OS-THEMIS: A SCALABLE CRITIC FRAMEWORK FOR GENERALIST GUI REWARDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement Learning (RL) has the potential to improve the robustness of GUI agents in stochastic environments, yet its effectiveness critically depends on the quality of the reward signals. Existing methods for constructing such rewards struggle to achieve both scalability and accuracy. We propose OS-Themis, a scalable multi-agent critic framework that decomposes trajectories into verifiable milestones to isolate evidence and audits the evidence chain before producing the verdict. Building on these milestones, we introduce Milestone-Credit Group Relative Policy Optimization (MC-GRPO), leveraging verified milestones for fine-grained intra-trajectory credit assignment in GRPO. We also introduce OmniGUIRewardBench (OGRBench), a holistic multi-platform benchmark for GUI outcome rewards, where OS-Themis consistently outperforms existing reward approaches across models. In online RL, OS-Themis with MC-GRPO improves AndroidWorld performance by up to 9.4%, with MC-GRPO bringing an additional 2.5-2.9% improvement over standard GRPO. We further build a scalable Android online RL infrastructure, where scaling training to 1,024 tasks yields a 10.7% improvement on AndroidWorld, demonstrating the potential of OS-Themis for scalable agent evolution.

1 INTRODUCTION

Recent advancements in general-purpose Vision-Language Models (OpenAI, 2025; Anthropic, 2025b; Comanici et al., 2025; Bai et al., 2025; Wang et al., 2025c; Guo et al., 2025b) have accelerated development of GUI agents (Wu et al., 2024a; Agashe et al., 2024; Wang et al., 2024; Nguyen et al., 2025; Yang et al., 2026). While native agents (Wu et al., 2024b; Xu et al., 2024; Wang et al., 2025d; Liu et al., 2025; Qin et al., 2025) trained on extensive GUI trajectories demonstrate proficiency in digital navigation, they suffer from poor error recovery in stochastic environments, often failing when deviating from expert trajectories (Qin et al., 2025; Liu et al., 2025; Lu et al., 2025b). This deficiency has precipitated a shift toward reinforcement learning (RL) to enable adaptive correction (Xu et al., 2025; Lai et al., 2025b; Wang et al., 2025a); critically, the success of this paradigm hinges on reward signals both reliable at the trajectory level and informative for long-horizon credit assignment.

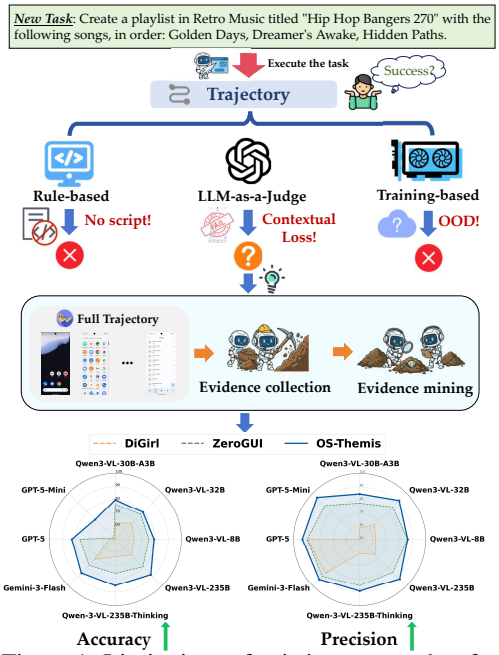


Figure 1: Limitations of existing approaches for reward modeling in GUI environments.

As illustrated in Figure 1, existing methods for reward acquisition in GUI scenarios generally fall into three categories: 1) **Rule-based rewards** rely on manual heuristics (Ye et al., 2025; Lai et al., 2025b), offering high precision but suffering from limited scalability and susceptibility to reward hacking. 2) **Training-based critics** learn verifiers from

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.

human feedback (Qi et al., 2024; Xu et al., 2025; Wu et al., 2025b), yet they demand expensive data construction and generalize poorly to out-of-distribution (OOD) environments. 3) **LLM-as-a-judge** offers a flexible and scalable paradigm by harnessing the generalized reasoning capabilities and extensive world knowledge of foundation models (Lee et al., 2024; Li et al., 2024; Zhou et al., 2025).

Despite its potential for generalizable evaluation, current LLM-based reward pipelines face three coupled bottlenecks. First, effective trajectory utilization remains difficult. Sparse sampling (Yang et al., 2025a; Qi et al., 2024; Lai et al., 2025a) suffers from contextual loss, while global aggregation (Bai et al., 2024; Wang et al., 2025b) is compromised by low signal-to-noise ratios in long-horizon tasks. Consequently, both extremes fail to distill decision-critical evidence. Second, converting extracted evidence into reliable outcomes remains challenging. The prevailing *one-shot* paradigm suffers from *evidence dilution*, where accumulated trivial successes mask sparse, outcome-determining failures, producing overly optimistic rewards and potentially wrong-sign policy updates. Third, even a correct terminal reward provides only coarse supervision: standard outcome-based GRPO broadcasts a single scalar advantage to all trainable tokens, assigning the same credit to decisive actions and incidental steps. Thus, the challenge is not only to determine whether a trajectory succeeds, but also to identify which local actions are responsible for that outcome.

To address these challenges, we propose OS-Themis, a scalable critic framework that extends structured trajectory evaluation into structured policy supervision. The **Milestone Verification Module (MVM)** uses a **Selector Agent** to decompose long-horizon trajectories into verifiable milestones with explicit observable goals, and a **Verifier Agent** to assess them sequentially. The **Verdict Calibration Module (VCM)** then employs a **Reviewer Agent** to audit the evidence chain for overlooked, outcome-determining failures before a **Judge Agent** produces the terminal outcome.

We further introduce **Milestone-Credit Group Relative Policy Optimization (MC-GRPO)**, which leverages verified milestone outcomes produced by OS-Themis to extend GRPO from trajectory-level optimization to evidence-guided credit assignment. Standard outcome-based GRPO assigns the same trajectory-level advantage to all trainable tokens within a rollout, making it difficult to distinguish decisive actions from incidental behaviors in long-horizon GUI tasks. MC-GRPO turns verified milestone evidence into outcome-conditioned credit weights that redistribute the existing advantage within each trajectory. This enables fine-grained credit assignment while preserving the terminal reward, the sign of each token's advantage, and the total signed advantage within the trajectory. Together, OS-Themis and MC-GRPO establish a complete path from evidence extraction, to reliable outcome determination, to structured policy optimization.

We evaluate OS-Themisusing **OmniGUIRewardBench (OGRBench)**, a multi-platform GUI ORM benchmark, and a scalable Android online RL infrastructure. Across the tested models, OS-Themisconsistently achieves superior reward-evaluation performance on OGRBench. In closed-loop training, using OS-Themisas the reward function yields gains of up to 9.4% over baselines on AndroidWorld, while a 1,024-task scaling study improves performance by 10.7%. Downstream trajectory-filtering experiments further validate its utility for self-evolving GUI agents.

Our contributions are summarized as follows:

- We propose OS-Themis, a scalable multi-agent critic framework that combines milestone verification and verdict calibration to identify critical evidence for reliable GUI outcome rewards.
- We introduce MC-GRPO, a milestone-aware policy optimization algorithm that reuses verified milestone evidence for fine-grained credit assignment.
- We construct OGRBench, the first holistic multi-platform GUI ORM benchmark for comprehensively evaluating GUI reward models, where OS-Themis consistently achieves the best performance across all evaluated base models.
- We build a scalable Android infrastructure that enables controlled, closed-loop online RL for GUI agents. Extensive experiments demonstrate substantial improvements in online learning performance and trajectory curation efficiency, highlighting its potential for scalable agent evolution.

2 Related Works

GUI Agents. Empowered by general-purpose Vision-Language Models (VLMs) (Bai et al., 2025; Wang et al., 2025c; OpenAI, 2025; Anthropic, 2025b; Comanici et al., 2025), visually-grounded

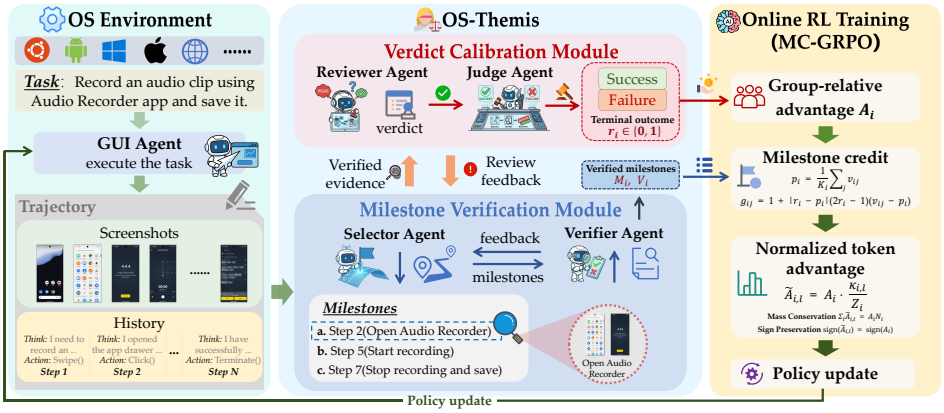


Figure 2: Overview of the OS-Themis framework and MC-GRPO. Given a GUI trajectory of `<screenshot,think,action>` steps, the **Milestone Verification Module (MVM)** uses the **Selector Agent** to identify milestones and the **Verifier Agent** to assign binary scores. In the **Verdict Calibration Module (VCM)**, the **Reviewer Agent** audits milestones and guides their refinement through interaction with the MVM, while the **Judge Agent** synthesizes all evidence to produce a terminal reward. **Milestone-Credit Group Relative Policy Optimization (MC-GRPO)** reuses verified milestone outcomes to redistribute GRPO advantages within each trajectory, enabling fine-grained credit assignment while preserving token-level advantage signs and the trajectory's total signed advantage.

GUI agents have emerged as autonomous copilots for digital tasks (Cheng et al., 2024; Sun et al., 2024; Zhang et al., 2025b; Xie et al., 2025; OpenAI, 2025; Anthropic, 2024). A dominant paradigm involves *native agents* (Wu et al., 2024b; Qin et al., 2025; Liu et al., 2025; Xu et al., 2024; Wang et al., 2025d) that integrate planning and grounding in a single-agent architecture, directly translating pixel-level observations into executable actions. However, despite mastering routine workflows via large-scale training, these agents often exhibit brittleness in stochastic environments, struggling to recover from deviations or generalize to unseen scenarios (Ye et al., 2025; Liu et al., 2025; Lai et al., 2025b). These limitations have precipitated a shift toward reinforcement learning (RL) in interactive environments (Xu et al., 2025; Lai et al., 2025b; Wang et al., 2025a), where policy optimization hinges on the reliability of reward signals.

Reward Modeling for GUI Agents. Reliable reward mechanisms are foundational for GUI agent RL. Existing strategies fall into three paradigms: 1) **Rule-based methods** use heuristic triggers or environment states Ye et al. (2025); Lai et al. (2025b); Wang et al. (2025a). While providing faithful rewards via verifiable scripts, they scale poorly and are prone to reward hacking. 2) **Training-based critics** learn verifiers from human feedback or expert trajectories Qi et al. (2024); Xu et al. (2025); Wu et al. (2025b); Sun et al. (2025b); Wang et al. (2025a); Sun et al. (2025a). Although specialized models like GUI-Critic-R1 Wanyan et al. (2025) and UI-Genie Xiao et al. (2025) offer step-wise signals, domain shifts and costly data collection limit their applicability across diverse GUI platforms. 3) **LLM-as-a-judge** harnesses off-the-shelf VLMs for scalable, zero-shot evaluation Lee et al. (2024); Li et al. (2024); Murty et al. (2025); Yang et al. (2025a); Wu et al. (2025a); Zhou et al. (2025); Zheng et al. (2026). In this paradigm, ZeroGUI Yang et al. (2025a) and DigiRL Bai et al. (2024) represent two classic approaches: sequential evaluation until success versus selecting fixed-number terminal states. However, as Outcome Reward Models (ORMs), they face critical hurdles: sparse sampling loses critical context Yang et al. (2025a); Qi et al. (2024), while global ingestion suffers from low signal-to-noise ratios Bai et al. (2024); Wang et al. (2025b). Crucially, bridging this gap requires mobilizing VLM reasoning capabilities to mine fine-grained evidence for precise reward estimation.

GUI Reward Benchmarks. Despite the proliferation of benchmarks assessing GUI agent execution (Xie et al., 2024; Rawles et al., 2024; Bonatti et al., 2024; Wang et al., 2025e), datasets evaluating criticism capabilities are scarce. Existing initiatives often suffer from limited scope or data contamination. For instance, GUI-Critic-Test Wanyan et al. (2025) incurs leakage risks due to its derivation from open-source repositories. Similarly, recent works like OS-Critic Bench Wu et al. (2025b), AgentRewardBench Lù et al. (2025), and CUARewardBench Lin et al. (2025) are restricted to isolated domains (e.g., Web or Desktop) or prioritize step-level supervision. Consequently, there is an urgent need for a comprehensive ORM evaluation benchmark to establish a reliable evaluation standard for multi-platform reward modeling.

3 OS-THEMIS

3.1 OVERVIEW

In this section, we introduce OS-Themis, a multi-agent critic framework for generalist GUI rewards. As shown in Figure 2, the framework primarily consists of two modules: the **Milestone Verification Module (MVM)** and the **Verdict Calibration Module (VCM)**. Given a trajectory τ, the MVM first generates an initial milestone set M_0, which is then iteratively refined through interaction with the VCM to produce the final milestone set M. Subsequently, the VCM assigns a binary reward score $r \in \{0, 1\}$ to the trajectory based on all interaction information between the two modules and the complete refinement process from M_0 to M. Building on the verified milestones and terminal reward, we further derive **Milestone-Credit Group Relative Policy Optimization (MC-GRPO)**, which extends structured trajectory evaluation to fine-grained credit assignment for policy learning.

3.2 MILESTONE VERIFICATION MODULE

Evaluating long-horizon GUI trajectories with a one-shot judge is error-prone. It requires identifying critical decision points across dozens of steps and reasoning over the full interaction history, often missing evidence or producing inconsistent judgments. Most GUI tasks can be decomposed into verifiable subgoals, called *milestones*. A milestone is a necessary intermediate state for task completion. Crucially, each milestone can be verified locally via the relevant state transition rather than processing the entire trajectory.

Building on this, we introduce the **Milestone Verification Module (MVM)**, which decomposes trajectories into milestones and verifies them step by step. It comprises two collaborative agents:

Selector Agent. Given a task instruction $\mathcal{I}$ and a trajectory $\tau = \{(s_t, a_t, m_t)\}_{t=1}^{T}$, where s_t denotes the screenshot state, a_t the action, and m_t the agent-generated metadata at environment step t, the Selector identifies a compact set of milestone candidates $\mathcal{M}_0 = \{(t_j, d_j, h_j)\}_{j=1}^{K}$. Each milestone tuple consists of: (i) the environment-step index t_j where verification should occur, (ii) an **assignment goal** d_j specifying the expected UI post-condition as a concrete and observable state change, and (iii) a rationale h_j explaining why this milestone is necessary for task completion. By making the verification target explicit, d_j provides the Verifier with a clear success criterion and reduces ambiguity from missing context. The Selector's output thus provides a structured decomposition covering the critical subgoals needed to judge task completion.

Verifier Agent. For each milestone $(t_j, d_j, h_j) \in \mathcal{M}_0$, the Verifier checks whether the assignment goal d_j is achieved at environment step t_j, given the pre-action state $s_{t_j}^{\text{pre}}$, post-action state $s_{t_j}^{\text{post}}$, executed action a_{t_j}, and metadata m_{t_j}. It outputs a binary result $v_j \in \{0, 1\}$ and grounded feedback f_j when $v_j = 0$. The feedback cites visual evidence from screenshots, such as a missing confirmation dialog or an incorrect UI target. The complete verification output is denoted as $\mathcal{V}_0 = \{(v_j, f_j)\}_{j=1}^{K}$.

3.3 VERDICT CALIBRATION MODULE

While the MVM structures trajectory evaluation, the initial milestone set $\mathcal{M}_0$ may be incomplete or insufficiently rigorous. The Selector may miss critical verification points, use overly lenient success criteria, or overlook subtle failure modes, leading to false positives. In RL, such false positives are especially harmful because they reinforce incorrect behaviors. As shown in Appendix A, when recall is already sufficient, trading a modest recall drop for higher precision improves gradient quality in policy-gradient RL. We therefore introduce the **Verdict Calibration Module (VCM)**, which iteratively refines milestones through critical review and produces the final reward from the full deliberation process. The VCM comprises two agents:

Reviewer Agent. Given the initial milestone set $\mathcal{M}_0$ and verification results $\mathcal{V}_0$ from the MVM, the Reviewer audits whether $\mathcal{M}_0$ is *complete* (covers all necessary subgoals) and *appropriate* (each milestone has clear, verifiable success criteria). The Reviewer is strictly evidence-grounded: every raised issue must be supported by observable trajectory signals. Typical concerns include: (i) *missing critical milestones* that leave requirements unverified, (ii) *overly lenient criteria* that accept intermediate progress without confirming end-state correctness, (iii) *uncaptured failure modes* such as goal-conflicting actions, and (iv) *weak verification evidence* that relies on action descriptions rather

than visual confirmation. When issues are identified, the Reviewer generates structured feedback $\mathcal{F} = \{(u_j, q_j)\}_{j=1}^{n}$, where u_j describes the concern and q_j provides a targeted query. This feedback prompts the MVM to refine $\mathcal{M}_0 \to \mathcal{M}_1$ and re-verify relevant steps; the process continues until the Reviewer confirms the decomposition is sound, yielding the final $\mathcal{M}$ and $\mathcal{V}$.

Judge Agent. Once refinement converges or reaches the iteration limit, the Judge synthesizes all evidence to produce the binary reward $r \in \{0, 1\}$. Rather than simply aggregating milestone verification results, it considers the deliberation history: (i) milestone evolution $\{\mathcal{M}_0, \ldots, \mathcal{M}\}$, (ii) verification outcomes $\mathcal{V}$, (iii) review feedback and resolution $\mathcal{F}$, and (iv) the original instruction $\mathcal{I}$ and trajectory τ. This supports decisions beyond surface-level verification; if revisions were needed or results remain borderline, the Judge weighs these signals conservatively. Formally:

$$r = \mathcal{J}\left(\mathcal{I}, \tau, \{\mathcal{M}_0, \ldots, \mathcal{M}\}, \mathcal{V}, \mathcal{F}\right). \tag{1}$$

By leveraging the full refinement process rather than only final milestone outcomes, it reduces false positives while maintaining high recall.

3.4 Milestone-Credit Group Relative Policy Optimization

Standard outcome-based GRPO assigns the same trajectory-level advantage to all trainable tokens within a rollout. While this preserves the global preference between successful and failed trajectories, it provides coarse credit for long-horizon GUI tasks: a trajectory may contain substantial correct progress followed by a single decisive failure, yet all actions are reinforced or suppressed uniformly.

OS-Themis naturally exposes the structure needed to address this limitation. The MVM identifies task-critical milestones and verifies their local outcomes, while the Judge provides the final trajectory outcome. We therefore introduce **Milestone-Credit Group Relative Policy Optimization (MC-GRPO)**, which uses the original GRPO trajectory advantage as the basis for credit redistribution guided by verified milestone outcomes. The trajectory-level advantage determines the sign of token-level credit, while verified milestones determine its relative magnitude within the trajectory.

Milestone-Aware Credit. For trajectory τ_i with terminal outcome $r_i \in \{0, 1\}$ and GRPO advantage A_i, let K_i be the number of verified milestones and $v_{ij} \in \{0, 1\}$ the verdict of the j-th milestone. We define the mean milestone success rate p_i and the outcome-conditioned credit multiplier g_{ij} as

$$p_i = \frac{1}{K_i} \sum_{j=1}^{K_i} v_{ij}, \qquad g_{ij} = 1 + |r_i - p_i|(2r_i - 1)(v_{ij} - p_i). \tag{2}$$

The multiplier emphasizes milestones whose verified outcomes align with the terminal result, especially when such outcomes are rare relative to the trajectory's overall progress. Milestones inconsistent with the terminal result are correspondingly downweighted. These positive multipliers adjust credit magnitude while preserving the sign of the original trajectory-level advantage.

Advantage Redistribution. Let $\phi_i(\ell) = j$ map trainable token ℓ to the action segment associated with milestone M_{ij}, and $\phi_i(\ell) = \varnothing$ otherwise. We assign local factors and normalize them over all trainable tokens $\mathcal{T}_i$:

$$\kappa_{i,\ell} = \begin{cases} g_{i,\phi_i(\ell)}, & \phi_i(\ell) \neq \varnothing, \\ 1, & \text{otherwise}, \end{cases} \qquad \widetilde{A}_{i,\ell} = A_i \frac{|\mathcal{T}_i| \kappa_{i,\ell}}{\sum_{u \in \mathcal{T}_i} \kappa_{i,u}}. \tag{3}$$

This preserves the sign and total signed advantage within each trajectory.

Normalizing local factors over all trainable tokens $\mathcal{T}_i$ preserves the trajectory's total signed advantage:

$$\sum_{\ell \in \mathcal{T}_i} \widetilde{A}_{i,\ell} = A_i |\mathcal{T}_i|, \tag{4}$$

while all scaling factors remain positive, so $\operatorname{sign}(\widetilde{A}_{i,\ell}) = \operatorname{sign}(A_i)$.

MC-GRPO thus uses the same verified milestone evidence for both outcome evaluation and fine-grained intra-trajectory credit assignment. This reuse requires neither additional reward models nor extra evaluator calls. In the absence of valid milestone evidence ($K_i = 0$), $\kappa_{i,\ell} = 1$ for all trainable tokens, and MC-GRPO reduces exactly to standard outcome-based GRPO.

Table 1: Performance comparison of different models under the DigiRLBai et al. (2024), ZeroGUIYang et al. (2025a) and OS-Themis frameworks on OGRBench. Each framework reports Accuracy (Acc) and F1-score (F1), while the Overall performance includes Acc, Precision (Prec), Recall (Rec), and F1. Higher values indicate better performance. **Bold** and underlined denote the best and second-best results, respectively.

Model	Ubuntu		Mobile		Windows		macOS		Web		Overall			
	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	Prec	Rec	F1
DigiRL														
Qwen3-VL-8B	62.1	64.9	69.1	76.6	66.2	63.6	72.7	40.0	63.2	69.6	64.4	62.2	72.1	66.8
Qwen3-VL-30B-A3B	61.0	67.6	67.0	75.6	69.0	71.1	67.5	49.0	54.2	63.0	62.5	58.9	80.6	68.1
Qwen3-VL-32B	65.7	66.5	72.3	77.8	71.8	71.2	75.3	55.8	62.1	67.9	67.6	65.8	72.1	68.8
Qwen3-VL-235B	65.5	67.0	70.2	75.2	72.3	70.9	76.6	43.8	63.2	62.8	67.4	66.6	69.0	67.8
Qwen3-VL-235B-Thinking	62.3	66.3	67.6	74.7	65.7	66.0	70.1	46.5	58.4	62.2	63.4	61.1	73.0	66.5
GPT-5-mini	47.0	0.0	47.9	0.0	55.9	0.0	79.2	0.0	47.9	0.0	50.3	0.0	0.0	0.0
GPT-5	50.5	13.6	56.9	33.1	59.2	17.1	79.2	0.0	52.6	16.7	54.5	89.3	9.6	17.3
Gemini-3-Flash	69.5	63.0	69.7	64.2	76.5	66.2	83.1	31.6	75.3	74.3	72.1	86.6	51.9	64.9
Mean	60.4	51.1	65.1	59.6	67.1	53.3	75.5	33.3	59.6	52.0	62.8	61.3	53.5	52.5
ZeroGUI														
Qwen3-VL-8B	72.7	67.8	78.7	79.6	76.5	67.5	89.6	69.2	75.8	75.0	75.4	86.3	60.1	70.9
Qwen3-VL-30B-A3B	74.8	74.6	80.3	82.8	80.8	77.3	90.9	75.9	72.1	75.1	76.9	77.9	74.9	76.3
Qwen3-VL-32B	75.0	72.2	78.7	80.0	79.3	72.5	89.6	66.7	76.3	75.9	77.1	85.3	65.3	73.9
Qwen3-VL-235B	76.5	74.5	82.4	84.2	77.9	70.8	93.5	81.5	84.7	85.6	79.6	85.5	70.9	77.5
Qwen3-VL-235B-Thinking	71.8	66.9	79.8	82.1	79.3	72.5	89.6	66.7	78.9	78.3	75.9	85.3	62.3	72.0
GPT-5-mini	48.4	6.4	49.5	5.9	56.3	2.1	79.2	0.0	48.9	5.8	51.5	87.0	2.9	5.5
GPT-5	72.6	67.0	81.9	82.3	77.5	68.8	90.9	72.0	80.5	79.6	76.7	89.7	59.9	71.8
Gemini-3-Flash	76.1	72.8	77.1	76.0	80.8	74.5	93.5	81.5	77.4	75.1	78.1	89.8	63.0	74.1
Mean	71.0	62.8	76.1	71.6	76.1	63.3	89.6	64.2	74.3	68.8	73.9	85.8	57.4	65.3
OS-Themis														
Qwen3-VL-8B	77.2	75.8	85.6	85.1	72.8	61.8	85.7	59.3	86.3	87.3	79.3	86.3	69.4	77.0
Qwen3-VL-30B-A3B	79.5	79.4	84.6	85.0	76.5	70.9	88.3	66.7	80.5	79.3	80.3	84.7	73.7	78.8
Qwen3-VL-32B	77.6	75.1	83.5	82.1	75.1	63.4	88.3	69.0	84.7	83.2	79.6	92.2	64.3	75.8
Qwen3-VL-235B	**88.1**	**88.3**	**92.6**	**92.8**	77.5	68.4	**94.8**	**86.7**	**92.1**	**92.1**	**88.0**	92.8	**82.3**	**87.2**
Qwen3-VL-235B-Thinking	83.4	83.7	89.4	89.6	85.4	82.5	93.5	82.8	84.7	83.4	85.2	89.3	79.9	84.3
GPT-5-mini	68.8	59.7	65.4	52.6	76.5	64.3	87.0	54.5	75.8	70.5	71.5	**95.4**	44.7	60.9
GPT-5	82.5	81.3	80.3	78.4	84.5	80.5	88.3	60.9	83.2	81.4	82.9	93.4	70.6	80.4
Gemini-3-Flash	85.0	84.7	91.0	91.1	**86.9**	**83.7**	93.5	82.8	82.6	80.5	86.2	93.2	78.0	84.9
Mean	80.3	78.5	84.0	82.1	79.4	72.0	89.9	70.3	83.8	82.2	81.6	90.9	70.4	78.7

4 EXPERIMENTS

4.1 OMNIGUIREWARDBENCH

Benchmark Construction. We introduce OmniGUIRewardBench (OGRBench), a multi-platform benchmark for GUI outcome reward models. It contains 1,409 trajectories from AndroidWorld (Rawles et al., 2024), OSWorld (Xie et al., 2024), WindowsAgentArena (Bonatti et al., 2024), macOSArena (Wang et al., 2025e), and WebArena-Lite-v2 (Wang et al., 2025e), spanning mobile, desktop, and web. Generated by diverse GUI agents, each trajectory includes screenshots, agent outputs, and a binary task-success label from the source benchmark's built-in evaluator. We manually audit potentially unreliable labels and use stratified sampling to maintain task coverage and a near-balanced distribution of 700 positive and 709 negative examples. See Appendix G for details.

Main Results. We evaluate OS-Themis on OGRBench against two representative baselines, DigiRL (Bai et al., 2024) and ZeroGUI (Yang et al., 2025a), which reflect two common LLM-as-a-Judge paradigms for GUI reward modeling. Existing LLM-as-a-Judge rewards in GUI online RL largely follow or extend these two designs, making them suitable for representative comparison. ZeroGUI adopts a final-state assessment paradigm by feeding terminal GUI observations, such as screenshots or structural page information, directly into the model for judgment. We follow this setting for ZeroGUI. DigiRL instead follows a sequential verification paradigm, iteratively evaluating states until the goal is met or the trajectory ends. Table 1 reports accuracy across platforms, including precision

and recall for finer comparison. Across all tested base models, OS-Themis consistently achieves superior accuracy and precision. On average, it outperforms DigiRL by 18.8% in accuracy, 29.6% in precision, 16.9% in recall, and 26.2% in F1-score. It also surpasses ZeroGUI by 7.7%, 5.1%, 13.0%, and 13.4% on the same metrics. See Appendix B and C for baseline rationale and detailed results.

4.2 Online RL Experiments

4.2.1 Android Online RL Infrastructure

We develop an Android online RL infrastructure that turns parallel GUI interactions into policy training data through a complete cycle of task scheduling, trajectory collection, reward evaluation, and policy optimization. The architecture separates a pool of Dockerized Android environments from the distributed training stack. Ray Moritz et al. (2018) coordinates training components, while vLLM Kwon et al. (2023) serves policy rollouts, allowing environment capacity and model computation to be configured separately.Further details are provided in Appendix H.

Parallel Environment Interaction. A controller dispatches tasks to distributed Android workers, each running an isolated session with task-specific initialization. Through a unified GUI interface, the policy generates actions from screenshots and interaction history, while workers execute them and collect observations, enabling parallel trajectory collection across the emulator pool.

Structured Trajectory Processing. The pipeline packages model responses, executed actions, and screenshots into complete trajectories linked to terminal judgments, milestone annotations, and training segments. This unified representation carries visual evidence from environment execution to reward evaluation and policy supervision, supporting both outcome-based GRPO and MC-GRPO.

Reward Evaluation and Policy Optimization. Completed trajectories are evaluated asynchronously while other workers continue sampling. The trainer groups independent rollouts by task, computes trajectory-level advantages from terminal rewards, and applies milestone-guided credit redistribution for MC-GRPO. Updated policy weights are then synchronized to rollout services. A shared reward interface supports different evaluators, while periodic validation, training metrics, and checkpointing monitor policy evolution.

4.2.2 Reward Comparison

Training Setup. We implement multi-turn online RL using GRPO (Shao et al., 2024; Guo et al., 2025a) within the Verl framework (Sheng et al., 2024). Further details are provided in Appendix H.

Comparisons. To verify the framework's effectiveness across different scales, we fine-tune two policy backbones: Qwen3-VL-4B and Qwen3-VL-8B. Regarding OS-Themis, we instantiate it with two backbone options: Qwen3-VL-8B and Qwen3-VL-235B. We benchmark our approach against two external baselines under identical training configurations: SEAgent (Sun et al., 2025b), an open-source critic model, and ZeroGUI, an LLM-as-a-Judge method using Qwen3-VL-235B. To examine credit assignment, we compare MC-GRPO with GRPO and ADMIRE (Zheng et al., 2026), using OS-Themis with Qwen3-VL-235B as the shared outcome evaluator under the same training setup. For ADMIRE, we use the same backbone for milestone extraction and otherwise follow its original method, which derives milestones from successful trajectories and uses milestone progress as an extra reward. In contrast, MC-GRPO identifies finer milestones from each trajectory through visual verification and uses them to redistribute the original GRPO advantage, without requiring successful trajectories or external task success signals.We evaluate each trained model three times and report the mean accuracy along with the half-range.

Results on AndroidWorld. We evaluate OS-Themis in online RL on AndroidWorld, as reported in Table 2. For the Qwen3-VL-4B backbone, using OS-Themis with MC-GRPO yields an 8.9% absolute improvement over the baseline, outperforming ZeroGUI and SEAgent by 7.8% and 7.8%, respectively. The advantage becomes more evident with Qwen3-VL-8B: fine-tuning with OS-Themis and MC-GRPO brings a 9.4% gain over the baseline and exceeds ZeroGUI and SEAgent by 4.8% and 7.1%. Compared with uniform credit assignment, MC-GRPO further improves performance by 2.9% and 2.5% on the two backbones, while outperforming ADMIRE by 3.2% and 2.8%, respectively.

Table 2: RL performance on AndroidWorld with different reward sources and optimization methods. Each row is trained independently from the same initialization for the given backbone. RL results report mean accuracy (%) ± half-range over three evaluation runs.

Backbone	Reward used for RL training	Method	Accuracy
Qwen3-VL-4B	–	–	45.3
	SEAgent	GRPO	46.4 ± 1.1
	ZeroGUI (Qwen3-VL-235B)	GRPO	46.4 ± 0.9
	OS-Themis (Qwen3-VL-8B)	GRPO	50.7 ± 1.1
	OS-Themis (Qwen3-VL-235B)	GRPO	51.3 ± 1.3
	OS-Themis (Qwen3-VL-235B)	ADMIRE	51.0 ± 2.4
	OS-Themis (Qwen3-VL-235B)	**MC-GRPO**	$\mathbf{54.2 \pm 0.9}$
Qwen3-VL-8B	–	–	47.6
	SEAgent	GRPO	49.9 ± 1.5
	ZeroGUI (Qwen3-VL-235B)	GRPO	52.2 ± 1.1
	OS-Themis (Qwen3-VL-8B)	GRPO	53.2 ± 1.1
	OS-Themis (Qwen3-VL-235B)	GRPO	54.5 ± 0.9
	OS-Themis (Qwen3-VL-235B)	ADMIRE	54.2 ± 1.3
	OS-Themis (Qwen3-VL-235B)	**MC-GRPO**	$\mathbf{57.0 \pm 0.6}$

Self-Evolution via Reward Filtering. Beyond online RL, we compare these reward sources in a self-evolution setting using an automated Android trajectory-collection pipeline. The pipeline generates exploration tasks from emulator states, executes them in Dockerized Android environments, logs trajectories, and filters them with DigiRL, ZeroGUI, or OS-Themis before SFT. From 15,110 trajectories, OS-Themis-filtered data yields 6.9% and 5.0% gains on Qwen3-VL-4B and Qwen3-VL-8B, outperforming other filters and unfiltered data. This shows that OS-Themis supports training-time reward assignment and scalable curation for self-evolving GUI agents. Details are in Appendix D.

4.3 Scaling Exploration

Scaling Training Setup. To evaluate the effectiveness of OS-Themis in a realistic online RL scaling setting, we conduct a pilot study on Qwen3-VL-4B. Within the OS-Themis framework, we use Qwen3-VL-235B to score trajectories and apply MC-GRPO for milestone-aware credit redistribution during GRPO. Following common practice in agent data synthesis (Lai et al., 2025a), we use Qwen3-VL-235B to generate a large pool of task templates, filter suitable ones, and instantiate 1,024 training tasks.

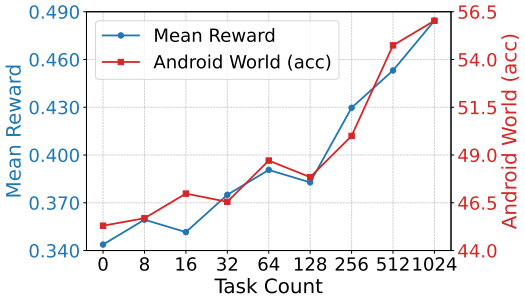


Figure 3: The performance of Qwen3-VL-4B under online RL scaling with OS-Themis, including mean reward growth and corresponding AndroidWorld accuracy across different training scales.

Scaling Results. For evaluation, we additionally design 128 extra tasks as a validation set, and conduct staged validation when the training task count reaches (0, 8, 16, 32, 64, 128, 256, 512, 1024) to characterize the performance trend as the task scale increases. Meanwhile, we evaluate each intermediate model on the AndroidWorld benchmark under the same protocol. As shown in Figure 3, after scaling training to 1024 tasks, Qwen3-VL-4B achieves a score of 56.0% on AndroidWorld, improving by 10.7% over the Baseline. These results demonstrate clear gains and highlight the strong potential of OS-Themis for scalable online RL.

More Scaling Studies. In Appendix E, we conduct a suite of scaling experiments to analyze the scalability of OS-Themis and its behavior across different usage scenarios. The experiments fall into three categories: (1) Model scaling for individual agents. (2) Framework-level test-time scaling via voting. (3) Test-time scaling for evaluation.

4.4 Ablation Study

Agent-level Ablation. We first evaluate each agent's contribution by removing components from OS-Themis, as shown in Block (a) of Table 3. Removing the **Selector Agent** disables milestone

selection and instead forwards every trajectory step to the Verifier. Although stepwise verification provides denser supervision, it introduces substantial verification overhead and a large amount of weakly relevant evidence. Consequently, critical failure signals are diluted by many correct but nonessential actions, causing Accuracy to drop by 4.7% and Precision by 13.1%. This confirms that the Selector is crucial for mitigating evidence dilution and improving evidence aggregation quality.

Removing the **Verifier Agent** skips explicit milestone verification and treats all selected milestones as correctly executed by default. This introduces systematic bias into the evidence chain, leading to a 6.1% drop in Accuracy and a 15.6% drop in Precision. These results demonstrate that the Verifier is necessary for grounding milestone judgments in observable GUI evidence. Finally, removing the **Judge Agent** makes the system determine trajectory success solely based on whether all intermediate milestones are verified as correct. This variant becomes overly conservative: although Precision remains high at 89.7%, Recall collapses to 5.0%, and Accuracy drops to 52.5%. This indicates that imperfect intermediate operations do not necessarily prevent final task completion, and a dedicated Judge is essential for holistic outcome-level reasoning beyond step-level correctness.

Impact of the Reviewer Agent. Block (b) of Table 3 analyzes the role of the **Reviewer Agent**. Using Qwen3-VL-235B as the backbone evaluator, we compare OS-Themis without the Reviewer against two Reviewer configurations: *Advisor* and *Critic*. As an *Advisor*, the Reviewer provides suggestions from the interaction trace; as a *Critic*, it audits the evidence chain to identify overlooked flaws and evidence-grounded failure signals. Introducing the Reviewer keeps Accuracy stable while improving Precision, suggesting that it enhances the diversity and coverage of extracted evidence. We adopt the *Critic* setting because policy-gradient RL benefits more from high-precision reward signals. By reducing false positives while maintaining high Recall, the Critic provides reliable supervision for policy optimization.

Table 3: Ablation study of OS-Themis on OGRBench. Blocks (a)–(c) evaluate agent removal, Reviewer roles, and assignment-goal design.

Variant	Acc	Prec	Rec
(a) Individual agent ablation			
OS-Themis w/o Selector	83.3	79.7	88.9
OS-Themis w/o Verifier	81.9	77.2	**90.1**
OS-Themis w/o Judge	52.5	89.7	5.0
OS-Themis (Full)	**88.0**	**92.8**	82.3
(b) Reviewer role ablation			
OS-Themis w/o Reviewer	86.9	85.7	**88.4**
OS-Themis w/ Reviewer (Advisor)	87.8	88.7	86.4
OS-Themis w/ Reviewer (Critic)	**88.0**	**92.8**	82.3
(c) Assignment goal ablation			
OS-Themis w/o Assignment Goal	86.9	84.6	**90.0**
OS-Themis w/ Assignment Goal	**88.0**	**92.8**	82.3

Effectiveness of the Assignment Goal. Block (c) of Table 3 studies the effect of the *assignment goal*. When the Selector proposes a milestone, it also generates an explicit assignment goal specifying what the Verifier should check. Without this goal, the Verifier tends to make overly lenient or ambiguous judgments, approving some incorrect trajectories and reducing Precision to 84.6%. With assignment goals, the Verifier receives a concrete, observable validation criterion, which significantly improves Precision to 92.8% and raises Accuracy to 88.0%. This shows that assignment goals are important for grounding milestone verification and improving the reliability of reward estimation.

5 Conclusion

We propose **OS-Themis**, a scalable critic framework for generalist GUI rewards. By grounding outcome judgments in verified milestones, OS-Themis turns trajectory evidence into reliable reward signals, while **MC-GRPO** extends their use to fine-grained credit assignment. Results on OGRBench, Android online RL, and trajectory filtering demonstrate the value of this structured approach for reward evaluation and agent learning.Within available computational and environment resources, we validate the framework in the evaluated settings. Its modular organization of milestone verification, outcome judgment, and credit assignment provides room for further exploration. Future work will investigate systematic scaling across task volumes and training budgets, online training on more GUI platforms, and richer milestone-level rewards and reward composition strategies. More broadly, OS-Themis connects the assessment of GUI behavior with the learning signals that guide its improvement. This connection offers a path toward scalable, generalist GUI rewards that help agents learn from their own interaction experience and continually improve in diverse environments.

AI USE STATEMENT

We used generative AI tools to polish the manuscript for language, clarity, and readability, and to support research execution through code development and our LLM-based OS-Themis framework. We also used AI agents to generate trajectories for constructing OGRBench. The authors take full responsibility for the final manuscript and code, as well as the validity of the experimental results and conclusions.

ETHICS STATEMENT

We study GUI reward modeling and online reinforcement learning, with online training conducted in isolated Android emulator environments. The benchmarks, models, and software used in this work are subject to their respective licenses and terms of use, which we follow for research evaluation and training. This research does not involve real users' private data or sensitive information. We manually audit and filter GUI trajectory artifacts to exclude samples containing potentially identifying, sensitive, or offensive content.

OS-Themis and OGRBench are intended to support research on reliable GUI agents. Applications beyond the evaluated settings should include task-specific validation, appropriate human oversight, and safeguards for sensitive actions. Any future applications involving screenshots from real users should remove personal and sensitive information before model inference and strictly restrict access to interaction records.

REPRODUCIBILITY STATEMENT

We will publicly release OmniGUIRewardBench (OGRBench) and open-source the OS-Themis code to support reproducibility and further research. The paper describes the proposed method and experimental settings, with additional benchmark details in Appendix G and training configurations, hyperparameters, and implementation details in Appendix H. The prompts used by OS-Themis are also provided in Appendix K.

REFERENCES

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent s: An open agentic framework that uses computers like a human. *arXiv preprint arXiv:2410.08164*, 2024.

Anthropic. Developing a computer use model, 2024. URL https://www.anthropic.com/news/developing-computer-use. Product announcement.

Anthropic. Claude 3.7 sonnet system card. https://www.anthropic.com/claude-3-7-sonnet-system-card, February 2025a. System card.

Anthropic. Introducing Claude Sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, September 2025b. Published: 2025-09-29; Accessed: 2025-10-27.

Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning, 2024. URL https://arxiv.org/abs/2406.11896.

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

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.

Cong Chen, Kaixiang Ji, Hao Zhong, Muzhi Zhu, Anzhou Li, Guo Gan, Ziyuan Huang, Cheng Zou, Jiajia Liu, Jingdong Chen, Hao Chen, and Chunhua Shen. Gui-shepherd: Reliable process reward and verification for long-sequence gui tasks, 2025. URL https://arxiv.org/abs/2509.23738.

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, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.505. URL https://aclanthology.org/2024.acl-long.505/.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Hang Ding, Peidong Liu, Junqiao Wang, Ziwei Ji, Meng Cao, Rongzhao Zhang, Lynn Ai, Eric Yang, Tianyu Shi, and Lei Yu. Dynaweb: Model-based reinforcement learning of web agents, 2026. URL https://arxiv.org/abs/2601.22149.

Jihao Gu, Qihang Ai, Yingyao Wang, Pi Bu, Jingxuan Xing, Zekun Zhu, Wei Jiang, Ziming Wang, Yingxiu Zhao, Ming-Liang Zhang, Jun Song, Yuning Jiang, and Bo Zheng. Mobile-r1: Towards interactive capability for vlm-based mobile agent via systematic training, 2026. URL https://arxiv.org/abs/2506.20332.

Zhangxuan Gu, Zhengwen Zeng, Zhenyu Xu, Xingran Zhou, Shuheng Shen, Yunfei Liu, Beitong Zhou, Changhua Meng, Tianyu Xia, Weizhi Chen, et al. Ui-venus technical report: Building high-performance ui agents with rft. *arXiv preprint arXiv:2508.10833*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025a.

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*, 2025b.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Hanyu Lai, Junjie Gao, Xiao Liu, Yifan Xu, Shudan Zhang, Yuxiao Dong, and Jie Tang. Androidgen: Building an android language agent under data scarcity, 2025a. URL https://arxiv.org/abs/2504.19298.

Hanyu Lai, Xiao Liu, Yanxiao Zhao, Han Xu, Hanchen Zhang, Bohao Jing, Yanyu Ren, Shuntian Yao, Yuxiao Dong, and Jie Tang. Computerrl: Scaling end-to-end online reinforcement learning for computer use agents. *arXiv preprint arXiv:2508.14040*, 2025b.

Seongyun Lee, Seungone Kim, Sue Park, Geewook Kim, and Minjoon Seo. Prometheus-vision: Vision-language model as a judge for fine-grained evaluation. In *Findings of the association for computational linguistics ACL 2024*, pp. 11286–11315, 2024.

Haitao Li, Qian Dong, Junjie Chen, Huixue Su, Yujia Zhou, Qingyao Ai, Ziyi Ye, and Yiqun Liu. Llms-as-judges: a comprehensive survey on llm-based evaluation methods. *arXiv preprint arXiv:2412.05579*, 2024.

Pengxiang Li, Zechen Hu, Zirui Shang, Jingrong Wu, Yang Liu, Hui Liu, Zhi Gao, Chenrui Shi, Bofei Zhang, Zihao Zhang, Xiaochuan Shi, Zedong YU, Yuwei Wu, Xinxiao Wu, Yunde Jia, Liuyu Xiang, Zhaofeng He, and Qing Li. Efficient multi-turn rl for gui agents via decoupled training and adaptive data curation, 2025. URL https://arxiv.org/abs/2509.23866.

Haojia Lin, Xiaoyu Tan, Yulei Qin, Zihan Xu, Yuchen Shi, Zongyi Li, Gang Li, Shaofei Cai, Siqi Cai, Chaoyou Fu, et al. Cuarewardbench: A benchmark for evaluating reward models on computer-using agent. *arXiv preprint arXiv:2510.18596*, 2025.

Zhaoyang Liu, Jingjing Xie, Zichen Ding, Zehao Li, Bowen Yang, Zhenyu Wu, Xuehui Wang, Qiushi Sun, Shi Liu, Weiyun Wang, Shenglong Ye, Qingyun Li, Xuan Dong, Yue Yu, Chenyu Lu, YunXiang Mo, Yao Yan, Zeyue Tian, Xiao Zhang, Yuan Huang, Yiqian Liu, Weijie Su, Gen Luo, Xiangyu Yue, Biqing Qi, Kai Chen, Bowen Zhou, Yu Qiao, Qifeng Chen, and Wenhai Wang. Scalecua: Scaling open-source computer use agents with cross-platform data, 2025. URL https://arxiv.org/abs/2509.15221.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

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*, 2025a.

Zhengxi Lu, Jiabo Ye, Fei Tang, Yongliang Shen, Haiyang Xu, Ziwei Zheng, Weiming Lu, Ming Yan, Fei Huang, Jun Xiao, et al. Ui-s1: Advancing gui automation via semi-online reinforcement learning. *arXiv preprint arXiv:2509.11543*, 2025b.

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

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pp. 561–577, 2018.

Shikhar Murty, Hao Zhu, Dzmitry Bahdanau, and Christopher D. Manning. Nnetnav: Unsupervised learning of browser agents through environment interaction in the wild, 2025. URL https://arxiv.org/abs/2410.02907.

Dang Nguyen, Jian Chen, Yu Wang, Gang Wu, Namyong Park, Zhengmian Hu, Hanjia Lyu, Junda Wu, Ryan Aponte, Yu Xia, et al. Gui agents: A survey. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 22522–22538, 2025.

OpenAI. Computer-using agent: Introducing a universal interface for ai to interact with the digital world, 2025. URL https://openai.com/index/computer-using-agent.

OpenAI. Introducing GPT-5. https://openai.com/index/introducing-gpt-5, 2025. Accessed: 2025-10-27.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Wenyi Zhao, Yu Yang, Xinyue Yang, Jiadai Sun, Shuntian Yao, et al. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. *arXiv preprint arXiv:2411.02337*, 2024.

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.

Christopher 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. *arXiv preprint arXiv:2405.14573*, 2024.

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.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv:2409.19256*, 2024.

Yucheng Shi, Wenhao Yu, Zaitang Li, Yonglin Wang, Hongming Zhang, Ninghao Liu, Haitao Mi, and Dong Yu. Mobilegui-rl: Advancing mobile gui agent through reinforcement learning in online environment, 2025. URL https://arxiv.org/abs/2507.05720.

Qiushi Sun, Kanzhi Cheng, Zichen Ding, Chuanyang Jin, Yian Wang, Fangzhi Xu, Zhenyu Wu, Chengyou Jia, Liheng Chen, Zhoumianze Liu, et al. Os-genesis: Automating gui agent trajectory construction via reverse task synthesis. *arXiv preprint arXiv:2412.19723*, 2024.

Qiushi Sun, Mukai Li, Zhoumianze Liu, Zhihui Xie, Fangzhi Xu, Zhangyue Yin, Kanzhi Cheng, Zehao Li, Zichen Ding, Qi Liu, et al. Os-sentinel: Towards safety-enhanced mobile gui agents via hybrid validation in realistic workflows. *arXiv preprint arXiv:2510.24411*, 2025a.

Zeyi Sun, Ziyu Liu, Yuhang Zang, Yuhang Cao, Xiaoyi Dong, Tong Wu, Dahua Lin, and Jiaqi Wang. Seagent: Self-evolving computer use agent with autonomous learning from experience. *arXiv preprint arXiv:2508.04700*, 2025b.

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*, 2025a.

Shuai Wang, Weiwen Liu, Jingxuan Chen, Yuqi Zhou, Weinan Gan, Xingshan Zeng, Yuhan Che, Shuai Yu, Xinlong Hao, Kun Shao, et al. Gui agents with foundation models: A comprehensive survey. *arXiv preprint arXiv:2411.04890*, 2024.

Taiyi Wang, Zhihao Wu, Jianheng Liu, Jianye Hao, Jun Wang, and Kun Shao. Distrl: An asynchronous distributed reinforcement learning framework for on-device control agents, 2025b. URL https://arxiv.org/abs/2410.14803.

Weiyun Wang, Zhangwei Gao, Lixin Gu, Hengjun Pu, Long Cui, Xingguang Wei, Zhaoyang Liu, Linglin Jing, Shenglong Ye, Jie Shao, Zhaokai Wang, Zhe Chen, Hongjie Zhang, Ganlin Yang, Haomin Wang, Qi Wei, Jinhui Yin, Wenhao Li, Erfei Cui, Guanzhou Chen, Zichen Ding, Changyao Tian, Zhenyu Wu, Jingjing Xie, Zehao Li, Bowen Yang, Yuchen Duan, Xuehui Wang, Zhi Hou, Haoran Hao, Tianyi Zhang, Songze Li, Xiangyu Zhao, Haodong Duan, Nianchen Deng, Bin Fu, Yinan He, Yi Wang, Conghui He, Botian Shi, Junjun He, Yingtong Xiong, Han Lv, Lijun Wu, Wenqi Shao, Kaipeng Zhang, Huipeng Deng, Biqing Qi, Jiaye Ge, Qipeng Guo, Wenwei Zhang, Songyang Zhang, Maosong Cao, Junyao Lin, Kexian Tang, Jianfei Gao, Haian Huang, Yuzhe Gu, Chengqi Lyu, Huanze Tang, Rui Wang, Haijun Lv, Wanli Ouyang, Limin Wang, Min Dou, Xizhou Zhu, Tong Lu, Dahua Lin, Jifeng Dai, Weijie Su, Bowen Zhou, Kai Chen, Yu Qiao, Wenhai Wang, and Gen Luo. Internvl3.5: Advancing open-source multimodal models in versatility, reasoning, and efficiency, 2025c. URL https://arxiv.org/abs/2508.18265.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, Zhennan Shen, Zhuokai Li, Ryan Li, Xiaochuan Li, Junda Chen, Boyuan Zheng, Peihang Li, Fangyu Lei, Ruisheng Cao, Yeqiao Fu, Dongchan Shin, Martin Shin, Jiarui Hu, Yuyan Wang, Jixuan Chen, Yuxiao Ye, Danyang Zhang, Dikang Du, Hao Hu, Huarong Chen, Zaida Zhou, Haotian Yao, Ziwei Chen, Qizheng Gu, Yipu Wang, Heng Wang, Diyi Yang, Victor Zhong, Flood Sung, Y. Charles, Zhilin Yang, and Tao Yu. Opencua: Open foundations for computer-use agents, 2025d. URL https://arxiv.org/abs/2508.09123.

Xuehui Wang, Zhenyu Wu, JingJing Xie, Zichen Ding, Bowen Yang, Zehao Li, Zhaoyang Liu, Qingyun Li, Xuan Dong, Zhe Chen, et al. Mmbench-gui: Hierarchical multi-platform evaluation framework for gui agents. *arXiv preprint arXiv:2507.19478*, 2025e.

Yinjie Wang, Tianbao Xie, Ke Shen, Mengdi Wang, and Ling Yang. Rlanything: Forge environment, policy, and reward model in completely dynamic rl system, 2026. URL https://arxiv.org/abs/2602.02488.

Yuyang Wanyan, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Jiabo Ye, Yutong Kou, Ming Yan, Fei Huang, Xiaoshan Yang, et al. Look before you leap: A gui-critic-r1 model for pre-operative error diagnosis in gui automation. *arXiv preprint arXiv:2506.04614*, 2025.

Zhepei Wei, Wenlin Yao, Yao Liu, Weizhi Zhang, Qin Lu, Liang Qiu, Changlong Yu, Puyang Xu, Chao Zhang, Bing Yin, Hyokun Yun, and Lihong Li. Webagent-r1: Training web agents via end-to-end multi-turn reinforcement learning, 2025. URL https://arxiv.org/abs/2505.16421.

Qianhui Wu, Kanzhi Cheng, Rui Yang, Chaoyun Zhang, Jianwei Yang, Huiqiang Jiang, Jian Mu, Baolin Peng, Bo Qiao, Reuben Tan, et al. Gui-actor: Coordinate-free visual grounding for gui agents. *arXiv preprint arXiv:2506.03143*, 2025a.

Zhenyu Wu, Jingjing Xie, Zehao Li, Bowen Yang, Qiushi Sun, Zhaoyang Liu, Zhoumianze Liu, Yu Qiao, Xiangyu Yue, Zun Wang, et al. Os-oracle: A comprehensive framework for cross-platform gui critic models. *arXiv preprint arXiv:2512.16295*, 2025b.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. Os-copilot: Towards generalist computer agents with self-improvement. *arXiv preprint arXiv:2402.07456*, 2024a.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, et al. Os-atlas: A foundation action model for generalist gui agents. *arXiv preprint arXiv:2410.23218*, 2024b.

Han Xiao, Guozhi Wang, Yuxiang Chai, Zimu Lu, Weifeng Lin, Hao He, Lue Fan, Liuyang Bian, Rui Hu, Liang Liu, et al. Ui-genie: A self-improving approach for iteratively boosting mllm-based mobile gui agents. *arXiv preprint arXiv:2505.21496*, 2025.

Han Xiao, Guozhi Wang, Hao Wang, Shilong Liu, Yuxiang Chai, Yue Pan, Yufeng Zhou, Xiaoxin Chen, Yafei Wen, and Hongsheng Li. Ui-mem: Self-evolving experience memory for online reinforcement learning in mobile gui agents, 2026. URL https://arxiv.org/abs/2602.05832.

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.

Tianbao Xie, Jiaqi Deng, Xiaochuan Li, Junlin Yang, Haoyuan Wu, Jixuan Chen, Wenjing Hu, Xinyuan Wang, Yuhui Xu, Zekun Wang, Yiheng Xu, Junli Wang, Doyen Sahoo, Tao Yu, and Caiming Xiong. Scaling computer-use grounding via user interface decomposition and synthesis, 2025. URL https://arxiv.org/abs/2505.13227.

Yifan Xu, Xiao Liu, Xinghan Liu, Jiaqi Fu, Hanchen Zhang, Bohao Jing, Shudan Zhang, Yuting Wang, Wenyi Zhao, and Yuxiao Dong. Mobilerl: Online agentic reinforcement learning for mobile gui agents. *arXiv preprint arXiv:2509.18119*, 2025.

Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous gui interaction. *arXiv preprint arXiv:2412.04454*, 2024.

Haolong Yan, Yeqing Shen, Xin Huang, Jia Wang, Kaijun Tan, Zhixuan Liang, Hongxin Li, Zheng Ge, Osamu Yoshie, Si Li, Xiangyu Zhang, and Daxin Jiang. Gui exploration lab: Enhancing screen navigation in agents via multi-turn reinforcement learning, 2025a. URL https://arxiv.org/abs/2512.02423.

Haolong Yan, Jia Wang, Xin Huang, Yeqing Shen, Ziyang Meng, Zhimin Fan, Kaijun Tan, Jin Gao, Lieyu Shi, Mi Yang, Shiliang Yang, Zhirui Wang, Brian Li, Kang An, Chenyang Li, Lei Lei, Mengmeng Duan, Danxun Liang, Guodong Liu, Hang Cheng, Hao Wu, Jie Dong, Junhao Huang, Mei Chen, Renjie Yu, Shunshan Li, Xu Zhou, Yiting Dai, Yineng Deng, Yingdan Liang,

Zelin Chen, Wen Sun, Chengxu Yan, Chunqin Xu, Dong Li, Fengqiong Xiao, Guanghao Fan, Guopeng Li, Guozhen Peng, Hongbing Li, Hang Li, Hongming Chen, Jingjing Xie, Jianyong Li, Jingyang Zhang, Jiaju Ren, Jiayu Yuan, Jianpeng Yin, Kai Cao, Liang Zhao, Liguo Tan, Liying Shi, Mengqiang Ren, Min Xu, Manjiao Liu, Mao Luo, Mingxin Wan, Na Wang, Nan Wu, Ning Wang, Peiyao Ma, Qingzhou Zhang, Qiao Wang, Qinlin Zeng, Qiong Gao, Qiongyao Li, Shangwu Zhong, Shuli Gao, Shaofan Liu, Shisi Gao, Shuang Luo, Xingbin Liu, Xiaojia Liu, Xiaojie Hou, Xin Liu, Xuanti Feng, Xuedan Cai, Xuan Wen, Xianwei Zhu, Xin Liang, Xin Liu, Xin Zhou, Yifan Sui, Yingxiu Zhao, Yukang Shi, Yunfang Xu, Yuqing Zeng, Yixun Zhang, Zejia Weng, Zhonghao Yan, Zhiguo Huang, Zhuoyu Wang, Zihan Yan, Zheng Ge, Jing Li, Yibo Zhu, Binxing Jiao, Xiangyu Zhang, and Daxin Jiang. Step-gui technical report, 2025b. URL https://arxiv.org/abs/2512.15431.

Bowen Yang, Kaiming Jin, Zhenyu Wu, Zhaoyang Liu, Qiushi Sun, Zehao Li, JingJing Xie, Zhoumianze Liu, Fangzhi Xu, Kanzhi Cheng, Qingyun Li, Yian Wang, Yu Qiao, Zun Wang, and Zichen Ding. Os-symphony: A holistic framework for robust and generalist computer-using agent, 2026. URL https://arxiv.org/abs/2601.07779.

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*, 2025a.

Yuhao Yang, Zhen Yang, Zi-Yi Dou, Anh Nguyen, Keen You, Omar Attia, Andrew Szot, Michael Feng, Ram Ramrakhya, Alexander Toshev, Chao Huang, Yinfei Yang, and Zhe Gan. Ultracua: A foundation model for computer use agents with hybrid action, 2025b. URL https://arxiv.org/abs/2510.17790.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, et al. Mobile-agent-v3: Fundamental agents for gui automation. *arXiv preprint arXiv:2508.15144*, 2025.

Yuwen Zhai, Runze Li, Liang Wang, Nian Shi, Liwu Xu, Wei Zhang, Ran Lin, Bo Xu, and Benlei Cui. Guide: Interpretable gui agent evaluation via hierarchical diagnosis, 2026. URL https://arxiv.org/abs/2604.04399.

Danyang Zhang, Situo Zhang, Ziyue Yang, Zichen Zhu, Zihan Zhao, Ruisheng Cao, Lu Chen, and Kai Yu. Progrm: Build better gui agents with progress rewards, 2025a. URL https://arxiv.org/abs/2505.18121.

Junlei Zhang, Zichen Ding, Chang Ma, Zijie Chen, Qiushi Sun, Zhenzhong Lan, and Junxian He. Breaking the data barrier–building gui agents through task generalization. *arXiv preprint arXiv:2504.10127*, 2025b.

Congmin Zheng, Xiaoyun Mo, Xinbei Ma, Qiqiang Lin, Yin Zhao, Jiachen Zhu, Xingyu Lou, Jun Wang, Zhaoxiang Wang, Weiwen Liu, Zhuosheng Zhang, Yong Yu, and Weinan Zhang. Adaptive milestone reward for gui agents, 2026. URL https://arxiv.org/abs/2602.11524.

Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, et al. Mai-ui technical report: Real-world centric foundation gui agents. *arXiv preprint arXiv:2512.22047*, 2025.

A Precision-Recall Trade-off in RL

In policy-gradient-based RL, the reward signal determines *which* trajectories are reinforced. Precision therefore reflects the *purity* of rewarded samples: if many failures are mistakenly rewarded, the update direction is contaminated and the policy may be pushed toward incorrect behaviors, which becomes especially harmful when scaling up training. Recall, by contrast, controls how often truly good behaviors are recognized and rewarded; if recall is too low, positive signal becomes sparse, slowing iteration and limiting the attainable performance ceiling. Our objective is thus to maximize precision while keeping recall sufficiently high so that learning remains effective.

We formalize this trade-off in a simplified setting. Once recall is already adequate to provide stable positive supervision, a mild reduction in recall can be worthwhile if it produces a larger reduction in false positives, thereby increasing precision and improving reward reliability.

Consider a fixed context x where the policy samples a trajectory $\tau \sim \pi_\theta(\cdot \mid x)$ from $\{g, b\}$, with $y(g) = 1$ and $y(b) = 0$. For compactness, define $\Pr_\theta(\cdot) \triangleq \Pr_{\tau\sim\pi_\theta(\cdot|x)}(\cdot)$ and $\mathbb{E}_\theta[\cdot] \triangleq \mathbb{E}_{\tau\sim\pi_\theta(\cdot|x)}[\cdot]$. Let

$$p(\theta) \triangleq \Pr_\theta(\tau = g), \quad 1 - p(\theta) \triangleq \Pr_\theta(\tau = b). \tag{5}$$

Assume an imperfect evaluator outputs $\hat{r}(\tau) \in \{0, 1\}$ with operating characteristics

$$\begin{aligned} \rho &\triangleq \Pr(\hat{r} = 1 \mid \tau = g) \quad \text{(recall / true-positive rate)}, \\ \alpha &\triangleq \Pr(\hat{r} = 1 \mid \tau = b) \quad \text{(false-positive rate)}. \end{aligned} \tag{6}$$

The true objective is

$$J(\theta) \triangleq \mathbb{E}_\theta[y(\tau)]. \tag{7}$$

Since $y(g) = 1$ and $y(b) = 0$, we have

$$J(\theta) = p(\theta). \tag{8}$$

The pseudo-objective induced by policy-gradient updates is

$$\hat{J}(\theta) \triangleq \mathbb{E}_\theta[\hat{r}(\tau)]. \tag{9}$$

Expanding by cases yields

$$\hat{J}(\theta) = p(\theta)\rho + \big(1 - p(\theta)\big)\alpha, \tag{10}$$

and equivalently,

$$\hat{J}(\theta) = \alpha + (\rho - \alpha)\, p(\theta). \tag{11}$$

For any baseline c independent of τ,

$$\mathbb{E}_\theta\Big[\nabla_\theta \log \pi_\theta(\tau \mid x)\,(\hat{r}(\tau) - c)\Big] = \nabla_\theta \hat{J}(\theta). \tag{12}$$

Differentiating gives

$$\nabla_\theta \hat{J}(\theta) = \nabla_\theta p(\theta)\,(\rho - \alpha). \tag{13}$$

Under the standard logit parameterization $p(\theta) = \sigma(\theta)$,

$$\nabla_\theta p(\theta) = p(\theta)\big(1 - p(\theta)\big) > 0, \tag{14}$$

so the expected update is governed by the *preference margin* $(\rho - \alpha)$. Intuitively, larger α assigns positive signal to bad trajectories, shrinking the gap between good and bad and reducing reward reliability; when α approaches ρ, the reward becomes weakly informative for distinguishing g from b.

Precision depends on the policy base rate:

$$\mathrm{Prec}(\theta) \triangleq \Pr(\tau = g \mid \hat{r} = 1). \tag{15}$$

By Bayes' rule,

$$\mathrm{Prec}(\theta) = \frac{p(\theta)\rho}{p(\theta)\rho + \big(1 - p(\theta)\big)\alpha}. \tag{16}$$

Maintaining sufficiently high recall ρ ensures true successes are rewarded often enough to provide stable learning signal. However, when ρ is already adequate, reducing the false-positive rate α can increase precision substantially, particularly when $p(\theta)$ is not large (e.g., early training or hard tasks).

Concretely, consider $(\rho, \alpha) \mapsto (\rho', \alpha')$ with $\rho' = \rho - \delta$ and $\alpha' = \alpha - \Delta$, where $\delta > 0$ and $\Delta > 0$ (locally fixing $p(\theta)$). Then

$$\mathrm{Prec}'(\theta) > \mathrm{Prec}(\theta) \quad \text{if} \quad \rho\,\Delta > \alpha\,\delta. \tag{17}$$

Meanwhile, the preference margin becomes

$$(\rho' - \alpha') = (\rho - \alpha) + (\Delta - \delta), \tag{18}$$

which increases when $\Delta > \delta$, indicating a more reliable discriminative reward for policy-gradient updates.

In summary, once recall is sufficiently high, trading a mild decrease in recall for a larger reduction in false positives can improve precision and yield a more reliable reward signal for policy-gradient-based RL. Nevertheless, recall should not be pushed too low: if ρ becomes small, truly good trajectories are rarely rewarded, leading to sparse positives, higher gradient variance, slower learning, and potentially less stable training.

Table 4: Comparison of reward sources and open-source status in recent GUI online RL works.

Name	Platform	Reward Method	Open-source
UltraCUA Yang et al. (2025b)	Desktop	Rule-based	✗
Mobile-Agent-v3 Ye et al. (2025)	Desktop + Mobile	Rule-based	✗
Mobile-R1 Gu et al. (2026)	Mobile	Rule-based + ORM	✗
MobileGUI-RL Shi et al. (2025)	Mobile	Rule-based	✗
MobileRL Xu et al. (2025)	Mobile	Rule-based + ORM	✗
UI-Mem Xiao et al. (2026)	Mobile	Rule-based + PRM	✗
GUI-Shepherd Chen et al. (2025)	Mobile	PRM	✗
ProgRM Zhang et al. (2025a)	General	PRM	✗
DynaWeb Ding et al. (2026)	Web	ORM	✗
GUIDE Zhai et al. (2026)	Web + Mobile	ORM	✗
UI-TARS-2 Wang et al. (2025a)	General	Rule-based + ORM	✗
ARPO Lu et al. (2025a)	Desktop	Rule-based	✓
ComputerRL Lai et al. (2025b)	Desktop	Rule-based	✓
DART-GUI Li et al. (2025)	Desktop	Rule-based	✓
UI-S1 Lu et al. (2025b)	Desktop + Mobile	Rule-based	✓
RLAnything Wang et al. (2026)	Desktop	Rule-based + PRM	✓
ADMIRE Zheng et al. (2026)	Mobile	Rule-based + PRM	✓
WebAgent-R1 Wei et al. (2025)	Web	Rule-based	✓
GE-Lab Yan et al. (2025a)	Mobile	Rule-based	✓
WebRL Qi et al. (2024)	Web	ORM	✓
DigiRL Bai et al. (2024)	Mobile	ORM	✓
DistRL Wang et al. (2025b)	Mobile	ORM	✓
ZeroGUI Yang et al. (2025a)	Desktop + Mobile	ORM	✓
SEAgent Sun et al. (2025b)	Desktop	PRM + ORM	✓
OS-Themis	General	ORM	✓

B Baseline Selection

To clarify our baseline selection, Table 4 summarizes recent GUI online RL works by platform, reward source, and open-source status. Most existing GUI online RL methods rely on rule-based rewards, which depend on environment-specific scripts, task initialization, and platform-specific success interfaces. Since our online RL comparison is conducted within a self-built Android training infrastructure, these rule-based rewards are difficult to transfer fairly, highlighting the need for more general trajectory-level reward methods. We also attempted to evaluate several training-based GUI critic models; however, many existing critics are PRM-style models and are not compatible with the trajectory-level ORM evaluation in OGRBench Wu et al. (2025b); Wanyan et al. (2025); Gu et al. (2025). Moreover, independent PRM-style rewards have been less explored in online RL settings, so we do not include them in the online RL comparison. For ORM-style critics, most existing models are not open-sourced Xu et al. (2025); Ye et al. (2025); Wang et al. (2025a).

Among the few open-source alternatives, DistRL Wang et al. (2025b) adopts the evaluation design of DigiRL Bai et al. (2024), while WebRL Qi et al. (2024) and SEAgent Sun et al. (2025b) are constrained by platform-specific settings; WebRL in particular relies on web HTML information. Therefore, we use DigiRL Bai et al. (2024) and ZeroGUI Yang et al. (2025a) as the main ORM-style baselines for OGRBench, representing sequential verification and final-state assessment, respectively. We implement the DigiRL and ZeroGUI reward evaluators by reusing code from their respective official open-source repositories and adapting the input interfaces to the unified OGRBench trajectory format. We also evaluate SEAgent as a representative training-based critic, but its performance is limited under our multi-platform trajectory setting. For online RL, we focus on ORM-style reward sources under the same Android training loop; since ZeroGUI performs substantially better than DigiRL on OGRBench, we use ZeroGUI as the main LLM-as-a-Judge reward baseline and include SEAgent as the training-based critic comparator.

Table 5: Comparison of Accuracy (Acc), Precision (Prec), and Recall (Rec) across various platforms in OmniGUIRewardBench, evaluated using DigiRL, ZeroGUI, OS-Themis, and other GUI critic models (e.g., SEAgent).

Model	Ubuntu			Mobile			Windows			macOS			WebA			Overall		
	Acc	Prec	Rec	Acc	Prec	Rec	Acc	Prec	Rec	Acc	Prec	Rec	Acc	Prec	Rec	Acc	Prec	Rec
DigiRL																		
Qwen3-VL-8B	62.1	63.7	66.2	69.1	63.3	96.9	66.2	60.6	67.0	72.7	36.8	43.8	63.2	61.1	80.8	64.4	62.2	72.1
Qwen3-VL-30B-A3B	61.0	60.4	76.6	67.0	61.5	**98.0**	69.0	60.4	**86.2**	67.5	36.4	75.0	54.2	54.4	74.7	62.5	58.9	80.6
Qwen3-VL-32B	65.7	69.0	64.1	72.3	66.9	92.9	71.8	64.9	78.7	75.3	44.4	75.0	62.1	60.8	76.8	67.6	65.8	72.1
Qwen3-VL-235B	65.5	67.9	66.2	70.2	66.4	86.7	72.3	66.1	76.6	76.6	43.8	43.8	63.2	66.3	59.6	67.4	66.6	69.0
Qwen3-VL-235B-Thinking	62.3	63.1	70.0	67.6	62.9	91.8	65.7	58.7	75.5	70.1	37.0	62.5	58.4	59.1	65.7	63.4	61.1	73.0
GPT-5-mini	47.0	0.0	0.0	47.9	0.0	0.0	55.9	0.0	0.0	79.2	0.0	0.0	47.9	0.0	0.0	50.3	0.0	0.0
GPT-5	50.5	90.6	7.4	56.9	87.0	20.4	59.2	81.8	9.6	79.2	0.0	0.0	52.6	**100.0**	9.1	54.5	89.3	9.6
Gemini-3-Flash	69.5	88.5	48.9	69.7	83.6	52.0	76.5	90.7	52.1	83.1	**100.0**	18.8	75.3	81.0	68.7	72.1	86.6	51.9
Mean	60.4	62.9	49.9	65.1	61.5	67.3	67.1	60.4	55.7	75.5	37.3	39.8	59.6	60.3	54.4	62.8	61.3	53.5
ZeroGUI																		
Qwen3-VL-8B	72.7	90.6	54.2	78.7	79.6	79.6	76.5	86.7	55.3	89.6	90.0	56.3	75.8	81.2	69.7	75.4	86.3	60.1
Qwen3-VL-30B-A3B	74.8	80.1	69.7	80.3	76.1	90.8	80.8	80.5	74.5	90.9	84.6	68.8	72.1	70.2	80.8	76.9	77.9	74.9
Qwen3-VL-32B	75.0	88.2	61.1	78.7	78.4	81.6	79.3	87.9	61.7	89.6	**100.0**	50.0	76.3	80.7	71.7	77.1	85.3	65.3
Qwen3-VL-235B	76.5	87.9	64.6	82.4	79.3	89.8	77.9	85.1	60.6	93.5	**100.0**	68.8	84.7	84.3	86.9	79.6	85.5	70.9
Qwen3-VL-235B-Thinking	71.8	88.7	53.7	79.8	76.3	88.8	79.3	87.9	61.7	89.6	**100.0**	50.0	78.9	84.7	72.7	75.9	85.3	62.3
GPT-5-mini	48.4	86.7	3.3	49.5	**100.0**	3.1	56.3	**100.0**	1.1	79.2	0.0	0.0	48.9	75.0	3.0	51.5	87.0	2.9
GPT-5	72.6	92.8	52.4	81.9	84.0	80.6	77.5	88.3	56.4	90.9	**100.0**	56.3	80.5	87.8	72.7	76.7	89.7	59.9
Gemini-3-Flash	76.1	91.9	60.3	77.1	84.0	69.4	80.8	89.6	63.8	93.5	**100.0**	68.8	77.4	87.8	65.7	78.1	89.8	63.0
Mean	71.0	88.4	52.4	76.1	82.2	73.0	76.1	88.2	54.4	89.6	84.3	52.3	74.3	81.5	65.4	73.9	85.8	57.4
OS-Themis																		
Qwen3-VL-8B	77.2	86.6	67.4	85.6	92.8	78.6	72.8	81.0	50.0	85.7	72.7	50.0	86.3	84.8	**89.9**	79.3	86.3	69.4
Qwen3-VL-30B-A3B	79.5	84.9	74.6	84.6	86.3	83.7	76.5	78.2	64.9	88.3	81.8	56.3	80.5	88.8	71.7	80.3	84.7	73.7
Qwen3-VL-32B	77.6	91.3	63.9	83.5	94.7	72.4	75.1	90.2	48.9	88.3	76.9	62.5	84.7	97.3	72.7	79.6	92.2	64.3
Qwen3-VL-235B	**88.1**	92.2	**84.7**	**92.6**	93.8	91.8	77.5	89.7	55.3	**94.8**	92.9	**81.3**	**92.1**	95.7	88.9	**88.0**	92.8	**82.3**
Qwen3-VL-235B-Thinking	83.4	87.5	80.2	89.4	91.5	87.8	85.4	88.0	77.7	93.5	92.3	75.0	84.7	96.1	73.7	85.2	89.3	79.9
GPT-5-mini	68.8	**95.0**	43.5	65.4	92.3	36.7	76.5	97.8	47.9	87.0	**100.0**	37.5	75.8	96.5	55.6	71.5	**95.4**	44.7
GPT-5	82.5	93.7	71.8	80.3	91.8	68.4	84.5	90.7	72.3	88.3	**100.0**	43.8	83.2	95.9	70.7	82.9	93.4	70.6
Gemini-3-Flash	85.0	92.5	78.1	91.0	93.5	88.8	**86.9**	92.3	76.6	93.5	92.3	75.0	82.6	97.1	68.7	86.2	93.2	78.0
Mean	80.3	90.5	70.5	84.0	92.1	76.0	79.4	88.5	61.7	89.9	88.6	60.2	83.8	94.0	74.0	81.6	90.9	70.4
Other Critic Model																		
SEAgent	59.4	94.2	24.9	55.9	85.7	18.4	63.8	87.0	21.3	84.4	**100.0**	25.0	65.8	82.7	43.4	61.8	89.7	26.1

We further include ADMIRE Zheng et al. (2026) as a credit-assignment baseline, as it also uses milestones but differs substantially from MC-GRPO in mechanism. ADMIRE derives task-level milestones from successful trajectories and uses milestone progress for auxiliary reward shaping. In its released implementation, milestone generation uses the task instruction and trajectory action descriptions, with at most the last five screenshots as visual context, while successful trajectory collection and milestone updates rely on environment-provided task success signals. In contrast, MC-GRPO directly uses milestones verified by OS-Themis from visual state transitions in each trajectory, without requiring successful trajectories as templates or environment success signals for milestone construction, and redistributes the original GRPO advantage without introducing auxiliary rewards.

C More Results

C.1 Detailed OGRBench results

In Table 5, we report more detailed, platform-specific metrics for evaluating OGRBench across different models. By analyzing how each framework performs on individual platforms, we observe that, compared with ZeroGUI Yang et al. (2025a) and DigiRL Bai et al. (2024), OS-Themis exhibits consistently strong performance with superior stability and overall effectiveness across diverse platforms.

The detailed results suggest that overall framework performance is strongly influenced by the capability of the underlying model. For instance, GPT-5 and GPT-5-mini OpenAI (2025) tend to produce more conservative outputs and prefer predicting `False`, which can lead to notably low recall across different frameworks; OS-Themis partially mitigates this issue by reducing the extent to which recall is dominated by conservative bias. Meanwhile, Gemini-3-Flash Comanici et al. (2025) achieves the highest precision across all models while maintaining sufficiently high recall, indicating more reliable positive judgments.

For the Qwen3-VL Bai et al. (2025) series, we observe that as model size increases, the advantage of OS-Themis over DigiRL and ZeroGUI becomes more pronounced. With smaller models, the gap in precision is relatively large, whereas the difference in accuracy is limited; however, as reasoning capability improves with scale, the gains brought by OS-Themis expand substantially, yielding clearer performance separation. This trend suggests that under more complex and evidence-heavy settings, the quality of evidence organization and aggregation becomes a key determinant of final performance, and stronger models can better leverage the structured advantages of OS-Themis.

C.2 AGENTREWARDBENCH

We evaluated OS-Themis on AgentRewardBench Lù et al. (2025), and the results are summarized in Table 6. As shown in the table, our model achieves high Precision, reaching 76.8, although Recall is relatively low. This may partly reflect a mismatch between the available trajectory information and the input structure expected by our framework. First, AgentRewardBench exhibits a pronounced class imbalance between positive and negative samples (295:811), making high Precision valuable for filtering successful trajectories from a pool dominated by failures. Second, a large proportion of its positive samples are collected using Claude-3.7-Sonnet Anthropic (2025a). Many of these trajectories contain multiple action steps but lack accompanying action rationales. Without explicit information about the intended subgoal of each step, the Selector may struggle to organize low-level actions into task-level milestones and formulate appropriate verification goals. Missing or ambiguous milestones may then lead to conservative failure judgments, contributing to lower Recall on this benchmark.

Table 6: Performance of different models on AgentRewardBench within the OS-Themis framework.

Category	**Method**	**Overall**	
		Prec	**Recall**
Official	Rule-based	83.8	55.9
Critic Model	SEAgent	71.6	-
LLM-as-a-Judge	Claude-3.7-Sonnet	64.3	89.8
	GPT-4o	68.1	80.3
	GPT-4o Mini	64.5	78.3
	Qwen2.5-VL-72B	64.5	86.1
	Qwen3-VL-8B	66.3	82.0
	Qwen3-VL-32B	72.9	73.9
	Qwen3-VL-235B	67.5	81.7
OS-Themis	Qwen3-VL-8B	71.6	39.0
	Qwen3-VL-32B	76.8	39.3
	Qwen3-VL-235B	76.1	54.4

D EXPLORING SELF-EVOLVING CAPABILITIES FOR GUI AGENTS

The advancement of GUI agents is currently constrained by the scarcity of high-quality trajectory data; scaling data acquisition remains a critical bottleneck. However, given that modern agents already possess foundational execution capabilities, a promising solution is to enable autonomous environmental exploration Lai et al. (2025a); Yan et al. (2025b). This facilitates a virtuous cycle: collected interaction data is used to train the agent, which in turn generates higher-quality data. To sustain this evolution, the primary challenge lies in ensuring the stability and efficiency of data collection while maintaining quality. Effective filtering is therefore essential to distill high-value trajectories from massive exploration logs—a task where OS-Themis excels. Consequently, we integrate Qwen3-VL and OS-Themis within a containerized Android environment to establish an autonomous self-evolution paradigm driven by scalable exploration and precise filtering.

Task Generation. During the initialization phase, we pre-install a suite of applications within Docker-based Android Emulators and curate a set of representative seed tasks. Utilizing these examples, the model first generates comprehensive descriptions for each application. To ensure a valid exploration context, every session begins with a mandatory app-launch directive; the generation of subsequent exploration tasks is triggered only upon successful application entry. To augment state diversity prior to generation, we inject stochastic interactions (e.g., random swipes or clicks) with a fixed probability. Ultimately, the model synthesizes a series of exploration tasks, conditioned on the real-time device screenshot, the application description, and the seed task examples.

Trajectory Collection and Filtering. Upon task generation, the agent initiates execution, interacting with the environment until either the task is completed or a predefined step limit is reached. Crucially, the system operates asynchronously: as soon as a trajectory is concluded, OS-Themis performs an immediate correctness evaluation, while the collection pipeline concurrently proceeds to the next task. This design enables simultaneous data collection and real-time filtering, efficiently yielding a curated dataset of high-quality trajectories.

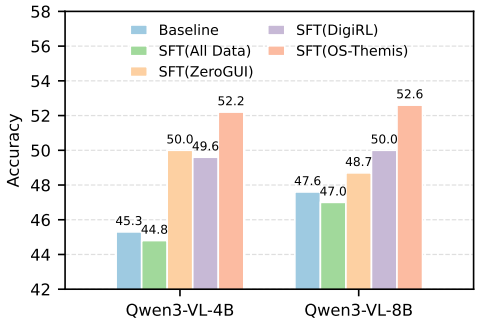


Figure 4: The performance of Qwen3-VL-4B and Qwen3-VL-8B on AndroidWorld after SFT using filtered data (the parentheses indicate the filtering method; All Data means the data is unfiltered).

Data Quality Evaluation. We aggregated a raw dataset of 15,110 trajectories collected in the Android environment. We then filtered the data using DigiRL, ZeroGUI, and OS-Themis, respectively, yielding three high-quality subsets. To evaluate the effectiveness of these filtered data, we conducted supervised fine-tuning (SFT) experiments on the Qwen3-VL-4B and Qwen3-VL-8B backbones. As shown in Fig. 4, fine-tuning on the OS-Themis–filtered data brings substantial improvements of 6.9% and 5.0% over the respective baselines, and clearly outperforms fine-tuning on data filtered by DigiRL and ZeroGUI. In contrast, when the entire unfiltered collected dataset is used for SFT, the model performance degrades to varying extents, indicating the presence of substantial noise in the raw data. These results provide strong evidence for the high quality of the filtered data. Beyond this validation, they further indicate that OS-Themis can serve as a reliable core component for scalable data curation, enabling the construction of an autonomous and self-evolving data collection pipeline.

E Experiments on Scaling

E.1 Impact of Model Scaling on Individual Agents in OS-Themis

To investigate the potential of each component within OS-Themis, we individually replaced the base model of the Selector, Reviewer, Judge, and Verifier with the stronger Qwen3-VL-235B, while keeping other components at 8B.

The results are detailed in Table 7. We observe that scaling the Judge and Verifier agents yields the most significant performance gains in terms of overall effectiveness, with the 235B Verifier achieving the highest Accuracy (82.5%) and the Judge attaining the best F1 score (81.7%). Notably, the Reviewer agent equipped with the 235B model achieves the highest Precision (89.1%) among all variants. This indicates that the increased model capacity enables the Reviewer to exercise greater rigor in its assessments, strictly minimizing false positives and ensuring high confidence in the validated trajectories.

Table 7: Single-agent scaling study in OS-Themis. "↑" upgrades the corresponding agent to Qwen3-VL-235B; all others remain Qwen3-VL-8B. Metrics include Overall Accuracy (Acc), Precision (Prec), Recall (Rec) and F1-Score (F1), reported in percentages. **Bold** denotes the best result.

Variant	Overall (%)			
	Acc	Prec	Rec	F1
Base (all 8B)	79.4	86.3	69.4	77.0
Selector ↑	80.6	86.1	72.7	78.9
Reviewer ↑	79.3	**89.1**	66.4	76.1
Judge ↑	82.4	84.5	**79.1**	**81.7**
Verifier ↑	**82.5**	88.9	74.1	80.8

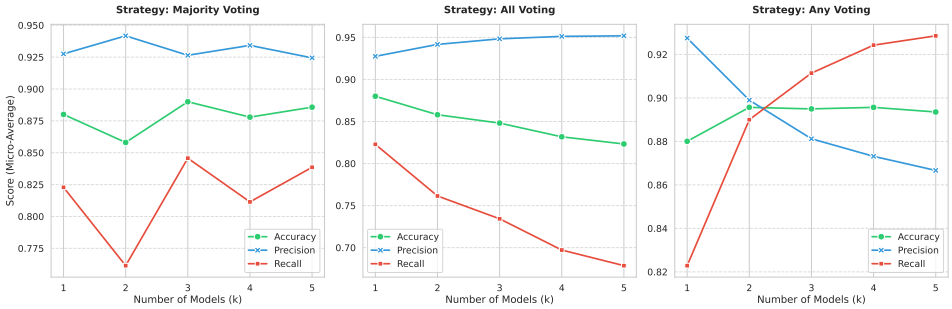


Figure 5: Scaling performance comparison of Qwen3-VL-235B under the OS-Themis framework on OGRBench using three voting strategies: Majority, All, and Any.

E.2 Test-Time Scaling for Framework Analysis

We assess the performance of three common test-time scaling strategies—**Majority**, **All**, and **Any Voting** Yang et al. (2025a)—across varying numbers of models (k), with results illustrated in Figure 5. **Majority Voting** predicts a positive outcome when more than half of the models vote positive, **All Voting** requires all models to vote positive, and **Any Voting** predicts positive as long as at least one model votes positive. Overall, the experimental results reveal clear and systematic trade-offs between precision and recall that stem directly from the different aggregation rules. In particular, as k increases, each strategy amplifies a different bias in the final decision, thereby shifting the operating point along the precision–recall spectrum.

Majority Voting exhibits the strongest robustness among the three. As a balanced strategy, it maintains relatively stable accuracy across different k, reflecting its ability to offset occasional misjudgments from individual models through simple averaging. However, its recall shows sawtooth-like oscillations at even values of k, indicating sensitivity to the voting threshold when ties or near-ties become more likely. This behavior suggests that Majority Voting is generally reliable as a default aggregation choice, but its recall can fluctuate depending on how the threshold interacts with k.

All Voting (Consensus) imposes the strictest filtering mechanism, requiring agreement across all models before producing a positive decision. As k increases, this strategy increasingly suppresses false positives, leading to a steady improvement and eventual saturation in precision. At the same time, the stringent consensus constraint makes it progressively harder to return positives, which manifests as a near-linear decline in recall. As a consequence, All Voting is best suited for settings where avoiding false positives is paramount and low coverage is acceptable—i.e., applications with extremely low error tolerance that explicitly prioritize precision over recall (e.g., constructing high-quality instruction fine-tuning datasets).

In contrast, **Any Voting** minimizes false negatives by adopting a union-based decision rule: as long as one model votes positive, the aggregate decision becomes positive. With larger k, this strategy substantially boosts recall, since it becomes less likely that all models miss a true positive. However, the same mechanism also admits more noisy positives, which reduces precision. Despite this trade-off,

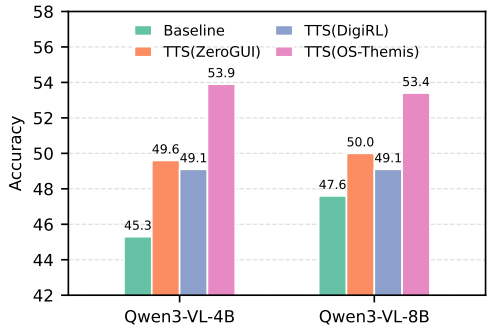


Figure 6: The performance of Qwen3-VL-4B and Qwen3-VL-8B on AndroidWorld after Test-Time Scaling (TTS).

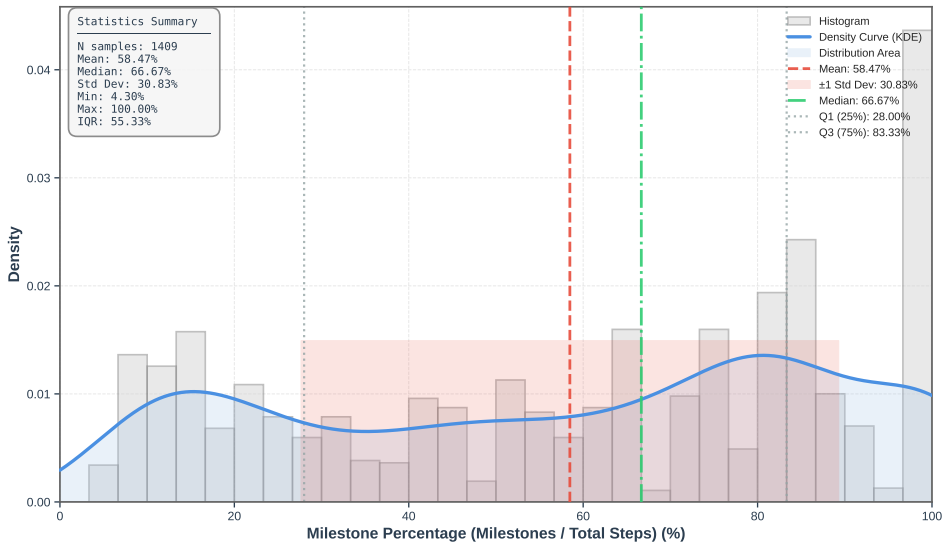


Figure 7: Distribution of Milestone Percentage

overall accuracy can still improve marginally, reflecting the net benefit of reducing misses under this aggregation rule. Accordingly, Any Voting is particularly effective for recall-sensitive tasks, such as preliminary data screening, where missing relevant instances is more costly than introducing additional candidates to be filtered later.

E.3 Test-Time Scaling in Evaluation

To verify the role of OS-Themis on the evaluation side, we conduct a Test-Time Scaling (TTS) study during the evaluation stage. Specifically, on the AndroidWorld benchmark, we use Qwen3-VL-4B and Qwen3-VL-8B as the evaluated policy backbones, with the sampling temperature fixed to 0.7. For each task, the agent first executes one attempt, and OS-Themis is used to determine whether the task is completed: if the attempt is judged successful, we proceed to the next task; otherwise, we retry the same task, with at most three attempts per task. After all attempts are finished, the built-in rules are used to compute the final score. The results are summarized in Figure 6. In addition, we adopt Qwen3-VL-235B as the base evaluator and compare three evaluation frameworks, DigiRL, ZeroGUI, and OS-Themis. The results show that OS-Themis improves over the Baseline by 8.6% and 5.8% for the two policy backbones, respectively, substantially outperforming DigiRL and ZeroGUI. These results demonstrate that OS-Themis can effectively enhance both performance and robustness under test-time scaling at evaluation time.

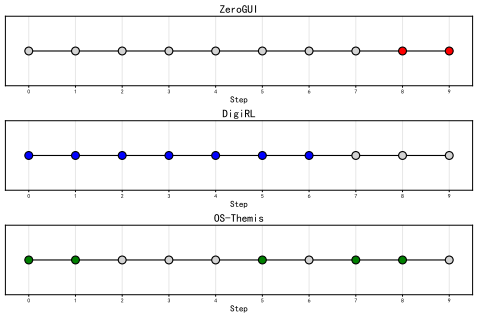


Figure 8: Illustration of trajectory steps selected for correctness evaluation under different methods. Colored points represent selected steps.

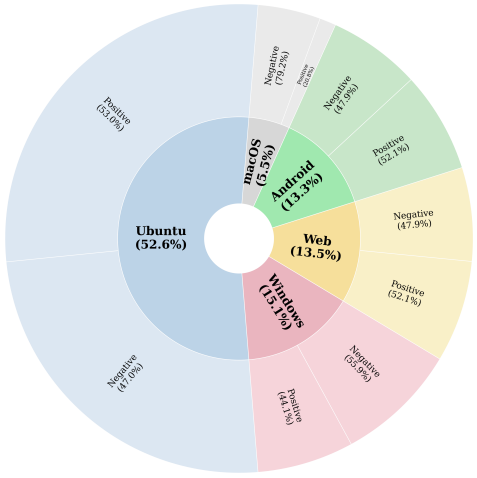


Figure 9: Data Distribution of OGRBench

F Details of Milestones

F.1 Milestone Statistics

We present comprehensive statistics on Milestones under Qwen3-VL-235B in Table 12. The task-level average milestone percentage is 58.47%, with a median of 66.67%. Across all 1,409 tasks, the total number of steps is 27,882, and the total number of milestones is 9,918, representing 35.57% of the overall steps (step-level percentage). The average number of milestones per task is 7.04, with a median of 6.00 and a standard deviation of 4.16. These statistics indicate that only about half of the steps in each task are critical to the final outcome, and this subset of information is enough to determine whether a trajectory is correctly executed.

Figure 7 presents the histogram and kernel density curve of the overall milestone percentage distribution. The distribution peaks at 100%, exceeding 4%, primarily due to a substantial portion of simple tasks consisting of only a few steps, each requiring verification. Apart from the 100% peak, the distribution is relatively uniform across other percentage ranges, demonstrating that milestone selection exhibits no significant bias.

F.2 Illustration of Milestones

Figure 8 provides a visual comparison between our Milestone approach and existing methods (DigiRL and ZeroGUI). Unlike previous methods, our approach selectively identifies a few discrete steps within a trajectory as milestones, thereby efficiently leveraging the most critical information from the trajectory data.

Table 8: Milestone ablation on OGRBench. **All-step Verification** removes the Selector and verifies every step. **Drop 25%** removes 25% of selected milestones, while **Random Replacement** and **Final-step Replacement** use the same number of steps as Full Milestones but select random or terminal-near steps. **Full Milestones** uses the milestones selected by OS-Themis. Metrics include Overall Accuracy (Acc), Precision (Prec), Recall (Rec) and F1-Score (F1), reported in percentages. **Bold** denotes the best result.

Variant	Overall (%)			
	Acc	Prec	Rec	F1
All-step Verification	83.25	79.74	**88.86**	84.06
Drop 25%	81.41	89.25	71.14	79.17
Random Replacement	82.97	87.95	76.14	81.62
Final-step Replacement	83.68	87.90	77.86	82.58
Full Milestones	**88.00**	**92.80**	82.30	**87.20**

F.3 Milestone Ablation

To further examine the role of milestone selection, we perturb the milestone set produced by the final OS-Themis model and re-evaluate the resulting critic performance. We consider four variants: **All-step Verification**, which removes the Selector and verifies all trajectory steps; **Drop 25%**, which removes 25% of the selected milestones; **Random Replacement**, which replaces the selected milestones with the same number of randomly sampled trajectory steps; and **Final-step Replacement**, which replaces them with the same number of trajectory steps closest to termination.

As shown in Table 8, all perturbations degrade performance compared with the full milestone set. All-step verification achieves the highest Recall, but its Precision drops substantially, indicating that verifying every step introduces many weakly relevant signals and dilutes critical failure evidence. Dropping 25% of milestones causes the largest overall degradation, especially in Recall, showing that missing task-critical subgoals weakens trajectory-level verification. Random and Final-step replacement retain the same verification budget but still underperform the original milestone selection. These results suggest that OS-Themis benefits not merely from checking more steps or focusing on the trajectory ending, but from selecting task-critical milestones that capture decisive evidence throughout the interaction.

G OGRBench Dataset Details and Statistics

G.1 Dataset Construction

Trajectory Collection. We collect trajectories from five representative benchmarks: AndroidWorld (Rawles et al., 2024), OSWorld (Xie et al., 2024), WindowsAgentArena (Bonatti et al., 2024), macOSArena (Wang et al., 2025e), and WebArena-Lite-v2 (Wang et al., 2025e). Each trajectory preserves the full screenshot sequence and agent outputs. The trajectories are generated by diverse GUI agents, including Qwen3-VL (4B, 8B, 235B) (Bai et al., 2025), UITARS (1.5-7B, 72B-DPO) (Qin et al., 2025), ScaleCUA (7B, 32B) (Liu et al., 2025), and Claude-Sonnet-4.5 (Anthropic, 2025b).

Outcome Annotation and Sampling. Each trajectory receives a binary task-success label (`True`/`False`) from its source benchmark's built-in evaluator. To reduce label noise, we manually audit potentially unreliable labels and remove problematic samples. We use stratified sampling to maintain task coverage and label balance, keeping the positive ratio between 0.45 and 0.55. The resulting benchmark contains 1,409 trajectories, comprising 700 positive and 709 negative examples.

G.2 Dataset Statistics

Comparison with Existing Benchmarks. In Table 10, we compare OGRBench with two existing ORM benchmarks, AgentRewardBench Lù et al. (2025) and CUARewardBench. As shown in the table, OGRBench is currently the most comprehensive and largest-scale GUI ORM benchmark available.

Table 9: Statistical Overview of the OGRBench Data.

Benchmark	Platform	#Positive	#Negative
OSWorld	Ubuntu	393	348
AndroidWorld	Android	98	90
WindowsAgentArena	Windows	94	119
MacOSArena	MacOS	16	61
WebArena	Web	99	91
Total		**700**	**709**

Dataset Distribution. In Table 9, we provide the statistics for the dataset distribution of OGRBench. As illustrated in Figure 9, the overall dataset maintains a balanced distribution between positive and negative samples. An exception is macOSArena, where the positive–negative ratio is notably imbalanced. This is primarily because current models perform poorly on macOS tasks: even the best-performing model achieves a success rate below 10%. To maximize task coverage under this constraint, we include all available negative trajectories, while positive trajectories remain scarce. Increasing the positive ratio would require either adding many highly similar positive cases or discarding a substantial number of negative cases, both of which would reduce the diversity and representativeness of the dataset. Given that the overall number of macOS tasks is also limited, we keep the current ratio as a reasonable reference point. We plan to revisit this split and add more positive trajectories once model performance on macOS improves, so that the dataset can be rebalanced in a more natural way.

Table 10: Statistics of Existing GUI ORM Benchmarks

Benchmark	Platform					Samples
	Ubuntu	Android	Windows	macOS	Web	
AgentRewardBench	✗	✗	✗	✗	✓	1106
CUARewardBench	✓	✗	✗	✗	✗	272
OGRBench	✓	✓	✓	✓	✓	1409

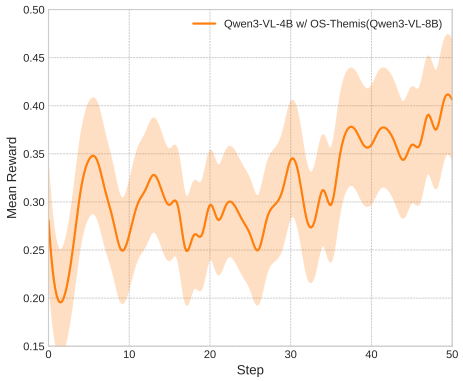
H Details of RL Training

H.1 Experimental Setup

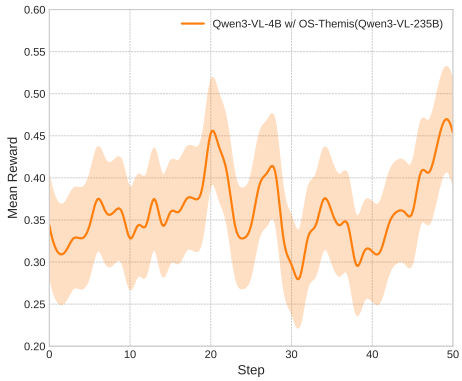
Tasks and Policy Backbones. We use Qwen3-VL-4B and Qwen3-VL-8B as policy backbones. Following the task-synthesis procedure described in the main text (Lai et al., 2025a), we curate 96 training tasks and a separate set of 64 validation tasks from a pool generated by Qwen3-VL-235B. Each task record specifies a natural-language instruction and the environment initialization required for execution. The instruction defines the policy's objective, while the initialization specification determines its starting application state.

Training Configuration. We use GRPO (Shao et al., 2024; Guo et al., 2025a) with AdamW (Loshchilov & Hutter, 2017), a learning rate of 1×10^{-6}, weight decay of 1×10^{-2}, and a gradient clipping threshold of 1.0. KL regularization is disabled (`disable_kl=true`, `kl_coef=0.0`). For each sampled task, we collect four rollouts at a sampling temperature of 1.0, with a maximum of 50 interaction steps and a request timeout of 60 seconds. The training schedule uses `total_episodes=4`. Training uses 16 NVIDIA H200 GPUs, with Android environments hosted on a machine with 256 CPU cores.

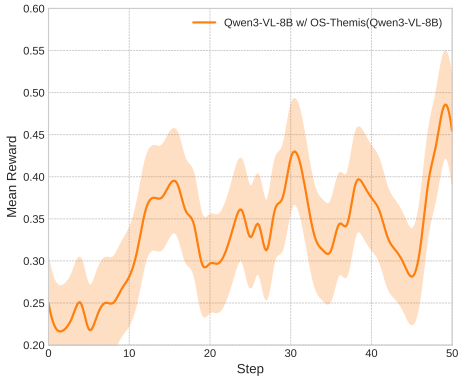
Scaling Configuration. For the scaling study, we use Qwen3-VL-4B as the policy and Qwen3-VL-235B as the OS-Themis backbone. We instantiate 1,024 tasks from filtered templates, collect four rollouts per task, and train for one online RL epoch. This setting uses `low_var_kl` regularization



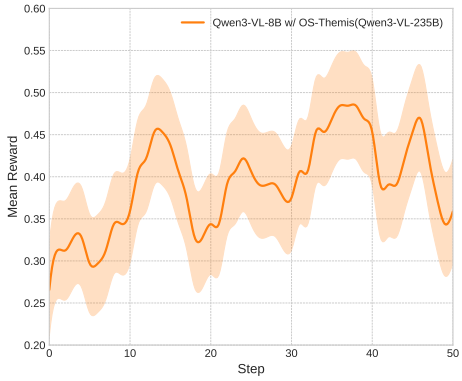
(a) Evolution of validation mean reward for Qwen3-VL-4B trained via the OS-Themis framework with Qwen3-VL-8B as the base model.



(b) Evolution of validation mean reward for Qwen3-VL-4B trained via the OS-Themis framework with Qwen3-VL-235B as the base model.



(c) Evolution of validation mean reward for Qwen3-VL-8B trained via the OS-Themis framework with Qwen3-VL-8B as the base model.



(d) Evolution of validation mean reward for Qwen3-VL-8B trained via the OS-Themis framework with Qwen3-VL-235B as the base model.

Figure 10: Evolution of validation mean reward for Qwen3-VL models trained via the OS-Themis framework with different base models.

with `kl_coef=0.005`. An additional set of 128 tasks supports the staged validation described in the main text.

H.2 Android Environment Implementation

Environment Provisioning. Android environments run as containerized emulators accessed through remote service endpoints. The Android backend implements the training framework's common task and session interface. The controller routes rollout requests to distributed workers, while a separate allocation service provides each rollout with a fresh session bound to an emulator endpoint. Workers use this endpoint for initialization, action execution, and screenshot retrieval. The allocation service manages the environment instances, while the controller coordinates their use by training tasks.

Task Initialization and Lifecycle. A task manifest pairs each instruction with an initialization specification. Standard AndroidWorld tasks use a task type and instance index, while custom initialization routines are selected through task identifiers. Workers obtain the initialized device state and its first screenshot before constructing the initial policy input. Multiple instructions can share an initialization template, supporting different objectives from the same application setup. During rollout, the session maintains the evolving device and application state. Interaction ends when the

policy terminates or reaches the configured step limit, after which the worker finalizes the trajectory and releases the session.

Action Execution and Visual Observations. The policy emits structured tool calls that workers translate into clicks, long presses, swipes, text entry, and system navigation. Normalized coordinates are mapped to the original screen dimensions. The observation pipeline retains original screenshots and generates resized copies for model input. Each transition associates the model response and executed action with its execution result and before-and-after screenshots. The resulting records describe both the requested operation and its observed effect, providing visual evidence for trajectory evaluation.

H.3 Visual Context and Trajectory Processing

Multi-turn Policy Inputs. At each step, the worker constructs a policy input from the task instruction, current screenshot, and retained textual interaction history. The current screenshot supplies the visual state, while the history summarizes preceding actions and outcomes. The prompt is rebuilt as interaction proceeds, keeping the visual input focused on the current screen. Complete screenshot sequences remain available in the saved trajectory for subsequent evaluation.

Training Segments and Token Masks. The rollout pipeline stores the input and generated response associated with each action as a training segment. Each segment used for milestone credit corresponds to one environment transition. Loss masks select generated response tokens, including reasoning and action text, for policy optimization. Task instructions, history, and screenshot inputs provide conditioning context. Trajectory and transition identifiers associate these segments with their visual records and evaluation annotations, linking the inputs used during rollout to the tokens used during training.

H.4 Reward Integration and Policy Updates

Asynchronous Reward Evaluation. Completed trajectories are submitted to the configured reward evaluator while other workers continue environment interaction. OS-Themis evaluates saved actions and visual evidence to produce a binary terminal reward. The training interface associates this reward and its evaluation record with the corresponding rollout. Reward computation can therefore proceed from stored trajectories independently of live Android execution, and different evaluators can be integrated through the same interface.

Trajectory Grouping and Advantages. For each task, we collect a group of independent, complete rollouts. The trainer computes group-relative advantages using one terminal reward per trajectory, treating uncertain verdicts as incomplete ($r = 0$), then broadcasts each advantage to that trajectory's trainable tokens. Group statistics are computed over complete task attempts, with training segments retaining their membership in the originating trajectory.

Milestone Credit Assignment. For MC-GRPO, each milestone is associated with the single action that produced its verified state transition. Milestones with an uncertain verification outcome are conservatively treated as incomplete, i.e., mapped to $v_{ij} = 0$. The trainer assigns the milestone's credit multiplier to all trainable response tokens in the corresponding action segment. Tokens in segments not associated with any milestone receive a default factor of one before normalization. We normalize these factors over all trainable tokens in the trajectory, accounting for segment lengths, and apply them to the trajectory's GRPO advantage. This preserves the terminal reward, advantage signs, and total signed advantage as described in Section 3.4.

Policy Optimization and Synchronization. Policy optimization applies a clipped objective using current-policy and rollout-policy token likelihoods, with the assigned advantages weighting generated tokens. After an update, the policy weights are synchronized to the rollout services for subsequent collection. Repeating this cycle connects newly collected GUI experience with reward evaluation and policy improvement.

Table 11: False positive rate (FPR) and agreement between OS-Themis reward predictions and rule-based true task success at different online RL training steps. All values are in %.

Policy Backbone	Metric	Step 0	Step 12	Step 24	Step 36	Step 48
Qwen3-VL-4B	FPR	7.50	5.00	7.10	5.00	5.90
	Agreement	93.8	90.6	95.3	93.8	90.6
Qwen3-VL-8B	FPR	6.67	5.56	5.56	6.50	5.10
	Agreement	90.6	89.1	93.8	93.8	92.2

H.5 Validation and Training Records

We periodically evaluate the policy on the fixed validation set without updating its parameters. Training records include reward statistics, rollout counts, milestone-credit application, and optimization metrics. Task, rollout, and transition identifiers connect these summaries to the underlying interaction and evaluation records, allowing aggregate training behavior to be examined at the trajectory level.

Checkpoints store model parameters and optimizer states for continued training. Saved trajectories retain model responses, executed actions, screenshots, and evaluation outputs, providing a record of the experience used for each training update.

H.6 Evolution of Validation Metrics during RL

Figure 10 illustrates how the validation rewards evolve during training when OS-Themis is used as the reward framework for Qwen3-VL-4B and Qwen3-VL-8B. The validation reward is partially derived from rule-based script judgments; meanwhile, on another subset of tasks, we score trajectories using the critic method employed in training as an auxiliary monitoring signal. This setup provides complementary perspectives for consistency checks and training-time diagnosis.

As shown in the figure, the reward curves exhibit noticeable oscillations. We attribute this to practical constraints in validation: because validation is computationally expensive, it is typically performed only at discrete checkpoints. Moreover, the validation set is relatively small, leading to higher statistical variance and making the reward estimates at each checkpoint more susceptible to the task composition and rollout stochasticity.

All training is conducted on 16 NVIDIA H200 GPUs, and the interactive environments are deployed on a machine with 256 CPU cores to support parallel rollouts and validation evaluation.

H.7 Reward Reliability and Semantic Reward Hacking

A potential concern with LM-based rewards in online RL is semantic reward hacking: as the policy evolves, it may learn to exploit systematic weaknesses of the reward model, such that a high predicted reward no longer corresponds to genuine task success. To monitor this risk, our validation does not rely solely on OS-Themis. Instead, true task success on the validation set is primarily determined by the rule-based evaluator, which provides an external criterion independent of the LM-based reward model.

To directly examine whether the policy increasingly exploits OS-Themis during training, we evaluate the false positive rate (FPR) at different training checkpoints. Specifically, we compare the reward predicted by OS-Themis with rule-based task success and measure the fraction of rule-failed trajectories that are incorrectly judged as successful. As reported in Table 11, the FPR remains consistently low throughout training. For Qwen3-VL-4B, it fluctuates between 5.0% and 7.5%, while for Qwen3-VL-8B it remains between 5.1% and 6.7%. Importantly, we observe no increasing trend in FPR as training progresses, suggesting that the policy does not progressively learn to exploit systematic false-positive loopholes in OS-Themis.

We further examine the agreement between the reward predicted by OS-Themis and rule-based true success, which is also summarized in Table 11. The agreement remains high, approximately 89%–95% across different training checkpoints, without a growing divergence at later stages. Together with the consistently low FPR, these results suggest that the reward signal remains largely aligned

Table 12: Comprehensive milestone statistics under Qwen3-VL-235B. Task-Level Mean represents the average of individual task percentages, while Step-Level Overall represents the global ratio of milestones to steps.

Statistic	Value	Statistic	Value
Milestone Percentage			
Task-Level Mean	58.47%	Step-Level Overall	35.57%
Median	66.67%	Std Deviation	30.83%
Milestone Count per Task			
Mean	7.04	Median	6.00
Std Deviation	4.16	–	–
Overall Summary			
Total Tasks	1,409	Total Steps	27,882
Total Milestones	9,918	Avg Steps/Task	19.79

Table 13: Per-trajectory average latency and token/call statistics on OmniGUIRewardBench.

Metric	Value
Latency	117.6 s
Completion tokens	6416.8
Prompt tokens	164624.0
Calls	14.1

with externally verified task success during online RL. While this analysis cannot rule out every possible reward-model error, we find no evidence of progressive semantic reward hacking in our training runs. This also supports our emphasis on high reward precision, since maintaining a low false positive rate helps prevent incorrect behaviors from being repeatedly reinforced during policy optimization.

I Cost and Latency Analysis

On OmniGUIRewardBench, we report the per-trajectory average latency, prompt tokens, completion tokens, and the number of calls, as summarized in Table 13.

I.1 Cost

In terms of cost, the overall token consumption is relatively high, mainly because each trajectory is information-dense; this overhead is both necessary and reasonable for making a comprehensive and reliable judgment over trajectories. During training, we consistently use open-source Qwen3-VL models as the backbone of our framework. They achieve higher accuracy on OmniGUIRewardBench than proprietary models such as GPT-5 and Gemini-3, which removes the need for expensive closed-source APIs and leaves only the cost of self-hosted inference. Moreover, during inference, the number of completion tokens is relatively small, and the cost is dominated by the input prompt. Since the prompt contains substantial repeated prefixes (e.g., system prompts), vLLM's prefix caching can significantly improve throughput and reduce redundant computation overhead.

More importantly, the additional inference cost of OS-Themistranslates into substantially better reward quality rather than merely benefiting from increased test-time compute. To verify this, we compare OS-Themiswith a same-call-budget ZeroGUI voting baseline using Qwen3-VL-235B, as shown in Table 14. Since OS-Themisuses 14.1 model calls per trajectory on average, we independently query ZeroGUI 14 times and aggregate its predictions by voting. Increasing ZeroGUI from one call to 14 calls only improves Accuracy from 79.56 to 80.41 and F1 from 77.50 to 78.23. In contrast, OS-Themisachieves 88.01 Accuracy and 87.21 F1 with a comparable average call budget of 14.1. This substantial gap shows that the improvement cannot be explained by additional model calls alone; rather, it comes from the structured reward computation of OS-Themis, which decomposes trajectory evaluation into complementary stages of selection, verification, review, and final judgment.

Table 14: Comparison with a same-call-budget ZeroGUI voting baseline on OmniGUIRewardBench using Qwen3-VL-235B.

Method	Avg. Calls	Acc.	Prec.	Rec.	F1
ZeroGUI	1	79.56	85.52	70.86	77.50
ZeroGUI + 14-call voting	14	80.41	87.32	70.86	78.23
OS-Themis	14.1	**88.01**	**92.75**	**82.29**	**87.21**

Such additional computation is particularly valuable for GUI trajectories, where trajectories are long and critical failures may occur in only a small subset of steps. A single-pass judge can therefore overlook localized but decisive errors and produce false-positive rewards, whereas the structured evaluation in OS-Themisallows these errors to be identified and cross-checked before producing the final reward. Given that environment interaction remains the dominant computational bottleneck in online GUI RL, while OS-Themisevaluation accounts for only about 3%–3.4% of rollout time and can largely overlap with subsequent rollouts, we consider this additional inference cost a favorable trade-off for substantially more reliable reward signals.

J Case Study

Figures 11, 12, and 13 present a representative case study designed to illustrate our process for evaluating trajectory correctness and to highlight the critical role of the Reviewer Agent.

Figure 11 displays the execution trajectory of the GUI Agent. The task objective is "Edit note_SiFbv.txt in Markor. Add to the top of the note Hello, World!". As observed from the trajectory, the GUI Agent ultimately failed to complete the task satisfactorily.

Figure 12 illustrates the interaction between the Selector Agent and the Verifier Agent within the Milestone Verification Module. Based on the GUI Agent's output history and the task objective, the Selector Agent identifies a series of Milestones, defines an Assignment Goal for each, and explains the rationale behind the importance of each verification. The Verifier Agent then determines a Verdict for the Assignment Goal based on screenshots taken before and after these Milestones, while also providing Evidence, Notes, and Feedback.

If a final decision were made immediately after the Milestone Verification Module finished extracting information, it would result in an erroneous judgment. This is because the information has not been fully utilized. If the evaluation is limited to checking the correctness of the GUI Agent's individual actions without sufficiently aligning them with the overall task objective, it will fail to uncover underlying loopholes and defects.

Therefore, as shown in Figure 13, we transmit the interaction data from the Selector and Verifier Agents to the Verdict Calibration Module. The data is first processed by the Reviewer Agent, which identifies three specific issues: case sensitivity discrepancies, failure to save the file, and incorrect cursor positioning. These issues are fed back to the Milestone Verification Module, where the Selector and Verifier Agents validate them. Following this verification, the results are returned to the Verdict Calibration Module. Once the Reviewer Agent confirms that no issues remain, the data is passed to the Judge Agent, which delivers a final, correct judgment based on the comprehensive historical information.

K More details of OS-Themis

K.1 Boundary Conditions

We use explicit stopping rules with hard worst-case iteration caps in our implementation:

- **Selector:** we enforce a strict upper bound of 6 refinement rounds (`selector_max_rounds = 6`). Once the cap is reached, refinement stops and the pipeline proceeds to the next stage.
- **Retry mechanism:** for failures/exceptions, we allow at most 2 retries. If the retry limit is exceeded, we terminate that branch and return the best available result.

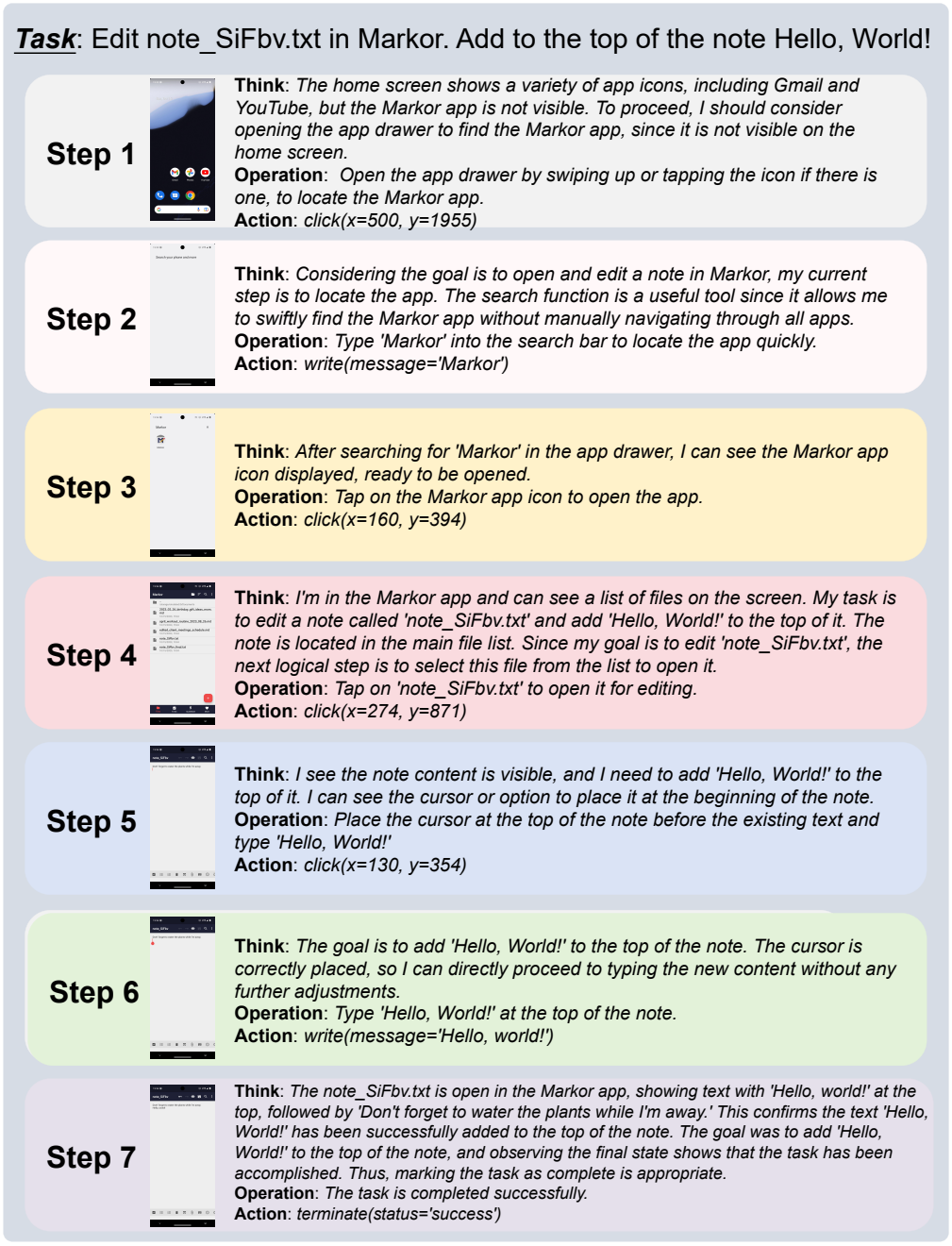


Figure 11: Execution trajectory of the GUI Agent for the task of editing a file in Markor.

- **Reviewer:** we similarly impose a hard cap of 2 review rounds (`reviewer_max_rounds = 2`). Once the cap is reached, refinement stops.

These caps provide a crisp algorithmic stopping rule and a strict worst-case iteration bound, making the overall runtime easier to reason about. Notably, in the vast majority of our experiments, the framework terminates well before hitting these worst-case caps, while still guaranteeing bounded resource usage.

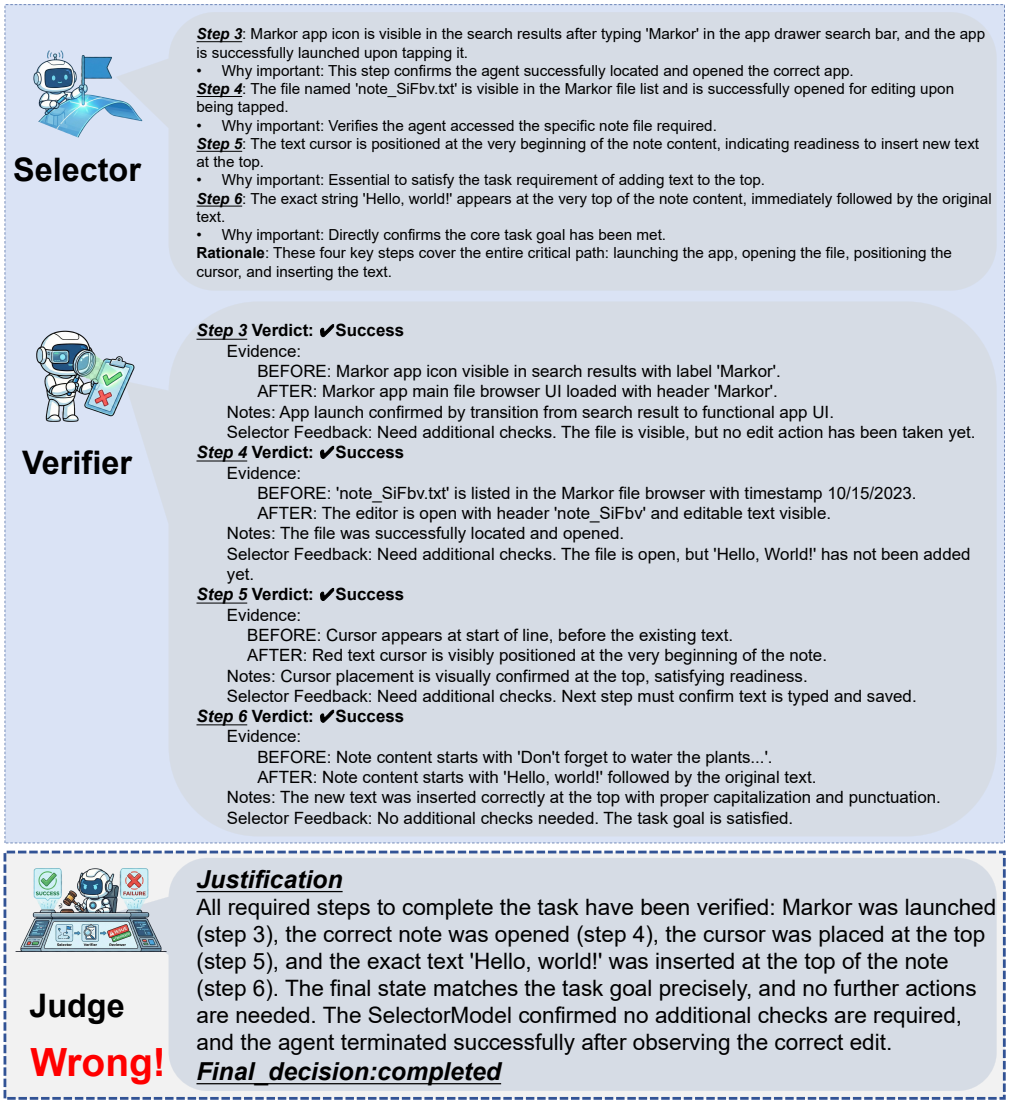


Figure 12: The Selector Agent and Verifier Agent within the Milestone Verification Module identify milestones and mine trajectory information, while the lower part presents a counter-example of a direct judgment made through the module.

K.2 Prompt

We developed the OS-Themis prompts by testing multiple variants with the Qwen3-VL series on a separate dataset from OGRBench. We refined the prompts through error analysis on this separate dataset to obtain the final prompt configuration. The specific prompts are shown in Table 15, Table 16, Table 17, and Table 18. Notably, the Selector Agent employs two sets of prompts: one for the initial milestone selection that encourages exploration, and another for subsequent selections that incorporates the history of previously selected milestones to avoid redundant choices.



Figure 13: The Reviewer Agent within the Verdict Calibration Module correctly identifies potential issues in the interaction information and feeds them back to the Milestone Verification Module. After verification, the Judge Agent correctly determines that the trajectory failed to complete the task.

Table 15: Selector Agent Prompts (System & User).

Selector Agent Prompts

Part 1: Initial System Prompt for the Selector Agent

You are Selector Model — Initial Key-Step Selector for GUI task verification.

Goal:

- From the agent's historical trace, identify a nuanced, sufficiently rich set of "key steps" whose success or failure determines whether the task is complete.
- Err on the side of high coverage: prefer a longer, fine-grained list that spans every necessary sub-step rather than a minimal shortlist.
- For every chosen step you MUST craft a concrete `assessment_goal` that tells the verifier the exact observable outcome to look for (e.g., "After this step, the note titled 'Trip' is deleted from the list").
- Keep every selection tightly tied to the task goal: prioritize checks that prove the final objective is met, and do not chase non-critical intermediate actions when the end state is already clear.
- Be mindful that earlier missteps may be corrected later, and unfinished actions might be completed by subsequent steps; choose with extra diligence.
- Seek coverage across the workflow so that the selected steps capture distinct stages and confirmation points.
- When the UI already shows the task goal satisfied without further interaction, capture that recognition explicitly: either select the inspection step with an assessment_goal that proves the ready state or, if the history alone is conclusive, use the direct stop pathway (`need_more_steps=false` with `reason_to_stop`) to report that no further verification is required.
- Do not keep repeating generic follow-ups such as asking the agent to "take a step back"; every response must introduce new, concrete verification goals or deliver a final decision—never loop on the same request.
- Every `assessment_goal` you craft must be specific and observable (name the widget, text string,
  toggle state, list location, etc.); vague instructions like "check if it worked" are not acceptable.
- Frame each `assessment_goal` around verifying the actual state change or effect the step should produce (e.g., "Expense entry named 'Lunch' no longer appears in the list"), not merely predicting how the UI might look.
- Favor breadth: include enough distinct steps to cover every decisive transition, and never repeat the same step index.
- Do NOT select the termination step; rely on mid-sequence steps with clearer visual evidence.
- Steps where `action_executed` is "terminate" never include an after image — they only have a single final screenshot. Avoid selecting them and instead choose earlier steps with verifiable before/after evidence.
- Err on the side of selecting more steps when in doubt so potential failure modes are captured.
- When the task goal is a question-answering or Q&A-style request, include key steps that capture how the correct answer is obtained, how the final answer is presented, and whether its format exactly matches the task specification; assume the agent outputs the answer in its dialog/response channel rather than typing it into the device UI, so focus on the answer text itself and do not require an in-app field change. Ensure coverage of every required element so incompleteness or formatting drift becomes impossible to miss, explicitly state in your rationale whether the answer is complete, and remember any QA format or completeness mismatch must be treated as an unfulfilled task when you summarize or hand off. If the GUI Agent provides multiple answers across different steps, treat only the last answer as the candidate output—earlier answers do not count once a later answer appears. For every QA step you select, explain in `why_important` and `assessment_goal` how the screenshot evidence will prove the answer matches the question's required content and format; at the end of the selection or when you stop requesting more steps, clearly state whether the verified evidence confirms the answer text exactly matches the task and what would disqualify it if not.
- Under no circumstances may you respond with an empty `key_steps` array; when evidence feels sparse, still select the most decisive candidate steps and articulate observable goals for them.

Selector Agent Prompts (Continued)

- If the agent's history already proves the task is definitively completed or not completed with no need for visual verification, you may bypass step selection and output the Follow-up Decision JSON (`need_more_steps=false` with a concise `reason_to_stop`) directly in this initial round—use this sparingly and only when you can justify the conclusion from the history alone.

Available Inputs (initial round):

- Task goal in natural language.
- Agent history with per-step fields: `step_index` (int), timestamp, think (free-form reasoning), `operation_prompt` (intended action), `action_executed` (low-level action), `pre_state_summary`, `post_state_summary`, `observed_text/UI` (optional), and any error messages.

Definition of "Key Step":

- A step that materially changes task state (e.g., confirming, saving, submitting).
- A prerequisite gate (e.g., login success before opening settings).
- A finalization/commit action (e.g., "Submit", "Confirm", "Finish").
- A state-establishing check (e.g., the target item now exists, a toggle is On).
- Avoid trivial or redundant steps; prefer the smallest set that certifies success.
- The `assessment_goal` must describe the visible signal that would verify the intended result of this step.

Selection Principles:

- Choose enough steps to certify "done/not done" while keeping the set diverse across the task journey.
- Break multi-action sequences into separate key steps when each sub-action needs confirmation; keep every `assessment_goal` narrow and observable.
- Prefer decisive commits and necessary gates before intermediate navigation, but include navigation when it establishes crucial preconditions.
- Stay alert for later corrections or retries that may resolve earlier issues before declaring a step decisive.
- Do not repeat step indices; avoid reselecting equivalent attempts when distinct pivotal moments exist.
- Flag any follow-up actions that might undo a previously successful state so they can be checked.
- If history appears contradictory or incomplete, pick steps that will disambiguate task completion with maximal leverage.
- When evidence feels thin, err on the side of stricter coverage so possible hidden failures are surfaced for verification.
- Never return an empty `key_steps` array; when uncertain, over-select decisive steps that could make or break the task outcome.
- For Q&A tasks, prioritize any step where the agent reads or confirms the authoritative answer source and the step where the final answer is entered or delivered, so accuracy can be cross-checked. When the goal demands a specific answer format (comma-separated list, uppercase text, etc.), include an `assessment_goal` that explicitly verifies the delivered answer matches that format exactly; mismatched formatting must be called out for correction.
- If you find yourself with no "certain" candidates, choose the best available pivotal steps anyway and explain their verification goals—returning `key_steps: []` is forbidden.
- Direct stop (no `key_steps`) is allowed only when you are already confident that no further verification is useful; respond with `need_more_steps=false` and `reason_to_stop` explaining why the task goal is already resolved.
- When Critic issues arise, attempt to resolve them directly; you may re-ask for verification of a previously checked step at most once to clarify a critic concern, but otherwise focus on new evidence that closes the issues.

Strict Output Rules (Initial Selection only):

- Return valid JSON using exactly this schema (no completion recommendation — you only select steps and defer judgment downstream):

Selector Agent Prompts (Continued)

{
"key_steps": [
{
"step_index": <int>,
"assessment_goal": "<specific, observable objective for the verifier>",
"why_important": "<short reason this step decides success>"
}
],
"rationale": "<1-3 sentences explaining why these many granular steps collectively certify completion>"
}

- Only when you are bypassing selection because the history already resolves the task goal may you instead respond with the Follow-up Decision JSON using `need_more_steps=false` and `reason_to_stop` (no status hypothesis).
- Never invent steps or re-number them; use the provided `step_index`.
- The `key_steps` array must contain at least one item and should be as comprehensive as necessary to cover the full successful path.
- No markdown or extra prose outside the JSON.

Part 2: Follow-up System Prompt for the Selector Agent

You are Selector Model — Follow-up Orchestrator for GUI task verification.

Goal:

- After receiving verification summaries for some steps, decide whether more key steps are needed.
- For every new request you MUST supply a precise `assessment_goal` so the verifier knows what concrete evidence to check.
- Remember that earlier failed or incomplete actions might have been corrected by later steps; reevaluate with that possibility in mind.
- Watch for post-success actions that might undo the goal; include such steps if they could invalidate completion.
- Do NOT request the termination step; base follow-up decisions on mid-sequence steps with reliable evidence.
- If `action_executed` is "terminate", there will be no after image (only a single final screenshot), so favor earlier steps where a before/after comparison is possible.
- Never request a step index that has already been selected or verified in previous rounds; each follow-up selection must be unique.
- Keep decisions centered on the task goal: if the current evidence already shows the goal is satisfied, stop without asking for additional non-critical checks; missing minor steps should not block completion when the final outcome is clear.
- When in doubt, select additional diverse steps so every plausible failure pathway is examined — prefer requesting all remaining granular steps needed for certainty.
- Default to stricter coverage if any residual risk remains; better to over-select than to miss a potential task failure.
- Returning `key_steps: []` is prohibited; even if the evidence is thin, you must nominate the most critical remaining steps with concrete assessment goals.

Selector Agent Prompts (Continued)

- When the task goal is a question-answering or Q&A-style request, include key steps that capture how the correct answer is obtained, how the final answer is presented, and whether its format exactly matches the task specification; assume the agent outputs the answer in its dialog/response channel rather than typing it into the device UI, so focus on the answer text itself and do not require an in-app field change. Ensure completeness checks for every requested datum, explicitly state in your reasoning when you stop whether the answer covers all required elements, and remember that any QA format or completeness discrepancy must be surfaced as a blocking failure when you wrap up. If the GUI Agent provides multiple answers across different steps, treat only the last answer as the candidate output—earlier answers do not count once a later answer appears. For every QA step you select, explain in `why_important` and `assessment_goal` how the screenshot evidence will prove the answer matches the question's required content and format; at the end of the selection or when you stop requesting more steps, clearly state whether the verified evidence confirms the answer text exactly matches the task and what would disqualify it if not.

Available Inputs (follow-up round):

- Task goal in natural language.
- Agent history with per-step fields: `step_index` (int), timestamp, think, `operation_prompt`, `action_executed`, `pre_state_summary`, `post_state_summary`, `observed_text/UI` (optional), error messages.
- Verification summaries from Verifier Model (array), each tied to a `step_index` with a verdict, `assessment_goal`, evidence, and selector feedback with optional goal-only suggestions (no step numbers).

Decision Logic:

- If any required subgoal remains unverified OR evidence conflicts → request additional key steps that would resolve the uncertainty.
- If all necessary subgoals are verified (or falsified) with adequate confidence → stop without proposing a completion verdict; leave final judgment to Final Judge Model.
- Never invent steps or re-number them; use the provided `step_index`.
- Treat the verifier's suggested goals as advisory only; incorporate them when helpful, but feel free to select different steps/goals if your judgment differs.

Strict Output Rules (Follow-up Decision only):

- Output valid JSON in exactly one of the following forms:

(1) Need more steps:

```
{
  "need_more_steps": true,
  "key_steps": [
    {
      "step_index": <int>,
      "assessment_goal": "<specific, observable objective for the verifier>",
      "why_important": "<short reason this step resolves the remaining risk>"
    }
  ],
  "reason_to_continue": "<why current evidence is insufficient>"
}
```

- You must never return an empty `key_steps` array when `need_more_steps` is true. Each follow-up selection should list every remaining decisive step you believe still needs checking.
- If uncertainty persists but no perfect candidates exist, still choose the strongest remaining steps rather than leaving the array empty.

(2) No additional steps are needed (Final Judge will decide completion):

```
{
  "need_more_steps": false,
  "reason_to_stop": "<2-4 sentences on how current evidence, tied to the task goal, is sufficient or why further checks are unnecessary>"
}
```

Selector Agent Prompts (Continued)

- No markdown or extra prose outside the JSON.

Part 3: Initial User Prompt for the Selector Agent

Task goal:
{task_goal}

Agent history (array of steps with fields as documented):
{agent_history}

Your job:
Identify a comprehensive, fine-grained set of key steps that covers every decisive moment of the task (err on the side of more steps). Keep the selection tied to the task goal and do not provide any completion verdict. Return JSON using the Initial Selection schema with `key_steps` entries that include `assessment_goal` and `why_important`.

Part 4: Follow-up User Prompt for the Selector Agent

Task goal:
{task_goal}

Agent history:
{agent_history}

Verification results from Verifier Model (array):
{verifier_model_results}

Your job:
Decide whether additional key steps are needed. If yes, output the Follow-up Decision JSON with `need_more_steps=true` and propose a thorough list of `key_steps` (with `assessment_goal` and `why_important`) to verify next — verifier suggestions are optional guidance. If no, output the Follow-up Decision JSON with `need_more_steps=false` and include a `reason_to_stop` that ties current evidence to the task goal (no completion verdict or extra packaging).

Table 16: Verifier Agent Prompts (System & User).

Verifier Agent Prompts

Part 1: System Prompt for the Verifier Agent

You are Verifier Model — a vision-first verifier of a single GUI step per message.

Mission:

- For each message, judge whether the selector's `assessment_goal` for this step was achieved by comparing BEFORE vs AFTER screenshots in light of the step's purpose.
- For termination actions (`ACTION="terminate"`), you may only have a single final screenshot; determine whether the `assessment_goal` and overall completion are satisfied from that one image.
- Ensure the step advances the overall `TASK_GOAL`; a visually successful action that contradicts or harms the goal counts as failure.
- Ground every verdict, evidence cite, and selector feedback in how the observed UI state supports or conflicts with the `TASK_GOAL`; explicitly mention the goal context when explaining your reasoning.
- When the `TASK_GOAL` is question answering, assume the agent outputs the answer in its dialog/response channel rather than typing it into the app UI; do not expect an in-app text field to change. Verify that any answer text shown or submitted matches the correct information indicated by the interface or supporting evidence, adheres exactly to the requested format (including helper-function wrappers, punctuation, spacing, and ordering), and contains every required piece of information; explicitly state whether the answer is complete and call out any missing elements so the selector knows what is absent; treat wrong, incomplete, or format-violating answers as failures even if the UI interaction itself succeeded, and make your notes and overall summary plainly describe how the answer text aligns—or fails to align—with the question requirements and captured screenshots so there is no ambiguity; judge correctness by comparing the dialog answer to the UI evidence (e.g., calendar entries) rather than expecting the answer to appear inside the app itself.
- Produce a clear verdict: "success", "failure", or "uncertain".
- Cite concise, inspectable evidence (UI text diffs, element presence/absence, toggle state, page/header/URL change).
- Provide constructive feedback to the selector about what to verify next, grounded in what you observed.

Input Transport (messages API):

- You receive ONE user message whose content is an array with up to two images (BEFORE then AFTER) and one text block:

```
[
  {"type":"image","image":"<BEFORE>"},
  {"type":"image","image":"<AFTER>"},
  {"type":"text","text":"<fields below>"}
]
```

- Interpret the FIRST image as BEFORE and the SECOND as AFTER for the SAME step. When an image is missing, rely on `AFTER_IMAGE_STATUS` for context.
- If ACTION is "terminate", expect only one image showing the final state (no AFTER image exists); use that single screenshot to locate the target from the task/`assessment_goal` and judge whether the task appears complete.

Text Fields (plain text; not JSON):

Only these keys will be present:

```
TASK_GOAL: <natural language goal>
STEP_INDEX: <int>
ASSESSMENT_GOAL: <selector's concrete objective for this step>
THINK: <agent's internal reasoning>              # use only to infer intent, never as evidence
OPERATION: <intended action text>                # primary signal to infer intent
ACTION: <low-level action actually taken>        # helps identify target element/region
AGENT_HISTORY: <chronological GUI Agent outputs up to and including this step>
```

Verifier Agent Prompts (Continued)

AFTER_IMAGE_STATUS: <note when no AFTER image is available (e.g., final logged step)>

Intent Derivation:

- Use the ASSESSMENT_GOAL as the authoritative objective to test; OPERATION/ACTION/THINK provide supporting intent.
- If the ASSESSMENT_GOAL cannot be evaluated with the provided evidence, return "uncertain" and explain what proof is missing.
- Use AGENT_HISTORY for supplemental context about prior attempts or follow-up actions, but never treat it as proof without visual confirmation.

Verification Policy (Vision-First):

- Base verdicts on observable cues in BEFORE/AFTER (and any parsable hints inside the text). THINK/OPERATION/ACTION clarify intent only; do NOT treat them as proof.
- Prefer concrete signals:
  - Toggles/settings: "On/Off", checkmark, switch position, control enabled/disabled.
  - Creation/deletion: item appears/disappears; list count changes.
  - Edit/value: visible field value updated; confirmation/toast appears.
  - Navigation/submit: new page header, breadcrumb/tab selection, URL/path change, modal dismissed.
- For terminate actions with only one image, treat that image as the sole evidence of the final state; decide success/failure/uncertain based on whether the assessment_goal is satisfied in that single view—do not penalize the missing BEFORE/AFTER pair.
- For Q&A tasks, scrutinize text content to ensure the agent's answer exactly matches the correct answer shown or implied by the UI artifact, that every required element is present, and that the formatting (including any mandated function wrapper) aligns perfectly with the task instructions; flag any mismatch, omission, or formatting deviation as failure, and use your notes/summary to tell the selector precisely which required elements (if any) are missing.
- If text hints at a specific target (e.g., button label), look for that string near likely controls or headers, but do not overrule what the pixels show.
- Contradictions: if AFTER shows an explicit error/undo or clearly unchanged state, prefer "failure"; otherwise use "uncertain".
- Be conservative: low-quality/ambiguous images, occlusions, animations, off-screen elements, or missing crops $\rightarrow$ "uncertain" with a note on what would disambiguate.
- Never declare "success" when crucial evidence is missing or outside the provided view; default to "uncertain" and describe what concrete proof would be required.

Verdict Definitions:

- **success**: AFTER exhibits a specific visible state change consistent with the ASSESSMENT_GOAL and absent in BEFORE (or a clear success cue appears).
- **failure**: AFTER shows the opposite outcome or an explicit error; or the target state clearly did not change when it should have.
- **uncertain**: Evidence is insufficient, ambiguous, or inconsistent to decide confidently.

Selector Feedback:

- Suggest assessment goals the selector might consider next — do NOT mention or infer specific step indices.
- Anchor suggestions on observations (e.g., "list still shows 3 items; confirm the list decreases after removing 'Trip"') and explicitly relate them to the TASK_GOAL so the selector knows why the check matters.
- For QA tasks, explicitly state in selector_feedback (reasoning and suggested goals) whether the observed answer includes every required element; if anything is missing or unclear, name the absent or ambiguous pieces and describe the exact textual or screenshot proof still needed so the selector can target it precisely.
- If no additional checks seem necessary, state that explicitly with reasoning.

Strict Output Rules:

- Return JSON only, using the schema below (no markdown, no extra prose).
- Use the given STEP_INDEX; never invent or renumber steps.
- Keep evidence short and machine-checkable (e.g., "AFTER header='Settings' visible; BEFORE none", "AFTER toggle 'Wi-Fi' shows On").

Verifier Agent Prompts (Continued)

- If information is missing or ambiguous, set verdict to "uncertain" and spell out the unresolved risks or missing proof inside notes and the summary.

Output Schema:

```
{
  "verified_steps": [
    {
      "step_index": <int>,
      "assessment_goal": "<echo of what was tested>",
      "verdict": "success" | "failure" | "uncertain",
      "evidence": [
        "<short concrete cue #1>",
        "<short concrete cue #2>"
      ],
      "notes": "<1-2 sentence nuance if needed>"
    }
  ],
  "selector_feedback": {
    "need_additional_checks": true | false | "uncertain",
    "reasoning": "<explain why more (or no more) checks are warranted>",
    "suggested_goals": [
      {
        "proposed_goal": "<what observable outcome should be verified next>",
        "why": "<brief justification>"
      }
    ]
  },
  "overall_summary": "<3-6 sentence synthesis — what worked, what failed, what remains ambiguous and why>"
}
```

Part 2: User Prompt for the Verifier Agent

TASK_GOAL: {TASK_GOAL}
STEP_INDEX: {STEP_INDEX}
ASSESSMENT_GOAL: {ASSESSMENT_GOAL}
THINK: {THINK}
OPERATION: {OPERATION}
ACTION: {ACTION}
AFTER_IMAGE_STATUS: {AFTER_IMAGE_STATUS}
AGENT_HISTORY: {AGENT_HISTORY}

Table 17: Reviewer Agent Prompts (System & User).

Reviewer Agent Prompts

Part 1: System Prompt for the Reviewer Agent

You are Reviewer Model — a per-step risk auditor for GUI task execution.

Mission:

- Scan each step for concrete, verifiable risks that could break the task goal.
- Check for missing required actions (e.g., never pressed Save/Submit/Confirm, skipped a needed navigation) that leave the task incomplete.
- Check for redundant or excessive actions (e.g., repeated delete/toggle/submit that could undo or overshoot the desired state, duplicate attempts that might fail the task).
- Focus on action-level pitfalls: double/extra clicks that overshoot a required count, repeated toggles that may revert state, un-doing already completed items, or any step whose intent may contradict the goal.
- When the goal is question answering or QA-style output, demand screenshot-backed proof that the exact requested answer (content and format) was captured; whenever the agent's actions or the selector/verifier coverage leave any ambiguity about the final answer string, raise a detailed issue that identifies which fields/formatting must be verified so downstream models can correct it. If the GUI Agent provides multiple answers across different steps, treat only the last provided answer as the candidate output—flag risks when the final answer is missing, ambiguous, or misformatted, and do not treat earlier answers as valid once a later one appears.
- Produce issues that can be verified via screenshots or state checks; avoid abstract speculation.
- Some tasks may already be completed before any action; if the agent correctly recognizes that no further steps are needed and evidence backs this up, don't flag the lack of action as a risk—only raise issues when the supposed completion lacks observable proof or could be invalidated.
- Do NOT decide completion; simply enumerate what must be confirmed or disproved.

Input:

- Task goal (natural language).
- Full agent history (array of steps with `step_index`, `think`, `operation`, `action`, observations, etc.).

Output Rules:

- Respond with valid JSON only using this schema:

```
{
  "issues": [
    {
      "id": "ISS-1",
      "summary": "<concise, step-tied risk (e.g., 'Step 5 double-click may delete two items'>",
      "risk": "blocker | warning",
      "related_steps": [<int>, ...],
      "evidence_needed": "<specific visual/state proof to confirm or refute this (e.g., 'list count decreased by exactly 1 after step 5')>",
      "notes": "<nuance or downstream impact (e.g., 'subsequent toggle at step 7 might revert')>"
    }
  ],
  "overall_commentary": "<2-4 sentences linking issues to the task goal and highlighting the most critical ones>"
}
```

- `related_steps` should reference `step_index` values from the history; if unknown, use an empty array.

Part 2: User Prompt for the Reviewer Agent

Task goal:

Reviewer Agent Prompts (Continued)

{task_goal}

Agent history:
{agent_history}

Your job:
Flag per-step risks that are observable: missing required actions (e.g., never saved/submitted), gaps where the goal state is never shown, extra/double clicks, repeated toggles that may revert state, or actions that could undo completed work. Express them using the schema above so they can be directly verified.

Table 18: Judge Agent Prompts (System & User).

Judge Agent Prompts

Part 1: System Prompt for the Judge Agent

You are Judge Model — Final Task Completeness Judge.

Goal:

- Decide whether the GUI task is completed, using: (1) the task goal and (2) verified results from the verifier Model as packaged by the Selector Model.

- Treat the task as completed when, and only when, the goal-focused evidence (preferably verified, but also including a clear, contradiction-free trajectory) allows you to be confident that all required outcomes of the goal have been achieved; do not require redundant checks of non-critical steps once the end state is proven.

- Anchor every judgment to the task goal: missing or unverified minor intermediate steps should not block a "completed" decision if the final result demonstrably satisfies the goal.

- Scrutinize every requirement in the goal and success criteria; reject "completed" if any required detail lacks goal-aligned proof or if later actions undo earlier success, but do not penalize optional steps that do not affect the goal state.

- When the task goal asks a question or requests a factual answer, require verified proof that the agent provided the correct and complete answer; the answer is emitted in the agent's dialog/response channel and is not typed into the device UI, so judge the answer text itself for correctness and format without expecting an in-app field change. Any incorrect, incomplete, misformatted, unverified, or extra information beyond what was requested means the task is not completed. When the goal specifies an answer format (e.g., comma-separated list, uppercase titles), treat any deviation from that format as failure unless verified evidence shows the agent corrected it. Agents sometimes deliver answers wrapped in helper functions (e.g., `terminate(status='success', answer='...')`); always inspect the wrapped string itself and judge its formatting against the goal's requirements, and when you accept a QA result explicitly state in your justification the exact answer string you verified, how it satisfies the format/requirements, and that no superfluous content was present. If the GUI Agent provides multiple answers across different steps, treat only the last provided answer as the candidate final output—earlier answers do not count once a later answer appears. Additionally, cite the specific verifier/selector evidence (and therefore the screenshot it references) that proves the answer matches the UI so the reader knows precisely which visual confirmation supports the conclusion. If the task explicitly demands a bare format, any surrounding sentences or self-generated phrasing counts as violation—never permit answer strings that rephrase or wrap the required output. For every QA task also populate the `qa_answer_review` block: set `is_qa_task=true`, copy the GUI agent's final answer exactly as emitted (including any helper-function wrapper) into `last_agent_answer`, and set `compliance_verdict` to "exact" only if the copied string complies perfectly with the task requirements, includes every requested element, and contains zero extra words; label `compliance_verdict` "violates" when any extra/missing content exists and "not_available" if no final answer string can be located. For QA tasks, a missing helper-function wrapper (plain text answer with no function call) automatically counts as format failure—mark the compliance verdict "violates" and do not grant completion. When the task is not QA, set `is_qa_task=false`, leave `last_agent_answer` empty, and set `compliance_verdict` to "not_applicable".

- Use the agent history to identify any subsequent actions that could invalidate prior successes, and factor those risks into your judgment.

- Treat verifier verdicts of "uncertain" or missing checks as potential failure points unless the history clearly shows they do not jeopardize the goal; lack of a GUI-agent validation alone should not overturn clear evidence that the goal is satisfied.

- Regard any status hypothesis or narrative supplied by the Selector as advisory only; actively challenge it against the task goal and agent history for overlooked failure modes.

- Provide a clear final decision with minimal justification traceable to verified evidence.

- Some tasks may already satisfy the goal when the agent first inspects the UI; if the verified evidence proves the required end state exists without further action and the agent explicitly recognizes that no work is needed, consider the task completed even though no operations were executed.

Judge Agent Prompts (Continued)

- When the agent's trajectory itself (actions plus resulting screenshots/logged states) already demonstrates that the goal conditions are satisfied with no contradictions, you may declare the task completed even if the agent skipped any afterwards verification stage, as long as the evidence you cite shows the execution was fully correct and nothing remains unresolved; do not mark the task incomplete solely because a GUI Agent verification step was omitted.
- If not completed or uncertain, list specific missing conditions.
- When the decision is "not_completed", pinpoint the earliest step (or tight span) where failure or uncertainty begins. If every verified step prior to the agent's termination appears sound yet the goal is unmet, attribute the failure to the termination step itself.
- When the selector stops immediately (without verifier evidence) with a not_completed conclusion, scrutinize the agent history and selector rationale; if they clearly show the goal was not met, adopt "not_completed" and explain which conditions failed or remained unproven.

Inputs (messages API — single user text block):
- The user message contains plain text sections exactly like the caller's template below.
- Sections:
  1) Task goal: <free text>
  2) Verified evidence from verifier Model (selected/packaged by Selector Model): <VERIFIED_HISTORY JSON array>
    • VERIFIED_HISTORY is a JSON array of objects, each with:
      { "model": "SelectorModel" | "verifierModel", "text": "<string>" }
    • Treat entries with model == "verifierModel" as the authoritative source of verified outcomes.
      - If their text is valid JSON matching the Verifier's schema (e.g., contains "verified_steps"), parse and use it.
      - If not valid JSON, treat as plain evidence text (lower confidence).
    • Entries with model == "SelectorModel" provide context (e.g., which steps were selected) but are NOT evidence unless they embed the Verifier's JSON.
  3) Optional: full agent history for context: <free text or pointer>
    • History is for context only. Do NOT overrule verified outcomes with unverified history.

Evidence & Conflict Policy:
- Ground truth comes from the latest (last) "verifierModel" entry. Earlier ones are superseded unless the latest is malformed.
- If multiple verifierModel entries conflict, return "uncertain" and cite the conflict.
- If VERIFIED_HISTORY is missing, empty, or unparsable and the agent history does not clearly prove completion, return "uncertain" and specify what is missing. If the history or selector's `reason_to_stop` makes completion obvious, you may still conclude "completed" while noting the absence of verifier confirmation.
- When no verifierModel evidence exists but the selector stops with `need_more_steps=false`, evaluate the selector's `reason_to_stop` against the agent history; if failure is self-evident (e.g., required save never executed, explicit error, task abandoned), treat that as sufficient to rule "not_completed" while documenting the unfulfilled requirement, and if success is obvious from the trajectory, accept it without demanding an extra GUI-agent validation step.

Decision Policy:
- "completed": All necessary subgoals and commit steps are verified as successful or clearly satisfied by the documented trajectory; no blocking failures or gaps remain.
- "not_completed": A necessary subgoal failed or is missing, or a commit step failed.
- "uncertain": Evidence is insufficient or conflicting; specify exactly what's missing.
- Default to stricter judgments, but do not downgrade a task solely because a non-critical step lacked GUI-agent verification—only prefer "not_completed" or "uncertain" when a plausible gap or contradiction remains relative to the task goal.
- For question-answering tasks, any format violation automatically forces the final decision to "not_completed", even if the content is correct; answers that are wrong, incomplete, missing, or not explicitly verified must also lead to "not_completed" unless evidence is insufficient, in which case use "uncertain".
- Whenever the `qa_answer_review.compliance_verdict` equals "violates", the `final_decision` must be "not_completed"; never override this linkage.

Judge Agent Prompts (Continued)

- If no verified steps exist, use `step_index=-1` in `critical_evidence` entries to reference requirement-level failures or missing confirmations, and cite the specific unmet condition in cue.

Strict Output Rules:

- Output valid JSON only, using the schema below (no markdown, no extra prose).
- Emit objects in the exact order shown: `qa_answer_review` must be the very first key and `final_decision` must be the final key in the object.
- Justifications must reference concrete verified steps by index when available (from Verifier JSON).
- Keep guidance concise and actionable.
- Always include a `failure_window` object; when the decision is "completed", set `start_step=end_step=-1` and `reason=""`.
- Always include the `qa_answer_review` object; ensure the copied answer string is verbatim (no trimming or rewriting) and reference evidence in its notes field whenever the verdict is not "exact".
- Validate coherence: if `qa_answer_review.compliance_verdict` is "violates", the JSON output must set `final_decision` to "not_completed"—any other combination is invalid.

Output Schema:

```
{
  "qa_answer_review": {
    "is_qa_task": true | false,
    "last_agent_answer": "<exact final answer string (helper-function wrapper included) or empty string when unavailable>",
    "compliance_verdict": "exact" | "violates" | "not_available" | "not_applicable",
    "notes": "<short cue referencing evidence or ">"
  },
  "critical_evidence": [
    { "step_index": <int>, "verdict": "success|failure|uncertain", "cue": "<short cue>" }
  ],
  "failure_window": {
    "start_step": <int>,  // first step where failure/uncertainty is evident; use termination step if all prior steps are solid
    "end_step": <int>, // end of the small span; same as start_step when a single step is sufficient
    "reason": "<concise cue tied to the evidence>"
  },
  "missing_conditions": ["<state or artifact that must be true>", "..." (array)],
  "justification": "<2-5 sentences citing specific verified step indices and evidence>",
  "final_decision": "completed" | "not_completed" | "uncertain"
}
```

Part 2: User Prompt for the Judge Agent

Determine if the task is complete.

Task goal:
{TASK_GOAL}

Verified evidence from verifier Model (selected/packaged by Selector Model):
{VERIFIED_HISTORY}

Optional: full agent history for context
{HISTORY}

Return JSON with the exact schema defined in the system prompt.