GSAR: Goal-State-Anchor Rewards for Mobile GUI Agents with Self-Evolving Data Synthesis

Long Zhang[1] **Yuhan Chen**[2] **Chaoran Zhang**[1] **Wanxia Cao**[2] **Kun Huang**[2]
Pengzhi Gao[2] **Wei Liu**[2] **Jian Luan**[2] **Chenliang Li**[1] **Lixin Zou**[1]*
[1]Wuhan University [2]Xiaomi Inc.
{zlongooo, chaoranzhang, cllee, zoulixin}@whu.edu.cn
{yuhanchen240, caowanxia123, huangkun813}@gmail.com
{gaopengzhi, liuwei40, luanjian}@xiaomi.com

Abstract

Vision-Language Models (VLMs) based GUI agents stand to benefit significantly from online reinforcement learning (RL). However, their training is bottlenecked by two fundamental issues: current data synthesis methods for GUI Agents rely on specific environments and struggle to generate diverse data, while existing evaluators either suffer from limited scalability or provide inaccurate and unreliable reward signals. To overcome these challenges, we introduce GSAR (Goal-State-Anchor Reward), a RL reward framework that supports scalable task generation and delivers reliable reward signals for stable and efficient policy optimization. Our approach features self-evolving data synthesis, which produces multiple environments through task execution and generates diverse tasks and goal states. Complementing this, a state-anchor mechanism automatically annotates task-relevant UI elements in successful goal states as reference anchors. During RL training, these reference anchors provide accurate, scalable reward signals that substantially enhance efficiency. Extensive evaluations demonstrate that our framework achieves over 90% accuracy on offline trajectory verification and performs closest to rule-based methods. Furthermore, agents trained using our reward framework exhibit strong performance on both AndroidWorld and our constructed benchmark, establishing a scalable approach for GUI agent training.

1 Introduction

Graphical User Interface (GUI) agents driven by Vision-Language Models (VLMs) (Bai et al., 2025; Hurst et al., 2024) demonstrate human-like competence in understanding and operating mobile device environments (Wang et al., 2025; Ye et al., 2025). As a promising paradigm for automating interactions across mobile and desktop platforms, GUI agents transform raw screen observations into actionable decisions through visual comprehension and functional reasoning. These agents operate without extensive fine-tuning or additional pretraining, highlighting their strong potential for broad real-world applications.

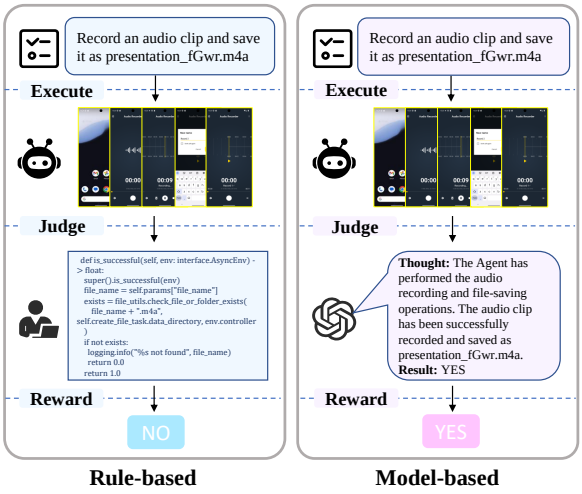


Figure 1: Rule-based and Model-based reward judge. Model-based judges may be inaccurate when context is limited or model capacity is insufficient.

Reinforcement Learning with Verifiable Rewards (RLVR) (Lambert et al., 2024; Shao et al., 2024; Yu et al., 2025; Feng et al., 2025), building on its success in domains such as mathematics and code reasoning, has recently emerged as an effective paradigm for training GUI agents (Bai et al., 2024; Xu et al., 2025). Scaling RLVR for GUI agents fundamentally relies on a unified training infrastructure: diverse task instantiation and strictly verified reward signals. While Large Language Models can trivially generate a vast array of task descriptions, grounding these open-ended instructions into executable training episodes requires configuring the corresponding initial environments. Currently, this environment setup and data collection process remains highly restricted by

* Corresponding author.

manually designed configurations (Sun et al., 2024; Lin et al., 2025). This labor-intensive dependency prevents the automatic generation of truly diverse environment-task pairs, which is the foundational prerequisite for large-scale policy learning.

Furthermore, evaluating agent performance across such an open-ended task space introduces an equally complex challenge. As shown in Figure 1, existing reward designs struggle to scale effectively. **Rule-based** methods require manually written execution functions to verify task completion (Rawles et al., 2024; Xie et al., 2024); while precise, they are completely impractical to manually scale alongside automatically generated tasks. Conversely, **model-based** evaluators attempt to automate outcome assessment (Bai et al., 2024; Wang et al., 2024), but they frequently suffer from instability, hallucination, or insufficient coverage of edge cases. These flaws lead to noisy supervision signals that severely impair policy convergence and degrade agent performance. Together, the inability to automatically instantiate diverse initial environments and the lack of scalable, accurate reward supervision form a systemic bottleneck that constrains the true potential of RL for GUI agents.

To address this challenge, we introduce GSAR (Goal-State-Anchor Reward), an RL reward framework that tackles both bottlenecks by enabling scalable task and environment generation and providing reliable reward signals for stable policy optimization. First, we leverage self-evolving data synthesis to generate diverse tasks and trajectories, while progressively evolving the environments through interaction, resulting in task, initial environment, and trajectory triplets in a fully automated manner. Next, the state-anchor mechanism automatically annotates task-relevant UI elements in the final states of successful trajectories, producing task, initial environment, and goal-state-anchor reference triplets. During online training, these references are used to provide accurate and semantically grounded reward signals. Both online and offline experiments demonstrate that our approach, by explicitly anchoring task-relevant UI elements in successful goal states, delivers robust and scalable reward feedback for GUI agent training.

In summary, our work makes the following primary contributions:

- We propose a self-evolving data synthesis mechanism that generates diverse environments and task trajectories while preserving initial environment snapshots, enabling reproducible and scalable reinforcement learning.
- We introduce a scalable Goal-State-Anchor reward framework for GUI agents that overcomes the limitations of prior judges to deliver accurate, generalizable signals for online reinforcement learning.
- We analyze the impact of reward accuracy on reinforcement learning and show that reward design critically affects agent performance.

2 Related Works

2.1 Data Synthesis for Mobile GUI Agents

Effective training of GUI agents requires diverse tasks that expose models to a wide range of interface elements and interaction patterns necessary for robust and generalizable performance. Following the introduction of sequential GUI data for mobile applications in Rico (Deka et al., 2017), research has increasingly focused on synthesizing GUI datasets. Several efforts (Rawles et al., 2023; Li et al., 2024; Lu et al., 2024) generated data for supervised fine-tuning (SFT) (Ouyang et al., 2022) models, but the scalability of manually curated datasets is hindered by costly and inefficient data collection. More recently, automated data synthesis approaches have been proposed to address this limitation. These methods first explore application interfaces via random or heuristic traversal, and then construct tasks based on the explored states, either by reverse-engineering instructions (Sun et al., 2024) or generating tasks directly from screenshots (Xie et al., 2025). Large pretrained models (e.g., GPT-4o (Hurst et al., 2024)) are then employed to execute these tasks and filter out incomplete or failed trajectories. However, these approaches still depend on manually preconfigured app environments and do not provide reproducible initial states required for stable reinforcement learning of GUI agents.

2.2 Reward System for GUI Agents

Recent research has increasingly integrated reinforcement learning (RL) into the training of GUI agents (Zhang et al., 2025; Chen et al., 2025), with the goal of reducing supervision requirements and enhancing generalization and long-term decision-making ability. In RL, reward signals are crucial for guiding policy optimization and shaping learning dynamics (Gao et al., 2024; DeepSeek-AI et al.,

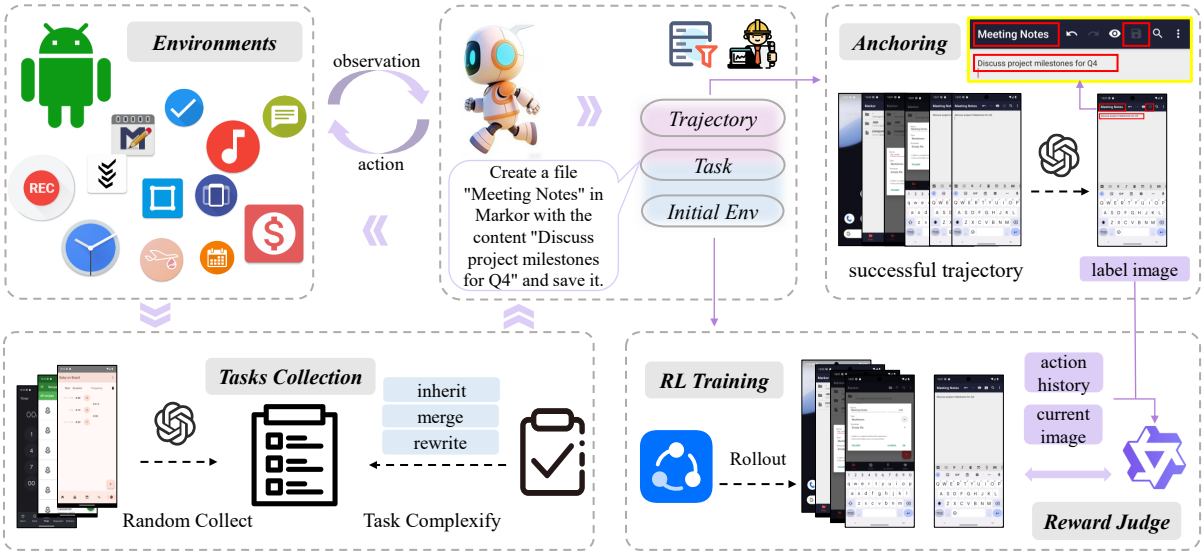


Figure 2: Overview of GSAR. (a) self-evolving data synthesis interacts with mobile environments to modify environment states and automatically generate tasks and trajectories. (b) state-anchor mechanism leverages successful trajectories to identify and anchor task-related UI elements in the goal state, providing more accurate reward signals for training.

2025; Huang et al., 2025). Initial reward designs for GUI agents primarily focused on grounding tasks (Yuan et al., 2025; Zhou et al., 2025), such as click position prediction (Lu et al., 2025) or intersection over union (IoU) measurement between predicted and target boxes (Liu et al., 2025). Yet these methods are limited in scope and lack validation on more complex high-level tasks. To support training of high-level GUI agents, subsequent studies have developed rule-based outcome rewards (Rawles et al., 2024; Luo et al., 2025) that provide reliable supervision through task-specific verification scripts. In addition, model-based reward functions have been explored for high-level tasks (Yang et al., 2025b; Shi et al., 2025), where an external model evaluates successful demonstrations or outcomes, providing rewards with high scalability across tasks.

Building upon model-based methods, we propose a reward framework designed to provide more accurate and reliable reward signals for RL training while maintaining scalability.

3 Method

In this section, we introduce GSAR (Goal-State-Anchor Reward) framework. An overview of the GSAR pipeline is illustrated in Figure 2. The framework operates through a seamless integration of two core phases. First, through **self-evolving data synthesis**, the framework progressively generates diverse tasks, execution trajectories, and their corresponding initial environments via interaction-driven evolution. Subsequently, the **state-anchor mechanism** leverages these trajectories for automatic reward annotation. By anchoring to goal states, this mechanism provides reliable and scalable reward supervision to close the learning loop.

3.1 Self-Evolving Data Synthesis

To synthesize data for training and to provide high-quality source data for reward annotation, we propose a self-evolving data synthesis framework that is both automatic and scalable. Unlike static or one-shot data generation pipelines, our approach explicitly models data construction as an evolving process. The key idea is to allow the task distribution, environment states, and trajectories to evolve together over time through continuous interaction with the GUI environment, closely mirroring how real-world applications are explored and used by humans. Task generation, execution, and trajectory collection are all driven by the model itself, enabling continuous expansion of the dataset without manual intervention. The self-evolving data synthesis framework is guided by the following principles:

Simulation of Real-World Usage. If an app has not undergone manual configuration or real user

interaction, the information contained in a freshly installed app is typically minimal and simplistic. To enable data collection in richer and more diverse environments, we modify the app state by executing multiple tasks. Through repeated rounds of task execution and environment updates, the system progressively evolves the app states. This iterative process produces a wide variety of environments, which subsequently support the collection of more diverse tasks.

Automatic Task Generation. We first explore the GUI environment of each app through random interaction, producing an exploration trajectory:

$$T = (s_0, a_0, s_1, a_1, \ldots, s_n),$$

where s_i denotes the environment state represented by the GUI screenshot, and a_i represents the action taken at step i.

Given the explored observations along the trajectory, a Vision-Language Model $\mathcal{M}$ generates a set of potential user tasks based on the visual input:

$$\mathcal{Q}_i = \mathcal{M}(s_i), \quad i \in \{0, \ldots, n\},$$

where $\mathcal{Q}_i = \{q_i^1, q_i^2, \ldots, q_i^k\}$ denotes the set of generated tasks conditioned on observation s_i.

For each generated task, we preserve the corresponding environment e as the initial state, forming task–environment pairs:

$$\mathcal{D}_{\text{task}} = \{(q, e) \mid q \in \mathcal{Q}_i\}.$$

The resulting pairs are used to initialize reinforcement learning training episodes.

Trajectory Collection and Filtering. After task creation, we employ GPT-4o to execute these tasks within their associated environments in order to obtain trajectory data while simultaneously altering the app states. Due to the inherent limitations of the model, some generated tasks may not be executable under the current GUI context, and certain collected trajectories may contain mistakes or incomplete steps. To mitigate this issue, we adopt the filtering strategy $\mathcal{F}$ proposed in OS-Genesis (Sun et al., 2024) to remove task–trajectory mismatches. As a result, we obtain triplets consisting of task q, initial environment e, and successful trajectory τ:

$$\mathcal{D}_{\text{traj}} = \{(q, e, \tau) \mid \mathcal{F}(q, \tau)\}.$$

Progression from Simple to Complex. Although numerous tasks can be produced from individual states, we observe that the model struggles to generate tasks that are both sophisticated (e.g., involving multiple sub-goals or strict execution order) and diverse (e.g., similar tasks with varying parameters). To address this limitation, we apply a task complexification process to tasks that were successfully completed in the previous iteration.

Given a completed source triplet (q_s, e_s, τ_s), where q_s denotes the source task instruction, e_s is the initial environment, and τ_s represents the successful execution trajectory, we generate a new task $q_{\text{complexify}}$ through one of three strategies—inheritance, composition, or rewriting:

$$q_{\text{complexify}} = \begin{cases} \mathcal{M}_{\text{inherit}}(q_s, \tau_s), & e = e_T, \\ \mathcal{M}_{\text{merge}}(q_s, \tau_s), & e = e_s, \\ \mathcal{M}_{\text{rewrite}}(q_s, \tau_s), & e = e_s. \end{cases}$$

Where e_T denotes the initial environment of the current iteration. $\mathcal{M}_{\text{inherit}}$ generates a follow-up task starting from e_T, $\mathcal{M}_{\text{merge}}$ augments the source task by combining it with additional subtasks starting from the original environment, and $\mathcal{M}_{\text{rewrite}}$ produces a task variant by modifying specific parameters of the source task while preserving its overall structure. Through this process, the task distribution progressively evolves from simple tasks toward more complex and diverse workflows, generating increasingly challenging tasks across iterations.

Further details and prompts for self-evolving data synthesis are provided in Appendix B and Appendix G.

3.2 State-Anchor Reward

While self-evolving data synthesis enables large-scale task and trajectory generation, effective online RL further requires accurate and scalable reward supervision. To this end, we introduce a state-anchor mechanism that supports automatic annotation with minimal human intervention.

Limitations in Action History Based Evaluation. In model-based reward evaluation, action history is often incorporated as contextual information to provide useful signals (e.g., the completion of intermediate steps), which helps the model better assess task progress. However, due to the limitations of current models, evaluators relying on historical context frequently suffer from false positives (FP). This issue is particularly pronounced in GUI scenarios,

where visually similar states may correspond to semantically different task outcomes. Compared with rule-based approaches, model-based methods are less reliable because the model lacks explicit references of completed task states, making it difficult to determine whether a task has truly been accomplished. Inspired by this observation, we introduce goal-state references that explicitly represent the expected outcome of a task. Specifically, we annotate key elements within the goal state and use them as reference signals, providing the evaluator with a form of "ground-truth answer" that improves the accuracy of model-based reward estimation.

Goal-State Acquisition. We obtain goal states from the self-evolving data synthesis stage by extracting the final screenshot of successful trajectories. Formally, given a successful task–trajectory triplet (q, e, τ) where $\tau = (s_0, a_0, \ldots, s_T)$, the goal state is defined as the terminal observation:

$$s_g = s_T.$$

For certain difficult tasks, models may fail to complete them during trajectory collection, leading to their removal during filtering. However, such tasks may still be beneficial for reinforcement learning. Therefore, for a subset of these challenging tasks, the final goal states are obtained through manual execution in order to provide reliable completion references.

Automatic Reward Annotation via Goal-State Anchoring. After obtaining the goal state, we automatically identify UI elements relevant to the task objective and use them to construct a goal-state-anchor reference. These anchored elements provide a structured representation of the expected task outcome.

Specifically, given the goal state s_g and its corresponding accessibility tree (a11y tree) A_g, we first overlay element indices from A_g onto the screenshot and then use $\mathcal{M}$ to identify the indices of task-relevant elements:

$$K_g = \mathcal{M}(\mathcal{I}(s_g, A_g), A_g),$$

where $\mathcal{I}$ denotes the operation of overlaying element indices on the screenshot. The a11y tree provides semantic and spatial information for linking UI elements to their corresponding indices.

We then retrieve the bounding boxes of the selected elements from A_g and use $\mathcal{A}$ to anchor their corresponding regions in the goal-state screenshot:

$$G = \mathcal{A}\left(s_g, \{\mathrm{bbox}(A_g^k) \mid k \in K_g\}\right).$$

The resulting G serves as the goal-state-anchor reference for reliable task completion verification. Importantly, the entire process is human-free, enabling large-scale and continual reward annotation.

Goal-State-Anchor Reward for Training. Once the anchored goal state is obtained, it can serve as a reference answer to assist the model in evaluating task completion. In addition, inspired by (Lai et al., 2025), we incorporate the action history as contextual information during evaluation. By combining the action history with the goal-state-anchor reference, the evaluator is provided with sufficient information to make more accurate judgments about task outcomes.

Formally, at time step t, the evaluator receives the current GUI state s_t, the action history $\tau_{0:t}$, and the goal-state-anchor label representation G, and outputs a binary reward indicating task completion:

$$y_t = \mathcal{E}(s_t, \tau_{0:t}, G), \quad r_t = \begin{cases} 1, & \text{if } y_t = 1, \\ 0, & \text{otherwise,} \end{cases}$$

where $\mathcal{E}$ denotes the evaluation model, $y_t \in \{0, 1\}$ indicates whether the task goal has been achieved, r_t is the reward at step t.

During training, this reward mechanism can be applied at any step of task execution to determine whether the desired outcome has been achieved, enabling reliable reward supervision for RL.

4 Experiments

4.1 Experiment Settings

Benchmarks. We use two widely used benchmarks, the AndroidControl (Li et al., 2024) and GUI-Odyssey (Lu et al., 2024), to evaluate the quality of our synthesized data. We report Type Match (TM) and Exact Match (EM) as the primary evaluation metrics. To evaluate the effectiveness of our GSAR for trajectory completion verification, we further construct an evaluation set from execution traces in AndroidWorld (Rawles et al., 2024). Specifically, we collect more than 300 trajectories containing both positive and negative examples, and report Accuracy and F1 score. To evaluate the effectiveness of the overall GSAR framework, we construct a benchmark consisting of 86 queries with their corresponding initial environment snapshots and annotated reference goal states. Half of

Model	Data Sources	AndroidControl-Low		AndroidControl-High		GUI-Odyssey	
		TM	EM	TM	EM	TM	EM
OS-Genesis-7B	Model	90.7	74.2	65.9	44.4	11.7	3.6
OS-Atlas-7B	Human&Model	73.0	67.3	70.4	56.5	91.8*	76.8*
Aguvis-7B	Human&Model	93.9	<u>89.4</u>	65.6	54.2	26.7	13.5
Qwen2.5-VL-7B	–	94.1	85.0	75.1	62.9	59.5	46.3
Ours (aw-app)	Model	**95.4**	**90.2**	<u>76.1</u>	**66.1**	**65.8**	**48.4**
Ours (extra-app)	Model	<u>95.3</u>	89.3	**76.3**	<u>65.2</u>	<u>65.4</u>	<u>47.6</u>

Table 1: Performance comparison on the AndroidControl and GUI-Odyssey benchmarks. Results are reported without the open action type. *OS-Atlas employs different train/test splits on GUI-Odyssey and is thus not directly comparable. **Bold** and <u>underline</u> indicate the best and second-best results.

the queries are used as the training split. We report the Success Rate (SR) on our self-built benchmark, with results verified through both manual assessment and GSAR.

Baselines. To evaluate the quality of the synthesized data, we selected three advanced open-source models, Os-Genesis (Sun et al., 2024), OS-Atlas (Wu et al., 2024), and Aguvis (Xu et al., 2024), for comparison with our fine-tuned model. Moreover, we compare our method with three representative model-based approaches for reward verification from prior work: DigiRL (Bai et al., 2024), DistRL (Wang et al., 2024), and StepCritic (Lai et al., 2025). Additionally, we include two baselines that use only the goal-state screenshot and only the anchored goal-state screenshot as input, denoted as GS-Only and GSA-Only.

Training Details. For the supervised fine-tuning (SFT) experiments on synthesized trajectory data, we adopt Qwen2.5-VL-7B (Bai et al., 2025) as the base model and fine-tune it using LoRA (Hu et al., 2021). For model-based reward evaluation, we use several vision-language models as judge models, including Qwen3-VL-8B, Qwen3-VL-32B (Yang et al., 2025a), GPT-4o (Hurst et al., 2024), and Gemini-2.5-Pro (Comanici et al., 2025). In the online reinforcement learning (RL) experiments, we adopt UI-TARS-7B-DPO (Qin et al., 2025) and GUI-Owl-7B (Ye et al., 2025) as the backbone models, and use GRPO (Shao et al., 2024) as the optimization algorithm, with Qwen3-VL-32B serving as the judge model. Detailed experimental settings are provided in Appendix C.

4.2 Main Results

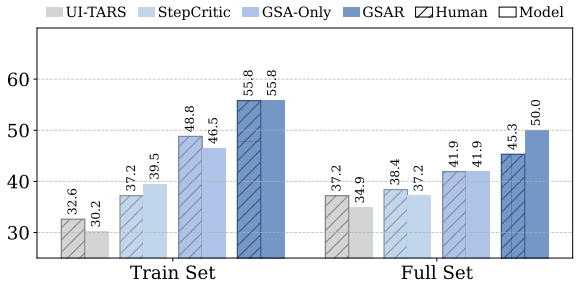
We first evaluate the fine-tuned model to validate data quality, then assess the accuracy of GSAR on offline trajectories, and finally verify that the entire pipeline scales effectively to online RL.

Evaluation of Synthesized Data Quality. We evaluate the effectiveness and scalability of synthesized data for GUI agent training. The model is fine-tuned on data collected from 20 AndroidWorld apps and further scaled using data from 40 selected open-source apps. As shown in Table 1, fine-tuning on synthesized data leads to consistent improvements across both benchmarks, with notable gains on the EM metric for the low and high splits of AndroidControl (+5.2% and +3.2%, respectively) and on the TM metric for GUI-Odyssey (+6.3%). When scaling to additional open-source apps, the model achieves similarly strong performance, although slightly lower due to distribution differences between datasets. These results indicate that **our synthesized data effectively improves model performance and generalizes well across diverse application domains.** Additional results on other base models are provided in Appendix D.

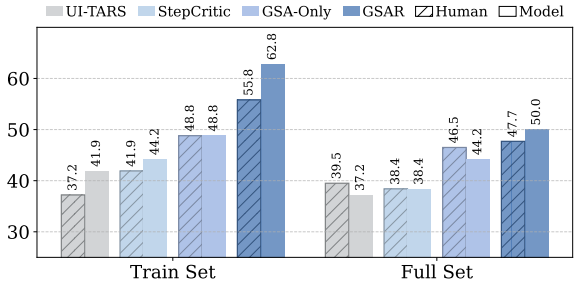
Offline Evaluation for GSAR. We conduct offline evaluations to assess the ability of GSAR in judging task completion, comparing it with three model-based reward methods as shown in Table 2. As model capabilities continue to improve, the accuracy and F1 scores of the three baseline methods also increase. However, they still lag behind the rule-based approach, whose accuracy and F1 are theoretically 100%. Providing the model with a reference state (GS-Only or GSA-Only) significantly enhances its evaluation capability. GSAR achieves the highest results in evaluating trajectory correctness, being the only method to surpass 90% in both accuracy and F1 for both open-source and closed-source models. This demonstrates that **GSAR most closely approximates rule-based evalua-**

Method	Qwen3-VL-8B		Qwen3-VL-32B		GPT-4o		Gemini-2.5-pro		Avg	
	Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1
DigiRL	67.0	64.9	71.4	65.9	63.8	56.2	71.1	64.3	68.3	62.8
DistRL	67.9	69.3	80.0	79.7	69.8	66.9	82.5	82.3	75.0	74.5
StepCritic	64.4	74.1	76.8	81.0	81.3	84.0	87.9	88.3	77.6	81.8
GS-Only	84.1	84.2	83.8	83.9	78.1	78.8	86.0	85.9	83.0	83.2
GSA-Only	87.6	88.2	86.4	86.9	81.3	80.9	89.8	90.1	86.3	86.5
GSAR	**90.2**	**91.0**	**92.1**	**92.4**	**91.1**	**91.3**	**92.7**	**92.7**	**91.5**	**91.8**

Table 2: Offline trajectory evaluation results of different model-based reward methods.

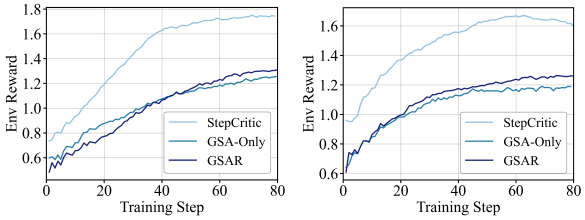


(a) UI-TARS-7B-DPO



(b) GUI-Owl-7B

Figure 3: Success rate on the self-built benchmark. *Model* indicates that GSAR is used as the evaluation method.



(a) Reward on UI-TARS. (b) Reward on GUI-Owl.

Figure 4: Online training reward curves of three reward mechanisms on the UI-TARS-7B-DPO and GUI-Owl-7B models. Each curve represents the per-step reward across training steps.

Method	Qwen3-VL-8B		Qwen3-VL-32B	
	Acc	F1	Acc	F1
w/o reference	64.4	74.1	76.8	81.0
w/o anchor	87.3	88.0	90.5	90.7
w/o history	87.6	88.2	86.4	86.9
GSAR (Ours)	**90.2**	**91.0**	**92.1**	**92.4**

Table 3: The results of the ablation study.

tion while retaining scalability advantages.

RL Training with the Full Framework. To further assess the effectiveness and efficiency of GSAR, we use our self-built benchmark, including triplets (task, initial environment, goal-state-anchor reference), for online RL. As shown in Figure 3, the model trained with GSAR as the reward function achieves a 23.2% improvement on the training split and an 8.1% improvement on the full task set compared to the UI-TARS. A similar trend is observed for GUI-Owl, with gains of 18.6% and 8.2%, respectively. Moreover, incorporating the goal-state-anchor reference substantially enhances task execution over StepCritic, while adding action history also improves training compared to GSA-Only. Verification using GSAR shows similar trends to human evaluation, with some metrics closely matching human judgments. Overall, **the proposed pipeline effectively enables scalable RLVR training in GUI agents and improves their performance.**

4.3 Ablation Study.

We perform ablation experiments on offline verification to assess the contribution of each component. Specifically, we test three variants: (1) w/o history, which removes the action history; (2) w/o anchor, which removes the anchor on the goal-state screenshot; and (3) w/o reference, which removes the goal-state screenshot.

As shown in Table 3, **combining the action history with the anchored reference screenshot as input yields the best performance, and removing either component leads to a noticeable**

Type	Answer	Delete	Normal	Total
Count	20	15	47	82
Accuracy	90.0	100.0	89.4	91.5

Table 4: Automatic annotation accuracy of goal-state anchoring across task categories.

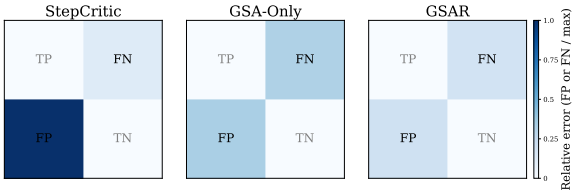


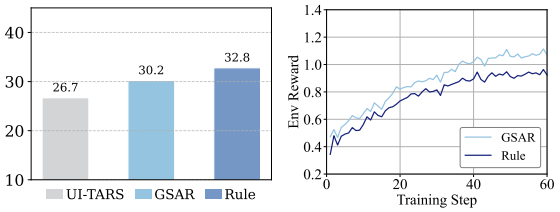
Figure 5: Comparison of false positives (FP) and false negatives (FN) across different reward designs. StepCritic reduces FN but leaves FP high, GSA-Only reduces FP but some FN remain, and combining both mechanisms balances the reward signal while mitigating both FP and FN errors.

degradation. Omitting the reference image leads to substantially lower accuracy compared to the rule-based approach, as the model loses information about the final task state. Excluding the anchor annotations from the completed reference also degrades performance, since VLMs may fail to perceive subtle visual differences between similar incomplete and completed screenshots. Additionally, because some tasks involve critical intermediate steps, discarding the action history removes this contextual information, negatively affecting the final decision.

5 Analysis Experiments

5.1 Annotation Accuracy by Task Category

Table 4 summarizes the accuracy of automatic goal-state annotation via GPT-4o across task categories. The overall annotation accuracy reaches 91.5%. Delete tasks achieve 100% accuracy because the deleted object no longer appears in the goal state, eliminating the need for fine-grained localization; therefore, these cases are treated as fully correct. For Answer and Normal tasks, accuracy is slightly lower, as the goal state may contain multiple small or scattered key elements that the model must individually identify and localize, posing a greater challenge. Overall, the automatic annotation pipeline provides reliable annotations across different task categories and can substantially reduce the need for manual annotation.



(a) Evaluation results (b) Reward curves

Figure 6: Evaluation results on AndroidWorld and online training reward curves of the UI-TARS model.

Reward Method	Average Rollout Time (s)
Rule-based	2754.92
GSAR (Model-based)	**2342.02**

Table 5: Average rollout time per training step for different reward methods.

5.2 Understanding the Role of Reward Signals in RL Training

We investigate the impact of reward signals on reinforcement learning by comparing the performance of models trained with different reward functions. Although StepCritic exhibits relatively low false negative rates in offline evaluation, it suffers from a notably high false positive rate (Figure 5). As a result, its reward signals during RL training are overly optimistic (Figure 4), leading to policies that fail to consistently translate into strong online performance and producing agents that are less stable and less reliable than ours (Figure 3). This suggests that providing more accurate reward signals can lead to greater gains in RL training.

5.3 Comparison with Rule-based Methods

Performance Comparison. We conduct a comparative study between GSAR and rule-based rewards on AndroidWorld (Rawles et al., 2024). We select a subset of 42 tasks as the training set, while evaluation is conducted on the full benchmark. As shown in Figure 6a, both GSAR and rule-based rewards improve the success rate over the UI-TARS-7B-DPO baseline (26.7%). GSAR increases the performance to 30.2% (+3.5%), while rule-based rewards achieve 32.8% (+6.1%) due to their near-noiseless verification signals. Meanwhile, the reward curves in Fig. 6b show that GSAR follows a trend highly consistent with rule-based rewards, demonstrating that it can provide stable optimization signals without relying on manually designed rules, offering better scalability and generalization.

Rollout Efficiency. Both methods perform reward evaluation on a step-wise basis during RL training. GSAR introduces additional latency due to model-based evaluation after each action. However, for most tasks, the latency introduced by model inference is lower than the overhead incurred by repeated ADB-based device access for rule-based verification. As shown in Table 5, GSAR therefore achieves lower overall rollout latency than the rule-based baseline.

6 Conclusion

We propose GSAR (Goal-State-Anchor Reward), a framework that integrates self-evolving data synthesis with a state-anchor mechanism. To enhance environmental diversity during data synthesis, we introduce an iterative mechanism that evolves environments through task execution and incorporate task complexification to generate diverse tasks. Furthermore, we leverage final states of successful trajectories to anchor task-relevant UI elements, providing richer context for model-based reward evaluation. Our experiments demonstrate that GSAR achieves over 90% accuracy in trajectory verification and improves performance over the base model in online RL training. Moreover, the reward curve of GSAR closely aligns with the rule-based reward, demonstrating its ability to provide stable optimization signals without relying on handcrafted rules. Overall, GSAR provides a scalable data synthesis paradigm and provides a reliable reward framework for training GUI agents.

Limitations

Our method addresses the scalability issues of rule-based approaches and the accuracy limitations of model-based approaches. However, it considers only a single final state as the success criterion for each task, whereas a task may be achievable through multiple alternative solutions. Moreover, for question-answering tasks, our method may assign a correct reward before the actual correct answer is produced. For tasks whose completion cannot be fully expressed visually (e.g., deleting certain items), it is difficult for the model to annotate reasonable task-relevant elements, and natural language or other modalities may be necessary to describe the final state. Future work should involve a more fine-grained categorization and handling of task types during data collection and training.

Meanwhile, both the automated task execution for trajectory collection and the filtering process are performed by VLMs (e.g., GPT-4o), eliminating the need for manual annotation. However, the quality of the generated trajectories is sometimes constrained by the capabilities of the models. For tasks that cannot be completed automatically, manual execution is still required to obtain the correct goal state if they are to be used for reinforcement learning. We expect that stronger GUI-capable models in the future will further improve the utilization of model-generated data.

References

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

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, and 8 others. 2025. Qwen2.5-vl technical report. *ArXiv*, abs/2502.13923.

Yuhan Chen, Yuxuan Liu, Long Zhang, Pengzhi Gao, Jian Luan, and Wei Liu. 2025. Step: Success-rate-aware trajectory-efficient policy optimization. *ArXiv*, abs/2511.13091.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Jun-Mei Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiaoling Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 179 others. 2025. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645:633 – 638.

Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Y. Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A mobile app dataset for building data-driven design applications. *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. 2025. Group-in-group policy optimization for llm agent training. *ArXiv*, abs/2505.10978.

Jiaxuan Gao, Shusheng Xu, Wenjie Ye, Weiling Liu, Chuyi He, Wei Fu, Zhiyu Mei, Guangju Wang, and Yi Wu. 2024. On designing effective rl reward at training time for llm reasoning. *ArXiv*, abs/2410.15115.

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

Wenxuan Huang, Bohan Jia, Zijie Zhai, Shaoshen Cao, Zheyu Ye, Fei Zhao, Zhe Xu, Yao Hu, and Shaohui Lin. 2025. Vision-r1: Incentivizing reasoning capability in multimodal large language models. *ArXiv*, abs/2503.06749.

OpenAI Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Mkadry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alexander Kirillov, Alex Nichol, Alex Paino, and 397 others. 2024. Gpt-4o system card.

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

Nathan Lambert, Jacob Daniel Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James Validad Miranda, Alisa Liu, Nouha Dziri, Xinxi Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Christopher Wilhelm, Luca Soldaini, and 4 others. 2024. Tülu 3: Pushing frontiers in open language model post-training. *ArXiv*, abs/2411.15124.

Wei Li, Will Bishop, Alice Li, Christopher Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. 2024. On the effects of data scale on ui control agents. *Advances in Neural Information Processing Systems 37*.

Mu Lin, Minghao Liu, Taoran Lu, Lichen Yuan, Yiwei Liu, Haonan Xu, Yu Miao, Yuhao Chao, and Zhaojian Li. 2025. Gui-rewalk: Massive data generation for gui agent via stochastic exploration and intent-aware reasoning. *ArXiv*, abs/2509.15738.

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

Quanfeng Lu, Wenqi Shao, Zitao Liu, Fanqing Meng, Boxuan Li, Botong Chen, Siyuan Huang, Kaipeng Zhang, Yu Qiao, and Ping Luo. 2024. Gui odyssey: A comprehensive dataset for cross-app gui navigation on mobile devices. *ArXiv*, abs/2406.08451.

Zhengxi Lu, Yuxiang Chai, Yaxuan Guo, Xiaojing Yin, Liang Liu, Hao Wang, Guanjing Xiong, and Hongsheng Li. 2025. Ui-r1: Enhancing efficient action prediction of gui agents by reinforcement learning.

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

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. 2022. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, and 16 others. 2025. Ui-tars: Pioneering automated gui interaction with native agents. *ArXiv*, abs/2501.12326.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, G. R. Lau, Marybeth Fair, Alice Li, Will Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy P. Lillicrap, and Oriana Riva. 2024. Androidworld: A dynamic benchmarking environment for autonomous agents. *ArXiv*, abs/2405.14573.

Christopher Rawles, Alice Li, Daniel Rodríguez, Oriana Riva, and Timothy P. Lillicrap. 2023. Android in the wild: A large-scale dataset for android device control. *ArXiv*, abs/2307.10088.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junmei Song, Mingchuan Zhang, Y. K. Li, Yu Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *ArXiv*, abs/2402.03300.

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

Qiushi Sun, Kanzhi Cheng, Zichen Ding, Chuanyang Jin, Yian Wang, Fangzhi Xu, Zhenyu Wu, Chengyou Jia, Liheng Chen, Zhoumianze Liu, Ben Kao, Guohao Li, Junxian He, Yu Qiao, and Zhiyong Wu. 2024. Os-genesis: Automating gui agent trajectory construction via reverse task synthesis. *ArXiv*, abs/2412.19723.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, Wanjun Zhong, Yining Ye, Yujia Qin, Yuwen Xiong, Yuxin Song, Zhiyong Wu, Bo Li, Chen Dun, Chong Liu, and 87

others. 2025. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *ArXiv*, abs/2509.02544.

Taiyi Wang, Zhihao Wu, Jianheng Liu, Jianye Hao, Jun Wang, and Kun Shao. 2024. Distrl: An asynchronous distributed reinforcement learning framework for on-device control agents. *ArXiv*, abs/2410.14803.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, and Yu Qiao. 2024. Os-atlas: A foundation action model for generalist gui agents. *ArXiv*, abs/2410.23218.

Bin Xie, Rui Shao, Gongwei Chen, Kaiwen Zhou, Yinchuan Li, Jie Liu, Min Zhang, and Liqiang Nie. 2025. Gui-explorer: Autonomous exploration and mining of transition-aware knowledge for gui agent. In *Annual Meeting of the Association for Computational Linguistics*.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *ArXiv*, abs/2404.07972.

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

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

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025a. Qwen3 technical report.

Chenyu Yang, Shiqian Su, Shi Liu, Xuan Dong, Yue Yu, Weijie Su, Xuehui Wang, Zhaoyang Liu, Jinguo Zhu, Hao Li, Wenhai Wang, Yu Qiao, Xizhou Zhu, and Jifeng Dai. 2025b. Zerogui: Automating online gui learning at zero human cost. *ArXiv*, abs/2505.23762.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, Jitong Liao, Qi Zheng, Fei Huang, Jingren Zhou, and Ming Yan. 2025. Mobile-agent-v3: Fundamental agents for gui automation. *ArXiv*, abs/2508.15144.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, and 16 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *ArXiv*, abs/2503.14476.

Xinbin Yuan, Jian Zhang, Kaixin Li, Zhuoxuan Cai, Lujian Yao, Jie Chen, Enguang Wang, Qibin Hou, Jinwei Chen, Peng-Tao Jiang, and Bo Li. 2025. Enhancing visual grounding for gui agents via self-evolutionary reinforcement learning. *ArXiv*, abs/2505.12370.

Zhong Zhang, Ya-Ting Lu, Yikun Fu, Yupeng Huo, Shenzhi Yang, Yesai Wu, Han Si, Xin Cong, Haotian Chen, Yankai Lin, Jie Xie, Wei Zhou, Wang Xu, Yuanheng Zhang, Zhou Su, Zhongwu Zhai, Xiao-Meng Liu, Yudong Mei, Jianming Xu, and 6 others. 2025. Agentcpm-gui: Building mobile-use agents with reinforcement fine-tuning. *ArXiv*, abs/2506.01391.

Yuqi Zhou, Sunhao Dai, Shuai Wang, Kaiwen Zhou, Qinglin Jia, and Jun Xu. 2025. Gui-g1: Understanding r1-zero-like training for visual grounding in gui agents. *ArXiv*, abs/2505.15810.

A Benchmarks

Here we provide additional details on the benchmarks used to evaluate GSAR.

AndroidControl. AndroidControl (Li et al., 2024) is a large-scale dataset for human-computer interface control on Android devices, containing 15,283 task demonstrations across 833 apps. Each task includes both high-level and low-level human-generated instructions, covering 14,548 unique tasks. The dataset enables analysis of agent performance on tasks of varying complexity, both within the training domain and out-of-domain. AndroidControl supports studies on how fine-tuning with more data affects performance, particularly highlighting the challenges of generalizing to unseen apps or higher-level tasks.

GUI-Odyssey. GUI-Odyssey (Lu et al., 2024) is a large-scale dataset for cross-app mobile GUI navigation, consisting of 8,334 episodes with an average of 15.3 steps per episode. It spans 6 mobile devices, 212 apps, and 1,357 app combinations. Each step includes detailed semantic reasoning annotations to support models in complex cross-app reasoning and decision-making. The dataset enables training and evaluation of agents capable of long-step, multi-app navigation tasks.

Following (Zhang et al., 2025), we removed steps related to the "open_app" action type in the evaluation of these benchmarks.

B Self-Evolving Data Synthesis

In this section, we present the algorithmic workflow of the self-evolving data synthesis framework used for generating tasks and collecting trajectories in mobile GUI environments. The framework iteratively collects tasks, evolves the GUI environment, and generates increasingly complex instructions based on previously collected trajectories. The workflow, summarized in Algorithm 1, illustrates how the system produces a diverse and high-quality dataset suitable for reinforcement learning training and goal-state–anchored reward evaluation.

We use GPT-4o (Hurst et al., 2024) for task generation, task complexification, and task execution. For AndroidWorld apps, we set the maximum iteration to 6. For additional open-source apps, we first allow the model to perform a single random exploration to generate trajectories, and then count the number of unique pages in the trajectory screenshots, mapping this number to a range of 1–6 to determine the maximum number of self-evolving iterations for that app. This approach ensures that apps with more diverse interactions undergo more iterations, while avoiding unnecessary exploration time for simpler apps.

Algorithm 1 Self-Evolving Data Synthesis

Require: Device serial d, package name p, maximum iterations T,
maximum tasks per page M_t, maximum breadth M_b, maximum depth M_d,
maximum steps per task $S_{\max}$
Ensure: Task set $\mathcal{Q}$ and trajectory set $\mathcal{D}$
Initialize task set $\mathcal{Q} \leftarrow \emptyset$
Initialize trajectory set $\mathcal{D} \leftarrow \emptyset$
Initialize previous trajectory set $\mathcal{D}_{prev} \leftarrow \emptyset$
for iteration $t = 1 \dots T$ **do**
  Pull environment E_t from device
  Collect candidate tasks $\mathcal{Q}_t = \textsc{CollectTask}(E_t, M_t, M_b, M_d)$
  if $t > 1$ and $\mathcal{D}_{prev} \neq \emptyset$ **then**
    for each trajectory $\tau \in \mathcal{D}_{prev}$ **do**
      if τ is completed and not complexified **then**
        Generate complexified tasks $\mathcal{Q}' = \textsc{ComplexifyTask}(\tau, E_t)$
        $\mathcal{Q}_t \leftarrow \mathcal{Q}_t \cup \mathcal{Q}'$
      end if
    end for
  end if
  Update task set $\mathcal{Q} \leftarrow \mathcal{Q} \cup \mathcal{Q}_t$
  Push environment E_t back to device
  Execute tasks $\mathcal{Q}_t$ with step limit $S_{\max}$
  Collect resulting trajectories $\mathcal{D}_t$
  Update trajectory set $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_t$
  Set $\mathcal{D}_{prev} \leftarrow \mathcal{D}_t$
end for
return $\mathcal{Q}, \mathcal{D}$

C Details of Experiments

Supervised Fine-Tuning. We provide the detailed experimental settings for the supervised fine-tuning (SFT) stage in Table 6, including the training data, model configuration, and key hyperparameters used during training. For all fine-tuned models, the training dataset contains 6,331 steps, covering both low-level and high-level instruction types, where the low-level instructions correspond to the action reasoning output.

Hyperparameters	All methods
Fine-tuning method	LoRA
LoRA rank (r)	8
LoRA α	16
Train batch size	128
Training epochs	3
Learning rate	1×10^{-5}
LR Scheduler	Cosine
Warmup Ratio	0.1
GPU numbers	8

Table 6: Training hyperparameters for SFT.

Offline Verification for GSAR. The datasets used to evaluate GSAR and other methods are derived from the evaluation trajectories of GUI-Owl-7B and Qwen2.5-VL-7B on the AndroidWorld benchmark. To better assess the differences between methods, we constructed a total of 315 trajectories, including 164 positive samples and 151 negative samples. For each task, the goal state is selected from the last step screenshot and anchored using GPT-4o. To ensure fairness, we strictly follow the experimental settings and prompts described in the original papers when using these baselines.

Hyperparameters	All methods
Train batch size	32
PPO batch size	256
Training steps	80
Max turn	20
Rollout numbers	8
Temperature	0.7
Learning rate	1×10^{-6}
KL coefficient	0.001
Clip ratio	0.2

Table 7: Training hyperparameters for RL.

Online Reinforcement Learning. The experimental configurations for the reinforcement learning (RL) stage are summarized in Table 7, including the backbone agent, reward evaluation model, optimization algorithm, and other key training parameters used during online RL. The experiments were conducted on two nodes, each equipped with $8 \times$ NVIDIA H100 GPUs.

To support large-scale environment interactions while decoupling the GUI environment from the training process, we deployed a service of 128 Android emulators for parallel trajectory sampling. Specifically, each of the two machines launched 64 emulators, providing a total of 128 emulators that enable the agent to interact with the environment independently of the training, thereby facilitating efficient training and data collection.

D Supplementary SFT Results

To further verify the generalization of our synthesized data across different base models, we fine-tune both UI-TARS-7B-SFT and UI-TARS-7B-DPO (Qin et al., 2025) on the same training data, as shown in Table 8. Across all three benchmarks, our method performs on par with, and in several cases surpasses, the strong UI-TARS baseline, and scaling to additional open-source apps yields comparable results. We note that the margins of improvement on UI-TARS are narrower than those observed on Qwen2.5-VL-7B (Bai et al., 2025) (Table 1), likely because UI-TARS is already specialized for GUI tasks and achieves substantially stronger baseline performance, leaving less room for further gains. These results confirm that the quality of our synthesized data is not tied to a specific base model, and the benefits transfer across different model initializations and capabilities.

E Additional Analysis Results

E.1 Analyzing the Effectiveness of Self-Evolving Data Synthesis

Impact of Self-Evolving Mechanism. To evaluate the effect of the self-evolving mechanism on page complexity, we sample 300 tasks each from early iterations and later iterations, as summarized in Table 9. For each task, one state is randomly sampled from its trajectory to compute page complexity metrics. The results show that pages from later iterations exhibit slightly higher edge density, increased image entropy, and more UI elements on average. These observations suggest that the self-evolving process gradually increases the visual and functional complexity of generated pages, leading to richer and more diverse environments.

Effect of Task Complexity. We evaluate the impact of task complexification on a filtered set of 339 samples for each setting, as shown in Table 10. After task complexification, both the average length of natural language task descriptions and the aver-

Model	Data Sources	AndroidControl-Low		AndroidControl-High		GUI-Odyssey	
		TM	EM	TM	EM	TM	EM
UI-TARS-7B-SFT	–	**95.05**	91.52	79.10	71.21	80.71	66.98
GSAR (aw-app)	Model	94.99	**91.53**	80.33	71.97	81.78	**68.01**
GSAR (extra-app)	Model	95.01	91.48	**80.76**	**72.41**	**81.90**	67.93
UI-TARS-7B-DPO	–	92.05	88.74	80.77	73.56	85.29	68.33
GSAR (aw-app)	Model	92.82	89.52	81.04	73.85	85.56	**71.28**
GSAR (extra-app)	Model	**93.33**	**90.01**	**81.28**	**74.09**	**85.90**	71.15

Table 8: Performance comparison of UI-TARS-7B-SFT and UI-TARS-7B-DPO fine-tuned on our synthesized data, evaluated on the AndroidControl and GUI-Odyssey benchmarks. Results are reported without the open action type. **Bold** and underline indicate the best and second-best results within each block.

Stage	Edge Density	Entropy	UI Elements
Early (Iter 1–2)	0.0240	1.90	22.68
Later (Iter 5–6)	**0.0295**	**2.11**	**24.56**

Table 9: Impact of self-evolving process on GUI page complexity.

Task Type	Task string length	Trajectory length
Normal	144.41	7.91
Complexified	**338.92**	**12.08**

Table 10: Effect of task complexity on average query string length and trajectory length.

Dataset	R=1	R=2	R=3	R=4	R=5	$R \geq 4$ Ratio
aw-app	44	281	214	126	328	45.7%
extra-app	110	342	196	83	229	32.5%

Table 11: Data utilization statistics of synthesized trajectories.

age trajectory length increase, demonstrating the effectiveness of the task complexification process.

Data Utilization Analysis We analyze the data utilization rate of trajectories generated by our self-evolving data synthesis pipeline, as shown in Table 11. The aw-app and extra-app datasets contain 993 and 960 tasks, respectively, and trajectories with reward score $R \geq 4$ are used for fine-tuning. The results show that a substantial portion of generated trajectories can be effectively utilized for training. The aw-app dataset achieves a higher utilization rate of 45.7%, benefiting from the carefully selected AndroidWorld applications with stable and complete functionalities. In contrast, the extra-app dataset has a lower utilization rate of 32.5%, since it consists of open-source applications, some of which have incomplete functionality or belong to relatively niche usage scenarios.

E.2 Offline Evaluation by Task Category

Table 12 reports classification results by task category using Gemini-2.5-Pro (Comanici et al., 2025). GSAR demonstrates consistently strong performance across all task types, highlighting both the varying difficulty of trajectory completion judgment across categories and the effectiveness of GSAR in handling them.

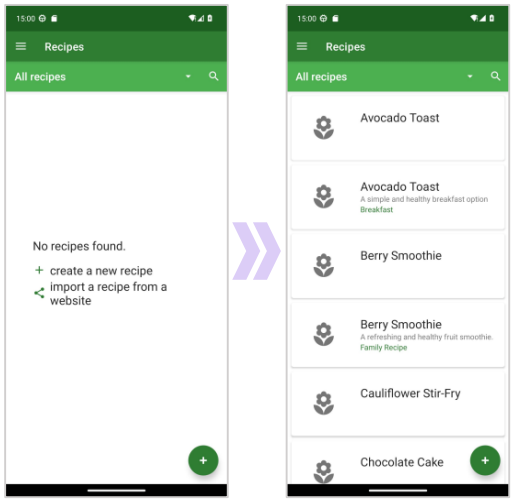
StepCritic outperforms both GSA-Only and GS-Only, indicating that action history plays a critical role in determining completion for Delete type tasks. Notably, GSA-Only performs worse than GS-Only on Delete tasks. This may be attributed to the fact that the added anchor boxes tend to cover large screen regions, causing the model to attend to irrelevant elements and thereby degrading judgment accuracy.

Answer tasks require a combination of semantic understanding and goal state reference for accurate judgment. DigiRL yields an F1 of zero on this task type because its frame similarity filter compares the last two screenshots and labels trajectories with minimal visual change as incomplete. Since Answer tasks typically exhibit little visual difference between the final two frames, DigiRL fails to identify any positive samples, resulting in zero true positives for both precision and recall, and consequently an F1 of zero. Comparing GS-Only and GSA-Only, anchoring task-relevant UI elements on the goal state brings substantial gains for Answer tasks.

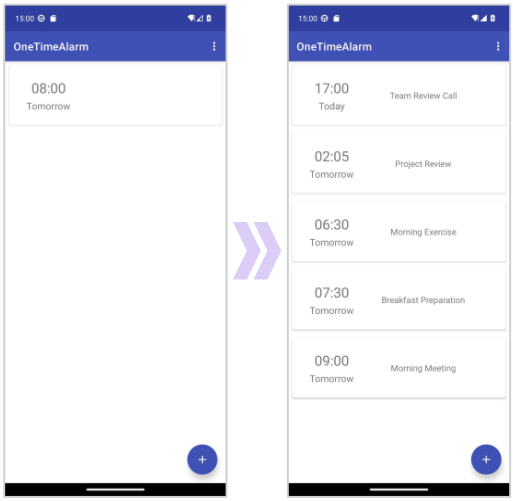
For Normal tasks, successful and incomplete trajectories exhibit distinct differences in key UI elements, allowing strong performance to be achieved even without action history.

Method	Delete		Answer		Normal		Overall	
	Acc	F1	Acc	F1	Acc	F1	Acc	F1
DigiRL	81.4	80.0	41.4	0.0	79.0	75.5	71.1	64.3
DistRL	86.4	86.2	75.7	80.0	83.9	82.1	82.5	82.3
StepCritic	96.6	96.7	77.1	77.8	89.3	89.7	87.9	88.3
GS-Only	91.5	92.1	77.1	79.0	87.6	86.7	86.0	85.9
GSA-Only	88.1	88.5	85.7	88.1	**91.9**	**91.5**	89.8	90.1
GSAR	**98.3**	**98.3**	**90.0**	**90.9**	**91.9**	**91.5**	**92.7**	**92.7**

Table 12: Offline trajectory evaluation results of different model-based reward methods across task categories.



(a) Broccoli



(b) OneTimeAlarm

Figure 7: Examples of environment state changes under self-evolving data synthesis.

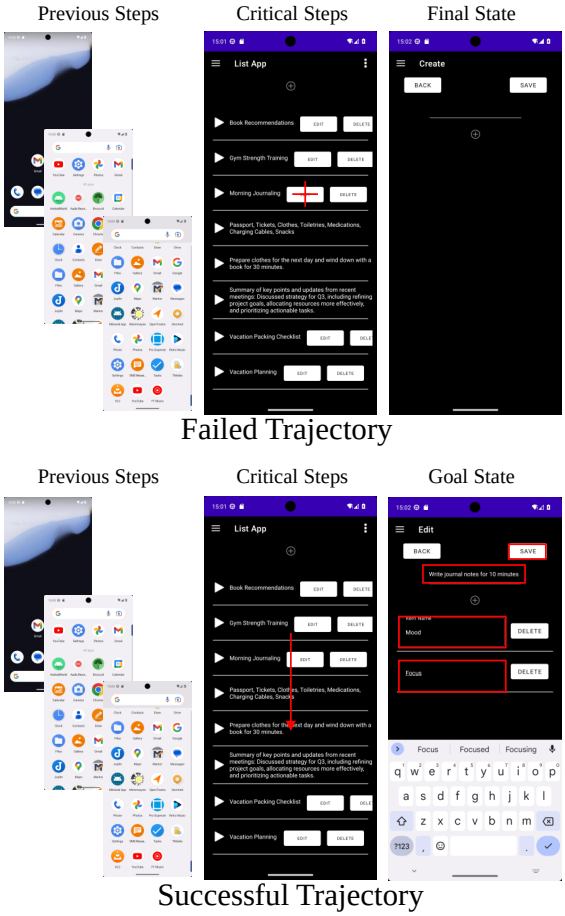


Figure 8: Comparison of UI-TARS-trained agents for task: In Mnemosyne, open the "Write journal notes for 10 minutes" item, set the first entry to "Mood" and the second entry to "Focus", without saving it.

F Case Study

F.1 Case Study for Self-Evolving

We show examples of the self-evolving mechanism illustrating how it modifies the environment state during the iterative process in Figure 7. It can be clearly observed that the app's main pages become increasingly diverse during the later stages of evolution, enabling the collection of more complex and varied tasks for subsequent training.

F.2 Case Study for Online Performance

To further illustrate the impact of different reward methods on agent behavior during online interactions, we present two representative case studies

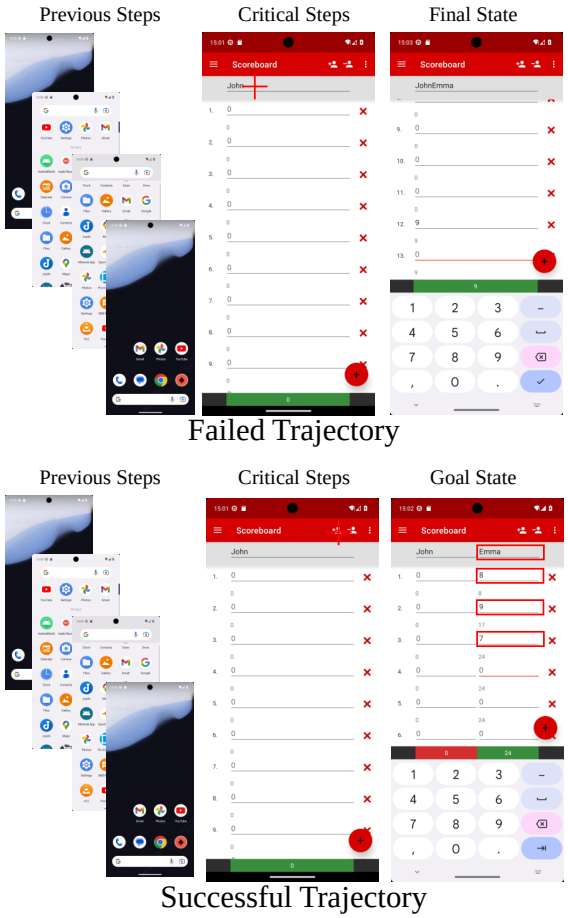


Figure 9: Comparison of GUI-Owl-trained agents for task: Add a new player named Emma to the Scoreboard app, and set her scores to 8 points in Round 1, 9 points in Round 2, and 7 points in Round 3.

in Figure 8 and Figure 9. One shows a failed trajectory from an agent trained with StepCritic, while the other demonstrates a successful trajectory from an agent trained with GSAR. These examples highlight how using our reward framework enables agents to perform complex mobile GUI tasks more reliably and accurately.

G Prompts

G.1 Prompting Template of Task Generation

You are an expert in analyzing mobile app user interfaces and generating clear, purposeful, and executable task instructions.
You will receive:
- The app name
- A current mobile screenshot
- UI component information extracted from the screenshot
- The action history from app launch to the current screen

Your goal is to analyze the current user interface and generate purposeful instructions based on the action history, representing the tasks a human user is likely to perform next.
Task types:
- Task-Oriented: A series of operations to achieve a specific goal.
- Question-Oriented: A series of operations that lead to answering a specific question.
Requirements:
- Each task must mention the app name and provide a high-level instruction without referencing any specific UI elements.
- You can combine multiple simple tasks (from the example tasks) into richer, more complex, and more diverse tasks.
- If the task involves entering information, specify the exact content to be input.
- DO NOT use words such as 'or', 'any', 'e.g.', or any other expression that makes the task description vague or ambiguous.
- DO NOT mention any non-existent files.
- DO NOT generate tasks related to user login, account registration, account creation, or authentication flows.
- Do not generate any tasks if the screen is a permission dialog, an app initialization popup, or unrelated to the current app.
- Generate between 0 and `{max_tasks}` tasks depending on the complexity of the screen and action history.
Context:
- App name: `{app_name}`
- UI components: `{accessibility_tree}`
- Action history: `{action_summary}`
Example tasks: `{examples`
Output format (strictly follow this format):
your analysis:
- A brief explanation of the navigation history, user intent, and key UI elements that affect the tasks.
your tasks:
1. Task description
2. Task description
3. Task description
...
Do not add any extra sections or commentary outside this format.

G.2 Prompting Template of Task Inherit

You are an expert at generating follow-up mobile app tasks based on an existing completed task.
You will receive:
- A user task trajectory, including the initial user goal, the last few screenshots, and a sequence of action summaries that indicate the task has already been completed.
Your goal is to generate a new task that STARTS FROM the environment state after the previous task is finished.
The new task should build upon the existing result and introduce additional steps or subtasks to continue the workflow.
Requirements:
- The new task must assume the previous task has already been successfully completed.
- The task must start from the current environment state shown in the screenshots.
- The task should extend the workflow by adding new steps or subtasks, without repeating the original task.
- The task must mention the app name and provide a high-level instruction without referencing any specific UI elements.
- If the task involves entering information, specify the exact content to be input.
- DO NOT use words such as 'or', 'any', 'e.g.', or any other expression that makes the task description vague or ambiguous.
- DO NOT mention any non-existent files.
Input:
- App name: `{app_name}`
- Previous completed user goal: `{initial_goal}`
- Action summary: `{action_summary}`
- Current environment screenshots: <image>
Output format (strictly follow this format):
Your analysis:
- A brief explanation of how the new task continues from the completed state and what new steps are added.
Output task:
<task>
(contain only a single task; if no task needs to be generated, write "No task")
</task>
Do not add any extra sections or commentary outside this format.

G.3 Prompting Template of Task Merge

You are an expert at generating more complex mobile app tasks by merging an existing task with additional subtasks.
You will receive:
- A user task trajectory, including the initial user goal, the last few screenshots, and a sequence of action summaries.
Your goal is to generate a NEW combined task that integrates the original task and newly added steps into a single, more complex task.
The initial environment of the new task corresponds to the environment state shown in the screenshots.
Requirements:
- The new task must explicitly include the original task goal and extend it with additional steps or subtasks.
- The task should describe a complete, merged workflow as a single high-level user goal.
- The task must mention the app name and provide a high-level instruction without referencing any specific UI elements.
- If the task involves entering information, specify the exact content to be input.
- DO NOT use words such as 'or', 'any', 'e.g.', or any other expression that makes the task description vague or ambiguous.
- DO NOT mention any non-existent files.
Input:
- App name: `{app_name}`
- Original user goal: `{initial_goal}`
- Action summary: `{action_summary}`
- Initial environment screenshots: <image>
Output format (strictly follow this format):
Your analysis:
- A brief explanation of how the original task and new subtasks are merged into a more complex task.
Output task:
<task>
(contain only a single task; if no task needs to be generated, write "No task")
</task>
Do not add any extra sections or commentary outside this format.

G.4 Prompting Template of Task Change

You are an expert at generating variant mobile app tasks by modifying parameters in an existing task instruction.
You will receive:
- A user task trajectory, including the initial user goal, the last few screenshots, and a sequence of action summaries.
Your goal is to generate a new task that has the SAME task structure as the original task,
but differs by changing specific parameters such as names, values, text content, or identifiers.
Requirements:
- The new task must keep the same overall workflow and steps as the original task.
- Only modify concrete parameters in the task instruction, such as file names, text content, titles, numbers, or labels.
- Do NOT add new steps or remove existing steps.
- The task must mention the app name and provide a high-level instruction without referencing any specific UI elements.
- If the task involves entering information, specify the exact content to be input.
- DO NOT use words such as 'or', 'any', 'e.g.', or any other expression that makes the task description vague or ambiguous.
- DO NOT mention any non-existent files.
Input:
- App name: `{app_name}`
- Original user goal: `{initial_goal}`
- Action summary: `{action_summary}`
- Environment screenshots: <image>
Output format (strictly follow this format):
Your analysis:
- A brief explanation of which parameters are modified and how they differ from the original task.
Output task:
<task>
(contain only a single task; if no task needs to be generated, write "No task")
</task>
Do not add any extra sections or commentary outside this format.

G.5 Prompting Template of label anchor

You are an expert in GUI understanding. Your task is to analyze the input image and UI Elements, and select the indices of the UI elements that are relevant to the instruction. These selected UI elements should uniquely identify the state after the instruction task has been completed.
Input Description:
- Instruction: A natural language description of the GUI task to be performed.
- UI Elements: The complete list of UI elements extracted from the final-state screen (including all element metadata).
- Image: The screenshot of the final state after completing the task, with UI element bounding boxes and indices overlaid to indicate their positions.
Requirements:
- Compare the image and the UI Elements list, and find UI elements whose content or visual role reflects the completion of the instruction.
- Select key UI elements that can indicate the final page state, such as navigation bars, menu titles, tab titles, selected buttons, status indicators, etc.
- Avoid selecting UI elements whose state may change dynamically due to time or network content.
- If you believe that no UI element in the screen is relevant to the instruction, stop the reasoning and output "No related elements".
Now output in the following format:
Thinking Process: Your thinking process
Output Index: [x, x, ...]
Input:
Instruction: `{query}`
UI Elements: `{ui_elements_info}`