FORGE: FEEDBACK-DRIVEN OPTIMIZATION AND REFINEMENT FOR GUI EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

GUI agents based on vision-language models have shown strong potential for general-purpose desktop automation, but their post-training remains constrained by the lack of scalable and reliably verifiable training data. Existing synthesis pipelines are largely static, making it difficult to adapt to new data quality issues and the evolving capabilities of trained models. We present FORGE, an end-to-end data flywheel that produces both high-quality supervised fine-tuning trajectories and verifiable reinforcement learning tasks. FORGE constructs diverse GUI tasks through scenario-behavior matching and pairs each task with an executable evaluator. Crucially, the pipeline evolves through its own feedback: recurring generation errors are distilled into reusable validation skills, while model failures on exploratory tasks inform subsequent data production. This closed loop progressively improves both data quality and task coverage. Using FORGE, we construct over 10,000 verifiable GUI tasks and evaluate the resulting models on OSWorld across two backbones. FORGE-8B and FORGE-4B achieve 51.2% and 40.4%, improving over their base models by 17.3% and 9.0%, respectively. We will release the dataset, code, and trained models upon acceptance.

1 INTRODUCTION

Large vision–language models (VLMs) have become a key foundation for graphical user interface (GUI) agents (Hong et al., 2024; Cheng et al., 2024; Wu et al., 2025; Lin et al., 2025a). Unlike conventional LLMs, GUI agents can perceive interfaces and interact with them through actions such as clicking and typing (You et al., 2024; Xu et al., 2024; Qin et al., 2025). This enables them to interact with existing software through human-facing interfaces without dedicated APIs or application-specific integrations. Recent advances highlight their potential for general-purpose automation across diverse software environments and cross-application workflows (Xie et al., 2024; Wang et al., 2025b; Ye et al., 2025a).

At present, an effective way to further improve GUI agents is through post-training (Wang et al., 2025a; Xu et al., 2026b; Cao et al., 2026). However, current approaches face two challenges. The first challenge is that most existing datasets are either manually collected or synthesized by LLMs. The former is costly and hard to scale, while the latter scales better but tends to generate increasingly repetitive tasks as the data volume grows (Wang et al., 2025b; Sun et al., 2025; Zhang et al., 2026c). This limits the diversity and complexity of the training data (Zhang et al., 2026a). Moreover, deterministic verification is especially important for RL training, because the evaluator directly provides the reward signal (Venuto et al., 2024; Dong et al., 2025; Huang et al., 2026). Yet reliable task evaluation remains difficult. LLM judges are flexible, but their decisions can vary across models and prompts and may also exhibit systematic biases. This reduces evaluation reproducibility and stability (Panickssery et al., 2024; Ye et al., 2025b; Bhat & Varma, 2026; Zhang et al., 2026b; Cui et al., 2026; Wei et al., 2026).

The second challenge is that open-source pipelines typically follow a staged workflow of task generation, trajectory collection, and post-hoc quality filtering (Sun et al., 2025; Zeng et al., 2025; He et al., 2026b; Gupta et al., 2026). Their generation and validation procedures are usually defined in advance. As illustrated in Figure 1, models at different capability levels exhibit distinct failure patterns. This suggests that their bottlenecks continuously shift as their capabilities improve. Such changes are difficult to cover with a fixed data generation procedure or a static task set. When

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

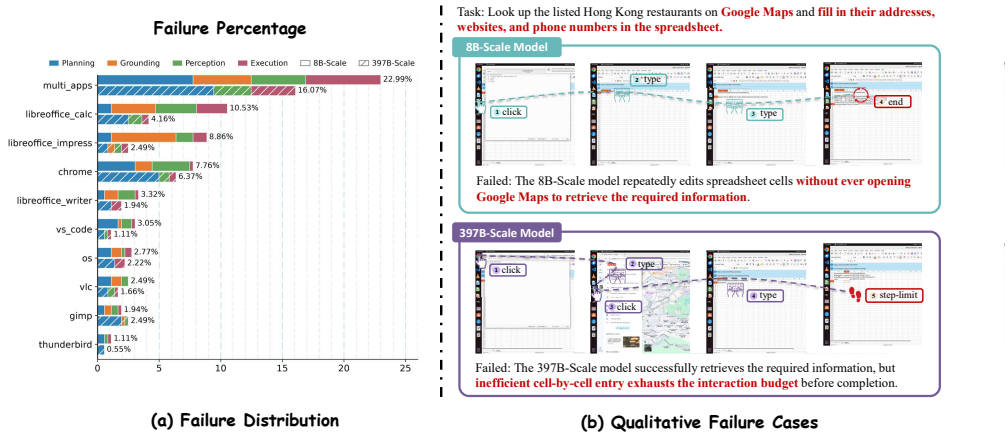


Figure 1: Failure analysis across model scales. (a) Distribution of failure categories on OSWorld. (b) On the same task, the 8B-Scale model fails to find a viable solution, while the 397B-Scale model identifies the correct solution but fails to finish due to inefficient cell-by-cell interaction.

new failures emerge, developers must diagnose them and update the pipeline (Liu et al., 2026; Lv et al., 2026). This process is particularly costly for GUI tasks, which often involve long interaction sequences and complex stateful environments (Xie et al., 2024; Cao et al., 2026; Lin et al., 2026). Moreover, statically generated tasks can become either too easy or too difficult as the agent improves, providing limited training signal (Qi et al., 2025; Xu et al., 2026b; Kang et al., 2026).

Recent studies have scaled data through automated synthesis or explored feedback-driven self-evolution. However, synthesized tasks often lack deterministic rewards, while self-evolving methods typically adapt only part of the pipeline. A unified framework for scalable, verifiable task generation is still missing. It should also adapt the data production pipeline as agent capabilities evolve. To bridge this gap, we develop **FORGE**, an end-to-end GUI data flywheel that jointly produces verifiable RL tasks and high-quality SFT trajectories. FORGE constructs atomic and compositional tasks through scenario-behavior matching, and pairs each task with an executable evaluator that directly verifies the final environment state. It further introduces a closed-loop self-evolution mechanism. Recurring generation failures are distilled into reusable validation skills, while model failures on free-composition tasks are used to improve subsequent data production. This allows the pipeline to continuously improve both data coverage and quality as the agent evolves.

We evaluate FORGE using Qwen3-VL-8B-Thinking and Qwen3-VL-4B-Thinking as base models. FORGE-8B and FORGE-4B achieve task success rates of 51.2% and 40.4% on OSWorld (Xie et al., 2024), with improvements of $+17.3\%$ and $+9.0\%$, respectively. These results demonstrate the effectiveness of FORGE across different model scales. Our contributions are as follows:

- We propose an end-to-end GUI data flywheel that jointly produces SFT trajectories and verifiable RL tasks. It combines scenario–behavior matching with three synthesis modes to generate diverse GUI tasks, yielding **10K** verifiable tasks.
- We introduce a closed-loop self-evolution mechanism that distills recurring error patterns into reusable validation skills. We further use free-composition tasks to expose model capability gaps and guide subsequent data production, improving task quality and coverage.
- We validate FORGE on OSWorld across two base models, with the 8B model improving from the low-30% range to a task success rate above 50%. Consistent gains across both model scales further demonstrate the generality of our approach.

2 Related Work

GUI Data Generation. High-quality data are essential for improving the capabilities of GUI agents. Recent efforts scale GUI training data through model-driven synthesis, including PC Agent-E (He et al., 2026a), AgentTrek (Xu et al., 2025), InSTA (Trabucco et al., 2025), OS-Genesis (Sun et al.,

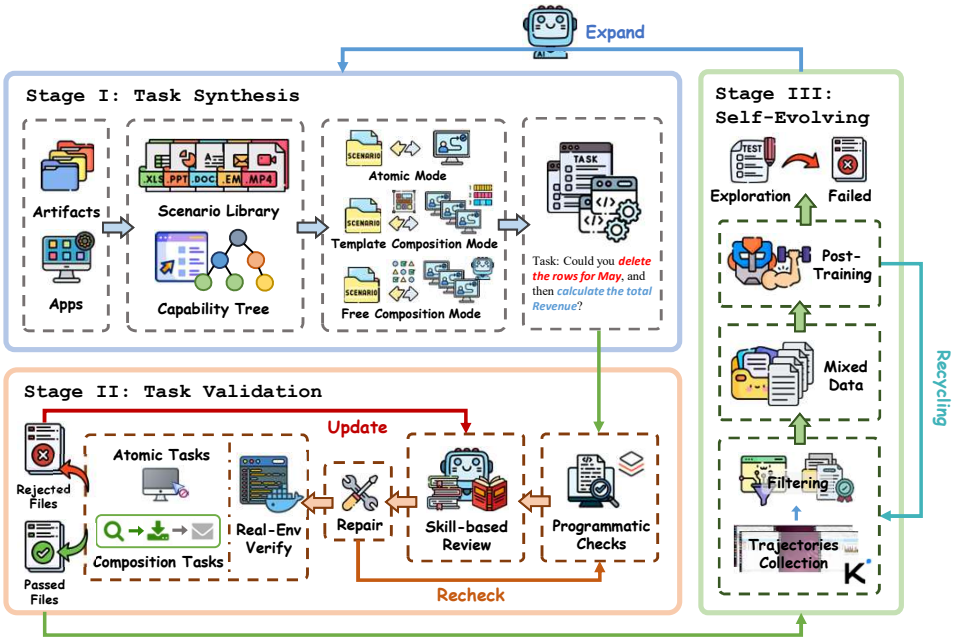


Figure 2: Overview of FORGE. The framework synthesizes and validates GUI tasks, then uses observed failures to improve validation and expand task coverage.

2025), Explorer (Pahuja et al., 2025), AndroidGen (Lai et al., 2025a), TongUI (Zhang et al., 2025a), AgentSynth (Xie et al., 2026a), GUI-ReWalk (Lin et al., 2025b), ProCUA-SFT (Jung et al., 2026), CUA-Gym (Wang et al., 2026a), AutoPlay (Ramrakhya et al., 2025), SynthAgent (Wang et al., 2025c), OpenMobile (Cheng et al., 2026), TreeCUA (Jiang et al., 2025), GUI-Genesis (Cao et al., 2026). Automated pipelines reduce manual effort, but many still rely on LLM/VLM-based judges for trajectory evaluation. Such supervision is difficult to use as a stable RL reward. FORGE addresses this issue with an end-to-end framework. It pairs each synthesized task with an executable evaluator. A skill-based validation mechanism further verifies and repairs generated tasks before use. This enables reliable task generation while scaling up task production.

Self-Improving. Recent work introduces feedback into different stages of GUI agent learning. WebRL (Qi et al., 2025) and MobileGen (Kang et al., 2026) adapt task generation to the current policy. Step-GUI (Yan et al., 2025) and LiteGUI (Wu et al., 2026) convert model-generated rollouts into new training supervision through filtering or distillation, while ComputerRL (Lai et al., 2025b) and UI-TARS-2 (Wang et al., 2025a) improve the policy through RL in interactive environments. UI-Evol (Zhang et al., 2025b) and Mobile-Agent-E (Wang et al., 2025d) refine reusable knowledge from past interactions to guide future decisions, whereas UI-Voyager (Lin et al., 2026) and BEPA (Wang et al., 2026c) exploit failed or off-policy experience to construct improved, policy-aligned supervision. These methods make individual components of the learning process adaptive, but typically focus on a particular component. FORGE instead evolves the *data production pipeline itself*. Recurring generation failures are distilled into reusable validation skills, while model failures on free-composition tasks are used to expand the capability tree and composition templates. As a result, both validation capability and task coverage are continuously updated across iterations.

3 METHOD

This section introduces FORGE, a self-evolving data flywheel that generates, validates, diagnoses, and evolves GUI agent data automatically. The scenario formulation and task definition are described in 3.1. Section 3.2 presents the synthesis framework for verifiable RL tasks and SFT trajectories. Section 3.3 introduces the self-evolving loop based on validation and exploration feedback. Finally, we detail the training strategies built upon the resulting data in 3.4.

3.1 SCENARIOS AND TASK FORMULATION

3.1.1 SCENARIO LIBRARY

We construct a scenario library to describe realistic environments that GUI agents interact with, spanning documents, images, and web pages across a set of desktop and browser applications. For each application, we collect its existing artifacts (e.g. spreadsheets, slides, source images, and web pages) and analyze their content and structural characteristics to identify representative usage contexts. Each artifact is paired with a corresponding scenario. To bootstrap the library efficiently, human annotators first analyze a subset of artifacts and author their scenario records manually; we then use a model to complete the rest. Each scenario record captures the artifact's metadata alongside three key fields used by downstream synthesis: a content description summarizing the artifact's structure and data, a set of property tags, and an initial screenshot of the application state.

3.1.2 SYNTHESIS MODES

Training data that support strong generalization require coverage across applications, function points, and complexity levels. Therefore, we design three synthesis modes that operate at different granularities to jointly achieve this coverage.

Atomic Mode. For each supported application, we organize its GUI operations into a hierarchical capability tree. The tree follows a coarse-to-fine structure: top-level nodes represent operation domains (e.g., cell operations, formatting, charts), intermediate nodes group related sub-categories, and leaf nodes correspond to atomic behaviors, which are the minimal units from which tasks can be synthesized. Each atomic behavior specifies a description, example instructions, and compatibility constraints (tags_required and tags_preferred) that govern which scenarios it can be paired with. This tree is constructed by exhaustively enumerating the application's GUI functionalities, ensuring complete coverage of its operation space.

In this mode, an atomic behavior is paired with a compatible scenario to produce one task instance. Let each behavior b declare required tags $\mathcal{T}_{\text{req}}(b)$ and preferred tags $\mathcal{T}_{\text{pref}}(b)$, and let each scenario s carry a tag set $\mathcal{T}(s)$. Compatible scenarios are those satisfying the required tags as a hard constraint,

$$\mathcal{C}(b) = \{\, s \mid \mathcal{T}_{\text{req}}(b) \subseteq \mathcal{T}(s) \,\}. \tag{1}$$

We then partition $\mathcal{C}(b)$ by whether a scenario matches any preferred tag,

$$\mathcal{P}(b) = \{\, s \in \mathcal{C}(b) \mid \mathcal{T}_{\text{pref}}(b) \cap \mathcal{T}(s) \neq \emptyset \,\}, \qquad \mathcal{O}(b) = \mathcal{C}(b) \setminus \mathcal{P}(b), \tag{2}$$

and define the preferred-tag matching score as:

$$r(b, s) = |\mathcal{T}_{\text{pref}}(b) \cap \mathcal{T}(s)|\,. \tag{3}$$

We select up to K scenarios by taking $\mathcal{P}(b)$ first and filling the remaining slots from $\mathcal{O}(b)$ when $|\mathcal{P}(b)| < K$:

$$\mathcal{S}(b) = \begin{cases} \mathcal{P}(b) \cup \mathcal{O}_{K-|\mathcal{P}(b)|}(b), & |\mathcal{P}(b)| < K, \\ \text{TopK}_{s \in \mathcal{P}(b)}\, r(b, s), & \text{otherwise.} \end{cases} \tag{4}$$

where $\mathcal{O}_m(b) \subseteq \mathcal{O}(b)$ denotes m scenarios drawn from $\mathcal{O}(b)$. Preferred tags thus act as a soft preference that biases selection without excluding valid scenarios. Given a target of N instances for behavior b, we allocate them evenly across $\mathcal{S}(b)$, generating $\lfloor \frac{N}{|\mathcal{S}(b)|} \rfloor$ tasks per scenario so that no single scenario dominates the data for that behavior.

Template Composition Mode. While atomic tasks cover individual operations, real user requests often span multiple steps. This mode produces multi-step tasks with controlled complexity using reusable composition templates. Human experts first design the templates, which are then extended by a foundation model following the same schema. To ensure that composed tasks follow a coherent execution order (e.g., sorting before computation and computation before saving), each template T is defined as an ordered sequence of category slots $T = (o_1, \ldots, o_m)$, where slot $o_i = (\mathcal{G}_i, [l_i, u_i])$ specifies a set of taxonomy categories $\mathcal{G}_i$ and a pick range. Each synthesis draws $k_i \sim \text{Uniform}\{l_i, u_i\}$ behaviors from the pool whose category falls under $\mathcal{G}_i$:

$$B_i = \text{Sample}_{k_i}\big(\{\, b \mid \text{cat}(b) \in \mathcal{G}_i \,\}\big), \qquad B = B_1 \circ B_2 \circ \cdots \circ B_m \tag{5}$$

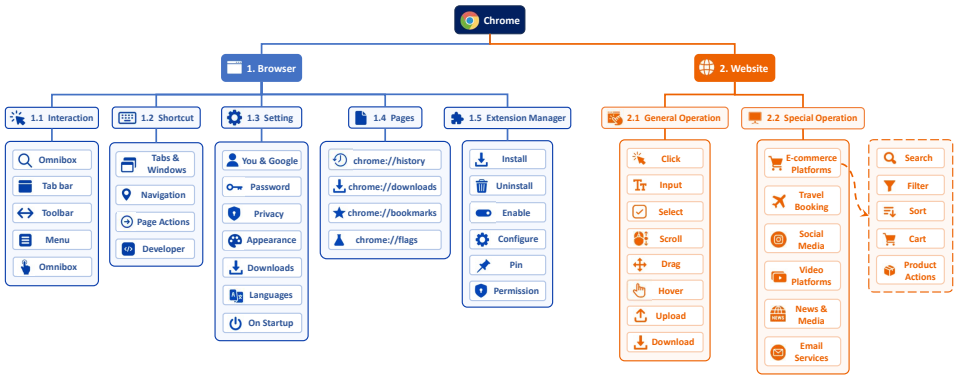


Figure 3: Example of the Chrome capability tree, hierarchically organizing GUI functionalities from high-level domains to fine-grained atomic behaviors for task synthesis.

The slot ordering is preserved in B, so the composed behaviors form a logical workflow rather than an arbitrary set of operations. Compatible scenarios are those satisfying the union of required tags:

$$\mathcal{C}(B) = \left\{ s \;\middle|\; \left(\bigcup_{b \in B} \mathcal{T}_{\text{req}}(b) \cup \mathcal{T}_{\text{req}}(T)\right) \subseteq \mathcal{T}(s) \right\}. \tag{6}$$

We distribute the target count N evenly across up to K scenarios from $\mathcal{C}(B)$. For each scenario, we re-sample B before prompting the model, producing different behavior compositions across scenarios. As the taxonomy grows, newly added behaviors automatically enter the candidate pool of their category and become reachable by existing templates.

Free Composition Mode. To discover useful combinations beyond existing templates and continuously expand the template library, we introduce a free composition mode that samples behaviors directly from the taxonomy. Let $\mathcal{G}^{\text{top}} = \{g_1, \ldots, g_M\}$ denote the top-level categories of the capability tree and $\mathcal{B}(g)$ the behaviors under category g. Each synthesis first draws a target size $n \sim \text{Uniform}(\{n_{\min}, n_{\max}\})$, then applies stratified sampling over a random permutation π of $\mathcal{G}^{\text{top}}$, taking one behavior from each of the first n categories:

$$B = \left\{ b_j \sim \text{Uniform}\big(\mathcal{B}(g_{\pi(j)})\big) \;\middle|\; j = 1, \ldots, \min(n, M) \right\}. \tag{7}$$

If fewer than n behaviors are collected, the remainder is filled by uniform sampling from the unused behaviors. Stratifying across top-level categories biases each draw toward operations from distinct functional domains, producing combinations that templates rarely cover. Scenario compatibility follows the same required-tag criterion as in Eq.6, and the sampled behaviors are passed to the model without any prescribed order, leaving it to organize them into a coherent request.

3.1.3 Task Formulation

We adopt the task format of OSWorld (Xie et al., 2024), allowing the synthesized tasks to be executed in GUI environments. Each task is represented as a tuple $\tau = (i, c, e)$. Here, i is a natural-language instruction describing the user's goal, and c contains the setup actions that initialize the virtual machine to a deterministic starting state. The executable evaluator e saves the final application state, extracts the target artifact from the VM, and verifies task completion using deterministic scripts, returning a score in $[0, 1]$. Each task also includes metadata about its provenance and properties, including the synthesis mode, source template or scene, composed atomic behaviors, complexity level, and instruction style. This metadata links each task to its generating components and is used by the self-evolving loop in Section 3.3. Detailed task schema is provided in Appendix A.1.

3.2 Synthesis Framework

3.2.1 Skill-Guided Task Validation

We first validate executable evaluators through stronger checks, ranging from lightweight programmatic inspection to semantic review and real-environment execution. Since model-generated evalu-

ators are untrustworthy by default and may fail to execute or misjudge task completion, deploying them without verification wastes environment resources and misleads downstream training.

Programmatic checks. We set five levels of programmatic checks for the configuration c and evaluator e of each task τ. L0–L2 perform static inspection of structure, name resolution, and action registration. L3 parses and analyzes the scoring code. L4 injects mocked state values and runs the code in a sandboxed subprocess.

Skill-based semantic review. Passing the programmatic checks only guarantees that an evaluator runs, not that it measures task quality. A judge model therefore reviews each evaluator against the instruction and configuration following a structured rubric (e.g., instruction clarity and task completion coverage). The review returns structured failure reasons and concrete fix suggestions rather than a bare score. Failing tasks are not immediately discarded. They are repaired using the reported reasons and suggestions, then re-validated from the programmatic checks onward. This repair-and-recheck cycle repeats until the task passes or a maximum number of rounds is reached.

Real-environment verification. Surviving tasks are finally executed in the real GUI environment to exclude false positives. For atomic tasks, we directly evaluate the initial state without executing any action. For composite tasks, we split the instruction at conjunctions and punctuation, drop the final segment, and execute only the truncated prefix, yielding a partial task. Let s denote the score returned by e on this state; we discard τ if $s > \delta$, since such an evaluator cannot separate partial from full completion and would produce misleading supervision.

3.2.2 Data Synthesis

Task generation. For each application, the generation pipeline selects one of the three synthesis modes in Sec. 3.1.2, assembles a prompt that embeds the sampled scenario context and behavior slots, and queries an LLM to produce an instruction together with its executable evaluator. Each candidate task is then passed through the validation pipeline of Sec. 3.2.1, and only validated tasks enter the trajectory synthesis stage.

Trajectory synthesis. We collect agent trajectories by running OSWorld on internally hosted virtual environments. Each rollout follows a thinking-then-acting paradigm: at every step the agent observes the current screenshot, produces a reasoning trace, and emits an action from a unified action space. The episode terminates when the agent reports the task as finished or the step budget is reached. We archive the screenshot, reasoning trace, and action of each step, together with the terminal status of the episode. It is important to note that the raw trajectories may contain noise from many sources, including device disconnections, network instability, account invalidation, malformed model actions, and unfinished tasks. Used without filtering, such data lead the model to imitate erroneous actions or treat environmental noise as part of normal execution. We therefore adopt a strict trajectory-level filter and retain an episode only if (i) its evaluator score exceeds the threshold δ, (ii) it terminates in a `finished` state rather than being cut off by an environment error, (iii) every step carries a valid screenshot, and (iv) every action parses under the unified action space. We then apply the step-level Reflector from OpenCUA (Wang et al., 2025b) to remove redundant or incorrect steps, and convert the remaining steps into SFT samples.

3.3 Self-Evolving Loop

A remaining challenge is that new quality issues continue to emerge as data production scales. These failure modes are difficult to fully cover with static validation rules. Meanwhile, without feedback from the trained model, the pipeline cannot adapt task generation to the model's evolving capabilities and may continue producing data with limited training value. We address both limitations through self-evolution. The pipeline collects failures from task generation and model evaluation and uses them to update the next synthesis round.

Validation Skill Evolution. As data production scales, generated tasks exhibit recurring quality issues that cannot be fully covered by predefined validation rules. We therefore collect failures arising during task generation and validation, cluster recurring error patterns, and distill them into reusable validation skills. These skills extend the existing validation rules, so similar defects can be detected and repaired earlier in later rounds.

Free Composition Exploration. Feedback from generation reveals little about capabilities outside the current task space. As the model improves, its failure patterns also change. A static task distribution can therefore become poorly calibrated and repeatedly sample tasks the model already solves. To expose these blind spots, every synthesis round additionally produces a batch of free-composition tasks (Section 3.1.2) that are withheld from trajectory collection and training. After each training iteration, the current model is evaluated on this batch to probe behavior combinations beyond the template library. We group failures by cause to identify capability gaps. These gaps guide updates to atomic behaviors, composition templates, and sampling strategies. The next round focuses on under-covered or weak capabilities. In this way, task difficulty and coverage evolve with the model.

3.4 Training Strategy

Failure prefix recycling. Failed rollouts from RL directly expose the model's capability boundary and hard states. However, these rollouts are often simply discarded. This wastes expensive environment interactions and misses an opportunity to convert hard states into targeted supervision. This issue is particularly pronounced in GRPO (Shao et al., 2024), where a group whose N rollouts all receive zero reward provides no discriminative learning signal. However, the student model often completes the first several steps correctly before diverging. Discarding such trajectories therefore overlooks the partial progress the model has already made. For each such group on task τ with rollouts $\{\rho_1, \ldots, \rho_N\}$, we extract from every rollout $\rho_i = (a_{i,1}, \ldots, a_{i,T_i})$ its longest correct prefix:

$$k_i^\star = \max\{\, k \le T_i : a_{i,1:k} \text{ are correct} \,\},$$

and retain $\rho_i^{[:k_i^\star]}$. Each prefix is then handed to a trajectory completion agent, which continues execution from the state reached at step $k_i^\star$ until the task succeeds or the step budget is exhausted. The completed trajectories are added to the SFT data of the next iteration.

Data composition. A GUI agent must both execute complex interface tasks and retain general reasoning and instruction-following capabilities. We therefore mix GUI-specific and general-purpose data to maintain both fine-grained grounding and high-level reasoning across domains. Although the general-purpose data are not directly tied to GUI environments, we find that their inclusion improves overall agentic performance. Moreover, GUI agent post-training also faces a distinct *reasoning–grounding trade-off*. While chain-of-thought (CoT) reasoning can improve high-level planning and decision making, recent studies have observed that it may degrade visual grounding (Zhou et al., 2026; Yang et al., 2026a). This is especially problematic for GUI agents performing long-horizon tasks. Each SFT iteration therefore mixes four data sources: (1) general instruction data, including question answering, mathematics, coding, and visual understanding; (2) grounding data for bounding-box prediction and element identification; (3) GUI trajectories synthesized in Section 3.2.2; and (4) teacher continuations recovered from the previous iteration's GRPO phase.

3.5 Post-Training

Our post-training pipeline consists of two stages. We first apply SFT to establish basic GUI interaction capabilities. We then optimize the model with GRPO using action-aware advantage scaling to improve GUI interaction quality.

Supervised Fine-Tuning (SFT). First, FORGE is trained on the mixed data. Directly training on long-horizon trajectories incurs substantial memory and computation costs. We therefore decompose each trajectory by retaining a sliding window of up to three recent interaction steps: earlier screenshots and dialogue turns are dropped, while their actions are preserved as compact text. Meanwhile, only the target action is supervised with SFT loss; preceding actions within the window serve as context and are masked. For the recovered trajectories, we supervise only the continuation segment: the prefix $\rho_i^{[:k_i^\star]}$ is kept as context but excluded from the loss. The training objective is:

$$\mathcal{L}_{\text{SFT}} = -\mathbb{E}_{\tau\sim\mathcal{D}} \sum_{t=k^\star(\tau)+1}^{T} \log \pi_\theta(a_t \mid s_{1:t-1}), \tag{8}$$

where $k^\star(\tau) = 0$ for standard SFT samples.

Reinforcement Learning (RL). SFT learns from successful demonstrations but treats every action in a trajectory as equally correct. It provides no signal to distinguish an effective action from a

Table 1: Comparison with proprietary and open-source models.

Model	Steps	OS	Office	Daily	Professional	Workflow	Overall
Proprietary models							
GPT-5.5 (OpenAI, 2026)	–	–	–	–	–	–	78.7
Claude Opus 4.6 (Anthropic, 2026)	–	–	–	–	–	–	72.7
Claude Sonnet 4.5 (Anthropic, 2025b)	100	70.8	72.6	61.4	63.2	49.5	62.9
Claude Sonnet 4 (Anthropic, 2025a)	50	45.8	43.5	55.1	55.1	28.5	43.9
Seed-1.8 (Seed, 2026)	100	66.7	68.8	67.1	71.4	42.4	61.9
OpenAI CUA (OpenAI, 2025)	50	70.8	23.0	37.2	51.0	15.8	31.3
CoACT-1 (Song et al., 2025)	50	70.8	60.6	54.1	69.4	42.4	56.4
Agent S2.5 w/ o3 (Agashe et al., 2025)	50	75.0	53.8	54.0	75.5	39.5	54.2
GTA1 w/ o3 (Yang et al., 2026b)	100	62.5	54.6	60.3	61.2	38.3	53.1
Jedi-7B w/ o3 (Xie et al., 2026b)	50	54.2	47.0	62.1	69.4	35.0	50.6
Open-source models							
Kimi K2.5 (Team et al., 2026)	100	73.9	69.2	66.4	73.5	46.1	63.3
EvoCUA-32B (Xue et al., 2026)	50	78.3	59.8	64.5	81.6	27.9	56.7
Mobile-Agent-v3.5 (Xu et al., 2026a)	–	–	–	–	–	–	56.5
UI-TARS-2 (Wang et al., 2025a)	100	41.7	61.1	62.1	61.2	34.1	53.1
ComputerRL-9B (Lai et al., 2026)	50	75.0	45.8	52.5	71.4	27.9	48.0
OpenCUA-72B (Wang et al., 2026b)	50	45.8	42.7	53.8	75.5	22.3	44.9
OpenCUA-32B (Wang et al., 2026b)	50	60.9	29.9	41.0	65.3	14.6	35.1
DART-GUI-7B (Li et al., 2025b)	30	62.5	39.3	50.8	73.5	16.7	42.1
Qwen3-VL-8B-Thinking (Bai et al., 2025)	50	–	–	–	–	–	33.9
Qwen3-VL-4B-Thinking (Bai et al., 2025)	50	–	–	–	–	–	31.4
Ours							
FORGE-Qwen3-VL-8B	50	**70.8**	**52.1**	**60.3**	**73.5**	**25.8**	**51.2**
FORGE-Qwen3-VL-4B	50	**58.3**	**39.3**	**47.4**	**65.3**	**18.3**	**40.4**

redundant one, nor does it penalize failure modes that appear at inference, such as premature termination or repeated actions. To address these limitations, we apply GRPO. Instead of directly imitating the labeled action, the policy samples candidate actions. We reweight their advantages using an action-aware coefficient that evaluates each step along multiple dimensions:

$$\alpha_t = \max\left\{0,\ F_t\left(w_{\text{type}}C_t + w_{\text{arg}}C_tQ_t^{\text{arg}} + w_{\text{desc}}Q_t^{\text{desc}}\right) - \lambda_{\text{term}}E_t - \lambda_{\text{rep}}L_t\right\}, \tag{9}$$

$$\tilde{A}_{i,t} = \begin{cases} \alpha_{i,t}A_i^{\text{grp}}, & A_i^{\text{grp}} \geq 0, \\ (1-\alpha_{i,t})A_i^{\text{grp}}, & A_i^{\text{grp}} < 0. \end{cases} \tag{10}$$

where $F_t \in \{0,1\}$ indicates format validity and $C_t \in \{0,1\}$ indicates action type correctness. Q_t^{arg} measures the quality of action arguments, such as coordinate distance for clicks or direction and magnitude similarity for scrolls, while Q_t^{desc} measures the semantic similarity of the action description. The last two terms, E_t and L_t, penalize premature termination and repeated actions. The resulting coefficient enables action-level credit assignment by reweighting the trajectory-level advantage according to action quality. Full details are provided in Appendix A.4.

4 Experiments

4.1 Setup

We apply FORGE to two models, Qwen3-VL-8B-Thinking and Qwen3-VL-4B-Thinking, and evaluate them on OSWorld. All training uses 32 H800 GPUs. Detailed settings are provided in the appendix. We compare FORGE against both proprietary and open-source models listed in Table 1.

4.2 Main Results

Table 1 reports the main comparison on OSWorld, including overall performance and category-level scores across OS, Office, Daily, Professional, and Workflow tasks. FORGE consistently improves performance across both base models. FORGE-Qwen3-VL-8B achieves **51.2%**, a 17.3-point gain over its backbone. FORGE-Qwen3-VL-4B reaches **40.4%**, improving by 9.0 points. These consistent gains across model scales demonstrate the generality of FORGE.

Table 2: Ablation on different training data composition.

Data Source	OSWorld (%)
GUI Only	38.0
GUI + Continuation	40.4
GUI + Grounding + Continuation	42.9
Full	**44.2**

Table 3: Effectiveness of each component in the validation pipeline.

Component	Atomic (%)	Composite (%)
Programmatic	$16.2_{\pm 2.1}$	$27.0_{\pm 1.2}$
Semantic	$65.0_{\pm 8.2}$	$63.0_{\pm 2.8}$
Real-Environment	$15.6_{\pm 5.8}$	$17.1_{\pm 1.1}$
Repair Δ Retention	$\mathbf{+67.6_{\pm 7.8}}$	$\mathbf{+64.0_{\pm 7.4}}$

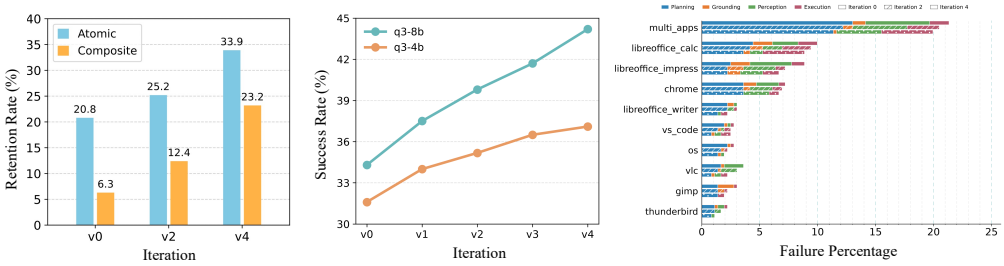


Figure 4: **Left:** Trajectory retention rate across iterations. **Middle:** OSWorld success rate across self-evolving iterations. **Right:** Failure distribution across application domains and iterations.

4.3 Ablation Results

In this section, we conduct all ablation experiments using SFT only. Unless otherwise specified, all variants use the same base model, training data scale, and SFT configuration.

Data composition ablation. We study the contribution of different data sources by progressively extending the training mixture. As shown in Table 2, model performance improves as additional data sources are incorporated, demonstrating the effectiveness of our mixed-data training strategy.

Validation pipeline ablation. We evaluate the validation pipeline on atomic and composite tasks. As shown in Table 3, programmatic checks and semantic validation identify most problematic tasks. More than 60% of these tasks are successfully repaired and retained, substantially improving data utilization. Real-environment verification further filters out tasks that remain unreliable under actual execution. These results demonstrate the effectiveness of our task validation pipeline.

Self-evolution ablation. We evaluate self-evolution across iterations. The v0 data serve as the baseline and are generated before any feedback-driven updates. For model comparisons, we keep the base model, SFT data size, and training settings fixed. As shown in Figure 4, trajectory retention increases for both atomic and composite tasks. From v0 to v2, stronger programmatic and semantic validation detects additional failure modes. From v2 to v4, FORGE uses model feedback to adjust composition difficulty and diversity. Models trained on data from later iterations achieve higher OSWorld success rates, with fewer failures across most application domains. The failure distribution also changes, suggesting that the model's capability bottlenecks shift across iterations. Free-composition tasks reveal these new weaknesses and guide subsequent data generation. These results support the effectiveness of FORGE's self-evolution mechanism.

5 Conclusion

This paper presents FORGE, an end-to-end data flywheel for GUI agent post-training that jointly produces SFT trajectories and verifiable RL tasks. FORGE combines executable task verification with a closed-loop self-evolution mechanism that improves validation capability and task coverage. Across two base models on OSWorld, FORGE-8B and FORGE-4B achieve task success rates of 51.2% and 40.4%, improving over their backbones by +17.3% and +9.0%, respectively. These results demonstrate the effectiveness and generality of FORGE across different models. We hope FORGE can provide a scalable foundation for future GUI agent data generation and post-training.

AI USE STATEMENT

Generative AI tools were used in two distinct ways. Within the proposed method, LLMs propose candidate GUI tasks, configurations, and executable evaluators; assist with semantic validation; and generate trajectories and failure-prefix continuations for training. These outputs are subject to the validation procedures described in the paper before use. In preparing the manuscript, generative AI tools were used only to improve the grammar, clarity, and readability of author-written text. They were not used for research ideation, method or experiment design, code development, data analysis, or the generation of experimental results, figures, or tables. The authors reviewed all AI-assisted edits and take full responsibility for the final content.

ETHICS STATEMENT

This work involves no human subjects, personally identifiable information, or private data. Our experiments use publicly available GUI-agent benchmarks and are conducted entirely in sandboxed environments. The trained agents are not deployed on real user systems. We acknowledge that GUI agents capable of interacting with real computer interfaces may pose dual-use risks if deployed without appropriate safeguards. We therefore recommend human oversight and explicit confirmation for sensitive or irreversible actions in real-world applications. We are aware of no conflicts of interest.

REPRODUCIBILITY STATEMENT

The implementation and training details are provided in Appendix A.9, including model initialization, data composition, SFT and RL configurations, reward definitions, and optimization hyperparameters. We also provide the prompts and procedures used for task synthesis, executable evaluator generation, hierarchical validation, and trajectory construction. GPT-5.5 is used for task and evaluator synthesis and validation, while Kimi-K2.6 serves as the trajectory executor for SFT trajectory collection and failure-prefix completion. Evaluation follows the standard OSWorld protocol, with task statistics and inference settings reported alongside the corresponding experiments. Code, generated data, and trained models will be released to facilitate reproducibility.

REFERENCES

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

Anthropic. System Card: Claude Opus 4 & Claude Sonnet 4. https://www.anthropic.com/claude-4-system-card, May 2025a.

Anthropic. Claude Sonnet 4.5 System Card. https://www.anthropic.com/claude-sonnet-4-5-system-card, September 2025b.

Anthropic. Claude Opus 4.6 System Card. https://www.anthropic.com/claude-opus-4-6-system-card, February 2026.

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

Savita Bhat and Vasudeva Varma. All prompts are created equal? evaluating robustness of llm judges against non-adversarial prompt variations. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 38730–38745, 2026.

Yuan Cao, Dezhi Ran, Mengzhou Wu, Yuzhe Guo, Xin Chen, Ang Li, Gang Cao, Gong Zhi, Hao Yu, Linyi Li, et al. Gui-genesis: Automated synthesis of efficient environments with verifiable rewards for gui agent post-training. *arXiv preprint arXiv:2602.14093*, 2026.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents. In *Proceedings of the*

62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 9313–9332, 2024.

Kanzhi Cheng, Zehao Li, Zheng Ma, Nuo Chen, Jialin Cao, Qiushi Sun, Zichen Ding, Fangzhi Xu, Hang Yan, Jiajun Chen, et al. Openmobile: Building open mobile agents with task and trajectory synthesis. *arXiv preprint arXiv:2604.15093*, 2026.

Chaoqun Cui, Jing Huang, Shijing Wang, Liming Zheng, Qingchao Kong, and Zhixiong Zeng. Agentic reward modeling: Verifying gui agent via online proactive interaction. *arXiv preprint arXiv:2602.00575*, 2026.

Guanting Dong, Keming Lu, Chengpeng Li, Tingyu Xia, Bowen Yu, Chang Zhou, and Jingren Zhou. Self-play with execution feedback: Improving instruction-following capabilities of large language models. In *International Conference on Learning Representations*, volume 2025, pp. 39286–39313, 2025.

Tanmay Gupta, Piper Wolters, Zixian Ma, Peter Sushko, Rock Yuren Pang, Diego Llanes, Yue Yang, Taira Anderson, Boyuan Zheng, Zhongzheng Ren, et al. Molmoweb: Open visual web agent and open data for the open web. *arXiv preprint arXiv:2604.08516*, 2026.

Yanheng He, Jiahe Jin, and Pengfei Liu. Efficient agent training for computer use. In *International Conference on Learning Representations*, volume 2026, pp. 123856–123877, 2026a.

Yifei He, Pranit Chawla, Yaser Souri, Subhojit Som, and Xia Song. Webstar: Scalable data synthesis for computer use agents with step-level filtering. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 516–533, 2026b.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for gui agents. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 14281–14290. IEEE, 2024.

Sian-Yao Huang, Li-Hsien Chang, Che-Yu Lin, and Cheng-Lin Yang. Beyond oracle: Verifier-supervision for instruction hierarchy in reasoning and instruction-tuned llms. *Advances in Neural Information Processing Systems*, 38:98362–98395, 2026.

Deyang Jiang, Jing Huang, Xuanle Zhao, Lei Chen, Liming Zheng, Fanfan Liu, Haibo Qiu, Peng Shi, and Zhixiong Zeng. Treecua: Efficiently scaling gui automation with tree-structured verifiable evolution. *arXiv preprint arXiv:2602.09662*, 2025.

Jaehun Jung, Ximing Lu, Brandon Cui, Muhammad Khalifa, Shaokun Zhang, Hao Zhang, Jin Xu, Amala Sanjay Deshmukh, Karan Sapra, Andrew Tao, et al. Procua-sft technical report. *arXiv preprint arXiv:2606.17321*, 2026.

Linjia Kang, Zhimin Wang, Yongkang Zhang, Duo Wu, Jinghe Wang, Ming Ma, Haopeng Yan, and Zhi Wang. Learning with challenges: Adaptive difficulty-aware data generation for mobile gui agent training. *arXiv preprint arXiv:2601.22781*, 2026.

Hanyu Lai, Junjie Gao, Xiao Liu, Yifan Xu, Shudan Zhang, Yuxiao Dong, and Jie Tang. Androidgen: Building an android language agent under data scarcity. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2727–2749, 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, 2025b. URL https://arxiv.org/abs/2508.14040.

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. In *International Conference on Learning Representations*, volume 2026, pp. 23553–23591, 2026.

Kaixin Li, Ziyang Meng, Hongzhan Lin, Ziyang Luo, Yuchen Tian, Jing Ma, Zhiyong Huang, and Tat-Seng Chua. Screenspot-pro: Gui grounding for professional high-resolution computer use. In *Proceedings of the 33rd ACM International Conference on Multimedia*, pp. 8778–8786, 2025a.

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

Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Stan Weixian Lei, Lijuan Wang, and Mike Zheng Shou. Showui: One vision-language-action model for gui visual agent. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 19498–19508. IEEE, 2025a.

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

Zichuan Lin, Feiyu Liu, Yijun Yang, Jiafei Lyu, Yiming Gao, Yicheng Liu, Zhicong Lu, Yangbin Yu, Mingyu Yang, Junyou Li, et al. Ui-voyager: A self-evolving gui agent learning via failed experience. *arXiv preprint arXiv:2603.24533*, 2026.

Guohong Liu, Jialei Ye, Pengzhi Gao, Wei Liu, Jian Luan, Yunxin Liu, and Yuanchun Li. Scalewob: Guiding gui agents with coding agents via large-scale environmental synthesis. *arXiv preprint arXiv:2605.25160*, 2026.

Bowen Lv, Xiao Liu, Yanyu Ren, Hanyu Lai, Bohao Jing, Hanchen Zhang, Yanxiao Zhao, Shuntian Yao, Jie Tang, and Yuxiao Dong. Scalecua: Scaling computer use agents with verifiable task synthesis and efficient online rl. *arXiv preprint arXiv:2607.11185*, 2026.

OpenAI. Computer-Using Agent. https://openai.com/index/computer-using-agent/, January 2025.

OpenAI. GPT-5.5 System Card. https://openai.com/index/gpt-5-5-system-card/, April 2026.

Vardaan Pahuja, Yadong Lu, Corby Rosset, Boyu Gou, Arindam Mitra, Spencer Whitehead, Yu Su, and Ahmed Hassan. Explorer: Scaling exploration-driven web trajectory synthesis for multimodal web agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 6300–6323, 2025.

Arjun Panickssery, Samuel R Bowman, and Shi Feng. Llm evaluators recognize and favor their own generations. *Advances in Neural Information Processing Systems*, 37:68772–68802, 2024.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, et al. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. In *International Conference on Learning Representations*, volume 2025, pp. 79791–79821, 2025.

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.

Ram Ramrakhya, Andrew Szot, Omar Attia, Yuhao Yang, Anh Nguyen, Bogdan Mazoure, Zhe Gan, Harsh Agrawal, and Alexander Toshev. Scaling synthetic task generation for agents via exploration. In *ICLR 2026*, 2025. URL https://arxiv.org/abs/2509.25047.

Bytedance Seed. Seed1. 8 model card: Towards generalized real-world agency. *arXiv preprint arXiv:2603.20633*, 2026.

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

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297, 2025.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, et al. Coact-1: Computer-using agents with coding as actions. *arXiv e-prints*, pp. arXiv–2508, 2025.

Qiushi Sun, Kanzhi Cheng, Zichen Ding, Chuanyang Jin, Yian Wang, Fangzhi Xu, Zhenyu Wu, Chengyou Jia, Liheng Chen, Zhoumianze Liu, et al. Os-genesis: Automating gui agent trajectory construction via reverse task synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5555–5579, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Ziwei Chai, Y Charles, HS Che, Cheng Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Brandon Trabucco, Gunnar Sigurdsson, Robinson Piramuthu, and Ruslan Salakhutdinov. Insta: Towards internet-scale training for agents. *arXiv preprint arXiv:2502.06776*, 2025.

David Venuto, Sami Nur Islam, Martin Klissarov, Doina Precup, Sherry Yang, and Ankit Anand. Code as reward: Empowering reinforcement learning with vlms. *arXiv preprint arXiv:2402.04764*, 2024.

Bowen Wang, Dunjie Lu, Junli Wang, Tianyi Bai, Shixuan Liu, Zhipeng Zhang, Haiquan Wang, Hao Hu, Tianbao Xie, Shuai Bai, Dayiheng Liu, Que Shen, Junyang Lin, and Tao Yu. Cua-gym: Scaling verifiable training environments and tasks for computer-use agents, 2026a. URL https://arxiv.org/abs/2605.25624.

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

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

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

Zezhou Wang, Ziyun Zhang, Xiaoyi Zhang, Zhuzhong Qian, and Yan Lu. From off-policy to on-policy: Enhancing gui agents via bi-level expert-to-policy assimilation. *ArXiv*, abs/2601.05787, 2026c. URL https://api.semanticscholar.org/CorpusID:284596886.

Zhaoyang Wang, Yiming Liang, Xuchao Zhang, Qianhui Wu, Siwei Han, Anson Bastos, Rujia Wang, Chetan Bansal, Baolin Peng, Jianfeng Gao, Saravan Rajmohan, and Huaxiu Yao. Adapting web agents with synthetic supervision. 2025c. URL https://arxiv.org/abs/2511.06101.

Zhenhailong Wang, Haiyang Xu, Junyang Wang, Xi Zhang, Ming Yan, Ji Zhang, Fei Huang, and Heng Ji. Mobile-agent-e: Self-evolving mobile assistant for complex tasks. *arXiv preprint arXiv:2501.11733*, 2025d.

Jinbiao Wei, Qianran Ma, Yilun Zhao, Xiao Zhou, Kangqi Ni, Guo Gan, and Arman Cohan. Opencomputer: Verifiable software worlds for computer-use agents. *arXiv preprint arXiv:2605.19769*, 2026.

Yubin Wu, Zicheng Cai, Liping Ning, Hua Wang, Zhi Chen, Yaohua Tang, and Hao Chen. Litegui: Distilling compact gui agents with reinforcement learning. *arXiv preprint arXiv:2605.07505*, 2026.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, et al. Os-atlas: Foundation action model for generalist gui agents. In *International Conference on Learning Representations*, volume 2025, pp. 5090–5108, 2025.

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

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

Tianbao Xie, Jiaqi Deng, Xiaochuan Li, Junlin Yang, Haoyuan Wu, Jixuan Chen, Wenjing Hu, Xinyuan Wang, Yuhui Xu, Zekun Wang, et al. Scaling computer-use grounding via user interface decomposition and synthesis. *Advances in Neural Information Processing Systems*, 38, 2026b.

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

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

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

Yiheng Xu, Dunjie Lu, Zhennan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. Agenttrek: Agent trajectory synthesis via guiding replay with web tutorials. In *International Conference on Learning Representations*, volume 2025, pp. 79822–79843, 2025.

Taofeng Xue, Chong Peng, Mianqiu Huang, Linsen Guo, Tiancheng Han, Haozhe Wang, Jianing Wang, Xiaocheng Zhang, Xin Yang, Dengchang Zhao, et al. Evocua: Evolving computer use agents via learning from scalable synthetic experience. *arXiv preprint arXiv:2601.15876*, 2026.

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

Rui Yang, Qianhui Wu, Zhaoyang Wang, Hanyang Chen, Ke Yang, Hao Cheng, Huaxiu Yao, Baolin Peng, Huan Zhang, Jianfeng Gao, et al. Gui-libra: Training native gui agents to reason and act with action-aware supervision and partially verifiable rl. *arXiv preprint arXiv:2602.22190*, 2026a.

Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, et al. Gta1: Gui test-time scaling agent. In *International Conference on Learning Representations*, volume 2026, pp. 48295–48323, 2026b.

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

Jiayi Ye, Yanbo Wang, Yue Huang, Dongping Chen, Qihui Zhang, Nuno Moniz, Tian Gao, Werner Geyer, Chao Huang, Pin-Yu Chen, et al. Justice or prejudice? quantifying biases in llm-as-a-judge. In *International Conference on Learning Representations*, volume 2025, pp. 102351–102390, 2025b.

Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. Ferret-ui: Grounded mobile ui understanding with multimodal llms. In *European Conference on Computer Vision*, pp. 240–255. Springer, 2024.

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

Bofei Zhang, Zirui Shang, Zhi Gao, Wang Zhang, Rui Xie, Xiaojian Ma, Tao Yuan, Xinxiao Wu, Song-Chun Zhu, and Qing Li. Tongui: Building generalized gui agents by learning from multimodal web tutorials. *arXiv preprint arXiv:2504.12679*, 2025a.

Long Zhang, Yuhan Chen, Chaoran Zhang, Wanxia Cao, Kun Huang, Pengzhi Gao, Wei Liu, Jian Luan, Chenliang Li, and Lixin Zou. Gsar: Goal-state-anchor rewards for mobile gui agents with self-evolving data synthesis. *arXiv preprint arXiv:2608.22847*, 2026a.

Sharon Zhang, R Kenny Jones, Jiajun Wu, and Maneesh Agrawala. Aggregating llm-based weak verifiers for spatial layout generation. *arXiv preprint arXiv:2606.05268*, 2026b.

Ziyun Zhang, Xinyi Liu, Xiaoyi Zhang, Jun Wang, Gang Chen, and Yan Lu. Ui-evol: Automatic knowledge evolving for computer use agents. *ArXiv*, abs/2505.21964, 2025b. URL https://api.semanticscholar.org/CorpusID:278959655.

Ziyun Zhang, Zezhou Wang, Xiaoyi Zhang, Zongyu Guo, Jiahao Li, Bin Li, and Yan Lu. Infiniteweb: Scalable web environment synthesis for gui agent training. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 28465–28492, 2026c.

Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, et al. Swift: a scalable lightweight infrastructure for fine-tuning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 29733–29735, 2025.

Yuqi Zhou, Sunhao Dai, Shuai Wang, Kaiwen Zhou, Qinglin Jia, and Jun Xu. Gui-g1: Understanding r1-zero-like training for visual grounding in gui agents. *Advances in Neural Information Processing Systems*, 38:95683–95705, 2026.

A Appendix

A.1 Example Generated Task Schema

Table 4 shows the task schema of our generated tasks, while Figure 5 shows an example of evaluation code. The executable judge checks whether cell E2 contains a valid formula computing the absolute difference between B2 and C2, and returns a binary reward of 1 for success and 0 otherwise.

Table 4: A representative anonymized FORGE task instance. The example preserves the generated task structure while masking task-specific identifiers, file names, URLs, and local paths.

Field	Anonymized Content
`id`	`<task-uuid>`
`snapshot`	`<target-desktop-application>`
`instruction`	Natural-language task instruction for the agent
`source`	`synthetic`
`config`	Prepare required directories, download the input media file, launch the target application with the specified initial state, and perform optional initialization actions.
`related_apps`	`<target-desktop-application>`
`postconfig`	Query transient application state required for evaluation, persist it for subsequent checking, and optionally terminate the application before evaluation.
`result`	Inspect the final environment state, including application configuration, generated files, file validity and properties, and the persisted playback state.
`code`	Executable task-specific judge that verifies the final state and returns a deterministic scalar reward.
`metadata`	Behavior taxonomy path, behavior identifiers, composition-template identifier, scene identifier, instruction style, task complexity and generation method.
`validation`	Programmatic and semantic validation status, number of repair rounds, validation pass flags, and semantic-validation score.

A.2 Data Statistics

Table 5 reports the number of scenarios and synthesized tasks across the ten domains in OSWorld (Xie et al., 2024). For each domain, we generate atomic and template-composition tasks following the synthesis strategies described above.

Table 5: Scenario and task counts per application.

Category	Application	Scenarios	Behaviors	Templates	Atomic Tasks	Composition Tasks
OS	os	20	262	20	1,101	545
Office	libreoffice_calc	42	247	37	1,341	158
	libreoffice_impress	45	259	32	718	93
	libreoffice_writer	22	260	40	615	99
Daily	chrome	25	120	–	2,071	-
Professional	gimp	13	189	34	594	213
	thunderbird	15	513	30	875	40
	vlc	15	291	28	700	222
	vs_code	26	276	32	382	254
Workflow	multi_apps	44	25	–	–	991
Total		**267**	**2,442**	**253**	**8,397**	**2,615**

Figure 6 summarizes the statistics of our constructed training data. After filtering, we obtain 119K SFT steps originating from 10K trajectories. Figure 6(a) shows the task distribution across applications. Figure 6(b) shows the action distribution over 119K interaction steps. *LeftClick* is the dominant action, accounting for 45.8% of all steps, followed by *Key*, *Type*, and *Click*, while more complex interactions such as dragging are relatively rare. Figure 6(c) shows the trajectory-length

```python
import re
import openpyxl


wb = openpyxl.load_workbook(result, data_only=False)
ws = wb["Sheet1"] if "Sheet1" in wb.sheetnames else wb.active

cell = ws["E2"].value

ok = False

if isinstance(cell, str) and cell.startswith("="):
    norm = re.sub(r"\s+", "", cell.upper()).replace("$", "")

    # Remove optional sheet qualifiers,
    # e.g., SHEET1!B2 or 'SHEET 1'!B2
    norm = re.sub(
        r"(?:'[^']+'|[A-Z0-9_ ]+)!",
        "",
        norm,
    )

    ok = bool(
        re.fullmatch(
            r"=ABS\((B2-C2|C2-B2)\)",
            norm,
        )
    )

print(1.0 if ok else 0.0)
```

Figure 5: Example of executable evaluator code for the instruction: "I'm checking the 2014 balance sheet row; in E2, use ABS to calculate the absolute difference between Current Assets (B2) and Fixed Assets (C2)."

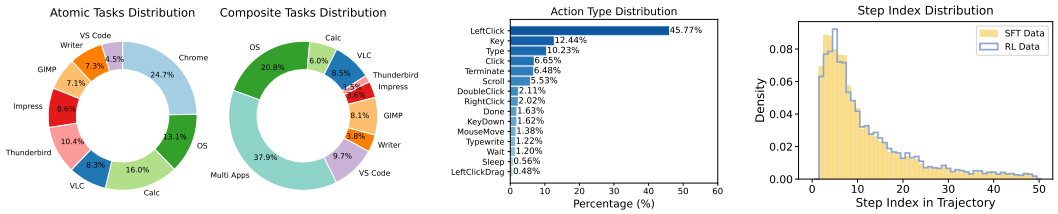
distributions of the SFT and RL data. Both exhibit similar long-tailed distributions, with average lengths of 12.0 and 12.2 steps, respectively, indicating that the two training stages cover tasks with comparable interaction complexity.

To further examine how task complexity changes during self-evolution, we analyze the complexity distribution of retained single-application tasks from v0 to v4. Figure 7 reports the distributions using the `complexity_level` labels recorded in task metadata. We normalize atomic and composite tasks separately within each iteration. Atomic tasks shift steadily toward higher complexity, while composite tasks show a similar trend from v1 onward. These results show that self-evolution also reshapes the task complexity distribution across iterations.

A.3 Specialized Scenarios

Most domains in FORGE follow a uniform scenario construction: each scenario is grounded in a concrete artifact file (a spreadsheet, a document, an image, etc.) that is downloaded into the virtual machine, and tasks are synthesized by pairing atomic or template behaviors with that artifact. Chrome and multi-application workflows require distinct construction strategies because their tasks are not anchored to a single offline artifact.

Chrome scenarios. Chrome uses a behavior-conditioned scene assignment strategy. Each atomic behavior is associated with a predefined subset of compatible scenes. These scenarios include blank and multi-tab states, `chrome://` pages, search engines, news sites, e-commerce platforms, social media, and developer websites. The candidate scenes are provided together with the behavior description to the expert model. The model then generates an instruction and selects the scene in which the task should be initialized. For example, a generic behavior such as `omnibox_search` can be instantiated in any scene, whereas `tab_pin` may be restricted to scenes containing multiple tabs. Once a scene is selected, FORGE constructs the corresponding initial browser state by



(a) (b) (c)

Figure 6: Statistics of the constructed training data. (a) Distribution of atomic and composite tasks across applications. (b) Distribution of the top-15 GUI action types. (c) Trajectory length distributions of the SFT and RL data.

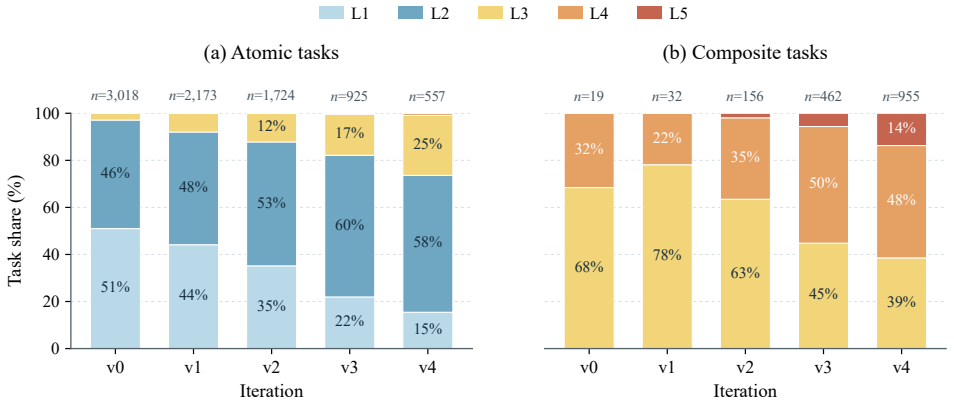


Figure 7: Task complexity distributions across self-evolution iterations. Each bar shows the proportions of complexity levels L1–L5 among retained tasks of the corresponding type and iteration; higher levels indicate greater complexity. Sample counts are shown above the bars.

launching Chrome and opening a URL sampled from that scene when applicable. This design allows the same behavior to be instantiated under diverse browser and web contexts while ensuring that scene-dependent behaviors are generated only in compatible states.

Multi-application scenarios. Cross-application tasks represent a distinct category in FORGE. Unlike single-application behaviors that target a specific GUI function within one application, each cross-application behavior describes a high-level information transfer or workflow pattern, such as extracting data from a spreadsheet and inserting it into a document, or downloading a file via the browser and processing it in a terminal. Multi-application tasks follow the same behavior–scene matching principle as single-application domains, but each scene describes a complete cross-application workspace, including the primary application state, related applications, available files and folders, and optional environment setup. This representation allows the expert model to synthesize instructions that naturally involve information transfer and coordinated operations across multiple applications. Behavior–scene compatibility is determined through tag matching, so each cross-application behavior is instantiated only in workspaces that provide the required applications and resources. Once a scene is selected, FORGE reconstructs the corresponding workspace by preparing the required files, folders, applications, and initial states.

A.4 Implementation Details

A.4.1 Action Space

We adopt the unified computer-use action space of Qwen3-VL, where each step outputs an action type together with its action-specific arguments. The action space includes keyboard operations (`key`, `type`), mouse operations (`mouse_move`, `left_click`, `right_click`,

Table 6: Unified action space of Qwen3-VL. Each action is specified by an action type and its corresponding arguments.

action type	arguments	details
key	keys	Press the specified keys in order and release them in reverse order.
type	text	Type the specified text using the keyboard.
mouse_move	coordinate (x, y)	Move the cursor to the specified screen coordinate.
left_click	coordinate (x, y)	Click the left mouse button at the specified screen coordinate.
left_click_drag	coordinate (x, y)	Click and drag the cursor to the specified screen coordinate.
right_click	coordinate (x, y)	Click the right mouse button at the specified screen coordinate.
middle_click	coordinate (x, y)	Click the middle mouse button at the specified screen coordinate.
double_click	coordinate (x, y)	Double-click the left mouse button at the specified screen coordinate.
triple_click	coordinate (x, y)	Triple-click the left mouse button at the specified screen coordinate.
scroll	pixels	Scroll vertically by the specified amount; positive values scroll up and negative values scroll down.
hscroll	pixels	Scroll horizontally by the specified amount.
wait	time	Pause execution for the specified number of seconds.
terminate	status	Terminate the current task with success or failure status.
answer	text	Return a textual answer to the user.

`middle_click`, `double_click`, `triple_click`, and `left_click_drag`), scrolling (`scroll` and `hscroll`), waiting (`wait`), task termination (`terminate`), and direct response (`answer`). Spatial actions additionally specify a screen coordinate (x, y), while other actions use arguments such as key combinations, input text, scroll distance, waiting time, or termination status. Following Qwen3-VL, we represent screen coordinates in a normalized 1000×1000 space. The complete action space is summarized in Table 6.

A.4.2 Action-Aware Scaling Details

We use the following default coefficients in Eq. (9): $w_{\text{type}} = 0.25$, $w_{\text{arg}} = 0.50$, and $w_{\text{desc}} = 0.25$. The premature-termination and repetition penalties are $\lambda_{\text{term}} = 0.5$ and $\lambda_{\text{rep}} = 0.3$, respectively. When the LLM judge is disabled, w_{desc} is folded into w_{arg}, resulting in a purely rule-based scaling score with $w_{\text{arg}} = 0.75$ and $w_{\text{desc}} = 0$. For each RL task, the corresponding successful SFT trajectory provides a step-aligned reference demonstration, whose labeled action is used to compute the action-aware scores.

The format indicator F_t is set to zero if the model output cannot be parsed as a valid `computer_use` tool call. The action-type indicator C_t is set to one only when the predicted action type matches the reference action type. Accordingly, both the argument score and the description score are gated by action-type correctness. The argument score $Q_t^{\text{arg}} \in [0, 1]$ is computed according to the action family. For coordinate-based actions, we use a smooth inverse-quadratic decay:

$$Q_t^{\text{arg}} = \frac{1}{1 + (d_t/\sigma)^2}, \tag{11}$$

where d_t is the Euclidean distance between the predicted and reference coordinates. During score calibration, we monitor `coord_hit@40` and coordinate errors over a sliding window of 256 steps. The observed median and mean errors are approximately 7 and 66 pixels, respectively, in the normalized 1000×1000 screen space. Based on these statistics, we set $\sigma = 40$, corresponding to a score of 0.5 at a 40-pixel error. This score is strictly positive and monotonically decreasing with

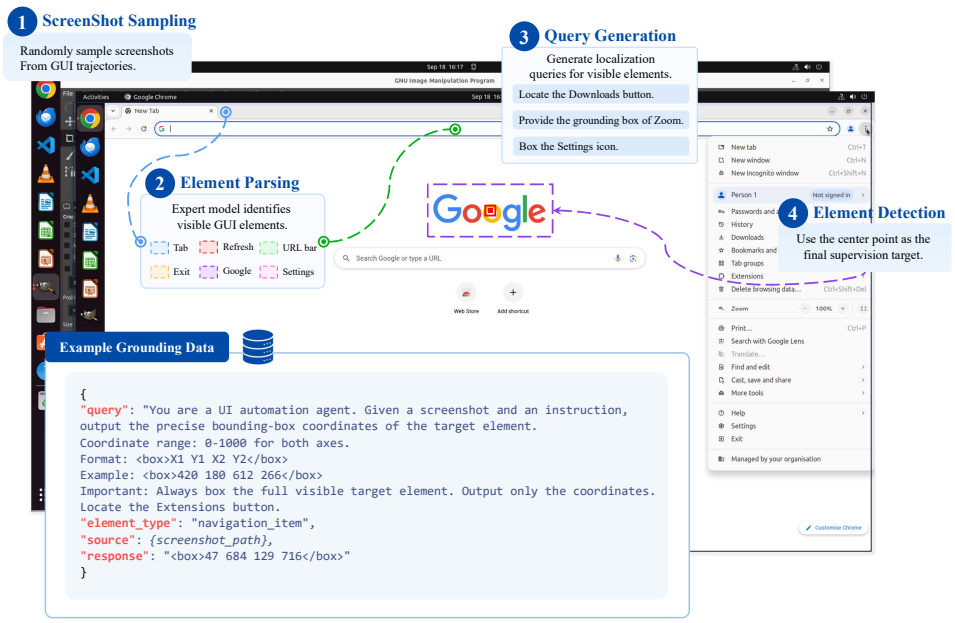


Figure 8: **Grounding data synthesis pipeline.** We sample screenshots from GUI trajectories, parse visible UI elements, generate localization queries for each element, and use the element center coordinates as supervision targets.

coordinate error, avoiding the dead zone caused by hard distance thresholds. For scrolling actions, `scroll` and `hscroll`, we decompose the score into direction and magnitude:

$$Q_t^{\arg} = \alpha_{\text{scroll}}\, \mathbb{I}[\text{sign}(\hat{p}_t) = \text{sign}(p_t)] + (1 - \alpha_{\text{scroll}}) \left(\frac{1}{1 + \frac{||\hat{p}_t| - |p_t||}{\max(|p_t|, 1)}} \right), \tag{12}$$

where $\hat{p}_t$ and p_t are the predicted and reference scroll amounts, and we set $\alpha_{\text{scroll}} = 0.5$ by default. This gives partial credit to predictions that scroll in the correct direction even when the magnitude is imperfect.

For text-producing actions, `type` and `answer`, we compute $Q_t^{\arg}$ using SequenceMatcher-based text similarity. For `key`, we compute the symmetric overlap between predicted and reference hotkey sets. For `wait`, we use relative numerical proximity:

$$Q_t^{\arg} = \frac{1}{1 + \frac{|\hat{v}_t - v_t|}{\max(|v_t|, 1)}}. \tag{13}$$

For `terminate`, $Q_t^{\arg} = 1$ only when the predicted termination status exactly matches the reference status, and is zero otherwise. If a reference argument is missing, empty, or NaN, we treat it as a don't-care field and assign full credit for that argument.

The description score $Q_t^{\text{desc}} \in \{0, 0.5, 1\}$ is produced by an optional LLM judge based on the natural-language `Action:` description. The judge checks whether the description is semantically consistent with the predicted tool call, the reference action, the task instruction, previous actions, and the current screenshot. It is invoked only when $C_t = 1$, so a fluent description cannot compensate for choosing the wrong action type.

The premature-termination indicator E_t is set to one when the model predicts `terminate` while the reference action is not `terminate`; otherwise it is zero. The repetition indicator L_t is set to one when the predicted action repeats the immediately preceding action while the reference requires a different operation. For coordinate actions, repetition is determined by matching the action type and checking whether the coordinates are within 10 pixels. For non-coordinate actions, exact action matching is used. If the reference itself requires repeating the previous action, then $L_t = 0$ and

Table 7: GUI grounding accuracy (%). Both benchmarks are single-turn and involve no multi-turn interaction. Baseline rows and the Backbone row are taken from the respective public reports.

Model	ScreenSpot-Pro	OSWorld-G
UI-TARS-7B (Qin et al., 2025)	35.7	47.5
EvoCUA-32B (Xue et al., 2026)	49.8	63.9
OpenCUA-72B (Wang et al., 2026b)	60.8	67.0
Qwen3-VL-32B-Thinking (Bai et al., 2025)	57.1	64.0
Qwen3-VL-8B-Thinking	46.6	56.7
FORGE-8B	**60.3**	**69.1**

no repetition penalty is applied. We additionally perform initialization-time monotonicity checks on the configured score bands to ensure that correctly matched actions remain strictly preferred to mismatched actions under the default coefficients.

A.4.3 Grounding Data Synthesis

To alleviate the reasoning-grounding trade-off, we further construct a set of grounding data to complement the main GUI training corpus. As shown in Figure 8, we begin by randomly sampling screenshots from existing GUI trajectories. These screenshots capture realistic interface states encountered during agent interaction. We then use a strong vision model (GPT-5.5) to parse all visible UI elements in each screenshot. For every detected element, the parser produces its category, semantic description, and bounding box. Based on these parsed elements, we automatically construct query-answer pairs by turning each element into a localization query, such as identifying a button, icon, or menu entry on the screen. Moreover, we convert each element's bounding box into the unified coordinate space used by Qwen3-VL, namely $[0, 1000] \times [0, 1000]$. Given a bounding box (x_1, y_1, x_2, y_2), we use its center point as the prediction target:

$$c_x = \frac{x_1 + x_2}{2}, \qquad c_y = \frac{y_1 + y_2}{2}.$$

After normalization to the target coordinate space, the final supervision target is:

$$\hat{c}_x = 1000 \cdot \frac{x_1 + x_2}{2W}, \qquad \hat{c}_y = 1000 \cdot \frac{y_1 + y_2}{2H},$$

where W and H denote the screenshot width and height. This process produces grounding samples in the form of $(\text{screenshot}, \text{query}, \text{center coordinate})$, which are mixed into SFT training to improve the model's grounding ability.

The results in Table 7 validate the effectiveness of our grounding data construction. Compared with the Qwen3-VL-8B-Thinking backbone, FORGE-8B improves grounding accuracy from 46.6% to 60.3% (+13.7) on ScreenSpot-Pro (Li et al., 2025a) and from 56.7% to 69.1% (+12.4) on OSWorld-G (Xie et al., 2026b). Despite using an 8B backbone, FORGE-8B also achieves grounding performance comparable to or better than that of substantially larger GUI agents. These results demonstrate that the constructed grounding data effectively strengthens visual localization ability and helps mitigate grounding degradation during GUI agent post-training.

A.4.4 Failure Taxonomy

This section provides detailed definitions of the failure categories used in Figure 1 and Figure 4. Each failed trajectory is assigned to a single primary category based on the dominant factor that determines task failure. The four failure categories are defined as follows:

Planning Error. The model fails to formulate an effective action strategy for completing the task. This includes either choosing an invalid approach or following a valid but inefficient plan. For example, when asked to fill a spreadsheet range, the model may repeatedly explore unrelated menus or edit the cells one by one instead of using a more efficient strategy.

Table 8: Expert human-judge audit of evaluator decisions on atomic and composite tasks. We randomly sample 10 evaluator-passed and 10 evaluator-rejected trajectories per application domain. Human Valid/Invalid reports the number judged successful/failed by expert human reviewers; FP/FN count evaluator false positives and false negatives against the human labels.

Domain	Pass	Human Valid	FP	Reject	Human Invalid	FN
Atomic						
Chrome	10	10	0	10	6	4
GIMP	10	9	1	10	7	3
LibreOffice Calc	10	8	2	10	8	2
LibreOffice Impress	10	10	0	10	6	4
LibreOffice Writer	10	9	1	10	5	5
OS	10	10	0	10	9	1
Thunderbird	10	9	1	10	6	4
VLC	10	10	0	10	7	3
VS Code	10	10	0	10	8	2
Total	**90**	**85**	**5**	**90**	**62**	**28**
Composite						
GIMP	10	9	1	10	7	3
LibreOffice Calc	10	10	0	10	8	2
LibreOffice Impress	10	7	3	10	7	3
LibreOffice Writer	10	10	0	10	6	4
OS	10	9	1	10	9	1
Thunderbird	10	6	4	10	10	0
VLC	10	9	1	10	10	0
VS Code	10	10	0	10	7	3
Total	**80**	**70**	**10**	**80**	**64**	**16**

Perception Error. The model fails to correctly recognize or interpret the current GUI state, including the current page, dialog, selection, application state, state changes, or task completion status. For example, it may mistake controls from a background application for those of the active window.

Grounding Error. The model understands the intended operation and identifies the correct semantic target, but fails to accurately localize or interact with the corresponding UI element. For example, it may attempt to click the correct button but select a nearby control instead.

Execution Error. The model has a valid plan and correctly identifies the interaction target, but fails while carrying out the required actions. For example, it may enter an incorrect value, repeat ineffective operations, or fail to complete the remaining steps before termination.

A.5 Expert Human-Judge Audit

To directly assess the reliability of our executable evaluators, we conduct an expert human audit of generated trajectories. For each application domain, we randomly sample 10 trajectories labeled as successful and 10 labeled as unsuccessful by the executable evaluator, and ask expert reviewers to inspect the task instruction, trajectory, and final environment state. We conduct the audit separately for atomic and composite tasks.

As shown in Table 8, the evaluator decisions agree with human judgments on **81.7%** of the 180 audited atomic trajectories and **83.8%** of the 160 composite trajectories. Among evaluator-passed trajectories, only 5 atomic and 10 composite cases are identified as false positives by human reviewers. The audit also shows that false negatives are more common than false positives, suggesting that our evaluators tend to favor conservative decisions. This behavior is desirable for trajectory collection, since false positives introduce incorrect supervision, whereas false negatives only reduce the amount of retained data. The low false-positive rate therefore supports the reliability of our executable evaluators for constructing high-quality training data.

Table 9: Comparison with proprietary and open-source models under different interaction budgets.

Model	Steps	OS	Office	Daily	Professional	Workflow	Overall
GPT-5.5 (OpenAI, 2026)	–	–	–	–	–	–	78.7
Claude Opus 4.6 (Anthropic, 2026)	–	–	–	–	–	–	72.7
Claude Sonnet 4.5 (Anthropic, 2025b)	100	70.8	72.6	61.4	63.2	49.5	62.9
Seed-1.8 (Seed, 2026)	100	66.7	68.8	67.1	71.4	42.4	61.9
CoACT-1 (Song et al., 2025)	50	70.8	60.6	54.1	69.4	42.4	56.4
Kimi K2.5 (Team et al., 2026)	100	73.9	69.2	66.4	73.5	46.1	63.3
EvoCUA-32B (Xue et al., 2026)	50	78.3	59.8	64.5	81.6	27.9	56.7
UI-TARS-2 (Wang et al., 2025a)	100	41.7	61.1	62.1	61.2	34.1	53.1
ComputerRL-9B (Lai et al., 2026)	50	75.0	45.8	52.5	71.4	27.9	48.0
OpenCUA-72B (Wang et al., 2026b)	50	45.8	42.7	53.8	75.5	22.3	44.9
DART-GUI-7B (Li et al., 2025b)	30	62.5	39.3	50.8	73.5	16.7	42.1
Qwen3-VL-32B-Thinking (Bai et al., 2025)	50	–	–	–	–	–	41.0
Qwen3-VL-4B-Thinking	50	–	–	–	–	–	31.4
FORGE-Qwen3-VL-4B	15	66.7	29.1	33.3	65.3	8.6	32.1
FORGE-Qwen3-VL-4B	30	54.2	35.0	43.6	61.2	15.1	36.6
Qwen3-VL-8B-Thinking	–	–	–	–	–	–	33.9
FORGE-Qwen3-VL-8B	15	62.5	32.5	37.2	63.3	12.9	34.6
FORGE-Qwen3-VL-8B	30	62.5	41.9	55.1	69.4	23.7	**45.2**

Table 10: Ablation of action-aware scaling components in RL.

Reward	Full	OS	Office	Daily	Prof.	Work.
Full	**51.2**	**70.8**	**52.1**	**60.3**	**73.5**	**25.8**
w/o L_t	48.8	66.7	47.0	59.0	73.5	24.7
w/o E_t, L_t	47.4	70.8	49.6	52.6	63.3	25.8
w/o $Q_t^{\text{desc}}, E_t, L_t$	46.3	70.8	44.4	53.8	69.4	23.7

A.6 Algorithm Overview

Algorithm 1 summarizes the self-evolving data production process in FORGE. Each iteration first synthesizes atomic and template-composition tasks for training, together with a separate set of free-composition exploration tasks. The generated tasks are then filtered or repaired by the current validation pipeline, after which successful trajectories are collected to update the policy. Failures observed during task generation are distilled into new validation skills, while failures on the exploration tasks are used to expand the atomic behavior set and composition templates. The validation rules and task space are updated after each iteration. Subsequent rounds can therefore target the model's current capability gaps more closely.

A.7 Effect of Interaction Budget

We further evaluate FORGE under different interaction budgets to examine its efficiency under constrained execution. As shown in Table 9, larger budgets improve task completion, particularly on applications requiring longer workflows. For the 8B model, success rates rise from 34.6% at 15 steps to 45.2% at 30 steps (+10.6%), and further to 51.2% at 50 steps (+6.0%). Similar trends are observed for the 4B model. These results suggest that FORGE remains effective even under constrained interaction budgets, while additional steps further benefit complex tasks.

A.8 Additional Ablation Results

This section provides further ablations of the action-aware advantage scaling mechanism and the two self-evolving components. Unlike the SFT-only experiments in Section 4.3, these experiments use the full SFT and RL post-training pipeline.

Table 11: Training hyperparameters. Left: supervised fine-tuning configuration. Right: GRPO configuration.

SFT

Hyperparameter	Value
Framework	MS-SWIFT
Hardware	32×H800
Training epochs	3
Learning rate	2×10^{-5}
LR schedule	Cosine
Warmup ratio	0.05
Maximum sequence length	16,384
Maximum image pixels	2,073,600
Precision	bfloat16
Sequence parallel size	4
Memory optimization	ZeRO-3
Gradient checkpointing	Enabled
Freeze vision encoder	Yes
Freeze vision–language aligner	Yes

GRPO

Hyperparameter	Value
Learning rate	1×10^{-6}
Task batch size	64
Rollouts per task	8
KL coefficient	1×10^{-2}
Entropy coefficient	0
Clip ratio	0.2/0.2
Clip-ratio cap	3.0
Rollout temperature	1.0
Maximum training steps	∼785
Maximum sequence length	14,336
Maximum prompt length	12,288
Maximum response length	2,048
Maximum interaction turns	50

Table 12: Ablation of the two self-evolving components.

Setting	Overall	OS	Office	Daily	Prof.	Work.
Full	**51.2**	**70.8**	**52.1**	**60.3**	**73.5**	**25.8**
w/o skill evolution	47.4	70.8	49.6	52.6	63.3	25.8
w/o exploration	44.6	62.5	41.9	53.8	73.5	20.4
w/o both components	41.4	62.5	41.6	52.6	53.1	19.1

A.8.1 Action-Aware Scaling Ablation

We further analyze the contribution of individual components in the action-aware advantage scaling mechanism. As shown in Table 10, removing the repetition penalty reduces the success rate from 51.2% to 48.8% (-2.4), while further removing the early-termination penalty lowers it to 47.4% (-3.8). When the description-level score is additionally removed, performance further drops to 46.3% (-4.9). Overall, these results show that both semantic action scoring and explicit penalties for undesirable actions contribute to more effective credit assignment during RL optimization.

A.8.2 Self-Evolution Components Ablation

We conduct an additional ablation on FORGE's two self-evolving components: validation-skill updates and free-composition exploration. Validation-skill updates add recurring errors from task generation and validation to the validation rules. Free-composition exploration evaluates the model on new behavior combinations and uses the observed failures to adjust behavior composition and task sampling. We compare four configurations: *Full*, which enables both components; *w/o skill evolution*, which keeps the validation skills fixed but retains free-composition exploration; *w/o exploration*, which disables free-composition exploration but retains validation-skill updates; and *w/o both components*, which disables the entire self-evolution process. All configurations use the same base model, post-training procedure, evaluation protocol, and training budget.

Table 12 reports the results. Removing either component reduces performance relative to the full system. Removing *skill evolution* reduces the overall success rate from 51.2% to 47.4%, while removing *exploration* reduces it to 44.6%. Disabling *both components* further reduces the success rate to 41.4%. These results show that both components contribute to FORGE's performance.

Algorithm 1 Self-Evolving Data Production

Input: scenario library $\mathcal{S}$, atomic behaviors $\mathcal{B}_0$, composition templates $\mathcal{M}_0$, validation skills $\mathcal{V}_0$, initial policy π_0, number of free compositions N, number of iterations R
for iteration $r = 0$ to $R - 1$ **do**
  Initialize training tasks $\mathcal{X}_r \leftarrow \emptyset$ and probe tasks $\mathcal{P}_r \leftarrow \emptyset$
  for atomic behavior $b \in \mathcal{B}_r$ **do**
    $\mathcal{C}(b) \leftarrow \{s \in \mathcal{S} \mid \mathcal{T}_{\text{req}}(b) \subseteq \mathcal{T}(s)\}$
    Select $\mathcal{S}(b) \subseteq \mathcal{C}(b)$ using preferred-tag matching
    Generate atomic tasks $\mathcal{X}^a$ from (b, s) for $s \in \mathcal{S}(b)$
    $\mathcal{X}_r \leftarrow \mathcal{X}_r \cup \mathcal{X}^a$
  end for
  for template $m \in \mathcal{M}_r$ **do**
    Sample B from $\mathcal{B}_r$ subject to template m
    Find compatible scenarios $\mathcal{C}(B, m)$
    Generate template-composition tasks $\mathcal{X}^m$ from (B, s) for $s \in \mathcal{C}(B, m)$
    $\mathcal{X}_r \leftarrow \mathcal{X}_r \cup \mathcal{X}^m$
  end for
  for $i = 1$ to N **do**
    Sample cross-category behavior composition $\widetilde{B}$
    Find compatible scenarios $\mathcal{C}(\widetilde{B})$
    if $\mathcal{C}(\widetilde{B}) \neq \emptyset$ **then**
      Generate free-composition tasks $\mathcal{P}^f$ for $s \in \mathcal{C}(\widetilde{B})$
      $\mathcal{P}_r \leftarrow \mathcal{P}_r \cup \mathcal{P}^f$
    end if
  end for
  $(\mathcal{X}_r, \mathcal{P}_r) \leftarrow \mathcal{V}_r(\mathcal{X}_r, \mathcal{P}_r)$
  Verify $(\mathcal{X}_r, \mathcal{P}_r)$ in real environment; retain passed as $(\widetilde{\mathcal{X}}_r, \widetilde{\mathcal{P}}_r)$
  $\mathcal{D}_r \leftarrow \text{Traj}_{\text{success}}(\pi_r, \widetilde{\mathcal{X}}_r)$
  Group failed tasks into error patterns $\mathcal{F}_r$
  Distill new validation skills $\Delta\mathcal{V}_r$ from $\mathcal{F}_r$
  $\mathcal{V}_{r+1} \leftarrow \mathcal{V}_r \cup \Delta\mathcal{V}_r$
  Train π_r on $\mathcal{D}_r$ to obtain π_{r+1}
  Evaluate π_{r+1} on $\widetilde{\mathcal{P}}_r$ and collect failures $\hat{\mathcal{F}}_r$
  $\mathcal{B}_{r+1} \leftarrow \mathcal{B}_r$; $\mathcal{M}_{r+1} \leftarrow \mathcal{M}_r$
  for $f \in \hat{\mathcal{F}}_r$ **do**
    if f reveals a missing atomic behavior b **then**
      $\mathcal{B}_{r+1} \leftarrow \mathcal{B}_{r+1} \cup \{b\}$
    else if f reveals a missing composition template m **then**
      $\mathcal{M}_{r+1} \leftarrow \mathcal{M}_{r+1} \cup \{m\}$
    end if
  end for
end for
Output: $\pi_R, \mathcal{B}_R, \mathcal{M}_R, \mathcal{V}_R$

A.9 Training and Evaluation Details

This section provides training and evaluation details, including data configuration, optimization settings, training infrastructure, and the OSWorld evaluation protocol. Table 11 summarizes the main hyperparameters for SFT and GRPO.

OSWorld Evaluation. Following common practice, we exclude the 8 Google Drive tasks from the 369-task OSWorld benchmark, yielding a final evaluation set of 361 tasks (OS 24, Office 117, Daily 78, Professional 49, Workflow 93). We use the official OSWorld agent loop and evaluator, with a maximum of 50 interaction steps per task and a screen resolution of 1920×1080. Model inference uses a temperature of 0.0 and a maximum generation length of 32,768 tokens.

Supervised fine-tuning. We perform full-parameter SFT using MS-SWIFT (Zhao et al., 2025) on 32 H800 GPUs. The training mixture contains 362,979 samples and is shuffled during training. The models are trained for 3 epochs with bfloat16 precision, and we use the checkpoint from the final epoch for evaluation. We use a global batch size of 512 for the 8B model and 2,048 for the 4B

model. The learning rate is set to 2×10^{-5} with a cosine schedule and a warmup ratio of 0.05. The maximum sequence length is 16,384. We enable gradient checkpointing and ZeRO-3 for memory optimization, and use sequence parallelism with size 4. The maximum number of image pixels is set to 2,073,600. As described in Section 3, only the target action is included in the SFT loss, while preceding interaction steps are retained as context and masked from supervision.

Reinforcement Learning. GRPO is implemented with verl (Sheng et al., 2025). For each task, GRPO samples a group of G rollouts, where each rollout receives a trajectory-level reward R_i from the executable evaluator based on task completion. We first compute the group-normalized advantage:

$$A_i^{\text{grp}} = \frac{R_i - \bar{R}}{\hat{\sigma}_R + \epsilon}. \tag{14}$$

Here, $\bar{R}$ and $\hat{\sigma}_R$ denote the mean and standard deviation of rewards within the group.

To provide finer-grained action-level supervision, we reweight the trajectory-level advantage with the action-aware coefficient $\alpha_{i,t}$ and apply the clipped GRPO objective only to executable-action tokens:

$$\mathcal{L}_{\text{GRPO}} = -\frac{1}{G} \sum_{i=1}^{G} \sum_{t=1}^{T_i} m_{i,t} \min \left(\rho_{i,t} \tilde{A}_{i,t}, \text{clip} \left(\rho_{i,t}, 1-\varepsilon, 1+\varepsilon \right) \tilde{A}_{i,t} \right). \tag{15}$$

The likelihood ratio is defined as

$$\rho_{i,t} = \frac{\pi_\theta(y_{i,t} \mid q, y_{i,<t})}{\pi_{\text{old}}(y_{i,t} \mid q, y_{i,<t})}. \tag{16}$$

Here, $m_{i,t} \in \{0, 1\}$ masks non-action tokens from the policy loss.

Before RL training, we use the SFT model to generate 8 rollouts for each candidate task and evaluate their task outcomes. We discard groups whose rollouts are either all successful or all unsuccessful, since such groups provide no informative within-group contrast for GRPO. This filtering retains 6,628 tasks for RL training.

We use a learning rate of 1×10^{-6} and a task batch size of 64. For each sample, we draw 8 rollouts with a sampling temperature of 1.0. The clipping range is set to $0.2/0.2$ with a clip-ratio cap of 3.0. We use a KL coefficient of 1×10^{-2} and no entropy regularization. The maximum sequence length is 14,336 tokens, consisting of at most 12,288 prompt tokens and 2,048 response tokens. Training runs for approximately 785 optimization steps.

A.10 Skill Examples

To illustrate the validation skills distilled from recurring failures in the data flywheel, we present three representative examples covering requirement coverage (Figure 9), task validity (Figure 10), and cross-application consistency (Figure 11). Each skill follows a unified structure consisting of a scope description, validation rules, failure criteria, and repair guidance. During validation, relevant skills are provided to the LLM reviewer together with the task instruction, configuration, and evaluator to identify failures and generate repair suggestions. Across iterations, recurring failure patterns are used to add or update skills as described in Appendix A.13. The LLM reviewer is used only for diagnosis and repair, while task retention is additionally constrained by programmatic checks and real-environment verification.

Skill Example: Check Whether The Evaluator Covers Every Task Requirement

---
name: osworld-requirement-coverage
description: Detect OSWorld evaluators that omit material task requirements or verify only an easy substep of a multi-step task.

Requirement Coverage Review

Skill Description
This skill checks whether the evaluator covers every material, verifiable requirement in the instruction rather than validating only a subset of the task. Break the instruction into separate requirements and verify that each one is evaluated.

Validation Rule
Check whether the evaluator covers requirements such as:
• the correct target object;
• the correct content or value;
• the correct location or range;
• the correct formatting or style;
• the correct filename and path;
• the correct extension or file type;
• the correct sorting, filtering, calculation, or transformation result;
• the correct save or export result;
• the correct UI state;
• the correct transfer of content between source and destination;
• every necessary outcome in a multi-step task.
A multi-step task must not be evaluated only by checking the easiest or most visible substep. For tasks that do not require preservation of the interaction process, evaluating the final state is usually sufficient, provided that the final state contains enough evidence to verify every material requirement.

Failure Criteria
The task must fail validation if the evaluator:
• omits one or more material requirements from the instruction;
• verifies only a partial result of a multi-step task;
• checks only an easy proxy condition while ignoring required outcomes;
• cannot provide sufficient final-state evidence that all required outcomes were completed.

Repair Guidance
Revise the evaluator so that every material requirement is explicitly covered.
When necessary:
• decompose the instruction into independent verifiable conditions;
• add checks for missing content, formatting, location, transformation, save, export, or transfer requirements;
• ensure multi-step tasks verify all required outcomes rather than only the final or easiest substep;
• use appropriate getters or additional state checks when the current final state does not provide enough evidence.

Examples
Suppose the instruction requires the model to:
1. modify spreadsheet data;
2. create a chart;
3. save the file to a specified path.
If the evaluator checks only that a chart exists, it misses required conditions and must fail validation.

Figure 9: Skill distilled from recurring evaluator failures in generated GUI tasks. The Failure Criteria field captures incomplete verification, such as omitting material requirements, checking only partial outcomes, or relying on easy proxy conditions.

Skill Example: Determine Whether The Task Itself Is Valid

---
name: osworld-task-validity
description: Detect invalid OSWorld GUI tasks caused by ambiguous instructions, missing targets, inconsistent initial state, or tasks already completed before execution.

Task Validity Review

Skill Description
This skill checks whether the combination of `instruction` and `config` forms a clear, executable, and verifiable GUI task before reviewing the evaluator. The task should have a uniquely identifiable target, a well-defined required outcome, and an initial state that is consistent with the instruction.

Validation Rule
Check whether the instruction clearly identifies:
• which file, document, webpage, email, script, worksheet, slide, layer, or UI element must be operated on;
• which specific object or range must be modified;
• what the required final state is;
• where the result must be saved or exported;
• whether the target can be uniquely identified when multiple candidate objects exist.

A short instruction may still be valid if the config and initial environment make its meaning uniquely resolvable.
Also use the config to verify that:
• the object referenced by the instruction actually exists;
• the target can be located in the initialized environment;
• filenames, paths, contents, and application state are consistent with the instruction;
• the task is not already completed before the model acts;
• the initial state does not already satisfy the intended completion condition;
• the target is accessible and uniquely identifiable;
• the config places the target in the correct path, application, and format.

Failure Criteria
The task must fail validation if:
• multiple targets or interpretations are equally reasonable but the evaluator accepts only one;
• the instruction is too vague to define an objective completion condition;
• the evaluator introduces requirements not stated in the instruction;
• the required object does not exist in the initialized state;
• the config initializes a different file, path, object, or application state from the one described by the instruction;
• the task is already completed in the initial state;
• the initial state already satisfies the intended evaluator condition.

Repair Guidance
Revise the instruction or config so that the task has a unique target, a clear final state, and a consistent initial environment.
When necessary:
• clarify the target file, object, range, or UI element;
• specify ambiguous references such as “latest”, “title”, or “script”;
• align filenames, paths, object identifiers, and application state between the instruction and config;
• remove pre-existing completed results from the initial state;
• ensure the required target exists and is accessible before execution.
Do not add hidden task requirements solely to make the evaluator easier to implement.

Examples
Ambiguous instructions include:
• “Change the title font to SimSun,” when the document contains multiple titles.
• “Run that script,” when the directory contains multiple scripts.
• “Modify the spreadsheet,” without specifying a sheet, range, or cell.
• “Open the latest file,” without defining how “latest” should be determined.
• “Organize it,” “fix it,” or “improve it,” without an objective completion criterion.

Invalid initial states include:
• the instruction asks to modify a sheet that does not exist;
• the instruction asks to run a script that is not present;
• the config already creates the fully correct target result;
• the filename created by the config does not match the filename in the instruction;
• the evaluator checks a different object from the one initialized by the config.

Figure 10: Skill distilled from recurring failures in generated GUI tasks. The Failure Criteria field captures ambiguous instructions, inconsistent initial states, and tasks already completed before execution.

Skill Example: Check cross-application and source consistency

---
name: osworld-cross-application-consistency
description: Detect OSWorld copy, download, transcription, and cross-application evaluators that cannot prove the correct source, complete transfer, or saved destination.

Cross-Application and Source Consistency Review

Skill Description
This skill applies to cross-application tasks, copying tasks, downloads, transcription, and other data-transfer tasks where the final destination state alone may not be sufficient proof of correct completion. The evaluator should be able to establish the relationship between the required source content and the final destination result.

Validation Rule
Check whether the evaluator verifies:
• the correct source object;
• the correct destination object;
• that the transferred content matches the source;
• that the transferred content is complete;
• that the model used the specific, latest, or matching object required by the instruction;
• that the destination artifact was correctly saved;
• that the model did not fabricate similar-looking content from another source.
The evaluator should verify both the correctness of the transferred content and its provenance whenever the instruction depends on a specific source.

Failure Criteria
The task must fail if the evaluator cannot reliably prove the relationship between the required source content and the destination result.
In particular, the evaluator is insufficient if it only checks for partial keywords, similar-looking content, or the existence of the destination artifact without verifying the required source and complete transfer.

Repair Guidance
When the evaluator cannot establish source-to-destination consistency, revise it to:
• identify and inspect the required source object;
• extract or obtain the expected source content;
• compare the destination content against the source;
• verify that the transfer is complete;
• verify that the destination artifact was saved to the required location.
Avoid relying only on weak proxies such as keyword presence when stronger source-based verification is available.

Examples
Example instruction:
> Copy the body of the latest invoice email from Thunderbird into Writer and save it.
Checking only whether the Writer document contains the word “invoice” is insufficient.
At minimum, the evaluator should verify:
• that the correct email was used;
• that the document content matches that email;
• that the content is complete;
• that the Writer file was saved to the correct path.

Figure 11: Skill distilled from recurring failures in cross-application tasks. The Failure Criteria field captures weak source-to-destination verification, such as checking only partial keywords or destination existence without validating provenance and complete transfer.

A.11 Instruction Generation Prompt

Task Generation Prompt

You are an expert in synthesizing training data for GUI agents.

Task Background
We are training a GUI agent that follows natural-language instructions by observing screenshots and interacting with graphical user interfaces.

Your task is to generate diverse training instructions for a given atomic GUI behavior.

Target Atomic Behavior
- Behavior name: {behavior_name}
- Category: {category}
- Description: {description}
- Reference instructions: {examples_text}

Current Environment
Application: {application}

{application_context}

{scene_constraint_section}

Generation Requirements
Generate exactly {count} diverse task instructions.

1. Language Distribution
Approximately 80% English and 20% Chinese.

2. Instruction Styles
Distribute the generated instructions approximately evenly across the following four styles:
- command: a direct instruction.
- casual: an informal user request.
- story: a request with brief contextual motivation.
- question: a question-form request expressing the intended operation.

3. Parameter and Content Diversity
- Vary relevant parameters such as numerical values, colors, percentages, angles, object references, or positions whenever applicable.
- Ground each instruction in the actual content and state of the current environment.
- Refer only to objects, text, regions, or other entities that actually exist in the provided environment description.
- Do not fabricate nonexistent content.

4. Complexity Distribution
- Level 1 (60%): simple and direct instructions.
- Level 2 (30%): instructions with additional context or specification.
- Level 3 (10%): instructions involving constraints or requiring contextual interpretation.

5. Challenging Cases
When compatible with the target behavior and current environment, include examples involving application-specific difficult cases described in:
{application_specific_guidelines}

Output Format
Return only a JSON array with exactly {count} elements. Each element must follow:

```
[
  {
    "instruction": "natural-language user instruction",
    "style": "command|casual|story|question",
    "complexity": 1
  }
]
```

Requirements:
- Do not wrap the output in Markdown code fences.
- The "instruction" field must contain only a natural user request and must not mention terms such as "atomic behavior" or "GUI agent".
- Each instruction must correspond to an executable operation in the current environment.

Figure 12: Prompt for GUI instruction generation based on scenario–behavior matching.

A.12 EVALUATOR GENERATION PROMPT

Evaluator Generation Prompt

You are an expert evaluator generator for the OSWorld automated GUI task evaluation system.
Given a task instruction and its environment config, generate an evaluator JSON object that verifies whether the task was completed correctly.

The evaluator uses a "code" field as a flexible Python metric. The framework first obtains observable state through OSWorld getters, injects the values as Python variables such as `result` and optional `expected`, then executes your code. You must decide whether the task needs gold generation from an initial state or can be evaluated from the final state.

Two valid code patterns:
1. Generate gold + evaluate: read an original file/state from `expected`, programmatically apply the requested operation, then compare the generated gold with `result`.
2. Evaluate only: directly inspect `result` or other fetched final state and print the score. Use this when gold generation is unnecessary.

Evaluator JSON Schema

```
{
  "postconfig": [...],
  "result": {"type": "<getter_type>", "...": "..."},
  "expected": {"type": "<getter_type>", "...": "..."},
  "code": "<python_code>"
}
```

`expected` is optional. Include it only when the evaluator needs an original file, initial state, or other reference state.

Field Descriptions

postconfig (optional)

Pre-evaluation steps executed before checking the result. Each step has "type" and "parameters".
Supported types:
- activate_window: {"window_name": "<title>", "strict": true}
- sleep: {"seconds": <float>}
- execute: {"command": ["python", "-c", "<code>"] or ["bash", "-c", "<cmd>"]}
- launch: {"command": [...] or "<shell_cmd>", "shell": true/false}
- close_window: {"window_name": "<name>", "strict": true, "by_class": true}

Common patterns:
- Save file before evaluation:
  execute python -c
  "import pyautogui; pyautogui.hotkey('ctrl', 's')"
- Wait for UI: sleep 0.5--3 seconds.
- Restart an application to check persistent settings: terminate and then relaunch the application.

expected (optional)

A reference state fetched before evaluation and injected as the Python variable `expected`. Include it only when the code needs an original file, initial state, or other reference state to generate gold or check a toggle/change.
Common examples:
- vm_file: {"type": "vm_file", "path": "<path_on_vm>", "dest": "<filename>"}
- Application/configuration/state getter types supported by OSWorld getters.
Do not invent an `expected` source. If a needed initial state is not present in Config, the task is not reliably evaluable.

result (usually required)

The final observable state after the agent operates, injected as the Python variable `result`. It may be a downloaded file path, command output, application configuration, accessibility tree, browser state, screenshot file path, or another observable state depending on the getter type.
Common examples:
- vm_file: {"type": "vm_file", "path": "<path_on_vm>", "dest": "<filename>"}
- vm_command_line: {"type": "vm_command_line", "command": ["<command>", "..."]}
- accessibility_tree: {"type": "accessibility_tree"}
- Application-specific types such as active_tab_info, bookmarks, vlc_config, and gimp_config_file.

code (required)

Python code that:
1. Decides whether to generate gold from `expected` or directly evaluate `result`.
2. Reads/parses the fetched states or files.
3. Prints a score (0.0--1.0) as the last line of stdout:
   - For exact-match tasks: print 1.0 if correct and 0.0 otherwise.
   - For similarity tasks: print a float between 0.0 and 1.0.

The variables `result` and optional `expected` are pre-injected before execution. `result` and `expected` may be paths or non-file states depending on the getter.

Application-specific Evaluation Skill

{evaluation_skill}

Now generate the evaluator for the following task:

Instruction:
{instruction}

Config:
{config}

{file_content_section}

Output ONLY the evaluator JSON object.

Figure 13: Prompt for executable evaluator generation. The prompt generates task-specific evaluators from the instruction and environment configuration, guided by application-specific evaluation skills and observable GUI states.

A.13 SELF-EVOLUTION PROMPT

Evolution Analysis Prompt

Your responsibility is to analyze the failure patterns of the GUI Agent on the current tasks, identify the model's current capability gaps based on that analysis, and update the data generation strategy for the next round. We are building a data flywheel for producing GUI tasks. The target repository is located at:
`{repo_root}`

The data flywheel primarily produces tasks through scene-behavior matching. The system maintains:

`data_flywheel/{app_dir}/behaviors/[0-9]*_*.jsonl`: atomic behavior library
`data_flywheel/{app_dir}/behaviors/scenes.jsonl`: scene definitions
`data_flywheel/{app_dir}/behaviors/composites.jsonl`: composition templates

After each training round, the current model performs rollouts on a batch of **Free Composition Tasks**. These tasks are not included in the current round's training data; they are used to explore the boundaries of the current model's capabilities.

Your task is to analyze these tasks and their failure trajectories, identify the model's current **Capability Gaps** from multiple failure cases, and update the data generation strategy for the next round accordingly. Examples include capabilities the current model systematically lacks, capabilities omitted from the existing data generation space, and data that needs to be generated in the next round.

Input

Results directory for this round: *`{probe_results_dir}`*

Each case contains a trajectory, result score, and possible failure diagnosis: *`result.txt`* / *`step_*.png`* / *`traj.jsonl`* / *`failure_diagnosis`*.

Step 1: Capability Gap Analysis

Cluster valid failure cases other than environmental anomalies, such as application crashes, network failures, or invalid accounts, and identify recurring capability gaps. Pay particular attention to distinguishing:

- MISSING_ATOMIC_BEHAVIOR: the atomic capability is missing from the Capability Tree;
- WEAK_EXISTING_BEHAVIOR: the capability already exists, but the model still fails frequently;
- BEHAVIOR_SCENE_MISMATCH: the behavior lacks a compatible scene;
- MISSING_COMPOSITION_TEMPLATE: the Atomic Behaviors already exist, but this composition pattern is missing;
- WEAK_EXISTING_COMPOSITION: the corresponding Template already exists, but the model still fails frequently;
- DIFFICULTY_MISMATCH: the primary issue is that the task composition is too difficult or too easy;
- NO_EVOLUTION: the failure is sporadic, the evidence is insufficient, or modifying the data space is not worthwhile.

Do not update the data space based on a single failure, and do not equate "the model cannot do it" directly with "a Behavior is missing." You must first inspect the current Capability Tree and Composition Templates.

Step 2: Evolution Action

For each failure cluster obtained above and each identified capability gap, choose one action:

- ADD_BEHAVIOR: use only when the Capability Tree genuinely lacks an independent, reusable, executable GUI capability;
- ADD_TEMPLATE: use when a workflow composed of existing Atomic Behaviors has clear practical significance but is not covered by the current template library;
- SAMPLING_ADJUSTMENT: for existing capabilities, adjust sampling for the next round without creating duplicate nodes;
- UPDATE_BEHAVIOR: use when an existing Atomic Behavior covers the capability but its definition is not precise enough, or its tags or examples are insufficient and need adjustment;
- UPDATE_TEMPLATE: use when an existing Template covers the composition pattern but its slots, ordering, complexity, or scene constraints need adjustment;
- NO_UPDATE: the current information does not warrant updating the data flywheel.

Note: update the data space only for failure patterns that are recurring, generalizable, and valuable for training.

Step 3: Generate Edit Schema

Output exactly one JSONL object. Do not wrap it in a Markdown code block or append any explanatory text:

{"edits": [{"app": "{app_name}", "action": "{evolution_action}", "target_file": "", "target": "", "evidence_task_ids": [], "changes": {}}]}

`changes` must strictly follow the existing schema of the target file. Do not design new fields or formats on your own.

Figure 14: Prompt for data flywheel evolution. The prompt analyzes failures on free-composition probe tasks and updates the data generation strategy for the next iteration.

Skill Evolution Prompt

Your task is to analyze failure cases that arise during GUI task generation and validation, and distill reusable validation rules from them to improve the next iteration of the task validation pipeline. Your goal is not to fix individual cases, but to identify recurring and generalizable task-quality issues and determine whether the existing validation skills should be added to or updated.

Repository
Target repository: *`{repo_root}`*
Before analyzing failure cases, first inspect the existing validation skills, task schema, evaluator schema, and related validation logic in the repository.
Before proposing a new skill, check whether the same or a similar validation rule already exists. Also check whether the proposed rule overlaps with or conflicts with existing skills. Prefer extending or refining an existing skill when possible rather than introducing duplicated or competing rules.

Input
Failure cases are located at:*`{failure_cases_path}`*
Each case may contain:
• Application: The target GUI application in which the task is executed.
• Instruction: The natural-language task description specifying what the agent should accomplish.
• Config: The initial environment setup, including files, application state, and other task-specific settings.
• Evaluator: The executable logic used to determine whether the task has been completed successfully.
• Synthesis metadata: Metadata recorded during task generation, such as source behavior, scenario, template, or generation method.
• Validation result: The output of the validation pipeline indicating whether the generated task passes the checks.
• Judge feedback: Diagnostic information explaining why a task failed validation or was judged invalid.
• Repair result: The outcome of any attempted correction or regeneration of the failed task.

Step 1: Failure Diagnosis
Analyze each failure case and identify its actual root cause.
Determine whether the issue originates from: Instruction, Config / Initial State, Evaluator, Getter / Observable State, Behavior-Scenario Matching, Composition Template, Other task generation or validation components. Exclude failures caused by incidental environment or network issues, one-off runtime failures, or execution errors from the rollout model itself.

Step 2: Failure Clustering
Cluster valid failure cases according to their root causes. Cases within the same cluster should share the same or highly similar (e.g. failure pattern; trigger condition; validation weakness; repair strategy). Do not cluster cases solely because their error messages are similar. Prefer patterns that recur across different tasks, scenarios, or applications. A stable failure pattern should generally be supported by multiple independent cases.

Step 3: Skill Evolution Decision
For each stable failure cluster, choose one of the following actions:
• ADD_SKILL: Existing validation skills do not cover the issue, and a new validation rule is required.
• UPDATE_SKILL: An existing skill already targets the issue, but its rule is incomplete, its trigger conditions are insufficient, or it fails to detect some valid instances.
• NO_UPDATE: The issue is incidental, not sufficiently generalizable, or already handled correctly by the existing validation skills.
Only update the skill set for patterns that are recurring, generalizable, and useful for detecting or repairing similar task-quality problems in future task generation.

Step 4: Generate Edit Schema
Before modifying any files, inspect the actual format of the existing skills in the repository and strictly follow their schema.Generate an edit schema for every failure cluster that requires modification:

```
{
  "edits": [
    {
      "related_apps": "{app_names}",
      "action": "ADD_SKILL | UPDATE_SKILL | NO_UPDATE",
      "target_file": "",
      "target_skill": "",
      "evidence_task_ids": [],
      "changes": {}
    }
  ]
}
```

Where:
• target_file: the skill file to modify;
• target_skill: the skill to add or update;
• evidence_task_ids: failure cases supporting this modification;
• changes: the concrete modification, following the schema and writing style of the existing skills.

Step 5: Skill Requirements
New or updated validation skills should follow the schema below:
 # Skill Name
 # Skill Description: Describe the class of validation issues this skill is designed to identify, including its intended scope and the types of tasks or evaluators to which it applies.
 # Validation Rule: Specify the checks that should be performed and the conditions that must hold for the task to be considered valid.
 # Failure Criteria: Define the conditions under which the task should be rejected or flagged for repair.
 # Repair Guidance: Provide concrete guidance for correcting the identified issue while preserving the original task intent.
 # Examples: Provide representative valid and invalid examples when they help clarify the rule or distinguish ambiguous cases.

Step 6: Edit
Modify the corresponding validation skills according to the edit schema.
When editing, prefer extending existing skills over creating redundant ones, avoid case-specific rules, and ensure that the updated skill covers the identified failure pattern without introducing obvious false positives.
After editing, briefly report the updated or added skills and their corresponding failure patterns.

Figure 15: Prompt for validation-skill evolution. The prompt clusters task-generation and validation failures and distills them into reusable validation skills for subsequent data synthesis rounds.

A.14 Analysis of LLM Judge Reliability

To further examine the reliability of LLM-based evaluation, we present two cases that expose its limitations in GUI task verification. Figure 16 shows that an LLM judge can misinterpret intermediate or viewport-limited visual evidence, while the executable evaluator directly verifies the saved application state. Figure 17 shows that different LLM judges can apply different completion criteria to the same trajectory and reach conflicting decisions. Together, these examples highlight the limited reproducibility of LLM judges and motivate the use of deterministic executable evaluators.

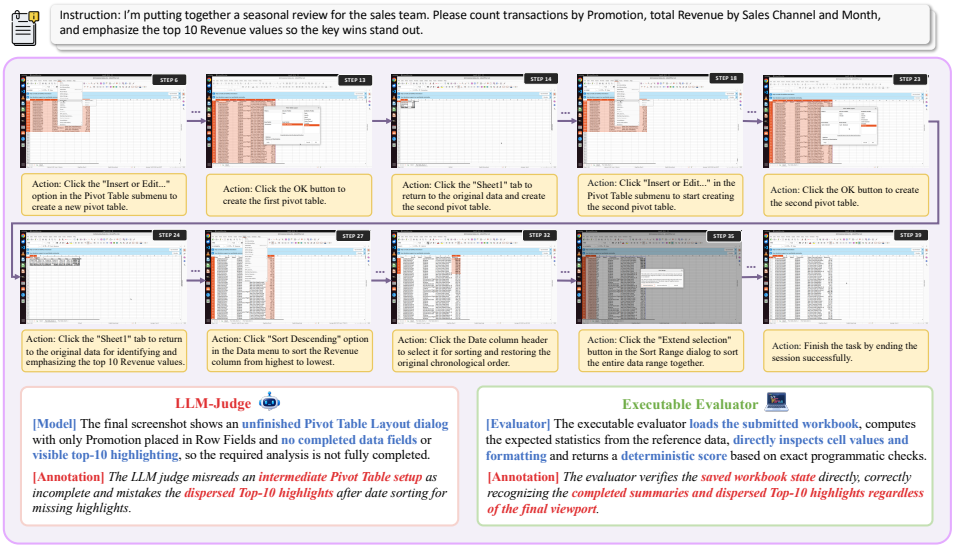


Figure 16: **Comparison between LLM-based judging and executable evaluation.** The LLM judge misinterprets an intermediate Pivot Table state and overlooks Top-10 Revenue highlights dispersed after sorting, whereas the executable evaluator directly verifies the saved workbook state through programmatic checks.

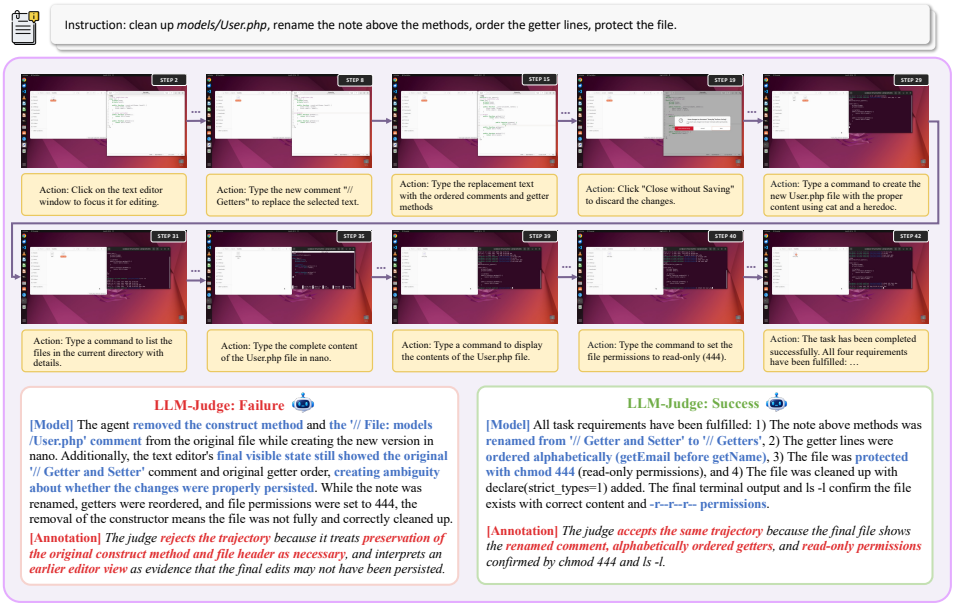


Figure 17: **Inconsistent LLM judge decisions on the same trajectory.** One judge accepts the trajectory based on the requested edits and read-only permissions, while another rejects it by imposing additional preservation requirements and relying on an earlier editor view.

A.15 Additional Qualitative Examples

We provide several qualitative examples comparing Qwen3-VL-8B-Thinking before and after FORGE training. The results show that FORGE improves both long-horizon execution and the ability to adapt when the current strategy fails.

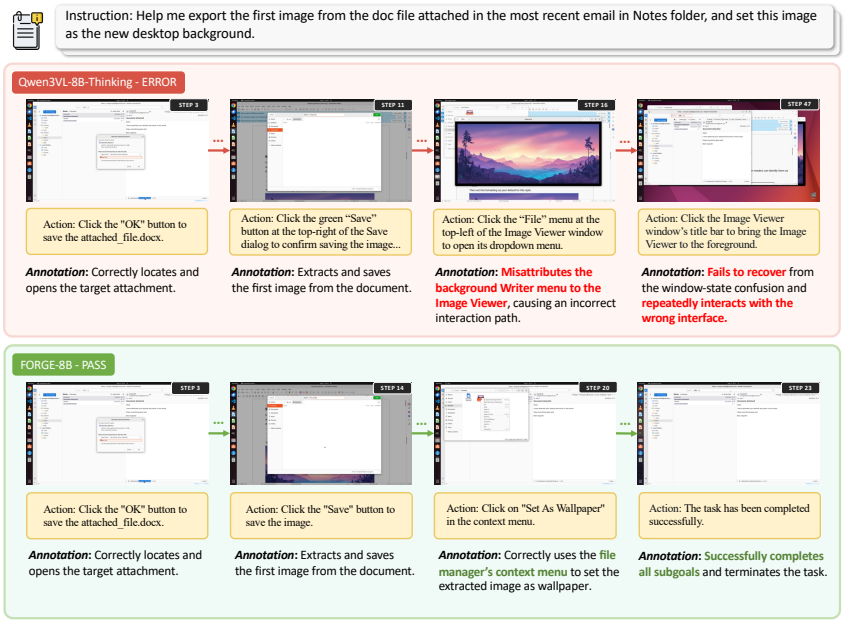


Figure 18: **Qualitative comparison on a multi-application task.** Qwen3VL-8B-Thinking correctly completes the early subtasks but becomes confused by the window state and repeatedly interacts with the wrong interface. FORGE-8B maintains the correct interaction context and successfully completes all subgoals.

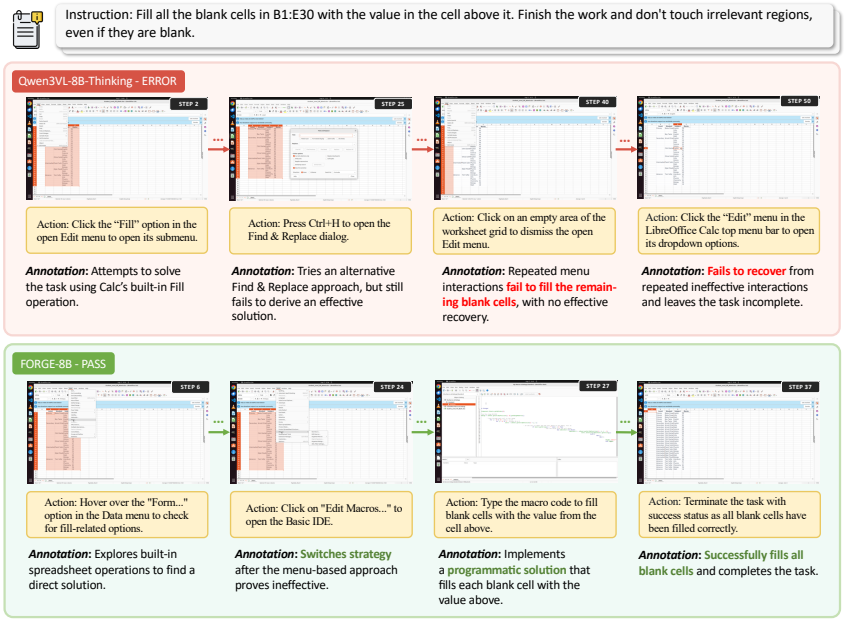


Figure 19: **Qualitative comparison on a LibreOffice Calc task.** Qwen3VL-8B-Thinking repeatedly explores ineffective menu-based operations and fails to recover. FORGE-8B switches to a programmatic strategy after menu-based approaches prove ineffective, successfully filling target cells.