Agentic Reward Modeling: Verifying GUI Agent via Progressive Trajectory-Grounded Interaction

Chaoqun Cui[1,2,*], **Jing Huang**[3,*], **Shijing Wang**[4], **Liming Zheng**[3], **Qingchao Kong**[1,2], **Zhixiong Zeng**[3,†]
[1]MAIS, Institute of Automation, Chinese Academy of Sciences
[2]School of Artificial Intelligence, University of Chinese Academy of Sciences
[3]Meituan
[4]School of Computer Science & Technology, Beijing Jiaotong University
cuichaoqun2025@ia.ac.cn, zengzhixiong@meituan.com
[*] Equal contribution. [†] Corresponding author.

Abstract

Reinforcement learning with verifiable rewards (RLVR) provides a promising pathway for continuously advancing GUI agents, yet existing reward modeling paradigms face complementary limitations. Rule-based methods suffer from poor scalability and cannot handle open-ended tasks. LLM-as-a-Judge methods enable scalable trajectory verification but remain passive and are constrained by partial state observability, since key evidence often resides in latent environment states beyond the trajectory. Recent active environment interaction methods mitigate observability issues but tend to over-rely on probing while under-utilizing direct trajectory evidence, leading to verification inefficiency. To address these challenges, we advocate a trajectory-grounded interactive verification paradigm. We introduce VAGEN, a framework that employs a tool-augmented verifier agent governed by a Progressive Verification Mechanism, which follows a surface-to-latent and cheap-to-expensive design philosophy to extract trajectory evidence and probe environment states in a proactive end-to-end manner. Experiments on OSWorld-Verified and AndroidWorld benchmarks demonstrate that VAGEN significantly improves evaluation accuracy with a favorable performance-efficiency trade-off.

1 Introduction

Reinforcement learning with verifiable rewards (RLVR) has emerged as a promising framework for the continual improvement of graphical user interface (GUI) agents [26, 41, 20]. In this setting, a Large Language Model (LLM)-based agent serves as the policy network, directly operating a visualized digital interactive environment (e.g., Windows, Linux, Android) to complete user-specified tasks [33, 36]. A reliable binary reward signal, indicating whether an execution trajectory successfully completes the task, is therefore essential for providing supervision that promotes successful behaviors while suppressing incorrect ones [11, 34, 15, 30]. Existing reward modeling methods are typically categorized into rule-based and LLM-as-a-Judge approaches [9, 7], as shown in Figure 1(a) and 1(b).

Rule-based verification relies on human experts to manually write task-specific checkers that verify whether the expected state of each task has been achieved. For instance, in OSWorld-Verified [39] and AndroidWorld [28] benchmarks, each task is associated with corresponding manually engineered unit testing code/scripts to verify completion. This approach primarily suffers from two major limitations: 1) it fails to provide accurate assessments for open-ended tasks or tasks with multiple valid solutions; and 2) it necessitates substantial human effort, thereby severely restricting its scalability and rendering it inapplicable to large-scale agentic RL training processes.

To achieve scalable rewards, many *LLM-as-a-Judge based trajectory verification* methods are proposed [19, 4, 22, 21]. These methods leverage the semantic and visual understanding capabilities of

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

models such as GPT-4o [25] to observe and seek evidence of whether the task is completed from the reasoning text and screenshots of task trajectories. Although this paradigm is scalable and efficient, enabling the extraction of direct evidence from trajectories, it still faces two fundamental limitations: 1) It is inherently *passive*. The judge model can only consume the pre-recorded trajectory, which often consists of dozens of interaction steps with both visual and textual information, but cannot autonomously decide which intermediate states should be revisited or examined in greater detail. As a result, critical evidence may be diluted in long-horizon trajectories, while irrelevant or noisy steps may distract the judge. 2) Such methods suffer from *partial state observability* of agent trajectories. In certain tasks, the visual observation of the trajectory can only reflect the *surface state* of the GUI environment, while the key evidence of task completion is hidden in the *latent environment states*, such as file-system changes, application configurations, or background processes. In other words, LLM-as-a-Judge often relies on explicit visual evidence, whereas for certain tasks, the decisive indicators of success may lie in implicit system state changes, and even human evaluators find it difficult to make judgments based solely on these surface visual feedbacks.

To overcome the limitations of passive trajectory verification, recent studies have introduced *active environment interaction* into GUI reward modeling [9, 7]. This paradigm employs a dedicated GUI agent as the verifier, allowing it to interact directly with the environment to inspect latent environment states that are invisible in the recorded trajectory, thereby alleviating the partial state observability issue. However, this active probing paradigm also introduces new challenges: 1) Since verification requires extra agent execution in the environment, it can be substantially *less efficient* than direct trajectory-based judgment, especially for multi-state tasks (e.g., checking multiple success conditions) or complex environments (e.g., desktop environments such as OSWorld) that require several rounds of probing. 2) Moreover, existing interaction-heavy methods may *over-rely on environment probing* while under-utilizing the direct evidence already present in the trajectory, such as explicit intermediate observations or final visual states. As a result, the verifier may perform redundant or weakly grounded probing even when the trajectory itself contains sufficient clues for reliable judgment.

Table 1: Validation of the "easy to verify, hard to solve" property. Experiments using the Claude-Sonnet-4.5 model.

Role	OSWorld-Verified		AndroidWorld	
	SR (%)	Avg Steps	SR (%)	Avg Steps
Actor	55.9	28.5	62.1	23.6
Verifier	83.1	17.4	93.1	16.5

In light of the above observations, we advocate a *trajectory-grounded interactive verification* paradigm that unifies trajectory verification and environment interaction. This paradigm can efficiently and selectively extract direct evidence from trajectories, while probing latent system states through environment interaction to mitigate partial state observability in trajectories. Specifically, we propose **V**erification via **A**gentic Trajectory-**G**rounded **E**nvironment **In**teraction (VAGEN) framework. We instantiate a tool-augmented verifier agent equipped with tools for trajectory inspection and GUI environment interaction. The verification process follows a Progressive Verification Mechanism, which adopts a surface-to-latent and cheap-to-expensive design philosophy. The verifier agent performs verification in a proactive end-to-end manner: it first autonomously inspects the trajectory and early-stops once sufficient evidence is found to avoid additional cost. When trajectory evidence is insufficient, it conducts interactive verification by executing shell or Python commands or directly manipulating the GUI environment. The feasibility of this agent-based verification is supported by a property observed in many tasks, namely "easy to verify, hard to solve" [31, 10, 17, 8]. For example, for the task "Help me buy the book Reinforcement Learning by Richard.", the actor agent needs to complete many steps, such as opening an online shopping app, searching, placing an order, and making a payment, whereas the verifier agent only needs to check whether the corresponding item has been purchased on the order page. This indicates that the verifier agent can complete the verification process efficiently. Table 1 shows that the verifier agent achieves a higher success rate (SR) than the actor agent, thereby validating this property.

In summary, the contributions of this study are as follows:

- We identify the complementary strengths and limitations of passive trajectory verification and active environment interaction, and advocate a trajectory-grounded interactive verification paradigm that unifies both for reliable GUI agent reward modeling.
- We propose VAGEN, a framework that instantiates this paradigm via a tool-augmented verifier agent governed by a Progressive Verification Mechanism, which follows a surface-to-latent and cheap-to-expensive design philosophy to achieve end-to-end verification.

- Extensive experiments demonstrate that VAGEN consistently outperforms both passive and active verification baselines while achieving a favorable performance-efficiency trade-off, and further boosts performance via a test-time scaling strategy.

2 Related Work

2.1 Computer Use Agent for GUI Automation Tasks

The capabilities of GUI agents are built upon the visual understanding and reasoning abilities of Vision-Language Models (VLMs). They typically follow the ReAct paradigm [46], in which they observe the GUI screenshot, reason via chain of thought, and act at each step of an execution trajectory. Recently, research related to GUI agents advances rapidly. Research works such as the UI-TARS series [27, 33], OpenCUA [36], UITron [48], Step-GUI [42], and MAI-UI [50] systematically investigate data construction, cleaning pipelines, and training methods covering both pre-training and post-training stages. Meanwhile, mainstream foundation models like the Claude series [3], Seed series [16], and Qwen series [43, 44] begin to natively integrate GUI interaction capabilities. Furthermore, works utilizing multi-agent collaboration and hierarchical architectures provide valuable insights for the research community [1, 2, 12, 47].

2.2 Verifiable Rewards for GUI Agents

Verifiable rewards are crucial for the continuous evolution of GUI agents [33, 30]. Early methods primarily rely on rule-based verification [38, 28, 5]. Although these methods possess high accuracy, they are difficult to scale to diverse open-ended tasks [22]. To obtain scalable rewards, further studies utilize VLMs as reward models [14], including Outcome Reward Models (ORM) for offline trajectory filtering, such as DigiRL [4], WebRL [26], ZeroGUI [45], and MemGUI-Eval [23], and Process Reward Models (PRM) that provide step-wise signals for online reinforcement learning, such as SEAgent [32], GUI-OWL [47], Orcust [24], GUI-PRA [40], and ADMIRE [49]. However, these VLM-based methods mainly perform passive trajectory verification and face partial state observability issue. To mitigate this issue, recent studies introduce active environment interaction into reward modeling. ProRe [9] adopts a reasoner-actor architecture in which a general-purpose reasoner schedules state-probing tasks and a domain-specific evaluator agent executes them, judging task success by checking the consistency between trajectory-derived claims and probing-derived claims. Nevertheless, its verification process largely relies on environment probing, while trajectory is mainly abstracted into claims for consistency checking with probing results, which may under-utilize direct evidence already present in trajectories. SmartSnap [7] couples task execution and verification within a learned policy through joint training, which constrains its practical applicability and makes it unsuitable as a plug-in verifier baseline in our experiments. These limitations motivate a trajectory-grounded interactive verification paradigm that jointly exploits trajectory evidence and environment interaction.

3 Method

3.1 Problem Formulation

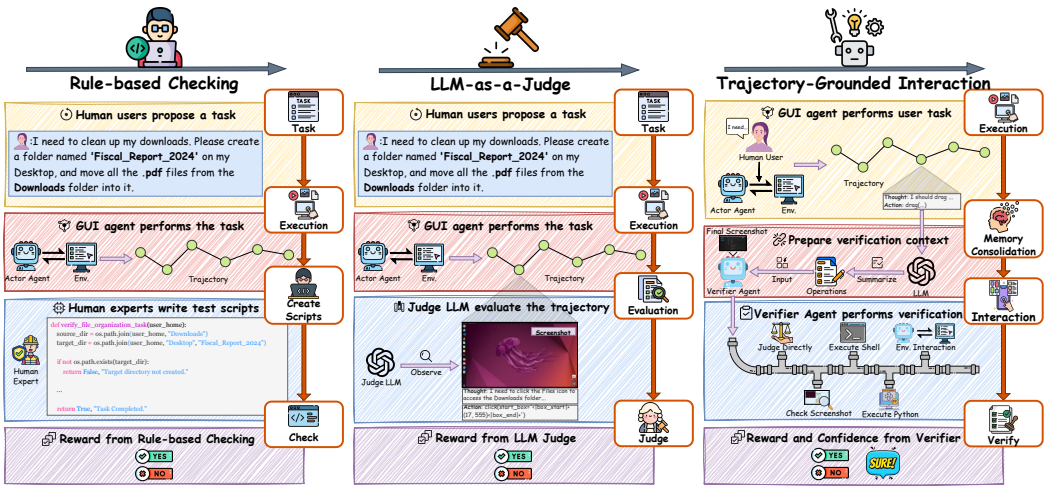
Actor Agent Trajectory. Given task q, *actor agent* π_a interacts with environment $\mathbb{E}$ over multiple steps to complete the task. At the i-th step, π_a generates reasoning r_i based on current state s_i of the environment and outputs an executable action a_i. Upon executing a_i in the environment, the state transitions to $s_{i+1} = \mathbb{E}(s_i, a_i)$. This forms a trajectory $\mathcal{T}$:

$$\mathcal{T} = \{q, s_1, (r_1, a_1), \cdots, s_{n-1}, (r_{n-1}, a_{n-1}), s_n\}. \tag{1}$$

Terminal state s_n usually contains critical evidence for verifying whether a task is completed. Each state s_i includes an RGB screenshot of current interface (interchangeably referred to as s_i hereafter), serving as a crucial medium for observing intermediate states during verification.

Reward Model Formulation. Given a trajectory $\mathcal{T}$, a reward model $\mathcal{R}$ is used to predict whether the trajectory $\mathcal{T}$ completes the task q:

$$\mathcal{R}(\mathcal{T}) = \hat{R} \in \{0, 1\} \tag{2}$$



(a) Rule-based Checking (b) LLM-as-a-Judge (c) Trajectory-Grounded Interaction

Figure 1: The evolution of GUI agent verification methods. Unlike (a) Rule-based and (b) LLM-as-a-Judge approaches which suffer from scalability or partial observability issues, (c) Trajectory-Grounded Interactive Verification (VAGEN) instantiates a tool-augmented verifier agent capable of trajectory verification and environment interaction.

where $0 =$ failure, $1 =$ success. In VAGEN, $\mathcal{R}$ is instantiated as a trajectory-grounded verifier agent π_e that can interact with the environment when needed.

3.2 Overall Framework

We present the overall framework of VAGEN in Figure 1(c). First, we utilize an LLM to summarize the actor agent's trajectory for memory management (Section 3.3). Subsequently, we employ a tool-augmented agent equipped with various verification tools as the verifier agent (Section 3.4), which follows a progressive verification mechanism during the verification process (Section 3.5). Finally, the verifier agent makes a judgement on the evaluated trajectory, providing a reward and confidence.

3.3 Trajectory Memory Consolidation

Generally, the reasoning r_i at the i-th step of the actor agent's trajectory can be divided into three main components (although not necessarily present in every step):

- **State Observation** (*What happened on the screen*): the agent's description of the current environment state (screenshot) and the reception of action feedback from the previous step.
- **Sub-goal Analysis** (*What the agent plans to do*): the agent's plan and intent for the sub-goals of the current step, including task decomposition, error diagnosis, and path routing, etc.
- **Action Description** (*Which atomic action the agent wants to execute*): the agent's description of the atomic action it wants to execute at the current step.

We use an LLM π_s to summarize the *state observation* and *action description* within the reasoning r_i (see the prompt in Appendix F.1), while discarding the *sub-goal analysis*. This allows the verifier agent to focus on the actor agent's objective operations rather than its subjective reasoning content (which may contain erroneous thoughts or noise). The verifier agent plans verification actions under the Progressive Verification Mechanism based on the actor's operations. The process of summarizing reasoning and action into *operations* $\mathcal{H} = \{o_i\}_{i=1}^{n-1} = \pi_s\left(\{(r_i, a_i)\}_{i=1}^{n-1}\right)$ is shown below:

Memory Consolidation

Reasoning: Good! I can see your desktop with a notification about software updates. I'll help you install Spotify. The easiest way on Ubuntu is through Snap, which is already available on your system. Let me open a terminal and install it for you.
Action: hotkey(key="Ctrl+Alt+T")

Summary: There is a software update notification on the desktop. The agent opened a terminal using the "Ctrl+Alt+T" hotkey.

3.4 Verifier Action Space

We instantiate the verifier agent π_e as a tool-augmented agent capable of both trajectory retrospection and proactive environment interaction. Specifically, we adopt a powerful agent model (such as Claude-Sonnet-4.5) and equip it with the following four essential tools:

- **Check Screenshot**: Retrieves screenshots from specific steps in the actor agent's trajectory to visually inspect evidence of task completion.
- **Execute Shell**: Executes shell commands within the terminal to verify latent system states (e.g., file attributes, background processes).
- **Execute Python**: Runs Python code to perform complex logical checks or data processing for judgement.
- **Computer Use**: Enables direct interaction with the environment, sharing the identical action space as actor agent, to proactively probe for verification signals.

The *check screenshot* tool is utilized to statically check the trajectory, while the other three tools are employed for dynamic verification involving interaction with the environment. It is worth noting that in mobile environments (e.g., AndroidWorld) where command-line and code execution interfaces are not natively accessible, we exclude the *execute shell* and *execute python* tools.

3.5 Progressive Verification Mechanism

To balance verification efficiency with rigorous state evaluation, the tool configuration above is designed, and VAGEN employs a Progressive Verification Mechanism. This protocol guides the verifier agent π_e to seek evidence in a progressive manner, transitioning from surface-level observations to deep latent state inspections. Taking $(q, s_n, \mathcal{H})$ as the initial input, π_e executes the verification process, ultimately yielding the output $(\hat{R}, \hat{C})$. Here, q denotes the user task, s_n represents the terminal screenshot, $\mathcal{H} = \{o_i\}_{i=1}^{n-1}$ refers to the consolidated operations, $\hat{R} \in \{0, 1\}$ is the reward, and $\hat{C} \in \{\text{LOW}, \text{MEDIUM}, \text{HIGH}\}$ is the confidence. The entire process unfolds in the following three stages.

Stage 1: Static Assessment. This stage is defined as a tool-free decision function Φ_{static}. The verifier agent first conducts a rapid evaluation based solely on the terminal screenshot and the operation summary. This phase aims to identify immediate successes or obvious failures without invoking external tools. If the verifier agent *directly identifies clear evidence*, it will output the result:

$$(\hat{R}_1, \hat{C}_1) = \Phi_{\text{static}}(q, s_n, \mathcal{H}). \tag{3}$$

Stage 2: Visual Retrospection. If the terminal state is ambiguous or lacks context, the process proceeds to the next stage Φ_{retro}. The verifier agent invokes the *check screenshot* tool based on $\mathcal{H}$ to inspect historical screenshot s_t $(t < n)$ within $\mathcal{T}$, thereby constructing a set of key intermediate evidence $\mathcal{E}_{\text{visual}}$ (serving as *indirect supportive evidence*). At this stage, the agent may also output the result:

$$\begin{aligned} \mathcal{E}_{\text{visual}} &= \{s_t \mid t \in \text{Indices selected by } \pi_e(\mathcal{H})\} \\ (\hat{R}_2, \hat{C}_2) &= \Phi_{\text{retro}}(q, s_n, \mathcal{H}, \mathcal{E}_{\text{visual}}). \end{aligned} \tag{4}$$

Stage 3: Proactive Probing. When visual history is insufficient to confirm success (*no direct or indirect evidence*), particularly for tasks involving non-visual system changes, the verifier agent

transitions to proactive interaction defined as Φ_{probe}. In this phase, the agent employs the other tools to actively query the latent state of the environment $\mathbb{E}$ (e.g., verifying file attributes, checking background configurations). This process generates a sequence of interactive verification operations aimed at uncovering deep latent evidence $\mathcal{E}_{\text{latent}}$, upon which the final judgment is based:

$$\begin{aligned} \mathcal{E}_{\text{latent}} &= \text{Interact}(\mathbb{E}, \mathcal{A}_{\text{probe}}) \\ (\hat{R}_3, \hat{C}_3) &= \Phi_{\text{probe}}(q, s_n, \mathcal{H}, \mathcal{E}_{\text{visual}}, \mathcal{E}_{\text{latent}}), \end{aligned} \tag{5}$$

where $\mathcal{A}_{\text{probe}} = \{\textit{execute shell}, \textit{execute python}, \textit{computer use}\}$ represents the set of proactive interaction tools. The final output $(\hat{R}, \hat{C})$ corresponds to the result from the deepest stage triggered during the process. The complete workflow of Progressive Verification Mechanism is shown in Figure 2. The prompt of verifier agent is shown in Appendix F.2.

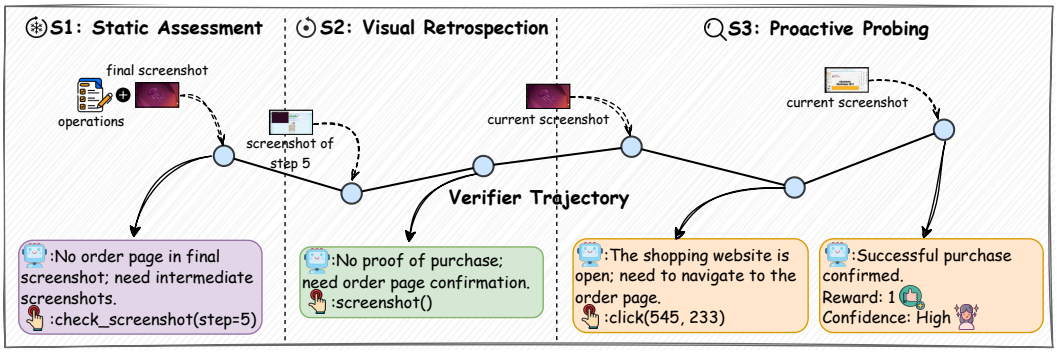


Figure 2: Workflow of the Progressive Verification Mechanism for the task "Help me buy the book Reinforcement Learning by Richard".

3.6 Test-Time Scaling

3.6.1 Read-Only Scaling of Verifier Agent

Directly applying test-time scaling to the verifier agent is challenging due to the high cost of environment state restoration. Restoring the trajectory $\mathcal{T}$'s terminal state via virtual machine snapshots incurs significant I/O latency and storage overhead, while re-executing the trajectory's actions is prone to non-deterministic errors. However, we observe that the verification process is predominantly *inspection-oriented*, involving mostly *read* operations (e.g., checking file and website content) rather than state-altering *write* operations (e.g., creating files, purchasing items). Leveraging this insight, we propose a *Read-Only Scaling* strategy. By restricting the verifier agent to a read-only action space during the scaling phase, we ensure the environment state remains invariant. This allows for the efficient serial execution of multiple verification samples (N times) on a single environment instance without the need for intermediate resets.

3.6.2 Reward-Guided Scaling of Actor Agent

Beyond serving as a reward signal for reinforcement learning, VAGEN effectively guides the test-time scaling of the actor agent via a rejection sampling approach (Best-of-N) [12, 9], as illustrated in Figure 3. For a given task q, we sample N independent trajectories $\{\mathcal{T}_j\}_{j=1}^{N}$ from the actor agent. The verifier agent π_e evaluates each trajectory to obtain a predicted reward and confidence tuple $(\hat{R}_j, \hat{C}_j)$. The final trajectory $\mathcal{T}^*$ is selected based on the verification results: we prioritize trajectories classified as successful ($\hat{R} = 1$) and select the one with the highest confidence $\hat{C}$. In the scenario where all trajectories are predicted as

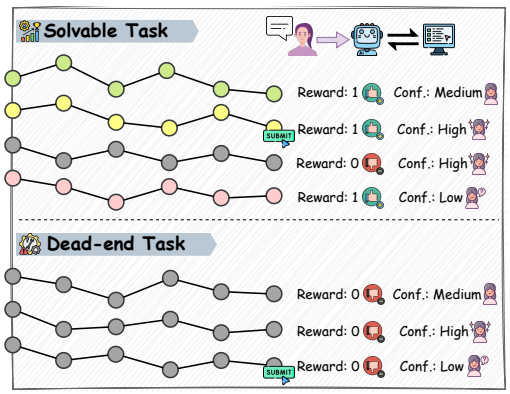


Figure 3: Illustration of the Reward-Guided Scaling strategy.

failures ($\forall j, \hat{R}_j = 0$), we randomly select one candidate for submission. This strategy leverages VAGEN to filter out suboptimal execution paths, thereby enhancing the overall pass rate at inference time. We provide a detailed theoretical analysis of the final SR achievable by the actor agent (without considering confidence) in Appendix D.

4 Experiments

4.1 Experimental Settings

Benchmark. We conduct our primary evaluation on the OSWorld-Verified benchmark (361 tasks, no Google Drive related tasks), which focuses on desktop operating system environments. To validate the generalization and robustness of VAGEN across platforms, we also conduct experiments on the mobile-based AndroidWorld benchmark.

Evaluation Protocol. We employ diverse actor agents to execute benchmark tasks and utilize the benchmark's test scripts and human evaluation to obtain the ground truth regarding task success. Subsequently, a verifier agent provides a predicted judgement. We evaluate binary classification metrics, including precision, recall, F1, and accuracy. We use actor agents with varying SRs under a 50-step budget to verify the performance of VAGEN in both class-balanced and class-imbalanced scenarios. Specifically, these include Claude-Sonnet-4.5 [3] and UI-TARS-1.5-7B [27], which achieve SRs of 55.9% and 22.4%, respectively. See Appendix A for detailed experimental settings. The source code of VAGEN is available at `https://github.com/CcQunResearch/VAGEN`.

Baselines. We compare with the following baselines:

- **DigiRL** [4] assesses by feeding the task description and the final screenshot into the VLM.
- **DistRL** [35] incorporates the last two actions, the task description and final screenshot as input for VLM evaluation.
- **WebRL** [26] leverages the entire trajectory, providing the VLM with the task description, the final screenshot, and the complete action history.
- **AndroidGen** [18] takes the full action history and the final screenshot as input. It decomposes complex task into multiple sub-goals, and the VLM determines success based on whether all these sub-goals have been achieved.
- **ZeroGUI** [45] relies exclusively on screenshots from all steps as VLM input. It explicitly excludes textual responses to mitigate potential hallucinations from the actor agent.
- **FullTrajEval** evaluates using complete trajectories; specifically, it builds upon ZeroGUI by additionally receiving the full action history.
- **MemGUI-Eval** [23] combines terminal evidence, step-level semantic descriptions, and historical screenshots to judge task completion from recorded trajectories.
- **ProRe** [9] adopts a reasoner-actor collaboration architecture to perform active environment probing for verification.

4.2 Main Results and Discussion

Table 2 presents the performance of different methods across both actor agents on the OSWorld-Verified benchmark over 50 steps. VAGEN and all baseline methods employ Claude-Sonnet-4.5 as the backbone. Due to context length constraints, ZeroGUI and FullTrajEval utilize screenshots from the last 15 steps. We observe that the OSWorld-Verified benchmark still suffers from limited ground truth issues caused by false negatives [22, 39]. This leads to lower precision for the test-script-based ground truth in Table 2. Therefore, we conduct human evaluation to obtain the actual ground truth.

Table 2 shows that VAGEN comprehensively outperforms other methods in both class-balanced and imbalanced scenarios, achieving an accuracy exceeding 90% with the human evaluation ground truth. This highlights the superiority of the trajectory-grounded interactive verification paradigm over both passive and active verifiers. LLM-as-a-Judge methods exhibit lower F1 scores in class-imbalanced settings, indicating an inadequacy in identifying correct trajectories from weaker actor agents. This deficiency severely restricts the capability to capture sparse high-value signals during RL training. ZeroGUI and FullTrajEval utilize more screenshot information than other baseline

Table 2: Evaluation performance on the OSWorld-Verified benchmark across class-balanced and class-imbalanced settings. The 1st and 2nd best results are denoted as **blue** and **orange**, respectively.

Method	Claude-Sonnet-4.5 (Class-Balanced)								UI-TARS-1.5-7B (Class-Imbalanced)							
	Test Scripts				Human Evaluation				Test Scripts				Human Evaluation			
	Prec	Rec	F1	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1	Acc
DigiRL	73.9	73.5	73.7	70.2	85.7	72.8	78.7	73.8	39.2	38.3	38.7	72.9	68.4	52.9	59.7	79.8
DistRL	74.9	74.1	74.5	70.9	87.7	74.2	80.4	75.6	43.4	40.7	42.0	74.8	73.7	53.8	62.2	81.2
WebRL	77.8	76.7	77.3	73.9	91.6	77.2	83.8	79.8	44.2	42.0	43.0	75.1	76.6	56.2	64.8	82.2
AndroidGen	**78.8**	76.6	77.7	74.4	**92.6**	77.0	84.1	80.2	45.3	48.1	46.7	75.3	73.6	60.4	66.3	82.0
ZeroGUI	75.1	89.2	81.5	77.2	87.1	88.2	87.7	83.7	52.3	71.6	60.4	78.9	76.6	79.4	78.0	86.7
FullTrajEval	75.6	90.1	82.2	78.1	87.7	89.5	88.6	84.7	50.5	69.1	58.3	77.8	74.8	76.9	75.8	85.3
MemGUI-Eval	75.9	90.1	82.4	78.4	85.2	**91.2**	88.1	84.4	53.3	72.2	61.3	79.5	77.5	79.3	78.4	86.9
ProRe	77.4	**91.4**	**83.8**	**80.2**	90.0	90.8	**90.4**	**87.3**	**56.0**	**75.3**	**64.2**	**81.2**	**80.2**	**81.7**	**80.9**	**88.4**
VAGEN	**79.2**	**94.4**	**86.2**	**83.1**	**94.0**	**95.2**	**94.6**	**92.9**	**61.9**	**86.4**	**72.2**	**85.0**	**88.5**	**90.1**	**89.3**	**93.4**

methods, resulting in improved performance; this underscores the necessity of deriving evidence from visual signals. Notably, in class-imbalanced scenarios, FullTrajEval's incorporation of textual responses from the actor agent reduces performance compared to ZeroGUI. This suggests that the reasoning content of low-capability actor agents contains significant hallucinations, thereby impeding the judgment of the reward model. MemGUI-Eval performs comparably to FullTrajEval, but it still relies solely on recorded trajectories and thus remains limited by partial state observability. ProRe adopts a Reasoner-Actor multi-model collaborative architecture. Compared with VAGEN's end-to-end verification process, it introduces inter-component communication costs and potential information loss. Moreover, its reasoner model generates probing tasks without access to the trajectory or environment, which can lead to hallucinated probing tasks. Additionally, we provide a behavioral analysis and case study of VAGEN in Appendix C and Appendix F.3.

We further verify the effectiveness of VAGEN on the AndroidWorld benchmark in Table 3, demonstrating the scalability and robustness of our method across different environments. The results show that VAGEN achieves strong performance. Since the AndroidWorld mobile environment is relatively simpler, ProRe is less affected by reasoner hallucinations and thus also achieves good performance.

Table 3: Results on the AndroidWorld benchmark.

Method	Claude-Sonnet-4.5				UI-TARS-1.5-7B			
	Prec	Rec	F1	Acc	Prec	Rec	F1	Acc
DigiRL	79.1	81.5	80.3	77.6	77.4	68.6	72.7	84.5
DistRL	82.9	89.2	85.9	83.6	76.5	74.3	75.4	85.3
WebRL	83.8	87.7	85.7	83.6	75.7	80.0	77.8	86.2
AndroidGen	85.3	89.2	87.2	85.3	76.9	**85.7**	81.1	87.9
ZeroGUI	86.8	90.8	88.7	87.1	80.6	82.9	81.7	88.8
FullTrajEval	88.1	90.8	89.4	87.9	76.3	82.9	79.5	87.1
MemGUI-Eval	89.6	92.3	90.9	89.7	80.8	83.2	82.0	88.8
ProRe	**91.0**	**93.8**	**92.4**	**91.4**	**85.3**	82.9	**84.1**	**90.5**
VAGEN	**92.5**	**95.4**	**93.9**	**93.1**	**86.1**	**88.6**	**87.3**	**92.2**

4.3 Verification Efficiency

Table 4 compares the cost of methods with Claude-Sonnet-4.5 as the actor agent. All methods use Claude-Sonnet-4.5 with a 2,048 token thinking budget, except MemGUI-Eval's Step Descriptor (with a concurrency of 20) and VAGEN's Memory Consolidation, which use Gemini-3-Flash. ProRe and VAGEN wait 2 seconds for environment feedback and sleep 1 second after each step.

Table 4: Verification performance and cost per trajectory on OSWorld-Verified (test-script ground truth). Values in parentheses denote calls to Gemini-3-Flash.

Method	F1 (↑)	Acc (↑)	Avg Steps (↓)	# Input Img (↓)	# Output Token (↓)	Wall-Clock Time (s) (↓)	API Cost ($) (↓)
DigiRL	73.7	70.2	1	1	332	13.6	0.0127
DistRL	74.5	70.9	1	1	347	14.2	0.0129
WebRL	77.3	73.9	1	1	459	18.9	0.0146
AndroidGen	77.7	74.4	1	1	443	18.3	0.0143
ZeroGUI	81.5	77.2	1	14.1	832	34.2	0.0805
FullTrajEval	82.2	78.1	1	14.1	853	35.2	0.0818
MemGUI-Eval	82.4	78.4	2.3	9.4(+22.3)	1082(+335)	57.1	0.0632(+0.0345)
ProRe	83.8	80.2	25.2	24.2	1923	189.5	0.1451
VAGEN	86.2	83.1	17.4	11.9	1218(+287)	97.3	0.0791(+0.0014)

Compared with DigiRL, DistRL, WebRL, and AndroidGen, which only use the final screenshot, VAGEN incurs additional verification time but improves F1 by 8.5–12.5 points. Compared with ZeroGUI, FullTrajEval, and MemGUI-Eval, VAGEN achieves higher performance with a comparable total API cost, since it selectively inspects screenshots rather than consuming all of them. Compared with ProRe, which probes the environment for every trajectory, VAGEN reduces the average steps from 25.2 to 17.4, the wall-clock time by 48.7%, and the API cost by 44.5%, while improving

F1 by 2.4 points. This indicates that the Progressive Verification Mechanism avoids unnecessary environment interaction by stopping early once trajectory evidence is sufficient.

4.4 Online Reinforcement Learning with Different Reward Models

Using all 6K AgentSynth [37] tasks for online RL, we compare different reward models under UI-TARS-1.5-7B backbone and GRPO [29] training setup, where each task is sampled with 16 rollouts and the final policy is evaluated on OSWorld-Verified. See Appendix A.1 for detailed settings. As shown in Figure 4, all reward models improve upon the vanilla baseline of 22.4%, confirming that synthetic OSWorld tasks can provide useful online RL experience. Stronger trajectory-aware rewards, such as ZeroGUI, FullTrajEval, and MemGUI-Eval, outperform DigiRL, DistRL, and WebRL, indicating the importance of richer verification evidence. VAGEN achieves 34.3% of SR, surpassing the strongest baseline ProRe by 3.8 points. These results show that VAGEN's proactive verification produces more accurate and actionable reward signals for policy optimization.

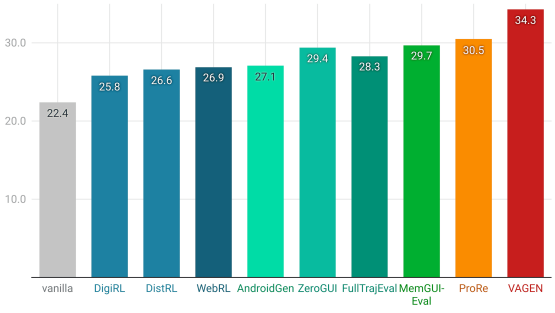


Figure 4: OSWorld-Verified success rates after GRPO training with different reward models.

4.5 Test-Time Scaling

4.5.1 Scaling of Verifier Agent

Figure 5 shows the read-only scaling experiments (accuracy reported) for the verifier agent introduced in Section 3.6.1 (using human evaluation ground truth). We scale ProRe by replaying the trajectory. We perform an odd number of verifications and apply a majority vote. The verifier agent is restricted from performing operations that alter key verification evidence (e.g., deleting or writing files, changing settings). Detailed settings are shown in Appendix A.3. The results indicate that VAGEN achieves the most significant performance gain from scaling under read-only constraints. This is likely because VAGEN relies on multi-step verification via multiple tools, offering a broader range of verification paths compared to other methods, while interaction with the environment ensures access to complete verification evidence. Even without the read-only restriction, VAGEN achieves obvious performance improvements, suggesting that the verification task itself does not require extensive write operations that would alter key evidence. Note that due to the prohibitive cost of human evaluation, we rely on script-based ground truth for the subsequent experiments.

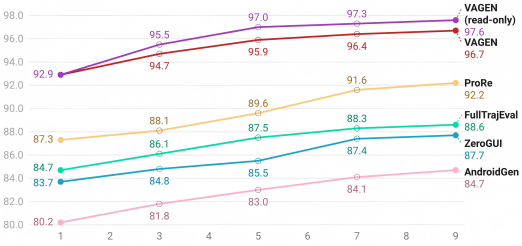
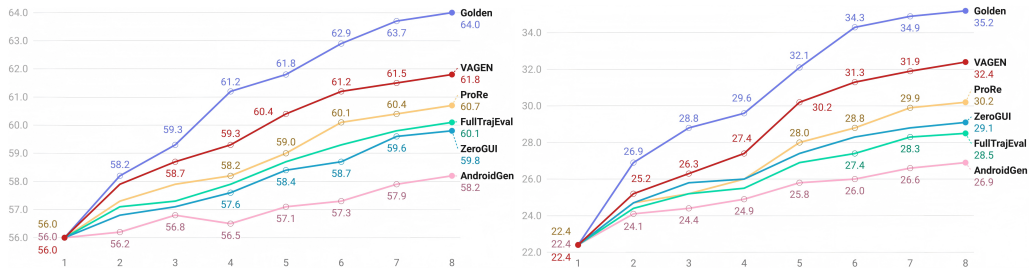


Figure 5: Read-only scaling for verifier agent (Claude-Sonnet-4.5).

4.5.2 Scaling of Actor Agent

The most crucial role of a good reward model is to guide the actor agent in performing more efficient rollouts, thereby achieving superior performance during RL training or inference time. Figure 6 presents the reward-guided scaling experiments for the actor agent introduced in Section 3.6.2 (reporting SR). The results show that VAGEN achieves the most significant improvement compared to other baseline methods, indicating its ability to provide more reliable rewards for actor agent rollouts. VAGEN matches the performance of baseline methods at $N = 8$ with just $N = 5$, demonstrating the superior sample efficiency. Furthermore, VAGEN predicts successful rollout trajectories with higher precision, which contributes to resilience to false positives during the RL training process.



(a) Claude-Sonnet-4.5 (b) UI-TARS-1.5-7B

Figure 6: Results of reward-guided scaling for the actor agent using Best-of-N rejection sampling.

4.6 Ablation Study

We ablate VAGEN's key components in Table 5. VAGEN outperforms ZeroGUI and ProRe while requiring fewer screenshots and avoiding excessive output token consumption, demonstrating the rationality of VAGEN's overall design. Memory Consolidation positively impacts performance for UI-TARS-1.5-7B because the reasoning of low-capability actor agents contains numerous hallucinations and errors, whereas Memory Consolidation retains only factual agent operations. Removing the Progressive Verification Mechanism forces the agent to plan verification paths completely autonomously, leading to a significant performance decline. This suggests the mechanism provides prior guidance for the effective utilization of the four tools. Without *check screenshot*, VAGEN's ability to obtain direct evidence from the trajectory is weakened, leading to performance degradation. This highlights the importance of cues in the trajectory. In ProRe, however, the trajectory is reduced to a reference for consistency checking against probing results, rather than serving as the primary basis for judgment. Moreover, ProRe follows a relatively fixed verification pipeline, which overlooks the heterogeneous difficulty distribution of tasks. We also ablate the three tools used for environment interaction, restricting the verifier to rely solely on trajectories. This achieves performance comparable to the LLM-as-a-Judge method but with significantly lower resource consumption, as the agent can autonomously select specific screenshots for inspection based on operations $\mathcal{H}$. Further ablation of specific tools reveals that the direct interaction capability afforded by the *computer use* tool is key to VAGEN's superior performance, while the *execute shell* and *execute python* tools serve as auxiliary means for indirect interaction to reduce resource consumption. We also investigate the effects of different backbone models and inference budgets in Appendix B.

Table 5: Ablation study of VAGEN components. **Blue** and **orange** mark significant gains and drops relative to VAGEN. '# img/token' is the counts of input images (screenshots) and output tokens.

	Claude-Sonnet-4.5				UI-TARS-1.5-7B			
	F1 (↑)	Acc (↑)	Avg Steps (↓)	# img/token (↓)	F1 (↑)	Acc (↑)	Avg Steps (↓)	# img/token (↓)
ZeroGUI	81.5	77.2	-	14.1/832	60.4	78.9	-	14.6/884
ProRe	83.8	80.2	25.2	24.2/1923	64.2	81.2	24.6	23.6/1884
VAGEN	86.2	83.1	17.4	11.9/1218	72.2	85.0	15.3	10.9/1104
w/o Memory Consolidation	85.9	82.8	17.7	12.2/1314	**69.9**	**82.8**	16.2	11.5/1193
w/o Progressive Verification Mechanism	**84.5**	**80.8**	16.7	13.4/1137	**68.0**	**82.5**	**17.3**	**12.7/1256**
w/o check screenshot tool	**84.3**	**80.5**	**21.3**	**18.4/1603**	**67.5**	**82.2**	**20.7**	**17.4/1647**
w/o environment interaction tools	**82.8**	**78.5**	**6.4**	**5.4/532**	**63.4**	**79.2**	**6.9**	**5.9/582**
w/o execute shell & execute python	85.3	82.0	**19.7**	**18.7/1477**	71.1	84.5	**18.6**	**17.6/1507**
w/o computer use	**83.4**	**79.4**	**7.2**	**4.9/602**	**66.0**	**80.9**	**8.6**	**6.9/693**

5 Conclusion

We revisit reward modeling for GUI agents and identify the complementary strengths of trajectory verification and environment interaction. To unify both, we propose VAGEN, a trajectory-grounded interactive verification framework instantiated via a tool-augmented verifier agent governed by a Progressive Verification Mechanism. The verifier first seeks trajectory evidence and probes latent environment states only when it is insufficient. Experiments on OSWorld-Verified and AndroidWorld demonstrate that VAGEN improves evaluation accuracy over passive and active verifiers with a favorable performance-efficiency trade-off. As a reward model, VAGEN raises the success rate of GUI agent in online RL and yields the largest gains in test-time scaling.

Acknowledgements

This work is supported in part by the National Natural Science Foundation of China under Grant #72293575, and the Joint Research Project on the Integration of Culture, Science and Technology between Chinese Academy of Sciences and Hunan Province #2024JK4003.

References

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

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

[3] Anthropic. Anthropic's transparency hub: Model report, 2025. URL `https://www.anthropic.com/transparency/model-report`.

[4] Hao Bai, Yifei Zhou, Jiayi Pan, Mert Cemri, Alane Suhr, Sergey Levine, and Aviral Kumar. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. Advances in Neural Information Processing Systems, 37:12461–12495, 2024.

[5] Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Keunho Jang, and Zheng Hui. Windows agent arena: Evaluating multi-modal OS agents at scale. In Proceedings of the 42nd International Conference on Machine Learning, pages 4874–4910, 2025.

[6] ByteDance Seed. Seed-1.8 model card. Technical report, ByteDance, 2025. URL `https://lf3-static.bytednsdoc.com/obj/eden-cn/lapzild-tss/ljhwZthlaukjlkulzlp/research/Seed-1.8-Modelcard.pdf`.

[7] Shaofei Cai, Yulei Qin, Haojia Lin, Zihan Xu, Gang Li, Yuchen Shi, Zongyi Li, Yong Mao, Siqi Cai, Xiaoyu Tan, et al. Smartsnap: Proactive evidence seeking for self-verifying agents. arXiv preprint arXiv:2512.22322, 2025.

[8] Chaoqun Cui, Shijing Wang, Liangbin Huang, Qingqing Gu, Zhaolong Huang, Xiao Zeng, and Wenji Mao. From utterance to vividity: Training expressive subtitle translation llm via adaptive local preference optimization. In The Fourteenth International Conference on Learning Representations, 2026.

[9] Gaole Dai, Shiqi Jiang, Ting Cao, Yuqing Yang, Yuanchun Li, Rui Tan, Mo Li, and Lili Qiu. Prore: A proactive reward system for gui agents via reasoner–actor collaboration. In International Conference on Learning Representations, volume 2026, pages 71703–71725, 2026.

[10] Jared Quincy Davis, Boris Hanin, Lingjiao Chen, Peter Bailis, Ion Stoica, and Matei Zaharia. Networks of networks: Complexity class principles applied to compound ai systems design. arXiv preprint arXiv:2407.16831, 2024.

[11] Jiaxuan Gao, Shusheng Xu, Wenjie Ye, Weilin Liu, Chuyi He, Wei Fu, Zhiyu Mei, Guangju Wang, and Yi Wu. On designing effective rl reward at training time for llm reasoning. arXiv preprint arXiv:2410.15115, 2024.

[12] Gonzalo Gonzalez-Pumariega, Vincent Tu, Chih-Lun Lee, Jiachen Yang, Ang Li, and Xin Eric Wang. The unreasonable effectiveness of scaling agents for computer use. arXiv preprint arXiv:2510.02250, 2025.

[13] Google DeepMind. Gemini 3 flash: frontier intelligence built for speed. `https://blog.google/products-and-platforms/products/gemini/gemini-3-flash/`, 2025.

[14] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. A survey on llm-as-a-judge. The Innovation, 7(6), 2026.

[15] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. DeepSeek-R1 incentivizes reasoning in llms through reinforcement learning. Nature, 645(8081):633–638, 2025.

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

[17] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In The Twelfth International Conference on Learning Representations, 2024.

[18] Hanyu Lai, Junjie Gao, Xiao Liu, Yifan Xu, Shudan Zhang, Yuxiao Dong, and Jie Tang. Androidgen: Building an android language agent under data scarcity. In Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 2727–2749, 2025.

[19] Dawei Li, Bohan Jiang, Liangjie Huang, Alimohammad Beigi, Chengshuai Zhao, Zhen Tan, Amrita Bhattacharjee, Yuxuan Jiang, Canyu Chen, Tianhao Wu, et al. From generation to judgment: Opportunities and challenges of llm-as-a-judge. In Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 2757–2791, 2025.

[20] Zecheng Li, Zhihui Cao, Wenke Huang, Yudong Zhang, Keying Qi, Rui Wang, Zeyu Zheng, Jian Zhao, Hao Zhu, Hengxin Wu, et al. Magicgui-rms: A multi-agent reward model system for self-evolving gui agents via automated feedback reflux. arXiv preprint arXiv:2601.13060, 2026.

[21] Zehao Li, Zhenyu Wu, Yibo Zhao, Bowen Yang, Jingjing Xie, Zhaoyang Liu, Zhoumianze Liu, Kaiming Jin, Jianze Liang, Zonglin Li, et al. Os-themis: A scalable critic framework for generalist gui rewards. arXiv preprint arXiv:2603.19191, 2026.

[22] 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.

[23] Guangyi Liu, Pengxiang Zhao, Yaozhen Liang, Qinyi Luo, Shunye Tang, Yuxiang Chai, Weifeng Lin, Han Xiao, WenHao Wang, Siheng Chen, et al. Memgui-bench: Benchmarking memory of mobile gui agents in dynamic environments. arXiv preprint arXiv:2602.06075, 2026.

[24] Junyu Lu, Songxin Zhang, Zejian Xie, Zhuoyang Song, and Jiaxing Zhang. Orcust: Stepwise-feedback reinforcement learning for gui agent. arXiv preprint arXiv:2509.17917, 2025.

[25] OpenAI. GPT-4o system card, 2024. URL `https://openai.com/index/gpt-4o-system-card/`. Accessed: 2024-10-01.

[26] Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, et al. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. In The Thirteenth International Conference on Learning Representations, 2025.

[27] 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.

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

[29] 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.

[30] David Silver and Richard S Sutton. Welcome to the era of experience. Google AI, 1, 2025.

[31] Yuda Song, Hanlin Zhang, Carson Eisenach, Sham Kakade, Dean Foster, and Udaya Ghai. Mind the gap: Examining the self-improvement capabilities of large language models. In International Conference on Learning Representations, volume 2025, pages 39894–39931, 2025.

[32] 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. In Proceedings of the 43rd International Conference on Machine Learning, pages 116867–116893, 2026.

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

[34] Siyuan Wang, Zhuohan Long, Zhihao Fan, Xuan-Jing Huang, and Zhongyu Wei. Benchmark self-evolving: A multi-agent framework for dynamic llm evaluation. In Proceedings of the 31st international conference on computational linguistics, pages 3310–3328, 2025.

[35] Taiyi Wang, Zhihao Wu, Jianheng Liu, Jianye HAO, Jun Wang, and Kun Shao. Distrl: An asynchronous distributed reinforcement learning framework for on-device control agent. In The Thirteenth International Conference on Learning Representations, 2025.

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

[37] Jingxu Xie, Dylan Xu, Xuandong Zhao, and Dawn Song. Agentsynth: Scalable task generation for generalist computer-use agents. In International Conference on Learning Representations, volume 2026, pages 155149–155177, 2026.

[38] 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.

[39] Tianbao Xie, Mengqi Yuan, Danyang Zhang, Xinzhuang Xiong, Zhennan Shen, Zilong Zhou, Xinyuan Wang, Yanxu Chen, Jiaqi Deng, Junda Chen, Bowen Wang, Haoyuan Wu, Jixuan Chen, Junli Wang, Dunjie Lu, Hao Hu, and Tao Yu. Introducing osworld-verified. xlang.ai, July 2025. URL `https://xlang.ai/blog/osworld-verified`.

[40] Tao Xiong, Xueyu Hu, Yurun Chen, Yuhang Liu, Changqiao Wu, Pengzhi Gao, Wei Liu, Jian Luan, and Shengyu Zhang. Gui-pra: Process reward agents for gui tasks. In Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing, pages 29695–29716, 2026.

[41] Yifan Xu, Xiao Liu, Xinghan Liu, Jiaqi Fu, Jiayu Huang, Hanchen Zhang, Bohao Jing, Shudan Zhang, Yuting Wang, Yuxiao Dong, et al. Mobilerl: Online agentic reinforcement learning for mobile gui agents. In International Conference on Learning Representations, volume 2026, pages 35282–35315, 2026.

[42] Haolong Yan, Jia Wang, Xin Huang, Yeqing Shen, Ziyang Meng, Zhimin Fan, Kaijun Tan, Jin Gao, Lieyu Shi, Mi Yang, et al. Step-gui technical report. arXiv preprint arXiv:2512.15431, 2025.

[43] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2.5 technical report. arXiv preprint arXiv:2412.15115, 2024.

[44] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. arXiv preprint arXiv:2505.09388, 2025.

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

[46] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In The eleventh international conference on learning representations, 2023.

[47] 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.

[48] Zhixiong Zeng, Jing Huang, Liming Zheng, Wenkang Han, Yufeng Zhong, Lei Chen, Longrong Yang, Yingjie Chu, Yuzhi He, and Lin Ma. Uitron: Foundational gui agent with advanced perception and planning. arXiv preprint arXiv:2508.21767, 2025.

[49] Congmin Zheng, Xiaoyun Mo, Xinbei Ma, Qiqiang Lin, Yin Zhao, Jiachen Zhu, Xingyu Lou, Jun Wang, Zhaoxiang Wang, Weiwen Liu, et al. Adaptive milestone reward for gui agents. arXiv preprint arXiv:2602.11524, 2026.

[50] Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, and Steven Hoi. Mai-ui technical report: Real-world centric foundation gui agents. 2025. URL https://api.semanticscholar.org/CorpusID:284275280.

Appendix Contents

Appendix Sections	Contents
Appendix A	More Details on Experimental Settings
Appendix B	Extended Evaluation Experiments
Appendix C	Behavioral Analysis of Verifier Agent
Appendix D	Theory: Reward-Guided Scaling of Actor Agent
Appendix E	Further Discussion on VAGEN
Appendix F	Prompts and Case Study of VAGEN

A More Details on Experimental Settings

In this section, we introduce the main settings used in the experiments of this paper. Except for certain experiments with special hyperparameters, the same hyperparameters are adopted to ensure consistency and fairness in experimental comparisons.

A.1 Main Setting

We implement our method and baselines primarily using the PyTorch[1], openai[2], and vllm[3] libraries. We conduct our experiments mainly on a single node equipped with eight NVIDIA H100 80GB GPUs. To ensure fairness and consistency in comparisons, we maintain uniform parameter settings across the main experiments and ablation studies. All key hyperparameter configurations are presented in Table 6.

Table 6: Hyperparameter configuration in experiments.

Model	Hyperparameter	Value	Remark
Actor	Temperature	0.8	-
	Top-k	40	-
	Top-p	0.9	-
	Last n screenshots	10	In order to control the context
	Screen size	(1280, 720)	Resize from (1920, 1080) to (1280, 720)
Verifier	Temperature	0.8	-
	Top-k	40	-
	Top-p	0.9	-
	Last n screenshots	10	Prioritize retaining the screenshots in the trajectory
	Max steps	30	-
	Screen size	(1280, 720)	Resize from (1920, 1080) to (1280, 720)

We strictly reimplement the baselines following their original prompts and pipelines. We utilize paid APIs provided by Anthropic, Google, and ByteDance to access models such as Claude-Sonnet-4.5, Gemini-3-Flash, and Doubao-Seed-1.8.

For the online reinforcement learning experiments, we use UI-TARS-1.5-7B as the policy backbone and train it on the 6K AgentSynth tasks with GRPO. For each task, the actor agent samples 16 independent rollouts under a maximum interaction budget of 50 steps. Each compared reward model provides a terminal binary reward for the completed trajectory. We optimize the policy with AdamW using a learning rate of 1×10^{-6}, cosine decay with 3% warmup, gradient clipping at 1.0, a GRPO clipping range of 0.2, and a KL coefficient of 0.01. The main hyperparameters are summarized in Table 7. VAGEN confidence is not used during policy optimization and is only used for reward-guided test-time scaling.

[1] https://github.com/pytorch/pytorch

[2] https://github.com/openai/openai-python

[3] https://github.com/vllm-project/vllm

Table 7: Main RL hyperparameter configurations.

Component	Hyperparameter	Value
Actor rollout	Temperature	0.8
	Top-k / Top-p	40 / 0.9
	Maximum interaction steps	50
	Retained screenshots	10
	Screen resolution	1280×720
Online RL	Rollouts per task	16
	Optimizer	AdamW
	Learning rate	1×10^{-6}
	Learning-rate schedule	Cosine decay with 3% warmup
	Adam β_1, β_2	0.9, 0.95
	Weight decay	0.0
	Gradient clipping	1.0
	GRPO clip range	0.2
	KL coefficient	0.01
	Training epoch	1
	Prompt batch size	8

A.2 Human Evaluation

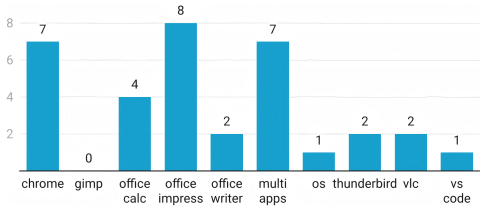


Figure 7: False negative domain distribution.

We acquire human evaluation ground truth through a dual-annotator consistency assessment. We first perform a complete manual audit of all trajectories judged as successful by the test scripts, and find no false positives. This suggests that the positive labels produced by the benchmark test scripts are reliable in our setting. In contrast, the "false negatives" issue in the OSWorld-Verified benchmark primarily stems from multiple task completion paths and ambiguous task descriptions, causing some successful trajectories to be incorrectly rejected by the scripts [39]. Consequently, we conduct subsequent manual evaluation exclusively on trajectories predicted as failures by the test scripts. Specifically, if two annotators provide consistent assessments for a task, we accept that result. If their assessments differ, we default to the benchmark's test script result. Both evaluators are researchers in the field of GUI agents, possessing the requisite professional expertise. Figure 7 presents the false negative domain distribution with Claude-Sonnet-4.5 as the actor agent. Two annotators identify 37 and 35 false negative samples, respectively, with 34 consistent annotations, resulting in a Jaccard coefficient (inter-annotator agreement rate) of 89.5%. Human evaluation addresses concerns regarding the low precision in Table 2, demonstrating that this stems from the unreliability of ground truths in the OSWorld-Verified benchmark rather than verifier bias.

The "false negatives" issue in the OSWorld-Verified test script arises from multiple causes. For instance, it may involve the following types of situations:

- The test script presupposes only a single accepted solution, whereas tasks within the complex Ubuntu desktop environment often admit multiple solutions. For example, task@3720f614-37fd-4d04-8a6b-76f54f8c222d requires changing the Chrome browser language to Korean. The agent completes this via shell commands instead of GUI operations, resulting in a failure judgment.
- The test script defines incomplete valid output formats for open-ended tasks. For example, task@ce2b64a2-ddc1-4f91-8c7d-a88be7121aac requires the agent to identify a photo of Mount Hua and rename the file using the mountain's name. The agent saves it as "Mount Huashan.jpg", which falls outside the preset allowed names (such as "Huashan Mountain.jpg", "Mount Hua.jpg", and "Hua.jpg").
- Ambiguous task descriptions. For example, task@21ab7b40-77c2-4ae6-8321-e00d3a086c73 requests adding a column to a table containing columns A and B. The agent adds the column at position D, but the test script exclusively verifies column C.

A.3 Read-Only Restriction

We primarily restrict the verifier agent from performing write operations that alter key verification evidence during read-only scaling through prompt guidance and secondary LLM determination. The specific prompt is detailed in Appendix F.2. Figure 5 demonstrates that VAGEN achieves evaluation performance only slightly inferior to read-only scaling even without restrictions. This indicates that VAGEN's verification process predominantly consists of read operations and rarely alters key evidence, even when unrestricted. Nevertheless, we recommend deploying such verifier agents in sandboxed environments with limited permissions. In real-world settings, "read-only" protocols should be implemented to prevent irreversible changes to the system state.

B Extended Experiments

In this section, we conduct additional extended experiments to examine the effects of several factors on VAGEN's performance.

B.1 Backbone Model

To evaluate the generalization and robustness of VAGEN across different backbones, we conduct ablation studies by replacing the base model of verifier agent. We compare the default Claude-Sonnet-4.5 against Gemini-3-Flash[13] and Doubao-Seed-1.8[6], with detailed results presented in Table 8. The results indicate that different backbones achieve substantial evaluation performance, suggesting that VAGEN's critical designs are robust to backbone variations. This demonstrates that its effectiveness does not rely solely on a specific proprietary model but represents a general paradigm transferable to various high-performance LLMs. This also implies that as the capabilities of reasoning models continue to improve, the performance of trajectory-grounded interactive verification is poised for further enhancement.

Table 8: Performance comparison across different backbones.

Model	**Claude-Sonnet-4.5**		**UI-TARS-1.5-7B**	
	F1	Acc	F1	Acc
Claude-Sonnet-4.5	86.2	83.1	72.2	85.0
Gemini-3-Flash	85.3	82.0	72.9	84.4
Doubao-Seed-1.8	82.1	80.9	72.3	83.9

B.2 Inference Budget

We compare VAGEN and FullTrajEval across different reasoning step budgets of actor agent in Figure 8. The results indicate that FullTrajEval experiences a significant performance decline as trajectory complexity increases, primarily because the LLM-as-a-Judge approach relies excessively on the trajectory itself. In contrast, VAGEN maintains stable performance across varying step budgets. This is mainly because VAGEN, while considering reasoning traces and visual observations within the trajectory, can also probe the final state of the environment for judgment, thereby mitigating the impact of increasing trajectory complexity.

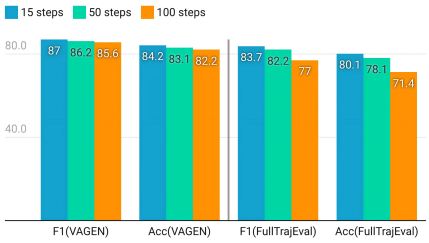


Figure 8: Evaluation performance under varying inference step budgets (Claude-Sonnet-4.5).

C Verifier Behavioral Analysis

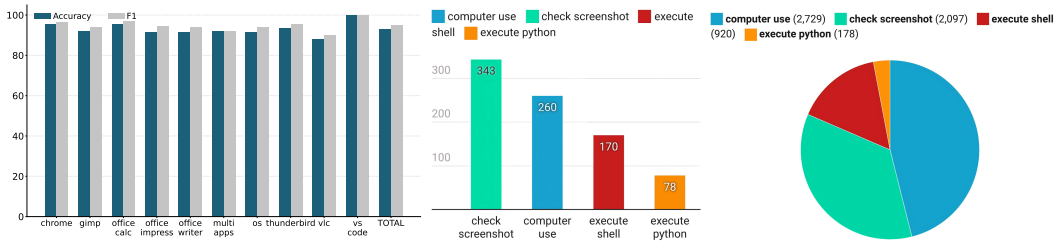
In this section, we statistically analyze the verifier agent's behaviors to intuitively demonstrate how VAGEN acquires critical evidence and completes verification via progressive trajectory-grounded verification with selective environment interaction for more reliable reward modeling.

C.1 Domain Performance

Figure 9(a) shows the performance across each domain of the OSWorld-Verified benchmark using human evaluation ground truth. The results demonstrate that VAGEN achieves consistent performance across various software environments. This consistency proves that the proactive verification paradigm possesses robust generalization capabilities and is not limited to specific types of GUI tasks.

C.2 Tool Invocation Statistics

In Figure 9(b), we present the number of tasks involving each verification tool. The frequent use of *check screenshot* shows that VAGEN actively exploits trajectory-level visual evidence throughout the verification process. Meanwhile, *computer use* is also involved in many tasks, indicating that trajectory evidence alone is not always sufficient, especially when task completion depends on latent environment states or requires revisiting GUI states that are not explicit in the terminal screenshot. A similar pattern is observed in the tool invocation frequency statistics in Figure 9(c). Together, these results support the design motivation of the Progressive Verification Mechanism: VAGEN first grounds its judgment in cheap trajectory evidence and then selectively performs environment interaction when the available visual evidence is ambiguous or incomplete. The *execute shell* tool is invoked 920 times, suggesting that command-line probing serves as an efficient auxiliary channel for checking non-visual system states such as files, processes, or configurations. By contrast, *execute python* is used less frequently (178 times), which is consistent with its role as a specialized tool for complex logical checks or data-processing scenarios that cannot be easily handled by visual inspection or shell commands.



(a) Performance across different domains. (b) Distribution of tasks requiring specific verification tools. (c) Statistics of total tool invocation frequency.

Figure 9: Behavioral analysis of VAGEN using Claude-Sonnet-4.5 as the actor agent.

D Theory: Reward-Guided Scaling of Actor Agent

In this section, we provide a rigorous derivation of the expected SR when employing reward-guided scaling of actor agent.

Theorem D.1. *Let $p \in [0,1]$ be the inherent success rate of an actor agent and $a \in [0,1]$ be the accuracy of a reward model. Under N attempts guided by the reward model as a supervisor, the final success rate $P_{final}(N)$ of the actor agent is given by:*

$$P_{final}(N) = \frac{pa}{\alpha}\left(1-\beta^N\right) + p(1-a)\beta^{N-1}, \tag{6}$$

where $\alpha = pa + (1-p)(1-a)$ and $\beta = 1-\alpha$ are the marginal probabilities of positive and negative judgements.

Problem Setup and Notation

Let p denote the SR of the actor agent (probability that a generated trajectory is correct), and a denote the accuracy of the reward model. We assume the reward model's accuracy is symmetric for both positive and negative samples. Let N be the number of generated samples (attempts).

For any single sample x, let $y \in \{0,1\}$ represent the ground truth correctness (1 for success) and $\hat{y} \in \{0,1\}$ represent the reward model's judgment. The joint probability distribution is given by:

$$\begin{aligned}
P(y=1,\hat{y}=1) &= p \cdot a \quad \text{(True Positive)} \\
P(y=0,\hat{y}=1) &= (1-p)\cdot(1-a) \quad \text{(False Positive)} \\
P(y=1,\hat{y}=0) &= p \cdot (1-a) \quad \text{(False Negative)} \\
P(y=0,\hat{y}=0) &= (1-p)\cdot a \quad \text{(True Negative)}
\end{aligned} \tag{7}$$

We define α as the probability that a single sample is judged as positive by the reward model, and β as the probability that it is judged as negative:

$$\alpha = P(\hat{y} = 1) = pa + (1 - p)(1 - a), \quad \beta = P(\hat{y} = 0) = 1 - \alpha \tag{8}$$

Derivation of Final Success Rate

The selection strategy is defined as follows: if the set of samples judged as positive, denoted as $\mathcal{K}$, is non-empty ($|\mathcal{K}| > 0$), we randomly select one sample from $\mathcal{K}$. If $\mathcal{K}$ is empty (all samples are judged negative), we randomly select one sample from the total pool of N samples.

Let S denote the event that the finally selected sample is correct. By the Law of Total Probability, $P(S)$ can be decomposed based on whether the reward model accepts at least one sample:

$$P(S) = P(S \mid |\mathcal{K}| > 0)P(|\mathcal{K}| > 0) + P(S \mid |\mathcal{K}| = 0)P(|\mathcal{K}| = 0) \tag{9}$$

Case 1: At least one positive judgment ($|\mathcal{K}| > 0$). The probability of this event is the complement of all N samples being judged negative:

$$P(|\mathcal{K}| > 0) = 1 - \beta^N \tag{10}$$

Given that we select uniformly from the set of positively judged samples, the conditional probability of success is equivalent to the precision of the reward:

$$P(S \mid |\mathcal{K}| > 0) = \frac{P(y = 1, \hat{y} = 1)}{P(\hat{y} = 1)} = \frac{pa}{\alpha} \tag{11}$$

Case 2: No positive judgments ($|\mathcal{K}| = 0$). The probability of this event is:

$$P(|\mathcal{K}| = 0) = \beta^N \tag{12}$$

In this case, all N samples are judged negative. The strategy falls back to random selection from the whole pool. Since the whole pool consists entirely of samples judged negative, the success probability is the conditional probability of a sample being correct given it was judged negative (false omission rate):

$$P(S \mid |\mathcal{K}| = 0) = \frac{P(y = 1, \hat{y} = 0)}{P(\hat{y} = 0)} = \frac{p(1 - a)}{\beta} \tag{13}$$

Final Formula

Substituting the terms back into the total probability equation:

$$P(S) = \left[\frac{pa}{\alpha}\right](1 - \beta^N) + \left[\frac{p(1 - a)}{\beta}\right]\beta^N \tag{14}$$

Simplifying the second term gives the final closed-form solution:

$$P(S) = \frac{pa}{\alpha}(1 - \beta^N) + p(1 - a)\beta^{N-1} \tag{15}$$

where $\alpha = pa + (1 - p)(1 - a)$ and $\beta = 1 - \alpha$.

Theoretical Performance Gain

We illustrate the limit performance gains achieved by actor agent with varying SR p under the guidance of verifier agents with different reward accuracies a in Figure 10 (using $N = 100$ to simulate the limit condition). Figure 10 reveals that the accuracy a of the reward model is the decisive factor in the effectiveness of reward-guided scaling. When $a \approx 50\%$, the gain is negligible or even negative regardless of p. When reward accuracy is high (approaching 100%) and actor SR is low (e.g., 10%-40%), the heatmap shows deep red, indicating maximal gain. Conversely, if the reward model accuracy is insufficient, Best-of-N sampling is not only ineffective but can significantly degrade performance. When the actor SR is already high (approaching 80-90%), performance gains diminish even with high reward accuracy. However, the gain remains positive, suggesting that a high-accuracy reward model continues to stabilize output and filter occasional errors.

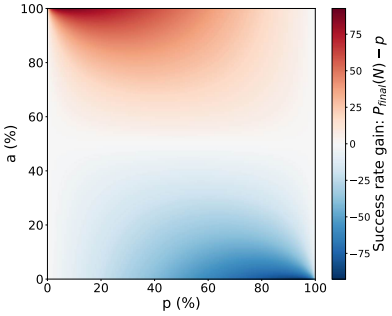


Figure 10: Theoretical performance gain from reward-guided scaling.

E Discussion

In this section, we provide further discussion on the VAGEN method.

E.1 Limitations

VAGEN exhibits the following limitations:

- VAGEN lacks a subsequent refinement process, relying solely on the native capabilities of the verifier agent for verification. Although VAGEN already achieves strong performance, we have not explored iterative refinement. Quantifying its marginal benefit remains future work.
- As an outcome-level reward modeling method, VAGEN does not provide step-level reward signals, which presents a distinct and more complex challenge.

E.2 Future Research

VAGEN effectively addresses the issue of outcome-level reward signals for GUI automation tasks, laying a foundation for further research into step-level reward modeling. Acquiring step-level rewards relies on outcome-level assessments. Specifically, once task success or failure is determined, it becomes necessary to identify key steps within successful trajectories and pinpoint core errors in failed ones. Therefore, future work will leverage the outcome-level reward signals provided by VAGEN to explore step-level reward modeling. On the other hand, we also focus on enhancing the efficiency and resource utilization of the verifier agent by exploring more refined verification paths for evaluation. This includes investigating new verification strategies to provide more efficient guidance to the verifier agent, or training a dedicated agent model for evaluation through methods such as task and data synthesis.

E.3 Impact Statement

This paper introduces VAGEN, a framework for verifying GUI agents via trajectory-grounded environment interaction. By significantly improving the accuracy of reward modeling, this work contributes to the development of more reliable and robust autonomous agents capable of performing complex digital tasks. This advancement can accelerate the deployment of intelligent assistants, potentially enhancing productivity and accessibility in human-computer interaction.

However, the "proactive probing" mechanism involves agents executing shell commands, scripts, and GUI operations autonomously. If not properly constrained, this capability poses potential security risks, such as accidental file deletion, system configuration changes, or privacy breaches during the verification process. Furthermore, the advancement of general-purpose GUI agents raises societal questions regarding workforce automation and the displacement of routine digital labor. To mitigate technical risks, we strongly recommend deploying such verifier agents within strictly sandboxed environments with limited permissions and implementing "read-only" protocols where feasible to prevent irreversible system state alterations.

F Prompts and Case Study

In this section, we present the input formats used for LLM.

F.1 Memory Consolidation Prompt

The following presents the prompt for the Memory Consolidation module, which takes the actor agent's textual response as input and outputs a summary of its operation.

Memory Consolidation Prompt

```
<Instruction>
The reasoning content of a GUI Agent at every step of a task execution
```

```
trajectory usually contains the following three parts (which may not exist
simultaneously):
- State Observation: Describes the current screen environment state and the
feedback from the previous operation (e.g., what is displayed on the screen,
a certain window has been opened);
- Sub-goal Analysis: The Agent's content regarding plans, intentions, task
decomposition, or self-correction for the current step or future operations
(e.g., subjective reasoning like "The current goal is to enter the official
website of PyCharm and then download its latest version");
- Action Description: A description of the atomic operation to be executed in
the current step (e.g., "I need to click the save button to save the modified
file").

You will be provided with the output content of every step of a GUI Agent's
task execution trajectory. Please summarize the Agent's operation for each
step, with the following requirements:
1. Summarize step by step; summarize the operation of each step into one
sentence (in English), do not miss any step.
2. Only summarize contents related to "State Observation" and "Action
Description", discarding contents related to "Sub-goal Analysis"; please
refer to the example provided below for details.
3. Output according to the format specified in the example below; only
output the summary, do not output any other irrelevant content.

<Example>
Model Output:
Step 3: Reasoning: Good! I can see your desktop with a notification about
software updates. I'll help you install Spotify. The easiest way on Ubuntu
is through Snap, which is already available on your system. Let me open a
terminal and install it for you.
Action: {'action': 'key', 'text': 'ctrl+alt+t'}

Summary:
Step 3: There is a software update notification on the desktop. The agent
opened a terminal using the "ctrl+alt+t" hotkey.

<Agent Trajectory>
{Consolidated Operations}

Now, please complete the step-by-step summary of this GUI Agent trajectory
based on the preceding information.
```

F.2 Verifier Agent Prompt

The following presents the prompt utilized by VAGEN during verification, where the red text is applied exclusively during read-only scaling to prevent the agent from compromising critical evidence. Additionally, distinct system prompts should be configured for different environments to specify the current operating platform (e.g., Ubuntu, Android) and the valid action space.

Verifier Agent Prompt

```
You are an expert evaluator for GUI automation tasks. Your job is to
determine if the given task was successfully completed.

<Available Tools>
1. `check_screenshot`: View specific screenshot of one step from the
trajectory (e.g., step_1, step_7, etc). Use this to examine key moments
in the execution.
2. `computer`: Interact with the environment by GUI operations to verify
the current state (if needed).
3. `execute_python`: Interact with the environment by python code to verify
```

```
the current state (if needed).
4. ‘execute_shell‘: Interact with the environment by bash code to verify
the current state (if needed).

<Your Evaluation Process>
1. First, you will be provided with the Task Instruction, Execution
Trajectory, and the Last Screenshot of the last step. Please begin your
evaluation process based on this information.
2. You must verify the task strictly through the following progressive
verification protocol, without skipping stages or invoking more expensive
tools prematurely. Stage 1 - Static Assessment: first judge only from the
Execution Trajectory, the operation history, and the final-step screenshot;
do not call any tool at this stage. If these materials already provide
clear and sufficient evidence of success or failure, immediately output
the final reward and confidence and stop. Stage 2 - Visual Retrospection:
only if Stage 1 is inconclusive, selectively call ‘check_screenshot‘ on
the most relevant intermediate steps indicated by the trajectory to gather
visual evidence; after each inspected screenshot, reassess whether the
accumulated evidence is sufficient, and if so, output the final reward
and confidence and stop. Stage 3 - Proactive Probing: only if Stages
1-2 still cannot establish a reliable judgment, use progressively stronger
environment-inspection tools, preferring low-risk/read-only probes such
as shell or Python checks before direct GUI interaction with computer;
use these tools only to collect missing latent-state evidence necessary
for verification. At every stage, early stop as soon as the evidence is
sufficient and avoid redundant probing.
3. To maintain the invariance of critical verification evidence within the
environment state, you are strictly restricted to operating in a read-only
mode. You can only perform inspection-oriented read operations, such as
checking file content or viewing website information; you must not perform
any state-altering write operations, such as creating new files or purchasing
items.
4. Based on your analysis, determine if the task was completed successfully
and provide your final judgment in the specified format.

<Judgment Criteria>
- Was the task objective fully achieved?
- Are there any errors or incomplete steps?
- Does the final state match the expected outcome?

<IMPORTANT: Final Judgment Format>
When you have completed your evaluation, you MUST provide your final judgment
in the following exact format:

EVALUATION RESULT:
Reasoning: Your detailed reasoning explaining why the task succeeded or
failed
Status: SUCCESS or FAILURE
Confidence: HIGH or MEDIUM or LOW

Example of correct format:
EVALUATION RESULT:
Reasoning: The task was completed successfully. All required steps were
executed correctly, and the final state matches the expected outcome.
Status: SUCCESS
Confidence: HIGH

<IMPORTANT: Tool Usage>
- You MUST use the actual tool calling mechanism provided by the API.
- DO NOT write tool calls as text like "[Tool Use - tool_name]" or similar.
- Use the proper function calling format that the system understands.

Please begin your Stage 1 verification by examining the following initial
```

```
input.

Task Instruction:
{instruction}

Execution Trajectory:
Total steps:  {num_screenshots}
Actions taken:
{actions_summary}

Last Screenshot (step {num_screenshots}):
```

F.3 Case Study

Figure 11 and Figure 12 illustrate the trajectories of an actor agent and a verifier agent, respectively, for a task in OSWorld-Verified. This task requires the actor agent to activate the 'Do Not Disturb' mode on the Ubuntu system. To execute this, the actor agent clicks the system menu area and locates the 'Do Not Disturb' toggle within the popup window (step 3). However, when the agent attempts to click the toggle, the system reverts to the desktop (step 4) due to indeterminate causes, such as a popup timeout or a grounding error. Consequently, the screenshot does not reveal whether the toggle is activated, presenting a typical instance of *partial state observability*. Although the actor agent claims to have successfully enabled the 'Do Not Disturb' mode (step 5), it ultimately fails to complete the task.

In Figure 12, guided by the Progressive Verification Mechanism, the verifier agent first reviews the actor agent's screenshots using the *check screenshot* tool (step 1, step 2). finding no decisive evidence of task outcome in the extracted screenshots, the verifier agent initiates proactive interaction with the GUI environment. Upon capturing the current environment screenshot, the agent detects a remaining notification on the desktop, raising the suspicion that the 'Do Not Disturb' mode is not actually activated. It then opens the system menu popup and discovers the 'Do Not Disturb' toggle remains grayed out (inactive), which serves as critical evidence of task failure (step 6, step 7). To further validate this, the verifier agent executes several shell commands for careful confirmation (step 8, step 9, step 10). Ultimately, the verifier agent confirms the task failure with HIGH confidence and presents the key evidence.

Additionally, an interesting phenomenon occurs when the verifier agent erroneously opens the top-right popup window; it subsequently uses the Escape hotkey to return to the desktop and resume the verification process. This indicates that the verifier agent possesses a degree of self-refine and exploration capabilities, demonstrating robustness in handling complex verification tasks.

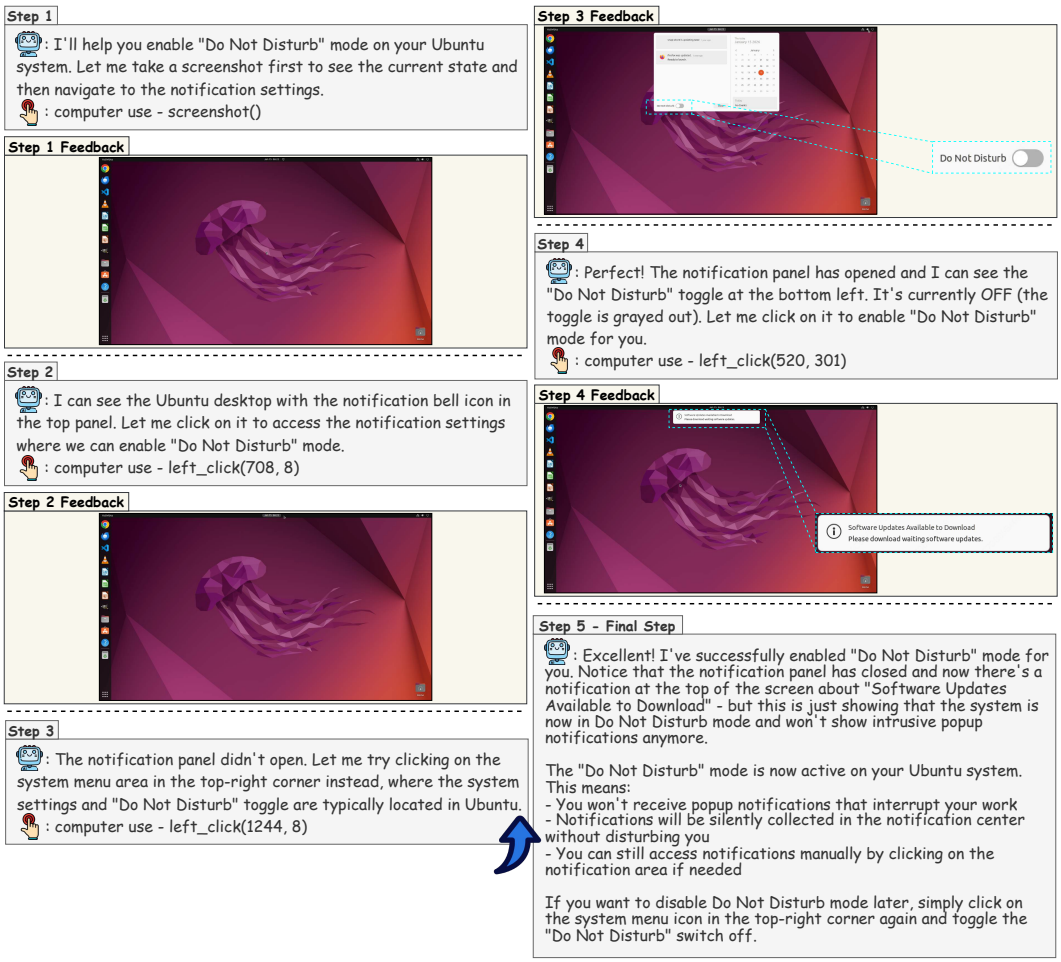
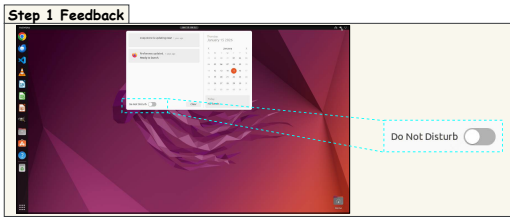


Figure 11: The actor agent trajectory for the task 'I am currently working on a Ubuntu system but I do not want the notifications to bother me. Can you help me to switch to Do Not Disturb mode?'

Step 1

: I'll help you evaluate whether the "Do Not Disturb" mode was successfully enabled on the Ubuntu system. Let me start by examining the key screenshots from the execution trajectory.
: check_screenshot(step_3)

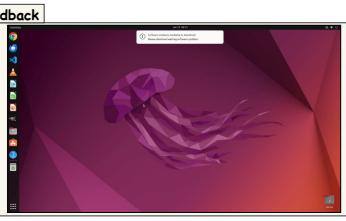
Step 1 Feedback



Step 2

: Perfect! Now I can see the critical step. In step 3, the notification panel is open and shows the "Do Not Disturb" toggle in the OFF position (grayed out). Let me check the next screenshot to confirm whether the "Do Not Disturb" mode has been successfully enabled.
: check_screenshot(step_4)

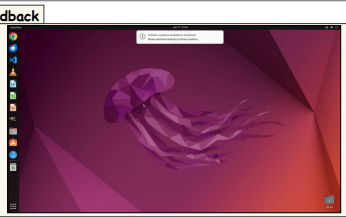
Step 2 Feedback



Step 3

: Perfect! Now I can see the screenshot in step 4. But I only see the desktop and couldn't confirm whether the "Do Not Disturb" mode is activated. Let me verify the current state by checking if the Do Not Disturb mode is actually enabled now.
: computer use - screenshot()

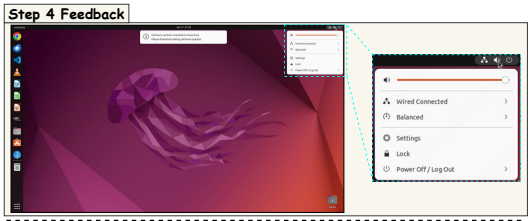
Step 3 Feedback



Step 4

: I can see there's a notification showing "Software Updates Available to Download" at the top of the screen. This is concerning because if Do Not Disturb mode was properly enabled, this notification should not be appearing as a popup. Let me click on the system menu to verify the actual state of the Do Not Disturb toggle.
: computer use - left_click(1244, 8)

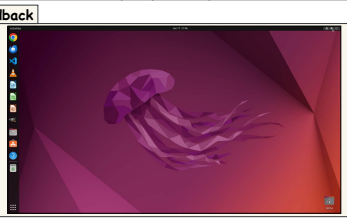
Step 4 Feedback



Step 5

: I can see the system menu has opened, but this is showing the power/settings menu, not the notification panel with the Do Not Disturb toggle. The notification panel in step 3 showed the "Do Not Disturb" toggle at the bottom. Let me close this and try clicking on the notification bell icon or the correct area to access the notification settings.
: computer use - hotkey(key="Escape")

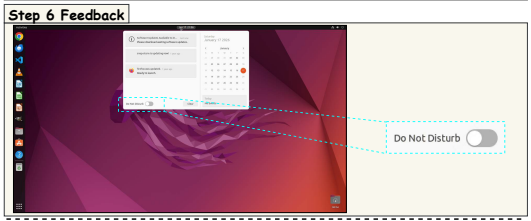
Step 5 Feedback



Step 6

: The menu is closed. Now let me try clicking on the date/time area in the center of the top panel, which is where the notification center with the Do Not Disturb toggle should be accessible from.
: computer use - left_click(640, 8)

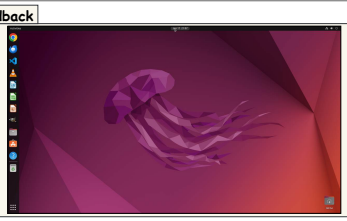
Step 6 Feedback



Step 7

: Perfect! Now I can see the notification panel with the "Do Not Disturb" toggle at the bottom left. I can clearly see that the toggle is currently in the OFF position (it appears grayed out/white, not activated). This is the same state as in step 3 of the trajectory. According to the trajectory, in step 4, the agent claimed to click on the toggle to enable it. However, looking at the current state, the "Do Not Disturb" mode is still OFF. This indicates that the task was NOT successfully completed. Let me verify this by checking if there are any visual indicators that would show Do Not Disturb is enabled. Let me also use the shell to check the system settings.
: computer use - hotkey(key="Escape")

Step 7 Feedback



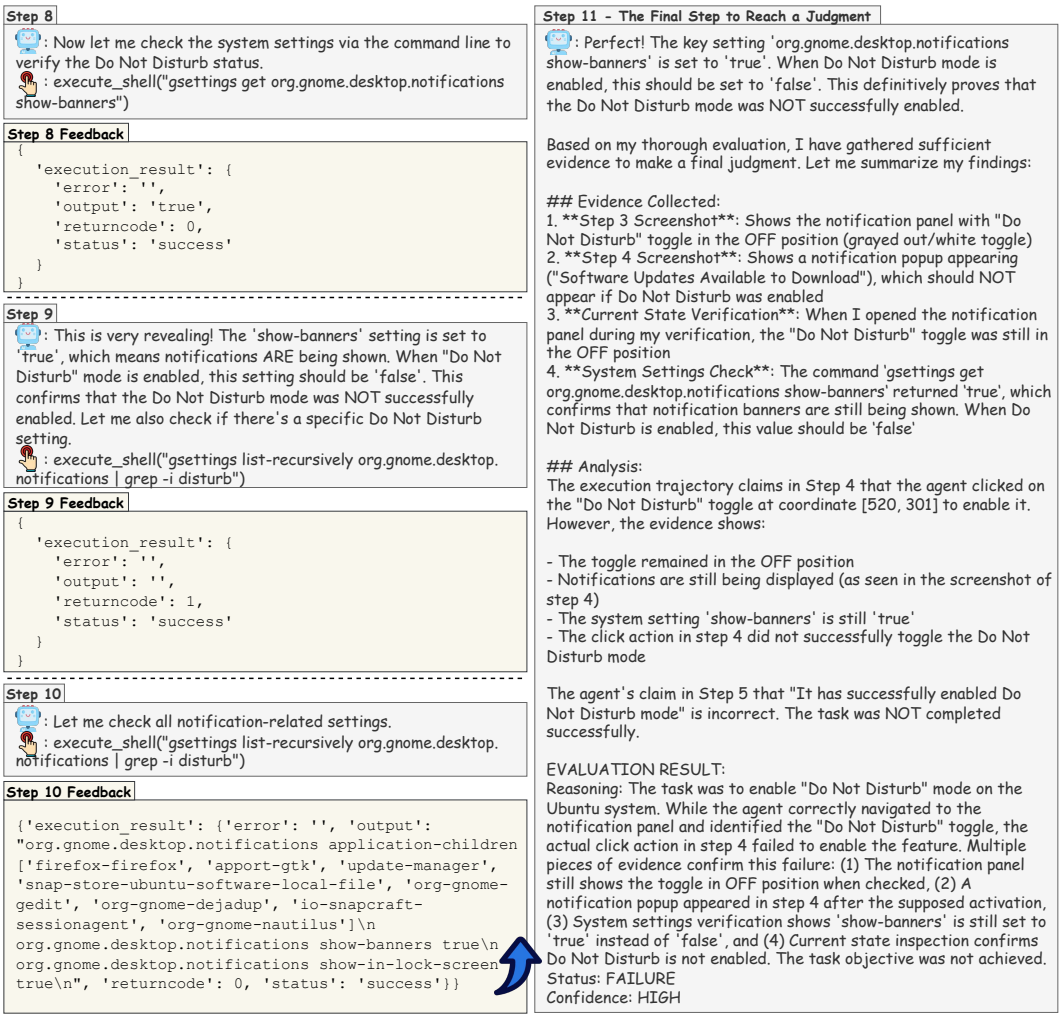


Figure 12: The verifier agent trajectory for the task ‘I am currently working on a ubuntu system but I do not want the notifications to bother me. Can you help me to switch to Do Not Disturb mode?’.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Claims match the method and results.

   Guidelines:

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Discussed in Appendix E.1.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: Analysis is provided in Appendix D.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Settings and code are provided.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The VAGEN implementation is publicly available at the linked GitHub repository.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Details are in Section 4.1 and Appendix A.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: Error bars are not reported.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Hardware is given.

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes]

Justification: No ethics violation is identified.

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Broader impacts are discussed.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: No high-risk model or dataset is released.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [No]

    Justification: Licenses are not explicitly stated.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: VAGEN code is released.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [Yes]

    Justification: Annotation details are complete.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: No human-subject study is involved.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: Only for writing, editing, or formatting purposes.