyang Liu, Jinguo Zhu, Hao Li, et al. Zerogui: Automating online gui learning at zero human cost. *arXiv preprint arXiv:2505.23762*, 2025.

Yan Yang, Xiangru Jian, Ziyang Luo, Zirui Zhao, Yutong Dai, Ziji Shi, Hanshu Yan, Jun Hao Liew, Silvio Savarese, and Junnan Li. Stateact: Program state, before pixels, for long-horizon computer-use agents. *arXiv preprint arXiv:2607.22798*, 2026.

Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. Gradient surgery for multi-task learning. *Advances in neural information processing systems*, 33: 5824–5836, 2020.

Chaoyun Zhang, Liqun Li, Shilin He, Xu Zhang, Bo Qiao, Si Qin, Minghua Ma, Yu Kang, Qingwei Lin, Saravan Rajmohan, et al. Ufo: A ui-focused agent for windows os interaction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 597–622, 2025.

Guijia Zhang, Yuxun Chen, Yuheng Qi, and Harry Yang. Do gui agents believe their eyes? diagnosing state-belief reliance on pixels versus structure. *arXiv preprint arXiv:2607.04334*, 2026a.

Jiayuan Zhang, Kaiquan Chen, Zhihao Lu, Enshen Zhou, Qian Yu, and Jing Zhang. Prune4web: Dom tree pruning programming for web agent. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 34710–34718, 2026b.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization, 2025. URL https://arxiv.org/abs/2507.18071.

Appendix

A Router Output Schema and Example

The router emits a single JSON object whose two top-level keys are `modality`$\in$ {`"screenshot"`,`"a11y_tree"`} and `keyword_weights`, a dictionary $\mathbf{w}_t = \{(k_{t,j}, w_{t,j})\}_{j=1}^{J_t}$ that is empty in the `"screenshot"` branch and holds at least three entries ($J_t \geq 3$) in the `"a11y_tree"` branch, with integer weights $w_{t,j} \in \{1, \ldots, 50\}$. The router does not emit any chain-of-thought tokens; the sequence-level GSPO ratio ρ_i in Eq. 5 is computed over exactly the tokens of this JSON object.

Visual-only decision. A minimal example of a visual-only routing decision:

```
{"modality": "screenshot", "keyword_weights": {}}
```

The actor then receives o_t^{vis} alone.

A11y-augmented decision. A minimal example of an a11y-augmented decision on an email-composition task:

```
{"modality": "a11y_tree",
 "keyword_weights": {"compose": 50, "send": 40,
                     "to": 30, "subject": 20}}
```

The rule-compressed tree $\bar{o}_t^{\text{a11y}}$ is filtered by Prune4Web (Zhang et al., 2026b) using $\text{score}_t(e) = \sum_{j=1}^{J_t} w_{t,j} f(k_{t,j}, e)$; the top-$K_{\text{node}}$ elements form $\tilde{o}_t^{\text{a11y}}$, which is concatenated with o_t^{vis} as the actor input.

Format reward. The reward $r_i^{\text{fmt}} \in \{0, 1\}$ used in Eq. 4 is 1 iff the raw router response parses as a JSON object with exactly these two keys, `modality` taking one of the two admissible values, and `keyword_weights` a mapping to integers in $\{1, \ldots, 50\}$ that is empty when `modality` is `"screenshot"` and holds at least three entries when it is `"a11y_tree"`; it is 0 otherwise.

B CMA Analysis and Implementation Details

Identification setting. At every routing step, the router observes both candidate information channels before emitting $a_i^{\text{rt}} = (m_i, \mathbf{w}_i) \in \mathcal{A}_{\text{rt}}$, so s_i^{rt} is a pre-treatment state. The treated variable is the modality gate M alone, and $Y_i(m)$ is the terminal return under $\text{do}(M{=}m)$ followed by the unmodified policies, so that $\mathbf{w}$, the actor, and the environment are all integrated out inside the potential outcome. We assume consistency and conditional modality unconfoundedness, $Y_i(m) \perp m_i \mid s_i^{\text{rt}}$, which identify $Q_M(s, m)$ of Eq. 2 from modality-conditional outcomes without a learned critic. Identifying the effect of a particular $\mathbf{w}$ would further require overlap in the continuous weight space and is outside the scope of *CMA*.

Proposition B.1 (Asymptotic unbiasedness of CMA)**.** *Let all members of anchor neighborhood $\mathcal{N}$ share router state s^* and satisfy conditional modality unconfoundedness, with keyword weights drawn from the current router policy $\pi_{\theta_{\text{rt}}}^{\text{rt}}(\mathbf{w} \mid s^*, m)$. If both $|\mathcal{N}_m|$ and $|\mathcal{N}|$ grow, then the estimator in Eq. 3 converges in probability to $A_M(s^*, m) = Q_M(s^*, m) - V_M(s^*)$, where*

$$V_M(s^*) = \sum_{m' \in \mathcal{M}} \pi_{\theta_{\text{rt}}}^{\text{rt}}(m' \mid s^*) Q_M(s^*, m'). \tag{8}$$

Proof. By consistency and conditional modality unconfoundedness, and because records in $\mathcal{N}_m$ draw $\mathbf{w}$ from $\pi_{\theta_{\text{rt}}}^{\text{rt}}(\mathbf{w} \mid s^*, m)$,

$$\mathbb{E}[G_i \mid i \in \mathcal{N}_m, s_i^{\text{rt}} = s^*] = \mathbb{E}[Y_i(m) \mid s_i^{\text{rt}} = s^*] = Q_M(s^*, m). \tag{9}$$

Algorithm 1 *CMA* and router update for one training iteration.

Require: $\pi^{\mathrm{rt}}_{\theta_{\mathrm{rt}}}$; current batch $\mathcal{B}^{\mathrm{rt}} = \{(\tau_i, \ell_i, t_i, s^{\mathrm{rt}}_i, a^{\mathrm{rt}}_i{=}(m_i, \mathbf{w}_i), r^{\mathrm{fmt}}_i, G_i)\}$; ε; $w_{\mathrm{ep}}, w_{\mathrm{cma}}, w_{\mathrm{fmt}}$

$\phi_j \leftarrow \phi_{\theta_{\mathrm{rt}}}(s^{\mathrm{rt}}_j), \forall j \in \mathcal{B}^{\mathrm{rt}}$ {centered, normalized current-router features}
for each task τ **do**
  group all j with $t_j{=}1$ into one $\mathcal{N}$ {identical router state}
  for each j with $t_j > 1$ **do**
    join first $\mathcal{N}$ with $1 - \phi_j^\top \bar{\phi}_{\mathcal{N}} < \varepsilon$, else open $\mathcal{N} \leftarrow \{j\}$ {$\bar{\phi}_{\mathcal{N}}$ is the centroid}
  end for
end for
for each $\mathcal{N}$ **do**
  $\mathcal{N}_m \leftarrow \{j \in \mathcal{N} : m_j{=}m\}, \forall m \in \mathcal{M}$
  $\hat{A}_M(\mathcal{N}, m) \leftarrow \mathrm{mean}_{\mathcal{N}_m} G - \mathrm{mean}_{\mathcal{N}} G$ if $\mathcal{N}_{\mathtt{vis}}, \mathcal{N}_{\mathtt{a11y}} \neq \emptyset$, else $0, \forall m \in \mathcal{M}$ {Eq. 3; single-modality $\mathcal{N}$ is not extrapolated}
end for
$\hat{A}_i \leftarrow w_{\mathrm{ep}}\big(G_i - \mathrm{LOO}_{\tau_i}(i)\big) + w_{\mathrm{cma}} \hat{A}_M(\mathcal{N}_i, m_i) + w_{\mathrm{fmt}} r^{\mathrm{fmt}}_i, \forall i \in \mathcal{B}^{\mathrm{rt}}$ {Eq. 4}
$\hat{A}_i \leftarrow \hat{A}_i - \mathrm{mean}_{j \in \mathcal{B}^{\mathrm{rt}}} \hat{A}_j, \forall i \in \mathcal{B}^{\mathrm{rt}}$ {no std. scaling}
$\theta_{\mathrm{rt}} \leftarrow \textsc{GspoStep}(\theta_{\mathrm{rt}}, \{\hat{A}_i\})$ {Eq. 5}
return θ_{rt}

The law of large numbers therefore gives $\hat{Q}_M(\mathcal{N}, m) \xrightarrow{p} Q_M(s^*, m)$. The pooled neighborhood follows the router's modality mixture, so

$$\hat{V}_M(\mathcal{N}) \xrightarrow{p} \sum_{m' \in \mathcal{M}} \pi^{\mathrm{rt}}_{\theta_{\mathrm{rt}}}(m' \mid s^*) Q_M(s^*, m') = V_M(s^*). \tag{10}$$

Subtracting the two limits proves the result. The limit is policy-dependent, since Q_M and V_M are defined under the current keyword-weight and actor policies. □

Approximate matching and graceful degradation. Later-step neighborhoods match nearby rather than identical states, introducing the usual representation-matching bias. The threshold ε controls the bias–coverage trade-off: smaller values improve state similarity but reduce the chance that both modalities occur in one neighborhood. If a neighborhood contains only one modality, we set $\hat{A}_M(\mathcal{N}, \cdot) = 0$, reducing its update to the episode-level baseline rather than extrapolating an unsupported counterfactual.

Optimization details. Step-1 states are grouped exactly by task ID. Later steps use single-pass greedy cosine clustering of current-router features at $\varepsilon = 0.1$, calibrated to balance within-neighborhood similarity and modality coverage. We subtract the batch mean from the combined advantage but do not divide by its standard deviation (Liu et al., 2025). The router uses the sequence-level ratio ρ_i in Eq. 5, which length-normalizes the full JSON response.

Full router-update procedure. Algorithm 1 collects the steps described in Section 3.2 and above into the routine executed once per training iteration: feature extraction with the current router, anchor construction, the *CMA* contrast of Eq. 3, the combined advantage of Eq. 4, and the GSPO step of Eq. 5.

C Shared-Update Compatibility and Adapter Decoupling

We prove Proposition 3.1 for one actor update with the rollout batch held fixed, so the result concerns the empirical GSPO surrogate rather than changes in the on-policy state distribution. Let $\mathcal{L}_{\mathrm{vis}}$ and $\mathcal{L}_{\mathrm{a11y}}$ be the differentiable, mixture-weighted branch losses evaluated at a shared adapter θ, and abbreviate $\mathbf{g}_{\mathrm{vis}} = \nabla\mathcal{L}_{\mathrm{vis}}(\theta)$, $\mathbf{g}_{\mathrm{a11y}} = \nabla\mathcal{L}_{\mathrm{a11y}}(\theta)$, and $\mathbf{g} = \mathbf{g}_{\mathrm{vis}} + \mathbf{g}_{\mathrm{a11y}}$. We call a direction $\mathbf{d}$ simultaneously compatible when both branch directional derivatives along $\mathbf{d}$ are negative.

Proposition C.1 (Shared-update compatibility; restatement of Proposition 3.1 with finite-step form)**.** *Suppose* $\mathbf{g}_{\mathrm{vis}}$ *and* $\mathbf{g}_{\mathrm{a11y}}$ *are nonzero. The shared mixed-batch direction* $-\mathbf{g}$ *is simultaneously com-*

patible if and only if

$$\langle \mathbf{g}_{\text{vis}}, \mathbf{g}_{\text{a11y}} \rangle > -\min\{\|\mathbf{g}_{\text{vis}}\|^2, \|\mathbf{g}_{\text{a11y}}\|^2\}. \tag{11}$$

Equivalently, for $\kappa = \|\mathbf{g}_{\text{vis}}\|/\|\mathbf{g}_{\text{a11y}}\|$,

$$\cos(\mathbf{g}_{\text{vis}}, \mathbf{g}_{\text{a11y}}) > -\min\{\kappa, \kappa^{-1}\}. \tag{12}$$

Moreover, suppose $\mathcal{L}_b$ *is* β_b*-smooth for branch* $b \in \{\text{vis}, \text{a11y}\}$*, and let* $\bar{b}$ *denote the other branch. If*

$$\delta_b = -\langle \mathbf{g}_b, \mathbf{g}_{\bar{b}} \rangle - \|\mathbf{g}_b\|^2 > 0, \tag{13}$$

then the shared update $\theta^+ = \theta - \eta\mathbf{g}$ *strictly increases* $\mathcal{L}_b$ *for every* $0 < \eta < 2\delta_b/(\beta_b\|\mathbf{g}\|^2)$.

Proof. Along the shared update path $\theta(\eta) = \theta - \eta\mathbf{g}$, the two directional derivatives at $\eta = 0$ are

$$\frac{d}{d\eta}\mathcal{L}_{\text{vis}}(\theta(\eta))\Big|_{\eta=0} = -\|\mathbf{g}_{\text{vis}}\|^2 - \langle \mathbf{g}_{\text{vis}}, \mathbf{g}_{\text{a11y}} \rangle, \tag{14}$$

$$\frac{d}{d\eta}\mathcal{L}_{\text{a11y}}(\theta(\eta))\Big|_{\eta=0} = -\|\mathbf{g}_{\text{a11y}}\|^2 - \langle \mathbf{g}_{\text{vis}}, \mathbf{g}_{\text{a11y}} \rangle. \tag{15}$$

Both are negative exactly when the inner product exceeds $-\|\mathbf{g}_{\text{vis}}\|^2$ and $-\|\mathbf{g}_{\text{a11y}}\|^2$, which is equivalent to Eq. 11. Dividing by $\|\mathbf{g}_{\text{vis}}\|\|\mathbf{g}_{\text{a11y}}\|$ yields Eq. 12.

For the finite-step claim, β_b-smoothness gives the two-sided quadratic remainder bound

$$\mathcal{L}_b(\theta - \eta\mathbf{g}) \geq \mathcal{L}_b(\theta) - \eta\langle \mathbf{g}_b, \mathbf{g} \rangle - \frac{\beta_b\eta^2}{2}\|\mathbf{g}\|^2. \tag{16}$$

By Eq. 13, $-\langle \mathbf{g}_b, \mathbf{g} \rangle = \delta_b$. Hence

$$\mathcal{L}_b(\theta - \eta\mathbf{g}) - \mathcal{L}_b(\theta) \geq \eta\delta_b - \frac{\beta_b\eta^2}{2}\|\mathbf{g}\|^2 > 0 \tag{17}$$

for the stated range of η. □

Proposition C.2 (Decoupled-adapter descent)**.** *Freeze the shared backbone and parameterize the two routed adapters by the product variable* $\psi = (\psi_{\text{vis}}, \psi_{\text{a11y}})$*, with* $\widetilde{\mathcal{L}}_{\text{vis}}(\psi) = \mathcal{L}_{\text{vis}}(\psi_{\text{vis}})$ *and* $\widetilde{\mathcal{L}}_{\text{a11y}}(\psi) = \mathcal{L}_{\text{a11y}}(\psi_{\text{a11y}})$*. Their lifted gradients are orthogonal:*

$$\left\langle (\nabla_{\psi_{\text{vis}}}\mathcal{L}_{\text{vis}}, 0), (0, \nabla_{\psi_{\text{a11y}}}\mathcal{L}_{\text{a11y}}) \right\rangle = 0. \tag{18}$$

If each branch loss is β_b*-smooth, its independent update* $\psi_b^+ = \psi_b - \eta_b\nabla_{\psi_b}\mathcal{L}_b$ *satisfies*

$$\mathcal{L}_b(\psi_b^+) \leq \mathcal{L}_b(\psi_b) - \eta_b\left(1 - \frac{\beta_b\eta_b}{2}\right)\|\nabla_{\psi_b}\mathcal{L}_b\|^2. \tag{19}$$

Thus both branch losses strictly decrease whenever their gradients are nonzero and $0 < \eta_b < 2/\beta_b$.

Proof. Because each routed loss depends on only its matching adapter, the two lifted gradients have disjoint support, proving Eq. 18. Applying the standard smoothness descent lemma separately to each adapter block gives Eq. 19. □

Remark C.1. *Equations 11–19 separate two claims that should not be conflated. Observation-format heterogeneity does not by itself prove gradient conflict; conflict is measured by the branch-gradient inner product and relative magnitudes. Once conflict occurs, disjoint routed adapters eliminate the direct cross-branch term at the trainable-adapter level. Consistent with this motivation, the full* COBRA *reaches 46.3% OSWorld-Verified success, compared with 41.9% for the single-LoRA multimodal control in Section 4.3.*

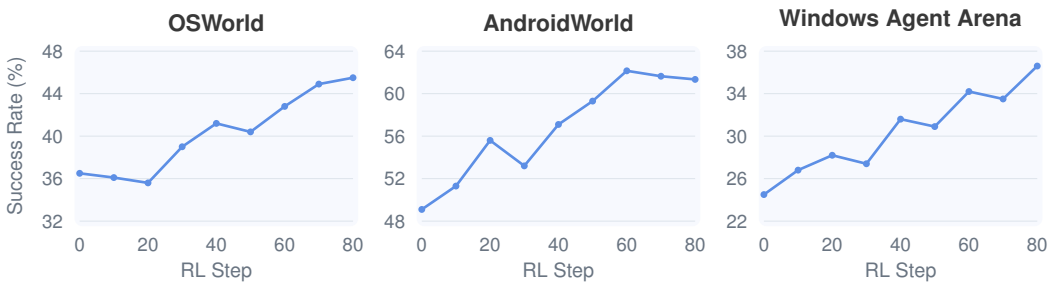


Figure 5: Online RL training curves of COBRA on the three benchmarks. The router and the actor are updated jointly. At every 10-step checkpoint, the saved policy is evaluated three times on the benchmark's full task set under the 30-step interaction budget; each point reports the mean success rate across the three runs.

D ONLINE TRAINING CURVES

Figure 5 shows how task success rate evolves over online training on OSWorld-Verified, AndroidWorld, and Windows Agent Arena. The router and the actor are updated jointly in the single loop of Section 3, so each curve tracks the routed system as a whole rather than either level in isolation. Every 10 training steps we take the saved policy and evaluate it three times on that benchmark's full task set under the same 30-step interaction budget used at test time; each plotted point is the mean over the three evaluation runs, which averages out part of the run-to-run variance caused by environment stochasticity.

All three benchmarks improve over the cold-start initialization and flatten in the later iterations, and the end points of the curves correspond to the COBRA rows of Table 1. The curves are thus a training-time view of results already reported there; we include them only to show that the gains accumulate steadily under joint router–actor updates rather than coming from a single favorable checkpoint.

E EVALUATION OUTSIDE THE ONLINE-TRAINING POOL

The main experiments compare observation strategies within a common task pool under matched training conditions; here we assess transfer to tasks excluded from updates. Online updates on OSWorld-Verified use 184 of the 361 released tasks. Table 3 re-evaluates the base actor, the visual-only and always-fused controls of Section 4.3, and COBRA on the released tasks that never enter any update, under the same 30-step interaction budget. Because the training pool follows the task-selection principle of ARPO (Lu et al., 2025), this set retains the tasks that the base actor either always or never solved in the 32 filtering rollouts, so absolute movement on it is compressed by construction: an always-solved task admits no gain, and a never-solved task supplies no positive terminal reward. COBRA remains the strongest variant on this subset. Visual-only online RL falls below its own untrained backbone ($25.6 \rightarrow 23.9$), so training a screenshot-only actor costs success on tasks the backbone had previously solved, whereas admitting structured evidence transfers positively: $+1.6$ points for always-on fusion and $+2.7$ for COBRA, which matches or exceeds every other variant in every domain. COBRA's gains therefore extend beyond its online-training tasks, complementing the controlled comparison of observation strategies in the main text.

F HYPERPARAMETERS

Table 4 reports the main online-training hyperparameters for COBRA. Each iteration uses 12 task groups with 12 rollouts per task. The two actor adapters share optimizer settings, while the router is optimized separately.

Table 3: OSWorld-Verified success rate (%, 30 steps) on the released tasks that are excluded from the online-training pool and therefore never receive gradient updates. All online-RL variants use the same cold-start initialization and training budget as in Table 2; Δ is taken against the base actor evaluated on this same task set.

Method	OS	Office	Daily	Prof.	Workflow	Avg
Qwen3-VL-8B-Thinking (base model)	70.0	12.9	31.4	83.3	8.5	25.6
Visual-only Online RL	70.0	12.9	28.6	66.7	10.2	23.9 (-1.7)
Always-fused Online RL	80.0	12.9	33.3	83.3	10.2	27.2 $(+1.6)$
Cobra (dual-LoRA + CMA router)	**80.0**	**13.7**	**37.1**	**83.3**	**10.2**	**28.3** $(+2.7)$

Table 4: Hyperparameters for COBRA training.

Symbol / setting	Value
Router history window n (screenshots and actions)	3
CMA cosine-distance threshold ε	0.1
Episode / CMA advantage weights $w_{\text{ep}}/w_{\text{cma}}$	1.0 / 2.0
Min keywords in the `ally` branch	3
Keyword weight range $w_{t,j}$	$\{1, \ldots, 50\}$
GSPO clip range $\epsilon_{\text{low}}/\epsilon_{\text{high}}$	5e-4
Actor LoRA rank / scaling α (each adapter)	64 / 128
Actor learning rate (`vision` / `ally`)	1e-5
Router learning rate	3e-6
Weight decay (actor / router)	0.01 / 0.01
GSPO update epochs (actor / router)	1 / 1
Actor mini-batch size (per adapter)	14
Router mini-batch size per GPU	28
Task groups per iteration	12
Rollouts per task per iteration N_{roll}	12
Max steps per episode (train / eval)	30 / 30
Max response tokens (actor / router)	1000 / 100
Rollout temperature / top-p	1.0 / 1.0

G HARDWARE AND RUNTIME

We implemented asynchronous online reinforcement learning on top of the DART-GUI framework (Li et al., 2025), with rollout generation and policy training running concurrently. Our experiments were run on six compute nodes, each equipped with eight NVIDIA H20 graphics processing units (GPUs), for a total of 48 GPUs. Two nodes (16 GPUs) were allocated to rollout generation, and the other four nodes (32 GPUs) were allocated to policy training. The rollout workers interacted with 300 parallel Tencent Cloud Agent Sandbox environment instances.[3]

Under this configuration, one complete online training iteration took approximately 1,100 seconds (18.3 minutes). Evaluating the actor's old-policy action log probabilities took approximately 180 seconds; training the router policy took 350 seconds; updating the actor policy parameters took 510 seconds; and saving checkpoints took 60 seconds. These are wall-clock times per iteration and sum to the reported iteration time. Because rollout generation overlapped with policy training, its runtime is not added as a separate sequential stage.

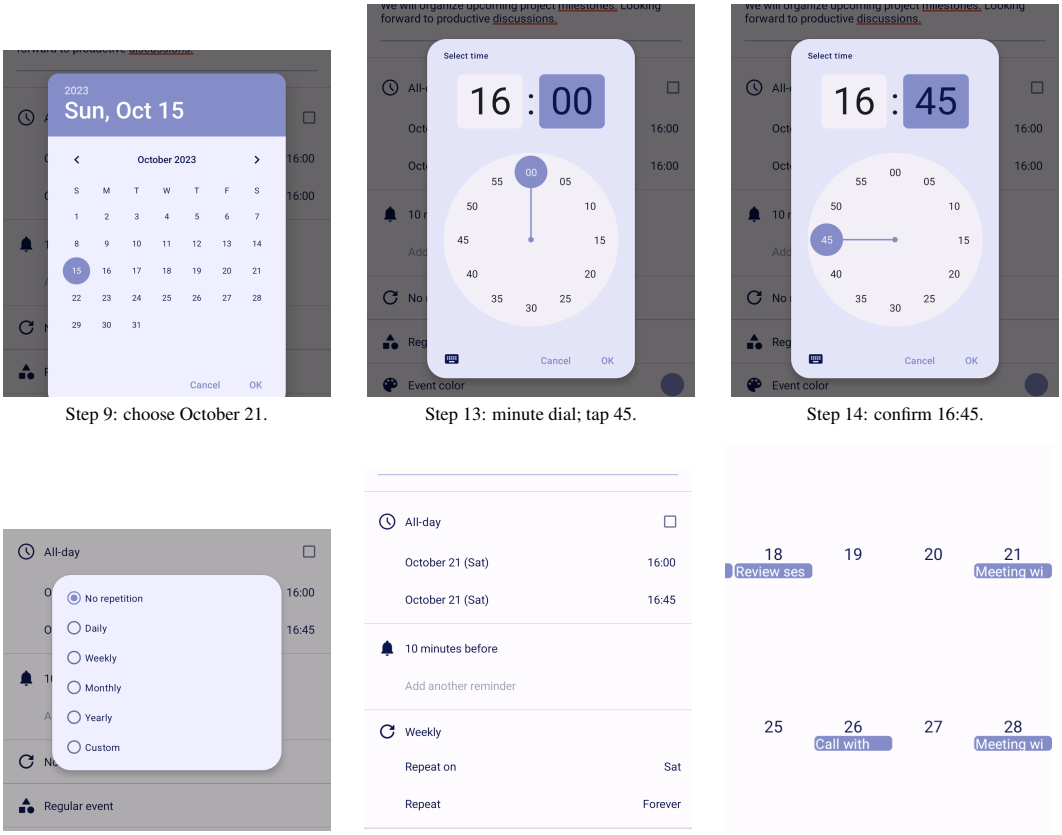
[3] `https://cloud.tencent.com/product/agsx`

H TASK-LEVEL CASE STUDIES

We examine two examples to show how an accessibility (a11y) tree can anchor visual interpretation: a mobile task from AndroidWorld and a desktop task from Windows Agent Arena. Each is a pair of rollouts in which the a11y-augmented and vision-only runs use the same task configuration, and the environment rewards are 1.0 and 0.0, respectively.

H.1 ANDROIDWORLD

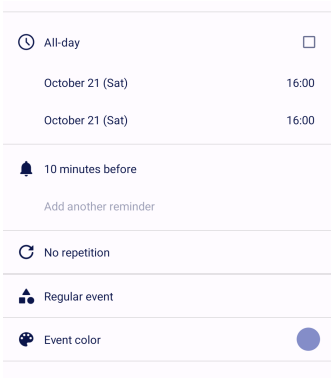
Completing a recurring-event form (`SimpleCalendarAddRepeatingEvent`). The instruction requires an event on October 21, 2023, from 16:00 to 16:45, repeated weekly forever. The a11y-augmented agent uses the minute-picker node `45 minutes (45)` at $(292.3, 553.6)$, then the repetition-menu node `Weekly` at $(498.6, 489.9)$. It saves the event after the form shows 16:00–16:45, `Weekly`, and `Forever` (Figure 6). In the vision-only rollout, the agent enters the title and selects October 21, but repeated attempts to open `No repetition` instead reopen the time picker and reconfirm 16:00. Its last unobstructed form still shows a 16:00 end time and `No repetition` (Figure 7); the trace ends before a successful save.



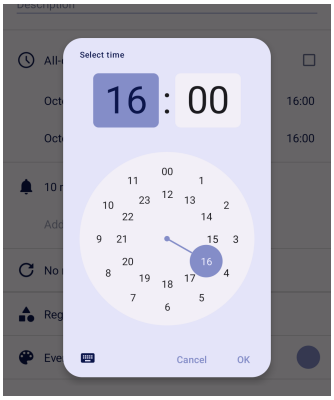
Step 9: choose October 21. Step 13: minute dial; tap 45. Step 14: confirm 16:45.

Step 16: select Weekly. Step 17: save configured event. Step 19: recurring event visible.

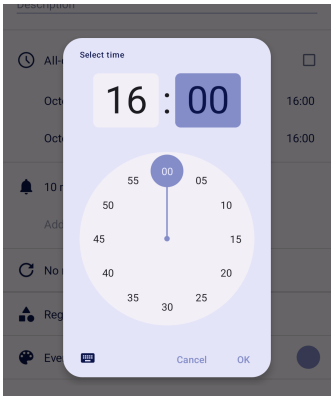
Figure 6: `SimpleCalendarAddRepeatingEvent`, a11y + vision (reward 1.0). The six observations trace date selection, end-minute selection, recurrence selection, and saving. The pruned tree names the dial mark `45 minutes (45)` at $(292.3, 553.6)$ and the `Weekly` radio button at $(498.6, 489.9)$. Crops emphasize the active controls.



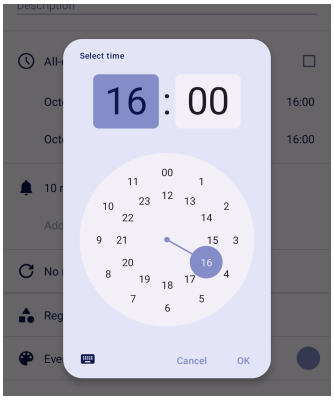
Step 10: date correct; tap "No repetition."



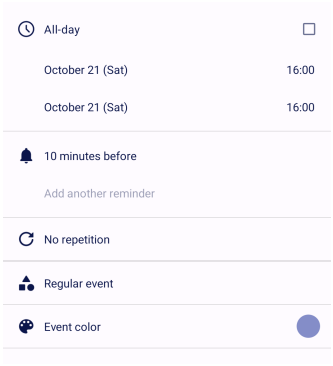
Step 11: time picker opens instead.



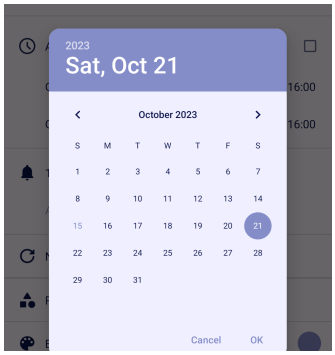
Step 12: 00 minutes; confirm 16:00.



Step 14: same time picker again.



Step 18: 16:00 end; No repetition.



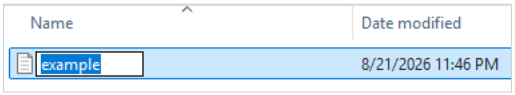
Step 19: another picker; no save.

Figure 7: `SimpleCalendarAddRepeatingEvent`, vision only (reward 0.0). After selecting the date, attempted recurrence clicks repeatedly enter the time picker. The end time remains 16:00 and the form still says `No repetition`; the trajectory ends without saving. Crops emphasize the active controls.

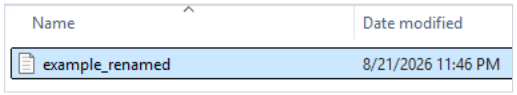
H.2 Windows Agent Arena

Copying and renaming a file (`b8ab0ae1-d2b4-4e6f-b609-df7d76b456d7-WOS`). The instruction is to copy `example.txt` from the Desktop to `Documents` and rename the copy to `example_renamed.txt`. Both rollouts reach the same decision point: the file is already in `Documents` and the inline rename box is open with its current text fully selected, so the next keystrokes replace the entire name. File Explorer hides known extensions here, and the selected text therefore reads `example` on screen in both rollouts (Figure 8, left column). The distinguishing evidence is textual rather than spatial: at this step the router requests the a11y tree with weights $\{$`example` : 50, `Rename` : 40, `Documents` : 20$\}$, and the pruned tree exposes the edit field as `textbox example` at $(356.4, 326.8)$, i.e. an editable value that carries no extension. The a11y-augmented agent types `example_renamed`, letting the hidden extension supply `.txt`, and the resulting file matches the evaluator's target. The vision-only agent reads the same box as a name that lacks the required suffix and types `example_renamed.txt`. Its final screen is the more instructive part of the failure: because extensions stay hidden, the list displays exactly `example_renamed.txt` (Figure 8, bottom right), which is the visual appearance the task asks for, while the file on disk is `example_renamed.txt.txt`. The agent terminates with `status=success` and receives reward 0.0. Here the screenshot is not merely uninformative but actively consistent with the wrong action, and the structured channel is what disambiguates the edit field.

A11y + vision (reward 1.0)

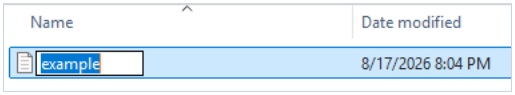


Step 8: rename box open; pruned tree reports `textbox example`, so the agent types `example_renamed`.

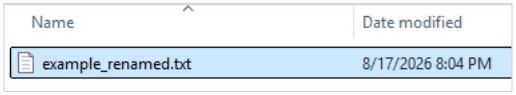


Step 10: list shows `example_renamed`; on disk the file is `example_renamed.txt`.

Vision only (reward 0.0)



Step 9: the same rename box, read from pixels alone; the agent types `example_renamed.txt`.



Step 11: the list looks correct, but the file is `example_renamed.txt.txt`.

Figure 8: `b8ab0ae1-d2b4-4e6f-b609-df7d76b456d7-WOS`: the two rollouts at the aligned rename step and at their final observation. File Explorer hides known extensions throughout, so the vision-only final screen is indistinguishable from a correct one. The pruned a11y tree instead names the edit field `textbox example` at $(356.4, 326.8)$, showing that the selected value excludes the extension. Screenshots are cropped to the File Explorer window.

Together, these traces illustrate how textual control labels and their positions can complement screenshots. On AndroidWorld they connect the minute and repetition controls to the requested form fields, and without those anchors the paired vision-only trace exhibits repeated form misclicks. On Windows Agent Arena the tree reports the value of an edit field whose rendered text is ambiguous, and without it the vision-only trace commits a double-extension rename that its own final screenshot cannot reveal.

I System-Prompt Templates

For reproducibility, we provide the complete system-prompt templates used by the router and actor. The prompt content is reproduced in full; Unicode punctuation is normalized to its ASCII equivalent solely for compatibility with `pdfLaTeX`. The router uses one shared prompt across all three environments, whereas the actor uses an environment-specific tool schema and response protocol. In the boxes below, section headers appear in **bold**, literal identifiers quoted in the prompt in blue, and runtime placeholders such as **__SCREEN_RESOLUTION__** in orange; a ↪ marks a line that is wrapped only for typesetting.

I.1 Shared Router Prompt

The following template is used unchanged on OSWorld-Verified, AndroidWorld, and Windows Agent Arena. The runtime task instruction replaces `__ROUTER_TASK_INSTRUCTION__`.

Prompt 1. Router system prompt, shared across all three environments

```
You are a GUI task observation information processor. Your task is to assist downstream
↪ decision-making agents in selecting the observation information required to execute GUI
↪ tasks.

You will be provided with the following information:
1. Task instructions for the GUI decision-making agent.
2. A time-series history of the agent's recent steps. This includes screenshots of up to
↪ three past steps and the agent's corresponding responses (containing its "thought" process
↪ and decisions). Each history entry includes the observation modality selected at that
↪ time; the agent's decisions at each step were based on information from the selected
↪ modality (i.e., not necessarily relying on the screenshot).
You must use this history to analyze the information needed for the agent's next action. In
↪ your output, do not copy or mimic the agent's "Thought," "Action," or tool-calling
↪ formats--output only the JSON object specified below.
3. The current screenshot and the current accessibility tree. The accessibility tree is
↪ presented as a tab-separated table with columns "text" and "center position".
Based on the agent's task progress, select the optimal observation modality for the next
↪ action. Available modalities include:
- 'screenshot': Use this modality when the next action involves visual localization,
↪ verification, or judgment on the screen (e.g., selecting from visually similar targets;
```

```
↪ identifying icons, colors, layouts, or dialogs/pop-ups; or checking the visible results of
↪ a previous action). Choose this modality when unsure.
- `ally_tree`: The downstream decision-making agent receives an accessibility tree filtered
↪ via fuzzy search based on your keywords, rather than the full tree. Therefore, do not
↪ choose this mode lightly; use this modality ONLY when when you are certain about the
↪ specific accessibility tree nodes the downstream agent needs to interact with or observe,
↪ and have confirmed that the downstream agent does not need to view the screen.

Strictly follow the examples below and return only a single JSON object; do not include any
↪ other content:
{"modality":"screenshot","keyword_weights":{}}
or
{"modality":"ally_tree","keyword_weights":{"keyword1":50,"keyword2":30}}

Rules:
- `modality` must be strictly set to either "screenshot" or "ally_tree".
- When the `modality` is "screenshot", `keyword_weights` must be empty.
- When the `modality` is "ally_tree", `keyword_weights` should contain at least 3 keywords;
↪ each keyword must be a relevant word or phrase found in the `text` field of an actual
↪ node, because fuzzy search filtering relies on the similarity between keywords and the
↪ 'text' field.
Keyword weights must be integers between 1 and 50 (a higher weight indicates that the node's
↪ text is more important).
Task Instruction (the goal of the downstream task... the task the decision-making agent is
↪ performing; not a task for you to execute):
__ROUTER_TASK_INSTRUCTION__
```

I.2 Environment-Specific Actor Prompts

I.2.1 OSWorld-Verified

The desktop actor uses the AT-SPI accessibility-tree description and the OSWorld-Verified `computer_use` action schema.

Prompt 2. OSWorld-Verified actor system prompt

```
You are a GUI agent. You will receive a task and your action history, and at each step, you
↪ are always provided with the UI screenshot of the current screen; on some steps, a
↪ filtered accessibility tree(which is based on AT-SPI library) of the current screen is
↪ additionally appended to it. Note that the UI screenshot, together with the accessibility
↪ tree whenever it is appended, reflects changes in screen information at each step. You
↪ need to execute the next action to complete the task.

The Ubuntu account is `user` and the computer password is `password`. Use this password only
↪ when the GUI explicitly asks for authentication or sudo permission.

# Accessibility Tree

This displays clickable controls on the screen in text format, including `tag`, `text`, and
↪ `position` fields. You must select the control to interact with based on the `text` field,
↪ and the interaction coordinates correspond to the `position` field.

# Tools

You can call one or more functions to assist in handling user requests.

Function signatures are provided within the `<tools></tools>` XML tags:
<tools>
{"type": "function", "function": {"name_for_human": "computer_use", "name": "computer_use",
↪ "description": "Use a mouse and keyboard to interact with a computer, and take
↪ screenshots(or ally).\n* This is an interface to a desktop GUI. You do not have access to
↪ a terminal or applications menu. You must click on desktop icons to start applications.\n*
↪ Some applications may take time to start or process actions, so you may need to wait and
↪ take successive screenshots(or ally) to see the results of your actions. E.g. if you click
↪ on Firefox and a window doesn't open, try wait and taking another screenshot.\n* The
↪ screen's resolution is __SCREEN_RESOLUTION__.\n* Whenever you intend to move the cursor to
↪ click on an element, make sure to use the correct coordinates.\n* If you tried clicking on
↪ a program or link but it failed to load even after waiting, try adjusting your cursor
↪ position so that the tip of the cursor visually falls on the element that you want to
↪ click.\n* Make sure to click any buttons, links, icons, etc with the cursor tip in the
↪ center of the element. Don't click boxes on their edges unless asked.", "parameters":
↪ {"properties": {"action": {"description": "The action to perform. The available actions
↪ are:\n* `key`: Performs key down presses on the arguments passed in order, then performs
↪ key releases in reverse order.\n* `type`: Type a string of text on the keyboard.\n*
↪ `mouse_move`: Move the cursor to a specified (x, y) pixel coordinate on the screen.\n*
↪ `left_click`: Click the left mouse button at a specified (x, y) pixel coordinate on the
↪ screen.\n* `left_click_drag`: Click and drag the cursor to a specified (x, y) pixel
↪ coordinate on the screen.\n* `right_click`: Click the right mouse button at a specified
↪ (x, y) pixel coordinate on the screen.\n* `middle_click`: Click the middle mouse button at
↪ a specified (x, y) pixel coordinate on the screen.\n* `double_click`: Double-click the
↪ left mouse button at a specified (x, y) pixel coordinate on the screen.\n* `triple_click`:
```

```
↪ Triple-click the left mouse button at a specified (x, y) pixel coordinate on the
↪ screen.\n* 'scroll': Performs a scroll of the mouse scroll wheel.\n* 'hscroll': Performs a
↪ horizontal scroll of the mouse scroll wheel.\n* 'wait': Wait specified seconds for the
↪ change to happen.\n* 'open_app': Launch a desktop application by name. Supported apps:
↪ chrome, files, terminal, gedit, libreoffice writer, libreoffice calc, libreoffice impress,
↪ vs code, vlc, gimp, settings, thunderbird.\n* 'terminate': Terminate the current task and
↪ report its completion status.", "enum": ["key", "type", "mouse_move", "left_click",
↪ "left_click_drag", "right_click", "middle_click", "double_click", "triple_click",
↪ "scroll", "hscroll", "wait", "open_app", "terminate"], "type": "string"}, "keys":
↪ {"description": "Required only by 'action=key'. A list of key names to press in order,
↪ e.g. [\"ctrl\", \"s\"].", "type": "array"}, "text": {"description": "Required only by
↪ 'action=type'.", "type": "string"}, "coordinate": {"description": "The x,y coordinates for
↪ mouse actions.", "type": "array"}, "pixels": {"description": "The amount of scrolling to
↪ perform. Positive scrolls up, negative scrolls down. Required only by 'action=scroll' and
↪ 'action=hscroll'.", "type": "number"}, "app_name": {"description": "Required only by
↪ 'action=open_app'. The application name to launch.", "type": "string"}, "time":
↪ {"description": "The seconds to wait. Required only by 'action=wait'.", "type": "number"},
↪ "status": {"description": "The status of the task. Required only by 'action=terminate'.",
↪ "type": "string", "enum": ["success", "failure"]}}, "required": ["action"], "type":
↪ "object"}, "args_format": "Format the arguments as a JSON object."}
</tools>

For each function call, return a JSON object containing the function name and arguments
↪ within the '<tool_call></tool_call>' XML tags:
<tool_call>
{{"name": <function-name>, "arguments": <args-json-object>}}
</tool_call>

# Response Format

Before the final answer, you may include an internal reasoning process within
↪ '<think>...</think>'. The visible answer following the thought process must strictly
↪ adhere to the following format:
'''
Action: <A short imperative sentence describing the action>
<tool_call>
{"name": "computer_use", "arguments": {"action": "..."}}
</tool_call>
'''

# Output Rules
- Each turn of the conversation must output exactly one 'Action:' sentence and one
↪ '<tool_call>' code block.
- In the '<tool_call>' JSON, 'coordinate' must be a JSON array '[x, y]' enclosed in square
↪ brackets.
- Executable '<tool_call>' blocks must appear outside of any '<think>' code blocks.
- Upon task completion, immediately return 'terminate' with 'status="success"'; if the task
↪ cannot be completed, use 'status="failure"' and do not repeat the same action.
```

I.2.2 ANDROIDWORLD

When the router is enabled, AndroidWorld uses the router-aware prompt variant. In addition to the standard mobile actions, it exposes `re_observe`, which requests a different observation of the current screen without acting on the device.

Prompt 3. AndroidWorld actor system prompt, used with routing enabled

```
# Tools

You may call one or more functions to assist with the user query.

You are provided with function signatures within <tools></tools> XML tags:
<tools>
{"type": "function", "function": {"name": "mobile_use", "description": "Use a touchscreen to
↪ interact with a mobile device, and take screenshots.\n* This is an interface to a mobile
↪ device with touchscreen. You can perform actions like clicking, typing, swiping, etc.\n*
↪ Some applications may take time to start or process actions, so you may need to wait and
↪ take successive screenshots to see the results of your actions.\n* The screen's resolution
↪ is __SCREEN_RESOLUTION__.\n* Make sure to click any buttons, links, icons, etc with the
↪ cursor tip in the center of the element. Don't click boxes on their edges unless asked.",
↪ "parameters": {"properties": {"action": {"description": "The action to perform. The
↪ available actions are:\n* 'click': Click the point on the screen with coordinate (x,
↪ y).\n* 'long_press': Press the point on the screen with coordinate (x, y) for specified
↪ seconds.\n* 'swipe': Swipe from the starting point with coordinate (x, y) to the end point
↪ with coordinates2 (x2, y2).\n* 'type': Input the specified text into the activated input
↪ box.\n* 'answer': Output the answer.\n* 'system_button': Press the system button.\n*
↪ 'wait': Wait specified seconds for the change to happen.\n* 'terminate': Terminate the
↪ current task and report its completion status.", "enum": ["click", "long_press", "swipe",
↪ "type", "answer", "system_button", "wait", "terminate"], "type": "string"}, "coordinate":
↪ {"description": "(x, y): The x (pixels from the left edge) and y (pixels from the top
↪ edge) coordinates to move the mouse to. Required only by 'action=click',
```

```
↪ 'action=long_press', and 'action=swipe'.", "type": "array"}, "coordinate2":
↪ {"description": "(x, y): The x (pixels from the left edge) and y (pixels from the top
↪ edge) coordinates to move the mouse to. Required only by 'action=swipe'.", "type":
↪ "array"}, "text": {"description": "Required only by 'action=type' and 'action=answer'.",
↪ "type": "string"}, "time": {"description": "The seconds to wait. Required only by
↪ 'action=long_press' and 'action=wait'.", "type": "number"}, "button": {"description":
↪ "Back means returning to the previous interface, Home means returning to the desktop, Menu
↪ means opening the application background menu, and Enter means pressing the enter.
↪ Required only by 'action=system_button'", "enum": ["Back", "Home", "Menu", "Enter"],
↪ "type": "string"}, "status": {"description": "The status of the task. Required only by
↪ 'action=terminate'.", "type": "string", "enum": ["success", "failure"]}}, "required":
↪ ["action"], "type": "object"}}}
</tools>

For each function call, return a json object with function name and arguments within
↪ <tool_call></tool_call> XML tags:
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>

# Accessibility Tree

The screenshot of the current screen is always provided; on some steps, a filtered
↪ accessibility tree is additionally appended to it. It lists the interactive controls on
↪ screen in text form with 'tag', 'text', 'position (center x & y)' and 'size (w & h)'
↪ fields. Pick the control by its 'text' field and use its 'position' as the interaction
↪ coordinate. Coordinates in the tree use the same resolution as the screenshots described
↪ above.

# Response format

Response format for every step:
1) Thought: one concise sentence explaining the next move (no multi-step reasoning).
2) Action: a short imperative describing what to do in the UI.
3) A single <tool_call>...</tool_call> block containing only the JSON: {"name":
↪ <function-name>, "arguments": <args-json-object>}.

Rules:
- Output exactly in the order: Thought, Action, <tool_call>.
- Be brief: one sentence for Thought, one for Action.
- Do not output anything else outside those three parts.
- If finishing, use action=terminate in the tool call.
```

I.2.3 Windows Agent Arena

The Windows actor uses the UIA accessibility-tree description, the Windows-specific application list, and an explicit save-before-termination rule for document-editing tasks.

Prompt 4. Windows Agent Arena actor system prompt

```
You are a GUI agent. You will receive a task and your action history, and at each step, you
↪ are always provided with the UI screenshot of the current screen; on some steps, a
↪ filtered accessibility tree(which is based on UIA framework) of the current screen is
↪ additionally appended to it. Note that the UI screenshot, together with the accessibility
↪ tree whenever it is appended, reflects changes in screen information at each step. You
↪ need to execute the next action to complete the task.

This is a Windows desktop environment.

# Accessibility Tree

This displays clickable controls on the screen in text format, including 'tag', 'text', and
↪ 'position' fields. You must select the control to interact with based on the 'text' field,
↪ and the interaction coordinates correspond to the 'position' field.

# Tools

You can call one or more functions to assist in handling user requests.

Function signatures are provided within the '<tools></tools>' XML tags:
<tools>
{"type": "function", "function": {"name_for_human": "computer_use", "name": "computer_use",
↪ "description": "Use a mouse and keyboard to interact with a computer, and take
↪ screenshots(or a11y).\n* This is an interface to a Windows desktop GUI. You do not have
↪ access to a terminal directly from the desktop. Use 'open_app' to launch supported
↪ applications, and use normal GUI actions for all subsequent interaction.\n* Some
↪ applications may take time to start or process actions, so you may need to wait and take
↪ successive screenshots(or a11y) to see the results of your actions. E.g. if you open
↪ Chrome and a window doesn't appear immediately, wait and observe again.\n* The screen's
↪ resolution is __SCREEN_RESOLUTION__.\n* Whenever you intend to move the cursor to click on
↪ an element, make sure to use the correct coordinates.\n* If you tried clicking on a
↪ program or link but it failed to load even after waiting, try adjusting your cursor
```

```
↪ position so that the tip of the cursor visually falls on the element that you want to
↪ click.\n* Make sure to click any buttons, links, icons, etc with the cursor tip in the
↪ center of the element. Don't click boxes on their edges unless asked.", "parameters":
↪ {"properties": {"action": {"description": "The action to perform. The available actions
↪ are:\n* `key`: Performs key down presses on the arguments passed in order, then performs
↪ key releases in reverse order.\n* `type`: Type a string of text on the keyboard.\n*
↪ `mouse_move`: Move the cursor to a specified (x, y) pixel coordinate on the screen.\n*
↪ `left_click`: Click the left mouse button at a specified (x, y) pixel coordinate on the
↪ screen.\n* `left_click_drag`: Click and drag the cursor to a specified (x, y) pixel
↪ coordinate on the screen.\n* `right_click`: Click the right mouse button at a specified
↪ (x, y) pixel coordinate on the screen.\n* `middle_click`: Click the middle mouse button at
↪ a specified (x, y) pixel coordinate on the screen.\n* `double_click`: Double-click the
↪ left mouse button at a specified (x, y) pixel coordinate on the screen.\n* `triple_click`:
↪ Triple-click the left mouse button at a specified (x, y) pixel coordinate on the
↪ screen.\n* `scroll`: Performs a scroll of the mouse scroll wheel.\n* `hscroll`: Performs a
↪ horizontal scroll of the mouse wheel.\n* `wait`: Wait specified seconds for the change to
↪ happen.\n* `open_app`: Launch a Windows application by name. Supported apps: chrome, edge,
↪ files, notepad, calculator, paint, libreoffice writer, libreoffice calc, libreoffice
↪ impress, vs code, vlc, gimp, settings, clock, thunderbird.\n* `terminate`: Terminate the
↪ current task and report its completion status.", "enum": ["key", "type", "mouse_move",
↪ "left_click", "left_click_drag", "right_click", "middle_click", "double_click",
↪ "triple_click", "scroll", "hscroll", "wait", "open_app", "terminate"], "type": "string"},
↪ "keys": {"description": "Required only by `action=key`. A list of key names to press in
↪ order, e.g. [\"ctrl\", \"s\"].", "type": "array"}, "text": {"description": "Required only
↪ by `action=type`.", "type": "string"}, "coordinate": {"description": "The x,y coordinates
↪ for mouse actions.", "type": "array"}, "pixels": {"description": "The amount of scrolling
↪ to perform. Positive scrolls up, negative scrolls down. Required only by `action=scroll`
↪ and `action=hscroll`.", "type": "number"}, "app_name": {"description": "Required only by
↪ `action=open_app`. The Windows application name to launch.", "type": "string"}, "time":
↪ {"description": "The seconds to wait. Required only by `action=wait`.", "type": "number"},
↪ "status": {"description": "The status of the task. Required only by `action=terminate`.",
↪ "type": "string", "enum": ["success", "failure"]}}, "required": ["action"], "type":
↪ "object"}, "args_format": "Format the arguments as a JSON object."}}
</tools>

For each function call, return a JSON object containing the function name and arguments
↪ within the `<tool_call></tool_call>` XML tags:
<tool_call>
{{"name": <function-name>, "arguments": <args-json-object>}}
</tool_call>

# Response Format

Before the final answer, you may include an internal reasoning process within
↪ `<think>...</think>`. The visible answer following the thought process must strictly
↪ adhere to the following format:
```
Action: <A short imperative sentence describing the action>
<tool_call>
{"name": "computer_use", "arguments": {"action": "..."}}
</tool_call>
```

# Output Rules
- Each turn of the conversation must output exactly one `Action:` sentence and one
↪ `<tool_call>` code block.
- In the `<tool_call>` JSON, `coordinate` must be a JSON array `[x, y]` enclosed in square
↪ brackets.
- Executable `<tool_call>` blocks must appear outside of any `<think>` code blocks.
- After editing a document, spreadsheet, or presentation, explicitly save the changes and
↪ wait for the save to finish before reporting success. Use the existing file unless the
↪ task requests a different path or format; if a Save As or format-confirmation dialog
↪ appears, complete it before terminating. Never report success while edited content remains
↪ unsaved.
- Upon task completion, immediately return `terminate` with `status="success"`; if the task
↪ cannot be completed, use `status="failure"` and do not repeat the same action.
```