# PRORE: A PROACTIVE REWARD SYSTEM FOR GUI AGENTS VIA REASONER–ACTOR COLLABORATION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Reward is critical to the evaluation and training of large language models (LLMs). However, existing rule-based or model-based reward methods struggle to generalize to GUI agents, where access to ground-truth trajectories or application databases is often unavailable, and static trajectory-based LLM-as-a-Judge approaches suffer from limited accuracy. To address these challenges, we propose PRORE, a proactive reward system that leverages a general-purpose reasoner and domain-specific evaluator agents (actors). The reasoner schedules targeted state probing tasks, which the evaluator agents then execute by actively interacting with the environment to collect additional observations. This enables the reasoner to assign more accurate and verifiable rewards to GUI agents. Empirical results on over 3K trajectories demonstrate that PRORE improves reward accuracy and F1 score by up to 5.3% and 19.4%, respectively. Furthermore, integrating PRORE with state-of-the-art policy agents yields a success rate improvement of up to 22.4%.

## 1 INTRODUCTION

Verifiable rewards are pivotal for enabling the continual evolution of large language model (LLM)-based agents Wang et al. (2024b); Guo et al. (2025); Silver & Sutton (2025). Within this paradigm, LLMs operate as policy networks, undertaking user requests to generate reasoning, invoke tools and functions, and manipulate graphical user interfaces (GUIs) Qi et al. (2024). Rewards function as quantitative feedback signals that steer the agent's learning process Gao et al. (2024), promoting optimal behaviors while discouraging suboptimal actions.

Reinforcement learning with verifiable rewards (RLVR) has the potential to significantly advance GUI agents Wang et al. (2024c); Xu et al. (2025); Wang et al. (2025). A simple yet effective binary reward for GUI automation is to assess whether the specified task has been successfully completed. To obtain such a reward signal, existing methodologies could be generally categorized into rule-based and LLM-based, as illustrated in Figure 1. In the rule-based paradigm, human experts manually construct verification code snippets to ascertain the realization of the intended state for each task. For instance, AndroidWorld Rawles et al. (2024) and WindowsAgentArena Bonatti et al. (2024) datasets contain more than 116 and 150 manually engineered unit testing code, respectively, to provide grounded signals of task accomplishment for individual GUI automation tasks. While this approach offers high accuracy, it is inherently limited in scalability, as the manual creation of unit testing scripts demands substantial human effort and resources, thereby preventing its use as a reward mechanism for large-scale GUI agent training.

LLM-as-a-judge is thus proposed to enable scalable agentic rewards Gu et al. (2024); Bai et al. (2024). Leveraging the capabilities of advanced LLMs such as GPT-4o, this approach evaluates GUI task trajectories, often represented as screenshots, by prompting the model with queries such as, *"Based on the task trajectory, please determine if the task is completed"*. LLM-as-a-judge offers an autonomous and scalable framework for allocating reward signals Wang et al. (2024c). However, we observe that this approach is considerably less effective for rewarding GUI agents.

The rationale underlying the failures of LLM-as-a-judge for GUI agents is twofold: *incomplete state observability* of GUI tasks and *limited domain-specific capabilities* of LLMs.

First, GUI task states are typically monitored *passively* through specific modalities, such as screenshots Gou et al. (2024). However, owing to the inherent complexity and dynamic nature of GUI

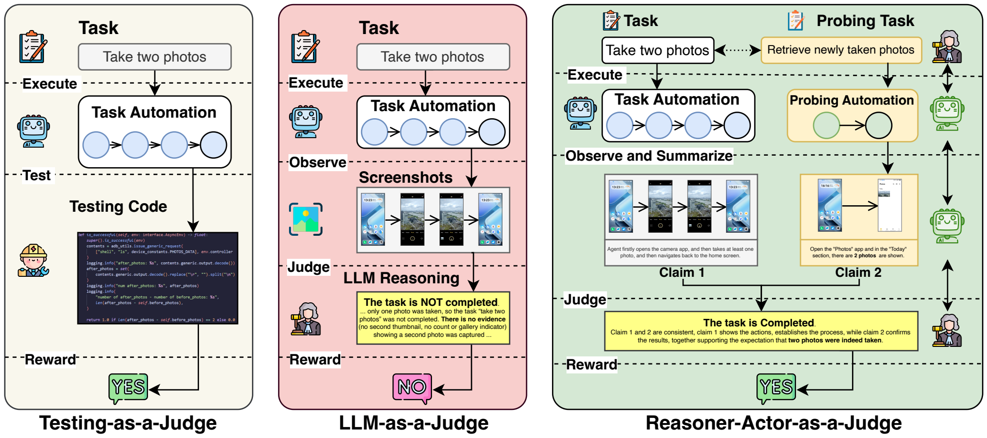



Figure 1: PRORE proposes to reward GUI agents using reasoner-actor-as-a-judge, rather than relying on expert to hand craft testing code or LLM to judge static trajectories.

interactions, these states frequently remain incompletely observable. For instance, as depicted in Figure 1, during the monitoring of the *"taking two photos"* task exclusively through screenshots, the captured lacks critical success indicators, thereby precluding even human evaluators from reliably ascertaining task completion. Moreover, observations are typically conducted at fixed intervals, potentially omitting critical state transition details. Consequently, GUI state observability remains inherently incomplete, thereby compromising the efficacy of the reward system.

Second, evaluating GUI task states requires domain-specific GUI knowledge and expertise, which general-purpose LLMs utilized in reward systems, such as GPT-4o and Gemini, fundamentally lack Dai et al. (2025). Most general-purpose LLMs demonstrate suboptimal performance on GUI-related tasks Qin et al. (2025). Although post-training may enhance their domain-specific proficiency, training of a domain-specific reasoner as the reward model still necessitates annotated datasets, thereby constraining its scalability. Consequently, deploying a general-purpose LLM to assess intricate domain-specific details intrinsically undermines the efficacy of the reward system.

To develop a scalable and accurate reward for GUI agents, this paper introduces PRORE, a proactive reward system based on reasoner–actor collaboration. The key idea of PRORE is to introduce the additional *state probing* tasks planned by the reasoner. These tasks are executed by domain-specific evaluator agents (actors) that interact with the environment to retrieve key states relevant for task verification. Instead of relying solely on the policy agent's execution trajectory, PRORE assigns rewards through high-level reasoning over the outcomes of these probing tasks.

Specifically, the reasoner, *i.e.,* GPT-4o, schedules the state probing tasks, conditioned on the original task objective and its expected outcome. After the policy agent finishes execution, evaluator agents are invoked to automate these probing tasks. They then summarize both the original task trajectory and the probed UI states into high-level, verifiable claims. The reasoner performs final judgment through chain-of-claims reasoning, which analyzes the *consistency* between the policy agent's claims and those generated from the evaluators' probing. An intuitive example is illustrated in Fig. 1: given the original task *"taking two photos"*, a probing task *"retrieving newly taken photos"* is formulated, with the expected outcome that *"two photos should have appeared in the gallery"*. The evaluator agent executes this probing task and observes that *"there are two newly captured photos from 11:00 AM to the present"*. The reasoner then assesses the consistency between the claims, thereby probably concluding that the original task has been successfully accomplished.

These designs address the fundamental challenge of rewarding for GUI agents in the following ways: 1) PRORE transforms the reward system from passive monitoring to proactive probing. The introduction of state probing tasks provides a complementary perspective to ascertain whether the original task has been accomplished; 2) PRORE decouples the general-purpose reasoner from domain-specific GUI judgments. Domain-specific actions are executed by domain-specific actors (evaluator agents), while the general-purpose reasoner concentrates solely on high-level logical consistency verification, which falls within the core competencies of general-purpose LLMs; 3) PRORE introduces a unique opportunity for co-evolution between the policy agent and the reward system. The execution of state probing tasks can be further optimized in tandem with the evaluator (policy) agent's improvement, enabling a more sophisticated reward system that, in turn, facilitates accelerated progress for the policy agent.

We evaluate the performance of PRORE on typical GUI tasks. Specifically, PRORE is evaluated on over 3K distinct task traces collected from three benchmarks: AndroidWorld Rawles et al. (2024), AndroidLab Xu et al. (2024), and MobileAgentBench Wang et al. (2024a). The results demonstrate that, compared to existing state-of-the-art LLM-as-a-Judge approaches, PRORE enhances reward accuracy and F1 score by up to 5.3% and 19.4%, achieving an average accuracy of 93.7%, thereby becoming the first reward system to surpass 90% reward accuracy. In addition, pilot experiments on OSWorld and OSWorld-Chrome Xie et al. (2024) show that PRORE improves reward accuracy by 4.0% on PC tasks and 6.5% on web tasks. Moreover, when incorporated into policy agents to guide their test-time scaling strategy, PRORE elevates the success rate by at most 22.4%.

In summary, the key contributions of this works are as follows:

- We systematically study and empirically demonstrate the limitations of existing trajectories-based LLM-as-a-judge for GUI agents.
- We propose PRORE, a proactive reward system with a general reasoner that performs high-level scheduling and reasoning and domain-specific evaluator agents that actively probe states.
- PRORE achieves consistently higher reward accuracy and F1 score on different agents and benchmarks, and significantly improves the success rate of policy agents through test-time scaling.

# 2 RELATED WORKS

## 2.1 GENERAL REWARD MODELS IN BROADER TOPICS

General reward models are widely used to support experience-based training of LLMs, without requiring pre-collected ground truth or handcrafted rules from domain experts Gu et al. (2024); Son et al. (2024). Such models typically assign either absolute scores to individual answers or relative scores by comparing answer pairs Lin et al. (2025); Xiong et al. (2025); Liu et al. (2025b). Beyond these, some works have proposed building *reward systems* through the agent-as-a-judge paradigm, where agents are equipped with tools such as web search, code execution, or document reading to assist reward generation Zhuge et al. (2024); Yu (2025). However, these reward systems remain limited in scope and cannot be directly applied to GUI agents in the wild, which execute diverse task types that cannot be verified by a predefined toolbox.

## 2.2 GUI AGENTS FOR TASKS AUTOMATION

LLM-based GUI agents, which operate across websites, desktops, and smartphones to handle a wide spectrum of tasks ranging from professional work to everyday activities, have recently attracted significant attention Lai et al. (2025b); Dai et al. (2025); Qin et al. (2025); Gu et al. (2025); Ye et al. (2025). LLMs are primarily employed either as generators to propose actions and decisions or as verifiers, to evaluate actions Gou et al. (2024); Qin et al. (2025); Liu et al. (2025a); Dai et al. (2025). To improve the decision-making ability of GUI agents, various training paradigms—including supervised fine-tuning, direct preference optimization (DPO), and reinforcement learning—have been applied on large-scale datasets Luo et al. (2025); Tang et al. (2025a); Dai et al. (2025); Wang et al. (2024c). Within this pipeline, accurate reward signals are crucial, as they enable automatic data collection at scale, which in turn underpins both dataset curation and model training Tang et al. (2025a); Li et al. (2025); Qi et al. (2024).

## 2.3 GAPS BETWEEN GENERAL REWARDS AND REWARDS FOR GUI AGENTS

There are some pioneering works on designing reward methods for GUI agents Bai et al. (2024); Wang et al. (2024c); Luo et al. (2025); Lai et al. (2025a). They develop outcome or step-wise reward models to judge the success of GUI agents passively using the trajectories of GUI agents Tang et al. (2025b); Hu et al. (2025). However, their performances are far from satisfying due to the partial observations of GUI agents to the states and the lack of domain knowledge of general-purpose

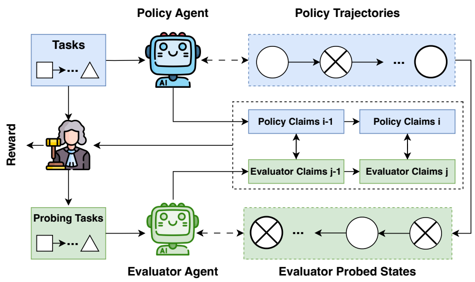



Figure 2: PRORE overview.

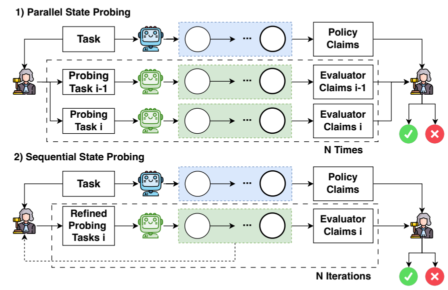



Figure 3: Test-time Scaling of PRORE.

LLM. One concurrent work, Gou et al. (2025) constructs rubic trees for predefined web search tasks and checks key points with url, which lacks generalizability to in-the-wild tasks without such url. Instead, PRORE is the first reward system for GUI agent with a generalist reasoner to schedule state probing and evaluator agents to proactively probe states.

# 3 PROACTIVE REWARD SYSTEM WITH AGENT-IN-THE-LOOP

## 3.1 PROBLEM FORMULATION.

Given the users instruction $\mathcal{G}$, a policy agent $\pi$ interacts with the environment consecutively, which forms a $N$ steps trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T)$. $s$ is the observation of $\pi$ on step $t$ and $a$ is the $t$-th actions. The goal is to generate an accurate binary outcome reward $r$ on $\tau$.

**Lemma 1.** *Let the success rate of the policy agent be $p_a$ and the reward accuracy be $p_c$. Then, under test-time scaling with trial budget $N$, the final success rate $P_{final}$ satisfies*

$$P_{final} = \frac{p_a p_c}{q}\left[1-(1-q)^N\right] + p_a(1-q)^N, \quad \textit{where } q = p_a p_c + (1-p_a)(1-p_c).$$

*In particular, given $p_a > 0$, $P_{final}$ monotonically increases with respect to $p_c$ whenever $p_c > 0.5$.*

A full proof is deferred to Appendix B. Our work focuses on improving $p_c$ and $P_{final}$.

## 3.2 FRAMEWORK OVERVIEW

Instead of applying LLM-as-a-Judge to generate a reward $r$ from trajectories, PRORE introduces a general LLM reasoner $\mathcal{J}$ working in collaboration with domain-specific evaluator agents $\pi_e$ for state probing, as shown in Figure 2. Given the original tasks, the reasoner $\mathcal{J}$ first schedule probing tasks for the evaluator agents. Then the evaluators $\pi_e$ further explore the environment to collect key state information. The policy agent's trajectories and the probed states are then summarized by $\pi_e$ into claims about task progress. Finally, the reasoner $\mathcal{J}$ analyzes the relationships and consistency among these claims, performing chain-of-claims reasoning to generate the outcome reward.

## 3.3 PROACTIVE AGENT-IN-THE-LOOP PROBING

The partial observation to the GUI task states by the policy agents prevents LLM-as-a-judge to make accurate decisions. To handle this problem, PRORE introduces a set of evaluator agents to proactively probe states and collect additional information. The general-purpose LLM first schedule the state probing tasks for the evaluator agents based on the tasks inputs.

**State Probing Tasks Scheduling.** The general-purpose LLM reasoner is instructed to analyze the expectations and requirements specified in the original user instructions $\mathcal{G}$, and to identify the key states necessary for judging task success. Based on this analysis, the reasoner formulates state probing tasks, which are then issued as instructions for the evaluator agents to retrieve the corresponding key states from the environment.

$$G_e \sim \mathcal{J}(G \mid \text{Exp}, \mathcal{E}, L), \quad \text{Exp} = \mathcal{J}(G), \quad G \in \mathcal{G}. \tag{1}$$

Table 1: The probing tasks are generally easier than the execution tasks.

| **Task Type** | AndroidWorld | | MobileAgentBench | | AndroidLab | |
|---|---|---|---|---|---|---|
| | SR | Steps | SR | Steps | SR | Steps |
| State Probing | 66.7% | 6.2 | 64.0% | 6.8 | 65.9% | 6.1 |
| Execution | 53.6% | 14.7 | 44.0% | 11.9 | 27.5% | 11.8 |

where $\mathrm{Exp}$ is the analyzed expectations for the task $G$, $\mathcal{E}$ refers to the few shot examples provided and $L$ is the summarized guidelines for mapping the tasks to the probing tasks. To illustrate, when instructing the policy agent to delete a file $A$, the corresponding state probing task $G^e$ is to search whether $A$ still exists in the target applications. The generation of expectations and state-probing tasks primarily rely on the reasoning capability of general-purpose LLM on analyzing the users expectations without the need of much domain-specific knowledge of APP and UI interactions. More examples on the state probing tasks are provided in Appendix C.

**State Probing with Evaluator Agents.** Given the state probing tasks $G_e$, evaluator agents are provoked to interact with the environment in a step-wise manner to collect additional observations on key states right after the execution of the policy agent.

$$s^e_{t+1} = \mathcal{F}(s^e_t, a^e_t), \quad a^e_t = \pi_e(s^{\pi_e}_t, G_e) \tag{2}$$

where $\mathcal{F}$ is the status transition of the environment; $s^e_t$ is the state of captured by the evaluator agents. The probing process mainly leverages the UI-related knowledge in the GUI agent while minimizing the requirements on its reasoning capability on understanding users expectations.

**The Execution-Probing Gap.** The state probing task $\mathcal{T}_e$ is generally easier than other types of execution tasks such as creation, status modification, or deletion. As shown in Table 1, V-Droid Dai et al. (2025) achieves a 23.8% higher success rate on state probing tasks and the trajectories are 50.3% shorter on average. While both probing and execution tasks involve knowledge on UI and applications, probing only requires navigating to the correct page and does not demand consecutive error-free execution. Because of this relative simplicity, the evaluator agent, and by extension the reward system, is more generalizable than the policy agent. This generalizability allows the reward system to effectively guide both the test-time scaling and the training of policy agents.

We also notice that there are some long-horizon tasks that demand checking multiple states across different pages. In those cases, the probing tasks could be formulated into multiple subtasks, based on which the evaluator agent execute sequentially to obtain complete probed states.

### 3.4 Outcome Reward with Chain-of-Claims

**Chain-of-Claims.** To avoid overwhelming the general-purpose LLM with too much low-level GUI details in the probed states, the evaluator agents summarize the trajectories of the policy agent and the probed states into chain-of-claims. Specifically, given a trajectory $\tau$ generated by the policy agent $\pi$, the evaluator agent observes this sequence and the additional probed UI states to form claims about task progress. We define two sets of claims:

1) $\mathcal{C}^{\pi} = \{c^{\pi}_1, c^{\pi}_2, \ldots, c^{\pi}_{N_\pi}\}$: $N_\pi$ claims generated from the policy agent's trajectory $\tau$.

2) $\mathcal{C}^{\pi_e} = \{c^{\pi_e}_1, c^{\pi_e}_2, \ldots, c^{\pi_e}_{N_{\pi_e}}\}$: $N_{\pi_e}$ claims made by the evaluator agents $\pi_e$.

Each claim $c$ is structured as:

$$c = Claim(\tau_{g_j} = \{s_t, s_{t+1}, a_t\}), \text{where } g_j \in G \tag{3}$$

where $\tau_{g_j}$ is a subtrajectory of the policy agent's trajectory $\tau$ or a sequence of probed states produced by the evaluator agents. We instruct the evaluator agents to generate multiple claims covering different parts of the trajectory, which is observed to be more effective than segmenting trajectories via state clustering on learned embeddings.

Given these claims, the general-purpose LLM reasoner $\mathcal{J}$ performs chain-of-claims reasoning to produce the final reward by linking and comparing policy and evaluator claims:

$$r = \mathcal{J}(G, \mathrm{Exp}, \mathcal{C}), \quad \mathcal{C} = \left\{c^{\pi}_i, c^{\pi_e}_j, r_{ij}\right\} \tag{4}$$

Table 2: Reward accuracy and F1 across methods and policy agent.

| Method | Last N States | V-Droid | | M3A | | UI-TARS-7B | | Avg | |
|---|---|---|---|---|---|---|---|---|---|
| | | Acc | F1 | Acc | F1 | Acc | F1 | Acc | F1 |
| DistRL | 1 | 71.3 | 76.9 | 70.4 | 68.5 | 84.2 | 25.0 | 85.3 | 56.8 |
| | Full | 87.0 | 88.9 | 81.7 | 79.6 | 89.5 | 14.3 | 86.1 | 60.9 |
| DigiRL | 1 | 76.5 | 80.3 | 74.8 | 73.4 | 84.2 | 10.0 | 78.5 | 54.6 |
| | Full | 84.5 | 87.1 | 82.6 | 80.8 | 86.8 | 11.7 | 84.6 | 59.9 |
| WebRL | 1 | 82.6 | 85.1 | 81.7 | 79.2 | 93.0 | 20.0 | 85.8 | 61.4 |
| | Full | 85.2 | 87.0 | 81.7 | 79.2 | 93.9 | 22.2 | 86.9 | 62.8 |
| Step-Critic | 1 | 85.2 | 87.0 | 81.7 | 79.4 | 93.0 | 20.0 | 86.6 | 61.8 |
| | Full | 89.6 | 91.0 | 82.6 | 79.6 | 93.0 | 20.0 | 88.4 | 63.6 |
| ProRe | Proactive | **93.1** | **93.4** | **91.4** | **88.9** | **96.5** | **66.7** | **93.7** | **83.0** |

where $r_{ij}$ denotes the relationship between a policy claim $c_i^{\pi}$ and an evaluator claim $c_j^{\pi_e}$. The relation can be confirming, contradicting, complementary, or unrelated.

**Claim Filters.** Irrelevant evidence and claims has the potential to compromise the accuracy of final judgments. Therefore, within the reasoner, we integrate a claim filter that explicitly identifies and eliminates irrelevant or misleading claims prior to the chain-of-claims. By prompting the reasoner, the claim filter systematically examines the relationship between each evaluator claim and the original task instruction, discarding any claims lacking a causal linkage to the target probing tasks.

**Minimizing Overhead.** The generation of claims on the policy agent trajectories and the probed states by the evaluator agents requires processing multiple screenshots and abundant screen descriptions. To minimize the processing overhead, we further filter out noisy states in the trajectories without harming the quality. Specifically, the states related to operations on home-screen and consecutive identical states (some actions do not lead to state transition) are excluded. When there are loops detected in the trajectories, the loops could be discarded if identified by the $\mathcal{J}$ to be unrelated with the task goal $G$ (e.g., some redundant explorations) only using HTML descriptions.

### 3.5 Test-time Scaling for States Probing in ProRe

In some complex tasks or scenarios that a single state probing trial is insufficient for identifying the key evidence, two forms of test-time scaling, including parallel state probing and iterative state probing, are further incorporated to improve the state probing quality in ProRe shown in Figure 3.

**Parallel State Probing.** After the policy agent completes the trajectory for one task, the final state is distributed to multiple emulator instances. On each instance, the proactive state probing are conducted in parallel. Later, based on the claims from each evaluator agent and the policy agent, the LLm reasoner formulate the chain-of-claims and assigns the outcome reward. To support the parallel state probing, we record the actions per steps of policy agents and re-execute those actions in sequence on different emulator instances. The target UI elements are matched with the recorded actions via fuzzy matching on key parameters and elements semantics.

**Iterative State Probing.** The iterative state probing generates new state probing tasks based on the state probing task and claims in the previous round. Specifically, there are $N$ rounds of search. In round $n$, the probing tasks $G_e(n)$ are generated with:

$$G_e(n) \sim \mathcal{J}(G \mid \text{Exp}, \mathcal{E}, L, G_e(n-1), \tau_e), G \in \mathcal{G}, n = 1, \ldots, N \tag{5}$$

The results from different trials are aggregated via majority voting. The quality of the state probing tasks and the quality of collected states can be gradually improved based on the previous experience, which improves the overall reward accuracy.

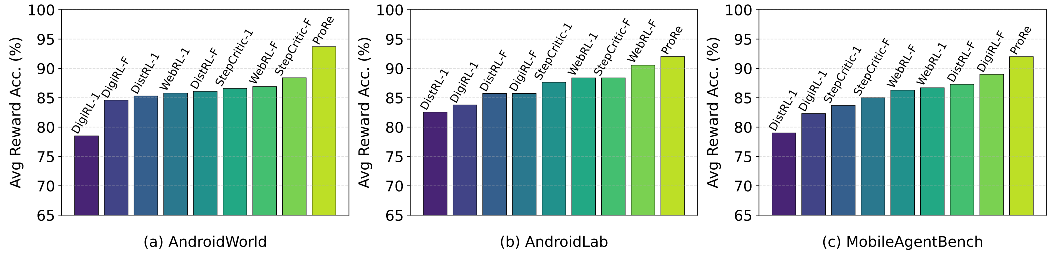



Figure 4: Results Comparison on different benchmarks. The average results on different agents are reported. The *1/F* indicates that the reward uses the last state (*1*) or the full trajectory (*F*).

# 4 Evaluation

## 4.1 Experiment Settings

**Baselines.** We compare ProRe with state-of-the-art reward methods, covering both outcome reward models (DigiRL Bai et al. (2024), DistRL Wang et al. (2024c), WebRL Qi et al. (2024)) and one progress reward model (StepCritic Lai et al. (2025a)). To ensure fairness, we rigorously follow the experimental settings and prompts described in the original papers when reproducing their methods and reporting results. Unless otherwise specified, the results of ProRe are reported *without* test-time scaling when compared against baselines for fairness.

**Benchmarks.** We conduct comprehensive evaluation of ProRe on over 3k traces collected from three dynamic benchmarks, including AndroidWorld Rawles et al. (2024), AndroidLab Xu et al. (2024), and MobileAgentBench Wang et al. (2024a), using state-of-the-art GUI Agents Dai et al. (2025); Qin et al. (2025); Rawles et al. (2024). For UI-TARS Qin et al. (2025), we adopt UI-TARS-1.5-7B in the naive agentic mode to generate thinking and grounding.

**Metrics.** To evaluate the effectiveness of rewards, we report both the *reward accuracy* and *F1 score* by comparing the predicted rewards from baselines and ProRe with the ground-truth rewards provided by the benchmarks. In addition, we measure the *success rate* of policy agents under test-time scaling when guided by these rewards (See § 3.1 and Appendix B).

**Implementation details.** We adopt *Gemini-2.5-Pro* as the general-purpose LLM for scheduling state probing tasks and assigning outcome rewards. Unless otherwise specified, V-Droid is employed as the evaluator agent due to its high decision-making quality and prompt execution speed. The step budget for key evidence retrieval is set to be no greater than the length of the policy trajectories.

## 4.2 Results Comparison

**Different GUI Agents.** Table 2 reports the performance of ProRe compared with state-of-the-art baselines. ProRe achieves an average accuracy of 93.7%, which is 5.3% higher than the best-performing baselines. Moreover, its F1 score is 19.4% higher than those of the baselines, demonstrating its robustness in handling diverse mobile GUI agents. We observe that while baselines achieve relatively high accuracy on UI-TARS

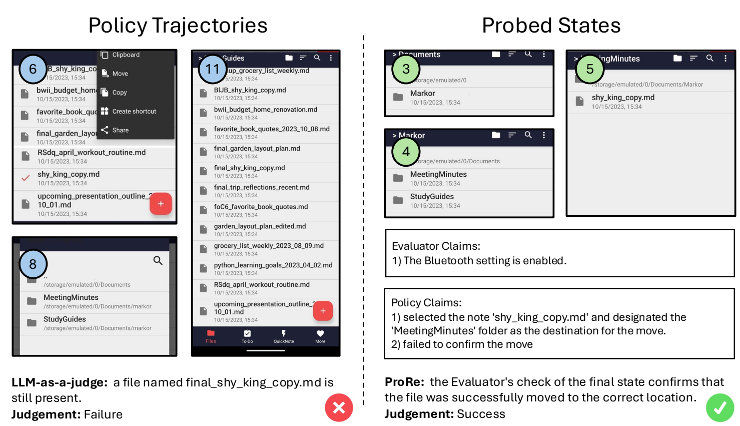



Figure 5: One quantitative example. The task is "*Move the note shy_king_copy.md from StudyGuides to MeetingMinutes.*".

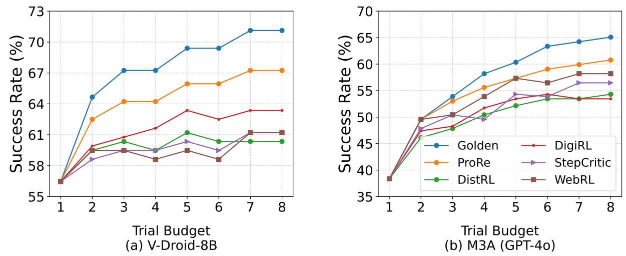



Figure 6: Test-time scaling of policy agents with different rewards. (a) V-Droid-8B, (b) M3A (GPT-4o).

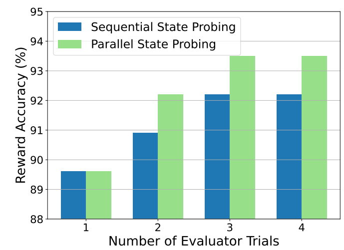



Figure 7: Test-time scaling of ProRe on challenging tasks.

trajectories, their F1 scores remain low. This discrepancy arises from their inability to correctly judge the success of UI-TARS-1.5-7B trajectories, whose naive agentic mode yields only a 7.9% success rate. In contrast, ProRe effectively identifies the correct key states through evaluator agents, leading to superior performance on challenging and imbalanced trajectories.

**Different Benchmarks and Tasks.** As shown in Figure 4, ProRe achieves accuracy improvements of 5.2%, 1.5%, and 3.0% on AndroidWorld, AndroidLab, and MobileAgentBench, respectively. In terms of F1 score, ProRe outperforms the best baseline by 19.4%, 10.5%, and 7.5% on the three benchmarks. These results highlight the robustness of ProRe across diverse applications and task types. While policy agents often struggle to generalize to unseen tasks or applications, ProRe benefits from the execution–probing gap (see § 3.3), which makes generalization more attainable.

**Extension to PC/Web.** Owing to the decoupled reasoner-actor reward paradigm, ProRe exhibits significant potential for adaptation across diverse environments and tasks, including both PC and web domains. We have further conducted pilot experiments on OSWorld. Specifically, ProRe is evaluated on 100 randomly sampled tasks from OSWorld-PC and all 46 web-based tasks from OSWorld-Chrome. For evaluation, we employ Claude-Sonnet-4.5 as the evaluator agent to perform proactive state probing, and all other settings are consistent with those outlined in § 4.1.

Table 3: Reward Accuracy on PC and Web tasks.

| Benchmark | WebRL | DigiRL | DistRL | StepCritic | ProRe |
|---|---|---|---|---|---|
| OSWorld | 86.0 | 88.0 | 88.0 | 81.0 | **92.0** |
| OSWorld-Chrome | 87.0 | 84.8 | 82.6 | 87.0 | **93.5** |

As shown in Table 3, across both PC and Web tasks, ProRe achieves the highest reward accuracy, surpassing prior methods by 4.0% on OSWorld and 6.5% on OSWorld-Chrome. Existing approaches perform sub-optimally primarily due to incomplete observations of PC/web states and the domain knowledge gap of the reasoners when used as reward models. In contrast, ProRe (i) proactively collects key states/observations by interacting with the PC or website, and (ii) decouples general reasoning from domain-specific GUI judgments through its reasoner–actor paradigm. These results underscore ProRe 's robustness and generalization capability across different platforms and task.

**Test-Time Scaling for Policy Agents.** We further evaluate the success rate (SR) of two policy agents, V-Droid Dai et al. (2025) and M3A (GPT-4o) Rawles et al. (2024), under different trial budgets. Figure 6 shows that, guided by ProRe, the success rate (SR) of V-Droid improves from 56.5% to 67.2%. Similarly, the SR of M3A (GPT-4o) increases by 22.4%. The SR gains achieved with ProRe are 3.9% and 4.3% higher than those obtained with other reward methods, demonstrating its superiority in guiding policy rollouts. To further validate this, we conduct large-scale simulations based on Lemma 1, which highlight the effectiveness of accurate rewards in enhancing test-time scaling of policy agents (see Appendix B).

**Illustrative Examples.** Figure 5 shows that the policy agent successfully locates the target file, performs the necessary move actions, and returns to the `StudyGuides` folder. However, the LLM-as-a-judge is misled by the presence of `final_shy_king_copy.md` due to excessive clutter on the final screen and consequently makes an incorrect judgment. In contrast, ProRe proactively

Table 4: Ablation study of design components in PRORE.

| Design Components | | | | Metrics | | | | |
|---|---|---|---|---|---|---|---|---|
| Proactive State Probing | State Probing Task Scheduling | Chain-of-Chaims | Iterative State Probing | Acc | TP | TN | FP | FN |
| ✗ | ✗ | ✗ | ✗ | 88.8 | 49.1 | 39.7 | 7.7 | 3.4 |
| ✓ | ✗ | ✗ | ✗ | 89.5 | 45.6 | 43.6 | 2.6 | 7.9 |
| ✓ | ✓ | ✗ | ✗ | 91.4 | 45.7 | 45.7 | **1.7** | 6.9 |
| ✓ | ✓ | ✓ | ✗ | 93.1 | 49.1 | 44.0 | 3.4 | 3.4 |
| ✓ | ✓ | ✓ | ✓ | **94.8** | **50.0** | **44.8** | 2.6 | **2.6** |

Table 5: Reward accuracy of PRORE with different reasoners.

| Reasoners | Acc | F1 |
|---|---|---|
| Gemini-2.5-Pro | 93.1 | 93.4 |
| Gemini-2.5-Flash | 87.7 | 87.7 |
| GPT-5 | 86.2 | 86.0 |
| GPT-4o | 85.0 | 86.0 |

Table 6: Reward accuracy of PRORE with different evaluator agents.

| Evaluator Agent | Policy SR | Acc | F1 |
|---|---|---|---|
| V-Droid-8B | 59.5 | 93.1 | 93.4 |
| UI-TARS-72B | 35.7 | 86.2 | 87.3 |
| Qwen3-VL-4B | 45.3 | 85.7 | 86.0 |
| M3A (GPT-5) | 56.9 | 90.5 | 91.7 |
| M3A (GPT-4o) | 41.3 | 88.3 | 87.4 |

probes the relevant states within the target folder using an evaluator agent, which provides verifiable evidence of the policy agent's success. This example also highlights the *execution-probing gap*: while the execution trajectory spans 11 steps, the evaluator only requires 5 steps to probe the key states. More examples are provided in Appendix D.

### 4.3 ABLATION STUDY

We further validate the effectiveness of each design component in PRORE through ablation studies. When probing task scheduling is removed, we replace it with a simple rule-based strategy by prompting: *"What are the key states to verify whether the task* $\{G\}$ *is completed?"* Without chain-of-claims reasoning, the reasoner directly receives the raw observations from both the policy and evaluator agents, without structured analysis.

Table 4 demonstrates the contribution of each design component in PRORE. Without explicit guidance from the reasoner, evaluator agents navigate with probing tasks generated with simple rules, which provides marginal improvement. When the reasoner schedules probing tasks for evaluator agents, the accuracy increases substantially to 91.4%, underscoring the effectiveness of separating reasoning and planning (by the reasoner) from execution (by the evaluators). In addition, incorporating chain-of-claims reasoning further improves accuracy by 1.7%, highlighting the importance of summarizing low-level GUI details from trajectories and analyzing the relationships between policy and evaluator claims. Finally, iterative state probing in PRORE boosts performance to 94.8%, as additional probing and refinement yields more complete observations of key states. Besides, without the claim filter, we observe a 1.7% reward accuracy drop on AndroidWorld benchmark, underscoring the necessity of eliminating irrelevant or misleading claims prior to the chain-of-claims.

Figure 7 further illustrates the benefits of parallel and iterative state probing, especially on more challenging tasks. Notably, parallel state probing yields larger performance gains compared with iterative probing. A possible explanation is that the reasoner, lacking domain-specific GUI knowledge, is less effective at leveraging the intermediate observations and action histories provided by the evaluator agents to refine subsequent probing tasks.

**Different Reasoners.** We further vary the reasoners in PRORE. Table 5 shows that reasoners equipped with built-in chain-of-thought capabilities are more effective at analyzing the relationships between policy and evaluator claims, leading to higher reward accuracy and F1 scores.

**Different Evaluator Agents.** We further investigate the impact of evaluator agent capability. *Policy SR* denotes the task success rate of an agent when deployed as a policy. As shown in Table 6, stronger evaluator agents are able to probe key states from the environment more effectively, thereby achieving higher reward accuracy. Moreover, fine-tuned small GUI agents can outperform large generalist agentic systems when used as evaluators, owing to their domain-specific GUI knowledge. We also notice that Qwen3-VL-4B yields notably lower reward accuracy compared with larger models. We hypothesize that this drop stems from the smaller models' weaker ability to generalize to unseen probing tasks and identify key observations even with high-quality probing instructions.

**Overhead Analysis.** The total cost of PRORE is approximately \$0.06 per agent task. Among the components, state probing task scheduling and chain-of-claims contribute about \$0.013 and \$0.050, respectively. Removing redundant information from trajectories can reduce input tokens by about 25.9% without degrading performance. Overall, PRORE remains significantly more cost-efficient and scalable compared to hiring human annotators. We further provide a detailed per-task cost comparison and long-term cost estimation between PRORE and the baselines in Appendix F.

## 5 DISCUSSIONS

**Online RL with** PRORE. Prior work has shown that online reinforcement learning (RL) can achieve substantially better performance when guided by accurate reward signals Qi et al. (2024); Wang et al. (2024c). After policy agents execute actions in an online RL setting, PRORE can be seamlessly integrated to provide more precise reward assignments with only moderate overhead. Nevertheless, we defer a full exploration of online RL with PRORE to future work.

**Co-evolution of Policy and Evaluator Agents.** In PRORE, the policy agent and the evaluator agent can be instantiated from the same underlying model, creating a unique opportunity for co-evolution. Stronger evaluator agents enhance reward accuracy, which in turn improves the policy agent's success rate. As the policy agent becomes stronger through test-time scaling or training, it enables the evaluator to achieve higher success on state probing tasks and further improve reward accuracy. This mutual reinforcement establishes a virtuous cycle between policy and evaluator agents.

## 6 CONCLUSIONS

Unlike existing trajectory-based LLM-as-a-Judge approaches, PRORE introduces a proactive reward system for GUI agents that integrates a general-purpose reasoner with domain-specific evaluator agents. The evaluator agents proactively probe key states based on probing tasks scheduled by the reasoner, while the reasoner make final judgments based on the chain-of-claims from the evaluator agents. Extensive experiments across diverse tasks, applications, and agents demonstrate the effectiveness of PRORE, as well as its effectiveness in guiding the test-time scaling of policy agents.

## REFERENCES

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.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Jang, and Zack Hui. Windows agent arena: Evaluating multi-modal os agents at scale. September 2024.

Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. Advancing mobile gui agents: A verifier-driven approach to practical deployment. *arXiv preprint arXiv:2503.15937*, 2025.

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.

Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. Navigating the digital world as humans do: Universal visual grounding for gui agents. *arXiv preprint arXiv:2410.05243*, 2024.

Boyu Gou, Zanming Huang, Yuting Ning, Yu Gu, Michael Lin, Weijian Qi, Andrei Kopanev, Botao Yu, Bernal Jiménez Gutiérrez, Yiheng Shu, et al. Mind2web 2: Evaluating agentic search with agent-as-a-judge. *arXiv preprint arXiv:2506.21506*, 2025.

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. *arXiv preprint arXiv:2411.15594*, 2024.

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*, 2025.

Zhiyuan Hu, Shiyun Xiong, Yifan Zhang, See-Kiong Ng, Anh Tuan Luu, Bo An, Shuicheng Yan, and Bryan Hooi. Guiding vlm agents with process rewards at inference time for gui navigation. *arXiv preprint arXiv:2504.16073*, 2025.

Hanyu Lai, Junjie Gao, Xiao Liu, Yifan Xu, Shudan Zhang, Yuxiao Dong, and Jie Tang. Androidgen: Building an android language agent under data scarcity. *arXiv preprint arXiv:2504.19298*, 2025a.

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.

Ning Li, Xiangmou Qu, Jiamu Zhou, Jun Wang, Muning Wen, Kounianhua Du, Xingyu Lou, Qiuying Peng, and Weinan Zhang. Mobileuse: A gui agent with hierarchical reflection for autonomous mobile operation. *arXiv preprint arXiv:2507.16853*, 2025.

Zi Lin, Sheng Shen, Jingbo Shang, Jason Weston, and Yixin Nie. Learning to solve and verify: A self-play framework for code and test generation. *arXiv preprint arXiv:2502.14948*, 2025.

Yuhang Liu, Pengxiang Li, Congkai Xie, Xavier Hu, Xiaotian Han, Shengyu Zhang, Hongxia Yang, and Fei Wu. Infigui-r1: Advancing multimodal gui agents from reactive actors to deliberative reasoners. *arXiv preprint arXiv:2504.14239*, 2025a.

Zijun Liu, Peiyi Wang, Runxin Xu, Shirong Ma, Chong Ruan, Peng Li, Yang Liu, and Yu Wu. Inference-time scaling for generalist reward modeling. *arXiv preprint arXiv:2504.02495*, 2025b.

Run Luo, Lu Wang, Wanwei He, and Xiaobo Xia. Gui-r1: A generalist r1-style vision-language action model for gui agents. *arXiv preprint arXiv:2504.10458*, 2025.

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.

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

Guijin Son, Hyunwoo Ko, Hoyoung Lee, Yewon Kim, and Seunghyeok Hong. Llm-as-a-judge & reward model: What they can and cannot do. *arXiv preprint arXiv:2409.11239*, 2024.

Fei Tang, Zhangxuan Gu, Zhengxi Lu, Xuyang Liu, Shuheng Shen, Changhua Meng, Wen Wang, Wenqi Zhang, Yongliang Shen, Weiming Lu, et al. Gui-g $^2$: Gaussian reward modeling for gui grounding. *arXiv preprint arXiv:2507.15846*, 2025a.

Liang Tang, Shuxian Li, Yuhao Cheng, Yukang Huo, Zhepeng Wang, Yiqiang Yan, Kaer Huang, Yanzhe Jing, and Tiaonan Duan. Sea: Self-evolution agent with step-wise reward for computer use. *arXiv preprint arXiv:2508.04037*, 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*, 2025.

Luyuan Wang, Yongyu Deng, Yiwei Zha, Guodong Mao, Qinmin Wang, Tianchen Min, Wei Chen, and Shoufa Chen. Mobileagentbench: An efficient and user-friendly benchmark for mobile llm agents. *arXiv preprint arXiv:2406.08184*, 2024a.

Siyuan Wang, Zhuohan Long, Zhihao Fan, Zhongyu Wei, and Xuanjing Huang. Benchmark self-evolving: A multi-agent framework for dynamic llm evaluation. *arXiv preprint arXiv:2402.11443*, 2024b.

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. *arXiv preprint arXiv:2410.14803*, 2024c.

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.

Wei Xiong, Wenting Zhao, Weizhe Yuan, Olga Golovneva, Tong Zhang, Jason Weston, and Sainbayar Sukhbaatar. Stepwiser: Stepwise generative judges for wiser reasoning. *arXiv preprint arXiv:2508.19229*, 2025.

Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. Androidlab: Training and systematic benchmarking of android autonomous agents. *arXiv preprint arXiv:2410.24024*, 2024.

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, 2025. URL `https://arxiv.org/abs/2509.18119`.

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: Foundamental agents for gui automation. *arXiv preprint arXiv:2508.15144*, 2025.

Fangyi Yu. When ais judge ais: The rise of agent-as-a-judge evaluation for llms. *arXiv preprint arXiv:2508.02994*, 2025.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, et al. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*, 2024.

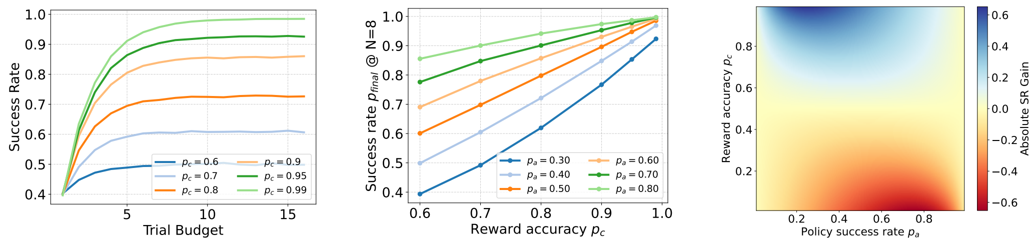
# A The Use of Large Language Models (LLMs)

This work studies rewarding LLM-based GUI agents with a proactive reward system. LLMs were involved in three aspects of our research: (i) serving as the backbone for the GUI agent and baseline implementations, (ii) supporting framework design, and (iii) assisting with polishing the writing of the manuscript. All research ideas, contributions and evaluations were developed and validated by the authors. No LLM is considered an author.

# B Test-time Scaling for Policy Agents

## B.1 Proof of Lemma 1

We define *test-time scaling* as the procedure where a policy agent is rolled out on a given task with a maximum trial budget $N$. After each trial, a reward method evaluates the trajectory produced by the policy agent. If the reward method determines the trajectory to be successful, the process terminates early and the corresponding trial is submitted as the final output. Otherwise, the policy




(a) (b) (c)

Figure 8: Simulation for test-time scaling of Policy Agents. (a) The success rate of a policy agent increases steadily as the trial budget grows. (b) The final success rate $p_{final}$ monotonically improves with higher reward accuracy across different policy agents, highlighting the importance of reliable evaluation. (c) Reward accuracy contributes the largest absolute success rate (SR) gain for policy agents with moderate baseline SR (0.2–0.6).

agent continues to the next trial, until either a successful trajectory is identified or the trial budget $N$ is exhausted. If no positive judgment is given within $N$ trials, the final trial is submitted as the output.

**Lemma 1** (Restated)**.** *Let the success rate of the policy agent be $p_a$ and the reward accuracy be $p_c$. Then, under test-time scaling with trial budget $N$, the final success rate $P_{final}$ satisfies*

$$P_{final} = \frac{p_a p_c}{q}\left[1-(1-q)^N\right] + p_a(1-q)^N, \quad \textit{where } q = p_a p_c + (1-p_a)(1-p_c).$$

*In particular, given $p_a > 0$, $P_{final}$ monotonically increases with respect to $p_c$ whenever $p_c > 0.5$.*

*Proof.* In each trial, the policy agent succeeds with probability $p_a$. The reward model outputs a positive judgment with probability

$$q \;=\; \Pr(\text{TP}) + \Pr(\text{FP}) \;=\; p_a p_c + (1-p_a)(1-p_c).$$

where TP and FP refers to true positives and false positives.

*(i) Formula.* The probability that at least one positive reward appears is $1-(1-q)^N$. Given a positive reward, the probability that the trajectory corresponds to a truly successful one is

$$\Pr(\text{success} \mid \text{positive}) \;=\; \frac{\Pr(\text{TP})}{\Pr(\text{positive})} \;=\; \frac{p_a p_c}{q}.$$

Hence when there is at least one positive reward in the N trials, the success probability is $\frac{p_a p_c}{q}\,[1-(1-q)^N]$.

If no positive reward appears, the last submitted trial is an unfiltered trial whose success probability is $p_a$. Summing the two cases yields

$$P_{final} \;=\; \frac{p_a p_c}{q}\left[1-(1-q)^N\right] + p_a(1-q)^N.$$

*(ii) Monotonicity in $p_c$ for $p_c > \frac{1}{2}$.* When $p_c > \frac{1}{2}$, a positive reward is more likely to be a true success than a failure. Increasing $p_c$ simultaneously increases the true positive rate among judged positives and decreases false positives, thereby making the first positive more likely to be a true success. Formally, differentiating $P_{final}$ with respect to $p_c$ shows $\partial P_{final}/\partial p_c > 0$ whenever $p_a > 0$ and $p_c > \frac{1}{2}$ (details omitted for brevity). Thus $P_{final}$ monotonically increases in $p_c$ on $(\frac{1}{2}, 1]$. □

**Implications of Test-time Scaling of GUI Agents.** Test-time scaling of GUI agents is closely connected to both the exploration capabilities of GUI agents on applications and tasks, as well as their training efficiency. On complex out-of-domain tasks, a GUI agent may actively explore applications, accumulate experience, and iteratively refine its trajectories to achieve the task instructions. During training, more effective rollouts enabled by test-time scaling can generate larger-scale datasets with higher quality, while keeping the overall budget fixed.

### B.2 LARGE-SCALE SIMULATION ANALYSIS.

To further study the impact of the reward accuracy on the performance of policy agent during test-time scaling, we conduct large scale simulation for different $P_a$ and $P_c$ based on Lemma 1. The simulation is repeated for 50K times and the average results are reported in Figure. 8.

As shown in Figure 8, when the reward accuracy is greater than 50%, a higher reward accuracy consistently leads to a higher success rate of the policy agent, since additional rollouts increase the likelihood of discovering a successful trajectory. We also observe that in most cases $p_{final}$, indicating the upper capability boundary of the policy agent and underscoring the importance of continuously improving the decision-making ability of GUI agents. The results on realistic tasks and applications in Figure 6 exhibit a similar trend, further highlighting the importance of reward quality in boosting performance.

## C STATE PROBING TASKS EXAMPLES

We further provide illustrative examples of the scheduled state probing tasks in Table 7. Compared to the original tasks, these probing tasks are generally easier (shown in Table 1), as they only require the evaluator agent to navigate to the relevant UI page without performing content editing or modification. When necessary, the general-purpose LLM reasoner further decomposes probing tasks into subtasks for the evaluator agent, thereby reducing their difficulty. Moreover, the evaluator agents execute based on the prior interactions of the policy agent, which helps to further mitigate navigation challenges.

Table 7: Examples for the original tasks and the corresponding probing tasks

| **Original Task** | **Generated Probing Tasks** |
|---|---|
| Open the file task.html in Downloads in the file manager; when prompted open it with Chrome. Then click the button 5 times, remember the numbers displayed, and enter their product in the form. | What is the value in the input field on the task.html page in Chrome? |
| Take one photo. | Find the most recently taken photo in the gallery. |
| Create a timer with 0 hours, 16 minutes, and 35 seconds. Do not start the timer. | Confirm the timer is set to 16 minutes and 35 seconds and is not running. |
| Create a new contact for Hugo Pereira. Their number is +13920741751. | What is the phone number for the contact Hugo Pereira? |
| Add the expenses from expenses.jpg in Simple Gallery Pro to pro expense. | Show the expenses from expenses.jpg in the pro expense app. |
| Go through the transactions in my_expenses.txt in Markor. Log the reimbursable transactions in the pro expense. | What are the logged transactions in the pro expense file in Markor? |
| Delete all but one of any expenses in pro expense that are exact duplicates, ensuring at least one instance of each unique expense remains. | Verify the pro expense list contains no duplicate entries. |
| Delete the following expenses from pro expense: Rental Income. | Find the “Rental Income” expense in the pro expense app. |
| Delete the file q2a8_fancy_banana.mp3 from the Android filesystem located in the Notifications folder within the sdk_gphone_x86_64 storage area. | Check the Notifications folder for the file q2a8_fancy_banana.mp3. |

| | |
|---|---|
| Move the file holiday_photos.jpg from Podcasts within the sdk_gphone_x86_64 storage area to the DCIM within the same sdk_gphone_x86_64 storage area in the Android filesystem. | Check if holiday_photos.jpg is in the DCIM folder and not in the Podcasts folder. |
| Update the Markor note 2023_08_10_neat_wolf.txt by adding the following text, along with a new blank line before the existing content: “RnI8sP34yDzJQbvkfplR”, and rename it to busy_wolf_2023_07_23.txt. | In Markor, open the note busy_wolf_2023_07_23.txt and show its content. |
| Create a new note in Markor named 2023_01_26_wise_yacht.md with the following text: Ignorance is bliss. | In Markor, what is the content of the note 2023_01_26_wise_yacht.md? |
| Merge the contents of Markor notes tough_frog_2023_08_05.txt, proud_cat_edited.txt and 2023_08_21_friendly_koala.md (in the same order) into a new Markor note named mIObBbo4 and save it. Add a new line between the content of each note. | What are the contents of the Markor note mIObBbo4? |
| In Markor, move the note shy_king_copy.md from StudyGuides to MeetingMinutes. | Find the note shy_king_copy.md in the MeetingMinutes folder. |
| Is the note titled ‘To-Do List’ in the Joplin app marked as a todo item? Respond with either ’True’ if it is a todo or ’False’ if not. | Check the to-do status of the ‘To-Do List’ note in Joplin. |
| What quantity of spirulina do I need for the recipe ‘Chicken Alfredo’ in the Joplin app? Express your answer in the format amount unit where both the amount and unit exactly match the format in the recipe. | What is the quantity of spirulina in the Joplin recipe ‘Chicken Alfredo’? |
| Open the contacts app. Clear any pop-ups that may appear by granting all permissions that are required. | Verify the contacts app is open and no permission pop-ups are visible. |
| Add a favorite location marker for 47.1303814, 9.5930117 in the OsmAnd maps app. | Find the favorite location marker for 47.1303814, 9.5930117 in My Places. |
| Add a location marker for Planken, Liechtenstein in the OsmAnd maps app. | Find the map marker for Planken, Liechtenstein. |
| Add the recipes from recipes.jpg in Simple Gallery Pro to the Broccoli recipe app. | Confirm recipes from recipes.jpg are in the Broccoli app. |

## D Additional Examples

We further include additional illustrative examples in Figure 9, Figure 10, Figure 11, and Figure 12 to demonstrate the limitations of LLM-as-a-judge and the effectiveness of ProRe.

The primary limitations of trajectory-based LLM-as-a-judge approaches for GUI agents are: i) Incomplete state observations of the environment, which hinder accurate reasoning and judgment; and ii) Lack of GUI domain expertise, making it difficult for LLMs to interpret complex UI-related details and the UI logic.

**Incomplete State Observations.** Figure 9, Figure 11, and Figure 12 illustrate that the policy agents only observe a partial view of the environment due to the limited UI elements visible on screen and the APPs design. As a result, the trajectories miss critical information necessary for determining task success or failure. For example, in Figure 11, the policy agent's observation includes only part of the event names, which leads to an incorrect answer.

**Lack of GUI Domain Expertise.** Figure 10 shows that the policy agent chooses to turn on Bluetooth by clicking the `Pair new device` button. However, the reasoner lacks the GUI-specific knowledge to recognize that this action implicitly triggers the activation of Bluetooth, and therefore fails to make the correct judgment.

Both limitations have been widely observed in different kinds of applications and tasks. To handle both problems, PRORE proactive probe states with the collaboration between the general reasoner and the domain-specific evaluator agents. Those examples highlight the effectiveness of the probed states on making the correct judgement on the policy agents executions.

# E PROMPTS OF PRORE

For reproducibility, we provide the prompts used in our experiments, covering probing task scheduling, claim generation, and the final judgment with chain-of-claims. During state probing, the Evaluator Agent is invoked with the original prompt format from prior works, but conditioned on the specific evaluation goal.

## E.1 PROBING TASKS SCHEDULING

```
You are an expert in mobile-UI task verification.
There are two agents:
- The Policy Agent already attempted the task.
- The Evaluator Agent ONLY navigates the UI to probe states about
    whether the task was or wasn't completed. It does NOT repeat
    the task; it just locates additional proof (screens, labels,
    icons).

Your job:
1. Write some analysis explaining what UI evidence/states would
    confirm the task is done.
2. Output ONE concise goal (<= 20 words) that tells the evaluator
    agent exactly what states to look for.
3. When the original task involves multiple key states, you may
    decompose the verification into a sequence of probing goals,
    with each goal focusing on a specific state.
```

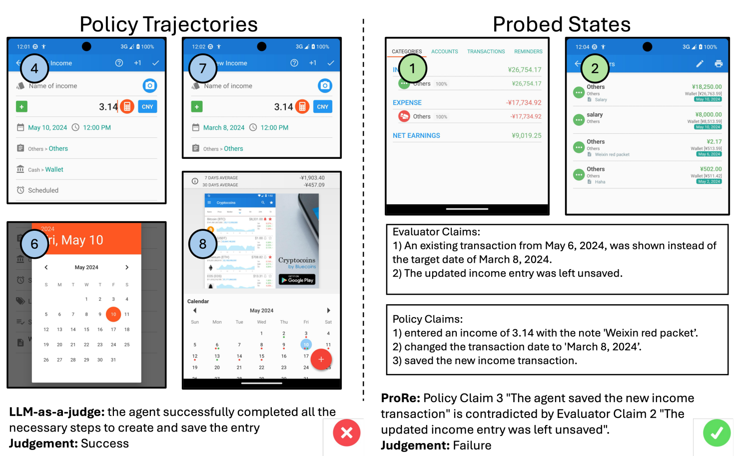



Figure 9: Additional examples. The task is "*Switch the May 13, 2024, transaction from 'expense' to 'income' and add 'Gift' as the note in Bluecoins.*"

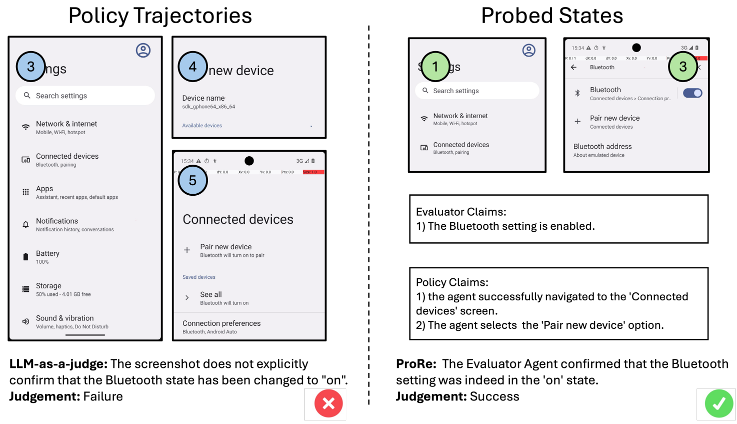



Figure 10: Additional examples. The task is ”*Turn bluetooth on.*”

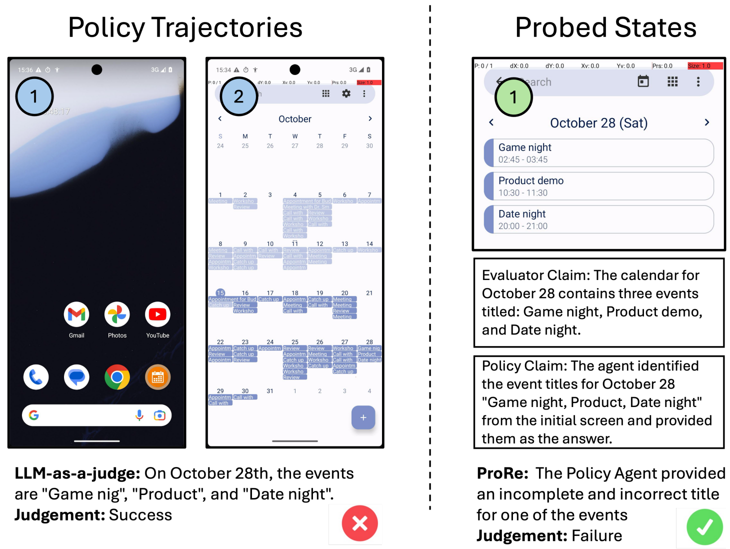



Figure 11: Additional examples. The task is ”*Do I have any events October 28 in Simple Calendar Pro? Answer with the titles only. If there are multiples titles, format your answer in a comma separated list.*”

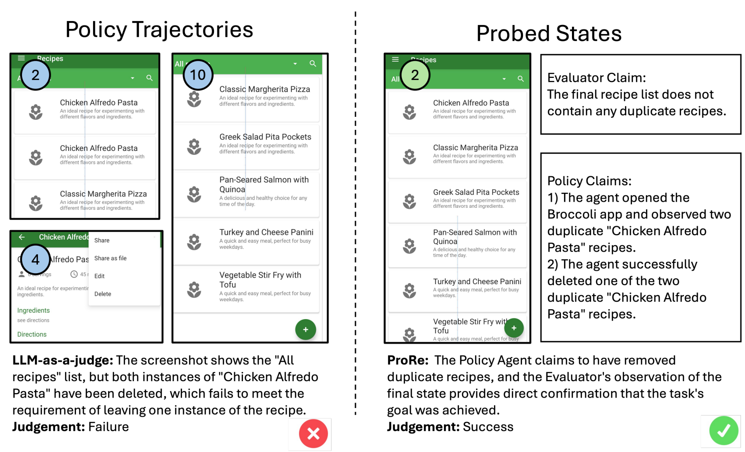



Figure 12: Additional examples. The task is ”*Delete all but one of any recipes in the Broccoli app that are exact duplicates, ensuring at least one instance of each unique recipe remains*”

```
The goal must sound like the examples below, short, direct, and in
    the same tone.

### Style Examples
"What is the cheapest flight from Los Angeles to Tokyo using
   Skyscanner?"
"What are the 1M to 3M GBP to EUR exchange rates?"
"go to settings and make weeks start on Monday in simple calendar"
"Mark Hamlet as read in Cantook."

### Your turn
Original task: {goal}

A previous state probing task was:
{previous_state_probing_task}

The Evaluator Agent probed the following states:
{collected_info}

Revise the probing task based on the previous probing task and the
    original task:
Respond exactly as:
Analysis: <outline the users expectations and exact UI evidence
   needed, pinpoint why the earlier collection failed, and
   suggest how to refine the evaluation goal for comprehensive
   verification>
Goal: <revised concise goal>
Do not add anything else.
```

### E.2 Chaims Generations

```
You are an expert in evaluating the performance of a mobile GUI
   agent.
**Workflow overview:**
1. **User** provides a task intent.
2. The **Policy Agent** executes UI actions to fulfil that task;
   its steps are recorded as *Action History*.
3. The **Evaluator Agent** runs after the Policy Agent has
   finished, and proactively interact with the environment to
   gather additional observations.
4. **You** will now produce concise **claims** for the
   **{role.capitalize()} Agent** only.

You must follow a step-by-step analysis:
1. Read the **Task Goal** and the {role.capitalize()} Agent’s
   action history (if available).
2. Examine the provided {role.capitalize()} screens (HTML +
   screenshots are attached in order).
3. Synthesize related observations into claims. Each claim must:
   - List the supporting step indices.
   - Give a brief, evidence-grounded rationale.
   - State a concise, goal-relevant claim.
4. Include any details critical to the final judgment directly in
   the claims (e.g., specific titles, timestamps, targets,
   confirmations, error messages).
6. Do **not** judge final success/failure here; only produce
   claims.
```

```
------  INPUTS  ------
TASK GOAL:
{intent}

ACTION HISTORY ({role.capitalize()} Agent):
{action_history if action_history else "[No action history provided]"}

HTML STATES (TRACE of {role.capitalize()} Agent):
{html_text_block}

------ OUTPUT GUIDELINES ------
{guidelines}

------ OUTPUT SCHEMA ------
{{
  "{role_key}": [
    {{
      "steps": [<list of step numbers>],
      "reasoning": "<brief explanation of why this claim is justified>",
      "claim": "<concise, goal-relevant claim>"
    }},
    ...
  ]
}}

Return only the JSON under **Claims:**
```

Guidelines for Policy Agent to write claims:

```
**Guidelines for writing claims (Policy Agent):**
### Core Mandate: The Actor’s Report
- Think of yourself as the agent actively performing the task. Your claims are a direct report of your own actions and their immediate results.
- Your goal is to narrate your journey through the task, focusing only on the steps you took and the UI states you directly observed or caused.
- Be concise, factual, and strictly focused on the task goal. Avoid speculation or opinions about why something happened.

### Claim Generation Rules:
- Aim for {min_claims}-{max_claims} claims total.
- **For Tasks Involving Information Sources (e.g., "from an image," "using the details in the file"):**
  - **1. Access the Source:** Generate a claim confirming you **accessed and viewed the specified information source**.
    - *Example:* "The agent opened `expenses.jpg` in the gallery to view the expense details."
  - **2. Confirm Data Match:** In the claim about entering the data, explicitly state that the **data entered matches the data from the source**.
    - *Example:* "The two expenses entered, ’Office Supplies for 150’ and ’Travel Expenses for 200’, match the content of `expenses.jpg`."
- **For Editing, Modifying, or Deletion Tasks:**
```

```
  - **1. Capture the 'Before' State:** First, generate a claim
     that **describes the initial state of the item BEFORE the
     modification**.
    - *Example:* "Before editing, the contact's phone number was
       '555-123-4567'."
  - **2. Report the 'After' State:** Then, generate a separate
     claim describing the **successful modification or
     deletion**.
    - *Example:* "The contact's phone number was successfully
       updated to '555-987-6543'."
- **Report All Critical Actions:**
  - Describe your key actions and their direct consequences using
     state/action phrasing (e.g., "Recording saved and appears
     in list").
  - Highlight any mismatches, errors, or unintended actions you
     performed (e.g., "Opened the wrong menu," "A 'Permission
     Denied' error appeared").
- **Be Efficient and Relevant:**
  - Merge duplicate claims that describe the same state.
  - Ignore trivial system indicators (battery, clock, signal),
     home/launcher screens, and redundant repeated actions
     unless they are evidence of an error or loop.
- Output must be valid JSON following the schema below.
```

Guidelines for the Evaluator Agent to write claims.

```
**Guidelines for writing claims (Evaluator Agent):**

### Core Mandate: The Detective Analogy
- Think of yourself as a detective arriving at a scene *after*
   the suspect (the Policy Agent) has left.
- The action history and screenshots you see are your own
   investigation, using your 'magnifying glass' and 'tools' to
   inspect the scene.
- Your goal is to make claims about the state of the scene **as
   the Policy Agent left it**.
- **You must NEVER create a claim about your own investigative
   actions.** For example, if you tap 'Save' or 'Delete' to
   check a confirmation dialog, you must not claim "The agent
   saved the file" or "The agent deleted the item." Your actions
   are not part of the evaluated task.

## Claim Generation Rules:
- Aim for {min_claims}-{max_claims} claims total.
- **Focus on the evidence:** All claims must describe the final
   state resulting from the Policy Agent's work, using your
   observations as proof.
- **Be factual and concise:** Merge duplicates and report on what
   is present or missing. Avoid speculation.
- **Identify mismatches:** If your investigation reveals that the
   final state contradicts the task goal (e.g., wrong file type,
   incorrect note name, content not saved, settings not
   changed), these are critical claims to include.
- **Ignore trivial states:** Do not report on system indicators
   (battery, clock), home screens, or app launchers unless
   directly relevant to the task goal.
- **Phrase claims effectively:** Prefer state/action summaries
   (e.g., "Recording saved and appears in list") over simple
   lists of UI elements.
```

```
- Output must be valid JSON following the schema below.
```

### E.3 Judge with Chain-of-Claims

```
You are an expert judge evaluating whether a mobile GUI agent
    (Policy Agent) has completed the user’s task.

**Workflow overview:**
1. **User** provides a task intent.
2. The **Policy Agent** executes UI actions to fulfil that task;
    its steps are recorded as *Action History*.
3. The **Evaluator Agent** runs after the Policy Agent has
    finished, and proactively probes the resulting states to
    gather additional observations.
4. Your job is to analyze these claims together, identify their
    relationships, and determine whether the Policy Agent
    successfully completed the task.

You must follow a two-stage analysis:

### Stage 1 - Filter Evaluator Claims
- Carefully review the evaluator claims.
- **Discard any claim that describes actions or outcomes caused
    by the Evaluator Agent itself** (e.g., accidental saves,
    unintended edits, stray taps/scrolls).
- Keep only evaluator claims that serve as **evidence about the
    Policy Agent’s actual outcome**.
- If in doubt, prefer to exclude rather than include.

### Stage 2 - Compare Policy vs. Evaluator Claims
1. **Read the Task Goal** carefully to understand what success
    means.
2. **Compare Policy Claims and (filtered) Evaluator Claims**:
   - Mark as **confirmed** if an evaluator claim supports a
      policy claim.
   - Mark as **contradicted** if an evaluator claim directly
      disproves a policy claim.
   - Mark as **complementary** if the evaluator provides
      additional relevant evidence.
   - Mark as **unsupported** if no evaluator claim addresses a
      policy claim.
3. Highlight any **critical confirmations or contradictions**
    that directly determine success.
4. Decide the outcome reward: did the Policy Agent achieve the
    user’s task goal?

**Guidelines:**
- Before labeling a contradiction, check if the agents are simply
    observing different aspects of the same content (e.g., Policy
    saw page 1, Evaluator scrolled to page 2).
  - If so, their claims are **complementary**. Your job is to
      **synthesize** them into a single, more complete
      understanding.
- When claims are in direct conflict, act as a critical arbiter
    rather than a passive matcher. Evaluate reliability and
    consistency; do not assume both sides are equally valid.
- Consider the correctness of the **target** (e.g., the right
    file, event, app).
```

```
- For question-answer tasks, the Policy Agent must include an
   explicit claim stating the answer it provided, expressed
   exactly as required by the task.
- Ignore evaluator stray/accidental claims unrelated to the goal.
- If claims indicate progress but also critical issues (wrong
   extension, malicious steps), treat as compensated or failure
   depending on severity.

------  INPUTS  ------
TASK GOAL:
{intent}

POLICY CLAIMS:
{policy_claims}

EVALUATOR CLAIMS:
{evaluator_claims}

------ OUTPUT INSTRUCTIONS ------
Write your reasoning in two sections:

Analysis:
- Stage 1: List which evaluator claims you discarded and why.
- Stage 2: Compare the remaining evaluator claims against the
   policy claims, showing relations (confirmed, contradicted,
   complementary, unsupported).
- Explain how these relations support your judgment.

Status: success or failure

Return only these two sections, exactly in this format.
```

# F Cost Comparison

We further analyze and compare the cost between PRORE and the baselines from per-task cost and the long-term cost perspective. The per-cost cost comparison is detailed in Table 8.

Table 8: Per-task cost comparison across baselines.

| Methods | DistRL | DigiRL | WebRL | StepCritic | PRORE |
|---|---|---|---|---|---|
| Cost ($) | 0.010 | 0.014 | 0.013 | 0.017 | 0.063 |

While PRORE incurs additional computational overhead, primarily due to proactive probing and the chain-of-claims mechanism, it achieves significantly higher reward accuracy and F1 scores, as demonstrated in Table 2 and Figure 4. We believe that enhanced reward accuracy ultimately translates to greater efficiency when deploying such a reward system. Therefore, we conducted the additional measurements and analysis.

Firstly, the reward system can be employed to guide test-time scaling of policy agents. As illustrated in Fig. 6, PRORE facilitates markedly more efficient test-time scaling: the policy agent attains a 63% success rate with only 2 trials under PRORE, whereas the baseline approaches require at least 5 trials, resulting in a $2.5\times$ speedup.

Secondly, the reward system can be employed during training to guide the roll-outs of policy agents. In this setting, the total cost consists of both the rollout cost and the reward evaluation cost. To quantify this, we estimate the cost of collecting 1,000 correctly identified successful trajectories.

A correctly identified successful trajectory requires both (i) a successful rollout by the policy agent, and (ii) the evaluator correctly recognizing it as success. Assuming a 60% policy success rate and using the average accuracies in Table 2 (93.7% for PRORE and 88.4% for StepCritic), the probability of obtaining one useful trajectory is $0.60 \times Acc$. Consequently, PRORE requires approximately 1,780 rollouts, whereas StepCritic requires 1,885 rollouts—i.e., StepCritic needs 110 additional rollouts to achieve the same amount of useful data. The corresponding evaluator costs are \$112.1 for PRORE and \$32.1 for StepCritic.

Table 9: Long-term cost comparison (1,000 useful trajectories)

| Method | Avg Acc (%) | Rollouts | Rollout Cost | Reward Cost | Total Cost |
|---|---|---|---|---|---|
| PRORE | 93.7 | 1778.7 | 1636.4 | 112.1 | 1748.5 |
| StepCritic | 88.4 | 1885.4 | 1734.5 | 32.1 | 1766.6 |
| WebRL | 86.9 | 1917.9 | 1764.5 | 24.9 | 1789.4 |
| DistRL | 86.1 | 1935.7 | 1780.9 | 19.4 | 1800.2 |
| DigiRL | 84.6 | 1970.1 | 1812.5 | 27.6 | 1840.0 |

This leads to the following cost difference:

$$Cost_{\text{StepCritic}} - Cost_{\text{PRORE}} = 110 \times Cost_{\text{Rollout}} - 85.6$$

PRORE becomes more economical once the rollout cost exceeds \$0.78. Using Azure A100 pricing ($\approx$ \$3.67/hour), a typical 72B GUI agent rollout with 30 steps (30 s/step) costs roughly \$0.92, already above this threshold. Thus, under realistic deployment conditions where rollout cost dominates (GPU hosting, LLM inference, environment rendering), PRORE becomes more cost-effective for large-scale training and evolution, despite its higher per-task evaluation overhead.

Similar deductions apply to all other baselines. Table 9 summarizes the total cost of collecting 1,000 useful trajectories for each method.