VISTA: VERIFIABLE SELF-CURRICULUM FOR TRAINING GUI AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Training GUI agents requires scalable supervision for long-horizon interactions in stateful environments, where task success is often difficult to specify and verify. Existing approaches rely on human annotations, large teacher models, or manually specified evaluators, limiting the scalability of training. We introduce **VISTA**, a lightweight framework for verifiable self-curriculum learning that enables GUI agents to improve through tasks they generate and verify themselves. Guided by its current capabilities, the agent synthesizes tasks, environment configurations, and executable evaluators. Task–evaluator pairs that pass execution-based verification provide supervision for reinforcement learning, and the updated agent guides the next round of task synthesis. This iterative process creates a curriculum that evolves with the agent. Our models, **VISTA-8B** and **VISTA-9B**, post-trained on Qwen3-VL-8B and Qwen3.5-9B, respectively, achieve task success rate gains of up to 4.7% on OSWorld-Verified and 4.3% on Windows Agent Arena over their respective base models. The resulting agents also improve on OSWorld-G and ScreenSpot-Pro without grounding-specific supervision and require fewer interaction steps on average. These results demonstrate the effectiveness of self-generated, execution-verified curricula for improving GUI agents.

1 INTRODUCTION

Computer-use/GUI agents (Xie et al., 2025b; Qin et al., 2025a; Yang et al., 2025b; Song et al., 2025; Agashe et al., 2025a;b) aim to autonomously operate within general-purpose computing environments by parsing visual input, understanding user intent, and executing actions through interfaces such as the mouse, keyboard, etc. Recent progress in multimodal foundation models has significantly improved the perception and reasoning abilities of such agents, yet their development remains bottlenecked by supervision as training a capable agent requires multi-turn interaction traces and reliable feedback about whether the final environment state satisfies the intended goal.

This supervision is difficult to obtain at scale. Meaningful computer-use tasks are stateful and tend to occur over long-horizons: a correct solution may require dozens of actions, intermediate observations may change dynamically, and success can often depend on fine-grained visual or structural properties of the environment. As a result, existing approaches commonly rely on human-in-the-loop data collection (Xie et al., 2025a; Wang et al., 2025; Nayak et al., 2025b) through methods like recording demonstrations, annotating trajectories, or manually checking outcomes. These pipelines are expensive, slow to scale, and difficult to keep up-to-date as software environments evolve. Some recent works aim to reduce human effort by using stronger teacher models as task evaluators (Cao et al., 2026; Wu et al., 2026; Zhang et al., 2026; Fan et al., 2026) or trajectory filters (Tang et al., 2025; Sun et al., 2025). However, these alternatives introduce their own limitations: large teacher models increase inference cost and make the training pipeline dependent on external model capabilities. This paradigm also restricts the VLM from evolving as their capabilities would be bound by the teacher model's.

To address these limitations, we propose **VISTA**, a simple, lightweight framework that aims to bootstrap a computer-use agent through a single, smaller VLM. Instead of relying on human annotators and large teacher models, **VISTA** uses a single small VLM as a self-contained synthesis engine. The model simultaneously generates task instructions and writes executable setup and evaluator code, while also attempting said task and validating whether the task, setup functions and

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.

Table 1: Comparison between **VISTA** and recent data synthesis frameworks. "●" indicates represents full support; "◐" represents partial support; "○" represents no support.

Method	Setting	Verifiable Tasks	Joint Task-Evaluator Synthesis	Reusable / Evolving Evaluators	No Need for External Teacher Model
GUI-GENESIS (Cao et al., 2026)	Web	●	●	◐	○
AutoWebWorld (Wu et al., 2026)	Web	●	◐	○	○
InfiniteWeb (Zhang et al., 2026)	Web	●	◐	◐	○
WebFactory (Fan et al., 2026)	Web	●	●	◐	○
ZeroGUI (Yang et al., 2025a)	Desktop	●	◐	○	○
EvoCUA (Xue et al., 2026)	Desktop	●	●	◐	◐
SEA (Tang et al., 2025)	Desktop	●	◐	○	○
SEAgent (Sun et al., 2025)	Desktop	◐	○	◐	○
VISTA (ours)	Desktop	●	●	●	●

evaluators are executable and relevant. Verified tasks and evaluators are stored in an evolving pool and reused as context for future synthesis rounds to further optimize subsequent rounds.

The central insight behind **VISTA** is that small VLMs may already possess complementary abilities that can form a scalable supervision loop: multimodal understanding of environment states, code generation to create executable checkers, and action prediction for attempting tasks. When properly grounded, the model itself can produce reliable training examples from self-contained and self-driven synthesis, turning open-ended data synthesis into an iterative bootstrapping process.

Our contributions can be summarized as follows:

- **Self-contained data synthesis for computer-use agents.** We introduce **VISTA**, a framework that uses a single small VLM to generate computer-use tasks, environment setup functions, and executable evaluators. Candidate examples are filtered through execution-based validation, reducing reliance on human annotation or stronger teacher models.
- **Co-evolving task and verifier generation. VISTA** jointly synthesizes tasks and their executable evaluators and can reuse verified task-evaluator pairs for future iterations. This iterative reuse allows both the task distribution and verification logic to expand over time. With **VISTA**, we synthesize 3.8K verifiable multi-turn agentic tasks.
- **Post-training with verifiable synthetic supervision.** We post-train Qwen3-VL-8B and Qwen3.5-9B on **VISTA**'s data to produce **VISTA-8B** and **VISTA-9B**. The models perform competitively against, and at times outperforms, other specialized computer-use agentic models of both similar and larger sizes on OSWorld and also show improvement over another online environment: WindowsAgentArena-V2. The results show that it also outperforms scale-matched training of their base forms on human-annotated data such as AgentNet. We also observe improvements on OSWorld-G and ScreenSpot-Pro despite **VISTA**'s focus on multi-turn agentic tasks rather than UI grounding specifically.

2 Related work

Data Curation for GUI Trajectories. Data curation for GUI agents is uniquely challenging. Datasets that offer fine-grained supervision for perception and grounding typically require substantial human annotation, making them expensive and difficult to scale. For desktop environments, Kapoor et al. (2024) introduces annotated multimodal GUI tasks with executable scripts derived from human demonstrations. Other datasets, such as UI-Vision (Nayak et al., 2025b), WinClick Hui et al. (2025), and AgentNet (Wang et al., 2025), provide densely labeled GUI elements and interaction trajectories for fine-grained perception–action evaluation. Similarly, ScreenSpot Cheng et al. (2024) and ScreenSpot-Pro Li et al. (2025a) focus on grounding evaluation in desktop settings with dense layouts and small interactive targets. Similarly, other mobile and desktop data efforts such as AndroidControl (Li et al., 2024), OmniACT(Kapoor et al., 2024), UI-Vision (Nayak et al., 2025a), etc. show the utility of scaling human demonstrations and annotations, but static trajectory matching can penalize alternative valid strategies and is expensive to collect. Overall, because these datasets are largely built with human-in-the-loop supervision, their construction effort limits scalability.

Data Curation for Verifiable GUI Tasks. Fully interactive computing environments also offer potential for data generation but can be slow due to their complexity and focus on specific environments. Early examples like MiniWoB++ (Liu et al., 2018) use human demonstrations and executable rewards, while later ones like Mind2Web (Deng et al., 2023) expanded to real sites and multi-turn navigation. Others like WorkArena (Drouin et al., 2024) further standardize browser-agent evaluation on knowledge-work tasks with multimodality and reusable action APIs. Execution-based benchmarks such as AndroidWorld (Rawles et al., 2024), OSWorld (Xie et al., 2025b), and Windows Agent Arena Bonatti et al. (2025) address this by initializing real applications and checking final environment state, but their evaluators are generally manually specified rather than synthesized and evolved by the agent itself. More recent web and GUI agents extend the idea of improving through autonomous synthesis of trajectories (Tang et al., 2025; Sun et al., 2025; Yang et al., 2025a), experience (Xue et al., 2026), and environments (Cao et al., 2026; Wu et al., 2026; Fan et al., 2026; Bai et al., 2026). **VISTA** differs by targeting general GUI tasks and by closing the loop not only over trajectories but also over reusable executable evaluators, using a single small VLM rather than external teacher judges or fixed hand-written checkers.

3 METHODOLOGY

3.1 PRELIMINARIES & SETUP

The agent's behavior can be formalized as a partially observable Markov decision process (POMDP) $(S, \mathcal{O}, \mathcal{A}, T, \mathcal{R})$ with state space S, observation space $\mathcal{O}$, action space $\mathcal{A}$, transition function $T : S \times \mathcal{A} \to S$, and reward function $\mathcal{R} : S \times \mathcal{A} \to \mathbb{R}$. Given the current observation $o_t \in \mathcal{O}$, an agent generates an executable action $a_t \in \mathcal{A}$, resulting in a new state $s_{t+1} \in S$ and a new partial observation $o_{t+1} \in \mathcal{O}$. The process repeats until the agent generates the termination action (i.e., DONE or FAIL) or when the maximum number of steps is reached ($t > t_{\text{max}}$). The evaluator or reward function $\mathcal{R} : S \times \mathcal{A} \to [0, 1]$ returns the value at the final step to check whether the agent achieves the task objective.

Table 2: Action space for GUI agents.

Category	Action	Description
Mouse	`single_click(x, y)`	Left-click at `(x, y)`
	`double_click(x, y)`	Double-click at `(x, y)`
	`right_click(x, y)`	Right-click at `(x, y)`
	`middle_click(x, y)`	Middle-click at `(x, y)`
	`triple_click(x, y)`	Triple-click at `(x, y)`
	`mouse_move(x, y)`	Move the cursor to `(x, y)`
	`left_click_drag(x, y)`	Drag with left button held to `(x, y)`
	`scroll(pixel)`	Vertical scroll by `pixel`
	`hscroll(pixel)`	Horizontal scroll by `pixel`
Keyboard	`type(text)`	Type string into the focused input field
	`key(key_str)`	Press special key or shortcut
Control	`wait(time)`	Pause for `time` seconds
	`done`	Terminate episode and indicate success
	`fail`	Terminate episode and indicate failure

Observation Space. The agent's observation space $\mathcal{O}$ specifies the information it can access at each step. In general, an observation includes (i) the task instruction and (ii) a representation of the on-screen state either as raw pixels (e.g., an RGB screenshot) and/or as structured interface descriptions (e.g., accessibility tree) with signals like text labels, bounding boxes, and element status that help the agent localize and interpret actionable UI elements. In this work, we focus on desktop GUI computer-use agents and consider screenshot-only observation space.

Action Space. Similar to previous works (Qin et al., 2025a; Wang et al., 2025; Yang et al., 2025b; Mei et al., 2025; Li et al., 2025b; Ye et al., 2025), we use keyboard, mouse movements and some control actions as the action space (see Table 2).

3.2 OVERVIEW OF **VISTA**

As illustrated in Figure 1, **VISTA** uses a single small VLM to simultaneously play three roles in our environment: a task creator, a verifier, and an actor policy. The task creator proposes computer-use tasks and executable checking logic, the verifier filters these candidates through environment execution, and the actor policy is optimized on the verified examples. This forms a closed loop

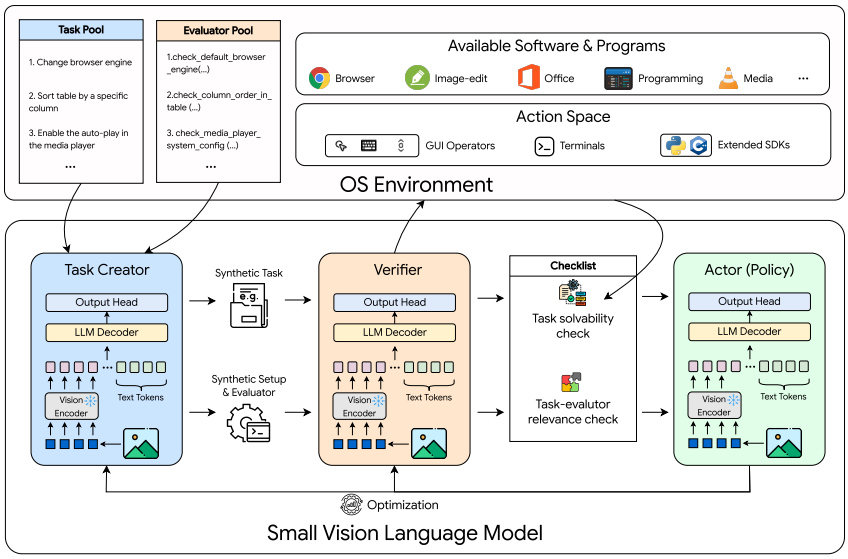


Figure 1: Overview of **VISTA**. A single small vision-language model is instantiated for three different roles: a task creator, a verifier, and an actor policy. The task creator proposes tasks and corresponding setup/evaluator code using examples and evaluators from existing pools. The verifier executes the generated setup and interacts with the env to check both task feasibility and task-evaluator relevance. Verified examples will be then used to optimize the policy of the actor.

where accepted tasks and reusable verifiers are continuously added back into the synthesis pools, allowing the supervision source to expand over time.

3.3 Data Synthesis and Verification

Task Creator. The task creator generates two items: an executable setup script and evaluator. Table 4 shows some examples of synthetic tasks along with setups and evaluators. The setup script prepares the initial environment state for the task, or placing the system into a particular UI state. The evaluator specifies how task completion should be checked. The task creator retrieves related examples from the task pool and reusable checking functions from the verifier pool to propose a new computer-use task. To avoid generating near-duplicate tasks, **VISTA** uses a vector store to store the previous generation memory to cover a broader range of aspects (e.g., task complexity, different layouts) of different software applications. Implementation details of how we build the task creator can be found at Appendix A.1.

Verifier. As the verifier, the VLM then validates each generated task candidate through three stages. First, it analyzes the synthesized setup and evaluator scripts, checking syntax, function signatures, argument types, required fields, and API availability, filtering out tasks with malformed function calls, unsupported APIs, or invalid arguments. Implementation details can be found in Appendix A.1. The verifier next performs a semantic consistency check between the task instruction and the evaluator, ensuring that the evaluator measures the intended outcome rather. Third, an initialized agent using the same VLM backbone interacts with the environment to test whether the setup and evaluator are executable in practice. Only candidates that pass all checks are accepted, after which the verified task and its evaluator are added back into the task and verifier pools.

Synthetic Data Characteristics. As shown in Figure 2, both the number of accepted tasks and the number of unique verifiers grow steadily across synthesis steps. This indicates that **VISTA** does not merely sample more trajectories from a fixed task distribution: the continued growth of unique verifiers reflects a gradual accumulation of verification logic rather than reliance on a small, static set of checkers. Figure 3 further illustrates the role of verification as quality control. De-duplication is the dominant reason for rejection, suggesting that task diversity remains the main bottleneck when generating tasks within a limited context window, while the remaining rejections, caused by erro-

Table 3: Analysis of the synthesized data. **(a)** Instruction-level semantic distance from AgentNet and **VISTA** to OSWorld, computed with Qwen3-Embedding-0.6B's embeddings (lower means more similar). Cos.: embedding-centroid cosine distance; MMD: RBF maximum mean discrepancy; FED: Fréchet embedding distance after PCA to 50 dimensions; Ratio: **VISTA** / AgentNet. **(b)** Human audit of the automatic filtering decisions on 200 synthesized candidates sampled across all 10 domains, with reference labels determined by majority vote among three experts. FP: accepted automatically but rejected by humans; FN: rejected automatically but accepted by humans.

(a) Distance to OSWorld (↓)

Metric	AgentNet	**VISTA**	Ratio
Cos.	0.1454	0.3150	2.17×
MMD	0.0170	0.0281	1.65×
FED	29.93	46.49	1.55×

(b) Human audit

Outcome	Count	%
Agreement	184	92
FP	12	6
FN	4	2

Table 4: A simplified and stylistic example of our synthetic tasks and evaluators. The task reward is evaluated based on the final state of the file/software after agent execution.

Initial State	Task Instruction	Success Criterion	Evaluator

	"Add a bookmark for the Wikipedia homepage titled 'Wikipedia' to the 'Favorites Bar' so it appears immediately below the address bar."	The Favorites Bar contains a bookmark named 'Wikipedia' whose URL is `www.wikipedia.org/`.	`bookmarks = get_bookmarks(env)`
			`rule = {`
			`  type: bookmark_bar_urls,`
			`  url:  wikipedia.org,`
			`  name:  Wikipedia`
			`}`
			`is_expect_bkmk(bookmarks, rule)`

neous setup or evaluator code, show how **VISTA** filters out invalid tasks. As we are synthesizing within the Linux environment, to avoid overfitting to task distribution of the eval set which uses the same environment (e.g., OSWorld), we compare the instruction distributions **VISTA** against it and another widely-used dataset for training GUI agents: AgentNet (Wang et al., 2025). As shown in Table 3, under all three metrics calculated over sentence embeddings (using Qwen3-Embedding-0.6B), **VISTA** is consistently *farther* from OSWorld than AgentNet, by a factor of 1.55× to 2.17×, suggesting **VISTA**'s synthesized data is less instruction-level similar to OSWorld; Besides, to further validate the accept/reject decisions made by **VISTA**'s automatic pipeline, we further validate its decisions with human annotation. We sample 200 synthesized candidates across all 10 domains and ask three human experts to independently audit each one along the same validity dimensions targeted by the automatic pipeline: task feasibility, setup correctness, task–evaluator relevance, evaluator completeness, and the false-positive and false-negative behavior of the evaluator. The reference label for each case is determined by majority vote. As shown in Table 3, the automatic decisions agree with human judgments on 92% of the cases. Most of the remaining disagreements are false positives (6%), i.e., a small fraction of invalid tasks may still pass the automatic filter, while false negatives account for 2% of the audited candidates. These results indicate that the automatic pipeline provides reliable quality control. Table 4 shows one minimal example of our synthetic tasks and evaluators, more examples and complete data synthesis and verification statistics are in Appendix A.2.

3.4 Optimization

We post-train the VLMs using Group Relative Policy Optimization (GRPO) (Guo et al., 2025). For each task instruction g, we sample a group of G rollouts $\{\tau_i\}_{i=1}^{G}$ from the old policy π_{old} in the VM environment. Each rollout is a multi-step GUI trajectory $\tau_i = \{(o_{i,t}, u_{i,t})\}_{t=1}^{T_i}$, where $o_{i,t}$ is the screenshot observation at step t, $u_{i,t}$ is the generated GUI action, and T_i is the trajectory length. The episode terminates when the agent outputs `DONE`/`FAIL` or reaches the maximum step budget. After execution, the task-specific evaluators check the final environment state and assigns an accuracy reward $R_{\text{acc}}(\tau_i)$. We additionally use a format reward $R_{\text{format}}(\tau_i)$ to encourage valid action outputs. The total rollout reward is

$$R(\tau_i) = \lambda R_{\text{format}}(\tau_i) + R_{\text{acc}}(\tau_i). \tag{1}$$

Algorithm 1 Pipeline of Synthesizing Verifiable Computer-use Tasks

Require: VM environment $\mathcal{E}$; software domains $\mathcal{S}$; action space $\mathcal{U}$; task pool $\mathcal{D}$; evaluator pool $\mathcal{V}$; task memory $\mathcal{M}$; synthesis budget N

for $i = 1$ to N **do**
  $s_i \sim \mathrm{Uniform}(\mathcal{S})$ ▷ select a software domain, e.g., browser, office, programming
  $(\mathcal{D}_i, \mathcal{V}_i) \leftarrow \mathrm{Retrieve}(\mathcal{D}, \mathcal{V}; s_i)$ ▷ retrieve examples and verifiers as synthesis demonstrations
  $g_i \leftarrow p_\theta^{\mathrm{task}}(\cdot \mid s_i, \mathcal{D}_i, \mathcal{V}_i)$ ▷ task creator proposes new computer-use task
  $m_i \leftarrow \phi(g_i)$ ▷ encode task into normalized memory key or semantic embedding
  if $\mathrm{Dup}(m_i, \mathcal{M}) = 1$ **then**
    continue ▷ reject if the task is semantic duplicate of previous tasks
  end if
  $(c_i^{\mathrm{setup}}, c_i^{\mathrm{eval}}) \leftarrow p_\theta^{\mathrm{code}}(\cdot \mid g_i, s_i, \mathcal{D}_i, \mathcal{V}_i)$ ▷ generate executable environment and task evaluator
  $e_i^0 \leftarrow \mathrm{ExecSetup}(\mathrm{Reset}(\mathcal{E}, s_i), c_i^{\mathrm{setup}})$ ▷ reset VM and instantiate initial task state
  $z_i^{\mathrm{setup}} \leftarrow \mathrm{CheckExec}(c_i^{\mathrm{setup}}, e_i^0)$ ▷ verify whether is executable in env
  $z_i^{\mathrm{rel}} \leftarrow \mathrm{CheckRelevant}(g_i, c_i^{\mathrm{eval}})$ ▷ verify whether evaluator measures intended task goal
  $v_i(e, \tau) \leftarrow \mathrm{ExecEval}(e, \tau, c_i^{\mathrm{eval}})$ ▷ wrap verified evaluator as reusable for future iteration
  if $z_i^{\mathrm{setup}} = 1 \wedge z_i^{\mathrm{solve}} = 1 \wedge z_i^{\mathrm{rel}} = 1$ **then**
    $x_i \leftarrow (s_i, g_i, c_i^{\mathrm{setup}}, c_i^{\mathrm{eval}}, v_i)$ ▷ construct accepted verifiable task instance
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{x_i\}, \quad \mathcal{V} \leftarrow \mathcal{V} \cup \{v_i\}$ ▷ feed verified task and evaluator back into pools
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{m_i\}$ ▷ record task to memory to prevent future duplicates
  else
    continue ▷ reject invalid, unexecutable, or evaluator-mismatched synthesis
  end if
end for
return $\mathcal{D}, \mathcal{V}, \mathcal{M}$

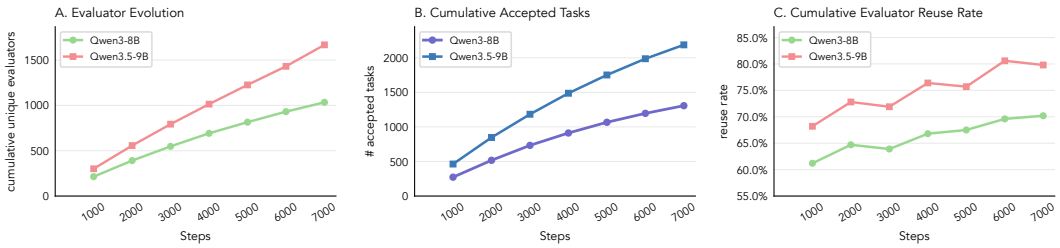


Figure 2: Synthesis dynamics of **VISTA**. Across both backbones, accepted tasks and unique evaluators grow together, while verifier reuse increases over time.

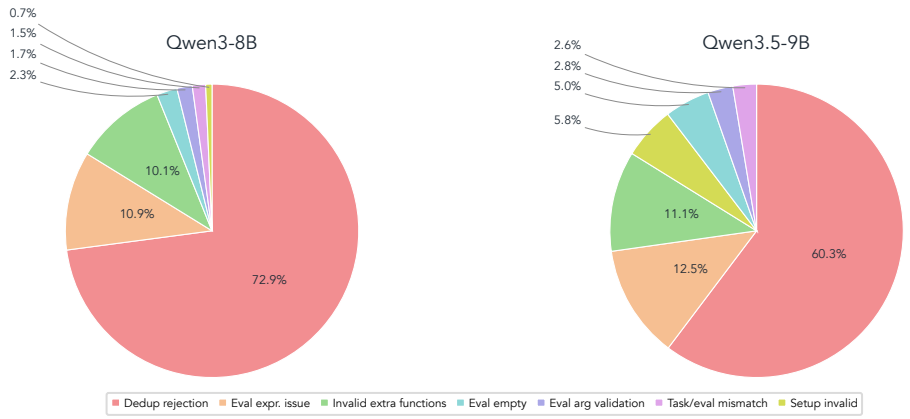


Figure 3: Distribution of reasons behind of task rejections during the data synthesis and verification process of **VISTA**.

For the group of rollouts sampled for the same task g, we compute the group-normalized advantage $\hat{A}_i = \frac{R(\tau_i)-\mu}{\sigma}$, where $\mu = \frac{1}{G}\sum_{j=1}^{G} R(\tau_j)$, and $\sigma = \sqrt{\frac{1}{G}\sum_{j=1}^{G}\big(R(\tau_j)-\mu\big)^2}$.

During rollout, computer-use agents produce multi-step GUI trajectories, we assign the trajectory-level advantage $\hat{A}_i$ to every action token generated in the rollout. Let $u_{i,t} = (u_{i,t}^1, \ldots, u_{i,t}^{L_{i,t}})$ denote the token sequence for the action emitted at step t. The GRPO objective is

$$J_{\mathrm{GRPO}}(\theta) = \frac{1}{G}\sum_{i=1}^{G}\frac{1}{\sum_{t=1}^{T_i} L_{i,t}}\sum_{t=1}^{T_i}\sum_{\ell=1}^{L_{i,t}}\min\left[\rho_{i,t,\ell}(\theta)\hat{A}_i,\ \mathrm{clip}\big(\rho_{i,t,\ell}(\theta), 1-\epsilon, 1+\epsilon\big)\hat{A}_i\right],$$

where

$$\rho_{i,t,\ell}(\theta) = \frac{\pi_\theta\big(u_{i,t}^{\ell} \mid g, o_{i,\le t}, u_{i,<t}, u_{i,t}^{<\ell}\big)}{\pi_{\mathrm{old}}\big(u_{i,t}^{\ell} \mid g, o_{i,\le t}, u_{i,<t}, u_{i,t}^{<\ell}\big)}.$$

The normalization by $\sum_{t=1}^{T_i} L_{i,t}$ prevents longer GUI trajectories from dominating the objective purely due to having more generated tokens. For each optimization step in post-training RL, we sample 8 prompts per batch and generate 16 rollouts per prompt. Each rollout conditions on observation tokens from up to the most recent three screenshots and all the action tokens, the observation tokens in other turns will be masked. The maximum generated response length is set as 1024 tokens. Additional details are provided in Appendix A.4.

4 Experiments

We post-train Qwen3-VL-8B-Instruct (Bai et al., 2025) and Qwen3.5-9B (Team, 2026) on our synthetic data. We evaluate on both task performance (success rate) and task completion efficiency (steps to success) of the models trained on **VISTA**-synthesized data (**VISTA-8B** and **VISTA-9B**, respectively) on online evaluation benchmarks (OSWorld-verified (Xie et al., 2025b), WindowsAgentArena-V2 (He et al., 2025)), and GUI grounding benchmarks (Xie et al., 2025a; Li et al., 2025a). For OSWorld-verified, our evaluation is conducted on the Ubuntu platform with screenshot-only mode. The screen size is 1920×1080 and the maximum number of steps is 50. More details of the benchmarks can be found at Appendix A.3. We also consider multiple open-sourced VLMs (similar and larger sizes) that are specifically optimized in computer-use scenarios, including UI-TARS (72B-DPO, 1.5-7B) (Qin et al., 2025a), OpenCUA (7B, 32B) (Wang et al., 2025), Scale-CUA (7B, 32B) (Liu et al., 2025), GUI-OWL (7B, 32B) (Ye et al., 2025) and DART-GUI (7B) (Li et al., 2025b) in our evaluation.

4.1 Overall Comparison

On OSWorld-Verified (Table 5), **VISTA** consistently improves both backbones over their base models. **VISTA-8B** raises the overall success rate of Qwen3-VL-8B-Instruct from 30.98% to 35.43% (+4.45), and **VISTA-9B** raises that of Qwen3.5-9B from 40.95% to 45.68% (+4.73), achieving the best overall result among all compared models and surpassing the strongest specialized baseline, DART-GUI-7B (42.13%). Notably, neither model regresses in any domain. The largest gains appear in domains that *require manipulating persistent software state or navigating application-specific functions*, such as Thunderbird (+13.33/+13.34), Writer (+8.70 for both), OS (+8.33/+8.34), and VSCode (+21.74 for **VISTA-9B**). These domains benefit most because the reward can be grounded in whether the final file or system state satisfies the intended goal. In contrast, Multi-app performance remains below 20% for both models, suggesting that long-horizon workflows spanning multiple applications remain challenging and are a promising direction for future task synthesis. The improvements also transfer to WAA-V2 (Table 6). Both **VISTA-8B** and **VISTA-9B** improve their base models by 4.25 points overall (8.04%→12.29% and 19.86%→24.11%, respectively), again without regression in any domain, and **VISTA-9B** achieves the best result in every domain (tied with Qwen3.5-9B on Utils) and the largest gain appears on Chrome (+11.76 for both models).

Results on grounding benchmarks (Table 7) are more nuanced. On OSWorld-G, **VISTA-9B** improves Qwen3.5-9B from 63.10% to 64.30%, with gains concentrated in Element Recognition and Layout Understanding, despite **VISTA**'s focus on multi-turn agentic tasks rather than UI grounding specifically. This aligns with the design of **VISTA**: many synthesized tasks require the model to explore software interfaces, identify actionable elements, and reason over layout structure before acting. Meanwhile, Text Matching and Fine-grained Manipulation slightly decrease, suggesting that action-centric synthetic data may improve structural grounding more than exact text matching or

Table 5: Performance comparison on OSWorld-Verified (Xie et al., 2025b) over 361 tasks (excluding Google Drive tasks). Each task is scored by its evaluator with a discrete value in $[0, 1]$; domain results are mean task scores (%), and Overall is averaged over all 361 tasks. All models use a maximum of 50 steps. Results for Qwen3-VL-8B-Instruct, Qwen3.5-9B, and our models are averaged over three independent runs; all other results are taken from the official leaderboard. **Bold** and underlined values denote the best and second-best results (ties share formatting). Δ: absolute gain over the corresponding base model.

Model	Chrome	GIMP	Calc	Impress	Writer	Multi_apps	OS	Thunderbird	VLC	VSCode	Overall
UI-TARS-1.5-7B (Qin et al., 2025a)	32.79	53.85	8.51	39.31	41.30	8.60	25.54	46.67	24.41	52.17	27.25
UI-TARS-72B-DPO (Qin et al., 2025a)	33.24	61.54	12.77	25.45	43.48	6.71	33.33	33.33	23.53	47.83	25.88
OpenCUA-7B (Wang et al., 2025)	38.61	43.59	13.22	32.60	33.33	12.11	43.47	42.22	28.31	47.10	28.20
OpenCUA-32B (Wang et al., 2025)	43.39	69.23	13.48	38.28	40.58	14.93	53.62	53.33	25.49	55.07	34.14
ScaleCUA-7B (Liu et al., 2025)	15.50	30.80	6.40	10.60	13.00	11.80	25.00	20.00	11.80	26.10	15.00
ScaleCUA-32B (Liu et al., 2025)	19.30	34.60	10.60	14.90	17.40	12.90	29.20	20.00	17.60	21.70	17.70
GUI-Owl-7B (Ye et al., 2025)	41.22	65.38	17.02	19.06	52.17	9.68	50.00	**66.67**	29.41	65.22	32.11
GUI-Owl-32B (Ye et al., 2025)	45.90	65.40	27.70	44.70	52.20	15.10	58.30	60.00	35.30	39.10	37.70
DART-GUI-7B (Li et al., 2025b)	52.09	**76.92**	19.15	48.80	60.86	16.69	62.50	60.00	**39.30**	69.57	42.13
Qwen3-VL-8B-Instruct (Bai et al., 2025)	36.78	50.00	25.53	36.00	39.13	12.90	41.67	40.00	23.53	52.17	30.98
Qwen3.5-9B (Team, 2026)	52.00	69.23	31.91	48.77	56.52	16.13	58.33	53.33	35.29	52.17	40.95
VISTA-8B	41.22	57.69	29.79	40.34	47.83	13.98	50.00	53.33	29.41	52.17	35.43
Δ (vs. Qwen3-VL-8B)	↑4.44	↑7.69	↑4.26	↑4.34	↑8.70	↑1.08	↑8.33	↑13.33	↑5.88	→0.00	↑4.45
VISTA-9B	**54.26**	69.23	**34.04**	**50.98**	**65.22**	**19.35**	**66.67**	**66.67**	35.29	**73.91**	**45.68**
Δ (vs. Qwen3.5-9B)	↑2.26	→0.00	↑2.13	↑2.21	↑8.70	↑3.22	↑8.34	↑13.34	→0.00	↑21.74	↑4.73

Table 6: Performance comparison on WindowsAgentArena-V2 (He et al., 2025) over 141 tasks. EEach task is scored by its evaluator with a value in $[0, 1]$; domain results are mean task scores (%), and Overall is averaged over all 141 tasks. All models use a maximum of XX steps. All results are averaged over three independent runs. **Bold** and underlined values denote the best and second-best results (ties share formatting). Δ: absolute gain over the corresponding base model.

Model	LibreOffice	Chrome	Edge	System	VSCode	VLC	Utils	Overall
Qwen3-VL-8B-Instruct (Bai et al., 2025)	0.00	15.69	10.26	15.28	8.77	7.14	8.33	8.04
Qwen3.5-9B (Team, 2026)	2.38	41.18	20.51	36.11	26.32	9.52	**19.44**	19.86
VISTA-8B	2.38	27.45	12.82	19.44	17.54	7.14	8.33	12.29
Δ (vs. Qwen3-VL-8B)	↑2.38	↑11.76	↑2.56	↑4.16	↑8.77	→0.00	→0.00	↑4.25
VISTA-9B	**4.76**	**52.94**	**28.21**	**40.28**	**29.82**	**11.90**	**19.44**	**24.11**
Δ (vs. Qwen3.5-9B)	↑2.38	↑11.76	↑7.70	↑4.17	↑3.50	↑2.38	→0.00	↑4.25

low-level manipulation precision. **VISTA-8B** also improves overall over Qwen3-VL-8B-Instruct, mainly through better Element Recognition and Layout Understanding. ScreenSpot-Pro shows a similar transfer pattern. **VISTA** improves the overall score for both backbones, but the gains are concentrated in domains closer to the synthesized task distribution, such as Development, Office, and OS. These domains overlap more with the software used during data synthesis, including browsers, office tools, system operations, and programming environments. In contrast, CAD and Scientific show slight degradation for **VISTA**, indicating that generalization to visually specialized domains are future directions to improve upon.

4.2 Synthetic Supervision vs. Human Annotation

We further compare **VISTA**-synthesized data with human-annotated data. Human-annotated datasets such as AgentNet provide demonstration trajectories without executable verifiers, so they can only be used for supervised fine-tuning (SFT); in contrast, each **VISTA** task is paired with a verified evaluator, which turns it into a reusable training environment that directly supports RL. For each backbone, we sample a scale-matched subset of AgentNet and train on it with SFT, and compare it against the base model and the model trained on **VISTA** tasks with RL. As shown in Figure 4, AgentNet improves over the base models, but **VISTA** yields larger gains on all four benchmarks. Notably, the gap is most pronounced on the online, long-horizon benchmarks: on OSWorld-Verified and WAA-V2, **VISTA** roughly doubles the gains of AgentNet for Qwen3-VL-8B (e.g., +4.4 vs.

Table 7: Grounding performance on OSWorld-G and ScreenSpot-Pro. For OSWorld-G, columns denote Text Matching, Element Recognition, Layout Understanding, Fine-grained Manipulation, and Refusal. For ScreenSpot-Pro, we report the average score for each domain. Bold and underlined values denote the best and second-best results in each column. Δ reports the absolute improvement over the corresponding base model.

	OSWorld-G						ScreenSpot-Pro						
Model	**Text**	**Element**	**Layout**	**Fine-gr.**	**Refusal**	**Overall**	**Dev.**	**Creative**	**CAD**	**Scientific**	**Office**	**OS**	**Overall**
UI-TARS-1.5-7B	68.7	57.0	57.4	49.1	0.0	56.0	31.8	40.2	31.8	47.2	65.6	33.2	42.0
UI-TARS-72B-DPO	69.2	60.0	59.8	45.4	0.0	57.1	40.8	39.6	17.2	45.7	54.8	30.1	38.1
OpenCUA-7B	66.1	59.3	57.9	42.8	0.0	55.3	51.9	45.1	50.5	56.6	60.9	33.8	50.0
OpenCUA-32B	70.0	63.0	64.4	47.1	0.0	59.6	56.8	51.0	54.9	61.6	66.1	39.8	55.3
ScaleCUA-7B	67.8	59.4	63.0	48.6	0.0	57.8	44.1	41.3	46.7	54.3	67.4	35.2	47.9
ScaleCUA-32B	69.0	63.7	66.8	50.4	0.0	60.6	55.8	55.4	55.2	63.4	73.9	<u>53.6</u>	59.2
GUI-Owl-7B	68.3	58.4	58.8	43.6	0.0	55.9	54.5	46.1	54.1	61.0	68.7	47.9	54.9
GUI-Owl-32B	69.7	61.2	61.0	46.1	0.0	58.0	62.5	45.5	54.0	63.8	71.8	**54.6**	58.0
Qwen3-VL-8B-Instruct	66.0	56.4	58.2	47.6	0.0	54.6	55.3	50.1	54.4	60.8	66.7	39.4	54.6
Qwen3.5-9B	**72.1**	<u>66.0</u>	<u>69.3</u>	**53.0**	0.0	<u>63.1</u>	<u>63.8</u>	**57.1**	**61.6**	**68.4**	<u>74.7</u>	46.9	<u>62.2</u>
VISTA-8B	65.4	58.6	61.0	47.0	0.0	56.2	58.1	51.5	53.7	59.9	68.1	41.1	55.4
VISTA-9B	<u>71.8</u>	**68.4**	**71.2**	<u>52.6</u>	0.0	**64.3**	**65.8**	<u>56.0</u>	<u>60.9</u>	<u>67.8</u>	**77.0**	53.5	**63.5**

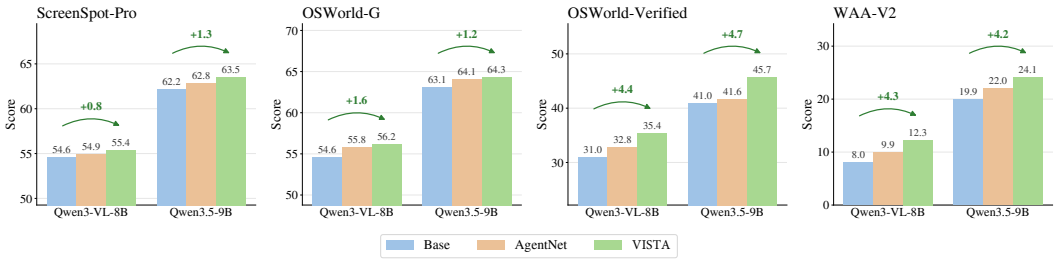


Figure 4: Comparison between post-training on human-annotated data and **VISTA**-synthesized data. **Base**: base model, **AgentNet**: base model post-trained by a scale-matched subset of human-annotated AgentNet tasks, **VISTA**: post-trained by our synthetic tasks.

+1.8 on OSWorld-Verified), whereas on the single-step grounding benchmarks the two are much closer. We attribute this to **VISTA** that can *provide on-policy supervision*. SFT imitates trajectories collected under a different policy, so the model only sees states reached by the demonstrator and never receives feedback on its own mistakes; once it deviates from the demonstrated path, errors compound over long horizons. RL on **VISTA** tasks instead lets the model learn from its own rollouts, including recovering from its own errors, and rewards any valid solution rather than a single demonstrated path. This matters little for single-step grounding, where imitation is already sufficient, but becomes critical for multi-step task execution.

5 Conclusion

We present **VISTA**, a self-contained framework for scaling supervision signals for computer-use agents with a single small vision-language model. Instead of relying on external teacher supervision (i.e., human or teacher model), **VISTA** bootstraps the capabilities of the VLM itself to synthesize computer-use tasks together with executable setup and evaluators. Experiments across OSWorld, OSWorld-G, and ScreenSpot-Pro show that this form of executable synthetic supervision improves both online task completion and grounding ability, and can outperform scale-matched human-annotated training data. Though powerful, **VISTA** still relies on interaction with realistic environments to ground the synthesis, generalization of this pipeline to scenarios where environment is inaccessible or has limited interface (such as SDKs or APIs) still remains challenging. Similarly, as noted in Section 4, certain tasks such as those that require more granular precise numerical or cell-level verification, will require extending **VISTA** to further improve generalization for computer-use agents is a promising direction for future work.

AI USE STATEMENT

In this work, we used generative AI tools to assist with code implementation and minor editing of the manuscript text. All research ideas, method design, experimental analysis, and final claims are the authors' own, and all AI-assisted content was reviewed and verified by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

6 ETHIC STATEMENT

All datasets used in this work are publicly available and contain no sensitive information. Our method applies LLM in computer-use scenarios. While stronger reasoning ability may be misused, we emphasize that this work is intended for beneficial research and responsible applications.

REPRODUCIBILITY STATEMENT

The appendix includes detailed experimental setups and hyperparameters so that others can reproduce our results. We also encourage the community to follow good research practices when using our code and data, to help maintain the reliability and transparency of future work.

REFERENCES

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent S: An Open Agentic Framework that Uses Computers Like a Human. In *International Conference on Learning Representations (ICLR)*, 2025a. URL https://arxiv.org/abs/2410.08164.

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, 2025b. URL https://arxiv.org/abs/2504.00906.

Hao Bai, Alexey Taymanov, Tong Zhang, Aviral Kumar, and Spencer Whitehead. Webgym: Scaling training environments for visual web agents with realistic tasks. *arXiv preprint arXiv:2601.02439*, 2026.

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

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Keunho Jang, and Zheng Hui. Windows agent arena: Evaluating multi-modal OS agents at scale. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 4874–4910. PMLR, July 2025.

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, Yantao Li, 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)*, 2024.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2023.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. Workarena: How capable are web agents at solving common knowledge work tasks? In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11642–11662. PMLR, 2024.

Sicheng Fan, Qingyun Shi, Shengze Xu, Shengbo Cai, Tieyong Zeng, Li Ling, Yanyi Shang, and Dehan Kong. Webfactory: Automated compression of foundational language intelligence into grounded web agents. *arXiv preprint arXiv:2603.05044*, 2026.

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

Yanheng He, Jiahe Jin, and Pengfei Liu. Efficient agent training for computer use, 2025. URL https://arxiv.org/abs/2505.13909.

Zheng Hui, Yinheng Li, Dan Zhao, Tianyi Chen, Colby Banbury, and Kazuhito Koishida. Winclick: Gui grounding with multimodal large language models, 2025. URL https://arxiv.org/abs/2503.04730.

Raghav Kapoor, Yash Parag Butala, Melisa Russak, Jing Yu Koh, Kiran Kamble, Waseem Alshikh, and Ruslan Salakhutdinov. Omniact: A dataset and benchmark for enabling multimodal generalist autonomous agents for desktop and web. *arXiv preprint arXiv:2402.17553*, 2024.

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. *arXiv preprint arXiv:2504.07981*, 2025a.

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

Wei Li, William E. Bishop, Alice Li, Christopher Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. On the effects of data scale on ui control agents. *arXiv preprint arXiv:2406.03679*, 2024. Introduces the AndroidControl dataset.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference on Learning Representations (ICLR)*, 2018. URL https://arxiv.org/abs/1802.08802.

Zhaoyang Liu, JingJing Xie, Zichen Ding, Zehao Li, Bowen Yang, Zhenyu Wu, Xuehui Wang, Qiushi Sun, Shi Liu, Weiyun Wang, et al. Scalecua: Scaling open-source computer use agents with cross-platform data. *arXiv preprint arXiv:2509.15221*, 2025.

Kai Mei, Jiang Guo, Shuaichen Chang, Mingwen Dong, Dongkyu Lee, Xing Niu, and Jiarong Jiang. R-wom: Retrieval-augmented world model for computer-use agents. *arXiv preprint arXiv:2510.11892*, 2025.

Shravan Nayak, Xiangru Jian, Kevin Qinghong Lin, Juan A. Rodriguez, Montek Kalsi, Rabiul Awal, Nicolas Chapados, M. Tamer Özsu, Aishwarya Agrawal, David Vazquez, Christopher Pal, Perouz Taslakian, Spandana Gella, and Sai Rajeswar. Ui-vision: A desktop-centric gui benchmark for visual perception and interaction. *arXiv preprint arXiv:2503.15661*, 2025a.

Shravan Nayak, Xiangru Jian, Kevin Qinghong Lin, Juan A Rodriguez, Montek Kalsi, Rabiul Awal, Nicolas Chapados, M Tamer Özsu, Aishwarya Agrawal, David Vazquez, et al. Ui-vision: A desktop-centric gui benchmark for visual perception and interaction. *arXiv preprint arXiv:2503.15661*, 2025b.

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*, 2025a.

Zengyi Qin, Jinyuan Chen, Yunze Man, Shengcao Cao, Ziqi Pang, Zhuoyuan Wang, Xin Sun, Gen Lin, Han Fang, Ling Zhu, et al. Osgym: Super-scalable distributed data engine for generalizable computer agents. *arXiv preprint arXiv:2511.11672*, 2025b.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. Androidworld: A dynamic benchmarking environment for autonomous agents, 2024.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, Ran Xu, and Caiming Xiong. Coact-1: Computer-using agents with coding as actions, 2025. URL https://arxiv.org/abs/2508.03923.

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

Liang Tang, Shuxian Li, Yuhao Cheng, Yukang Huo, Zhepeng Wang, Yiqiang Yan, Kaer Huang, Yanzhe Jing, and Tiaonan Duan. Sea: Self-evolution agent with step-wise reward for computer use. *arXiv e-prints*, pp. arXiv–2508, 2025.

Qwen Team. Qwen3.5: Accelerating productivity with native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, et al. Opencua: Open foundations for computer-use agents. *arXiv preprint arXiv:2508.09123*, 2025.

Yifan Wu, Yiran Peng, Yiyu Chen, Jianhao Ruan, Zijie Zhuang, Cheng Yang, Jiayi Zhang, Man Chen, Yenchi Tseng, Zhaoyang Yu, et al. Autowebworld: Synthesizing infinite verifiable web environments via finite state machines. *arXiv preprint arXiv:2602.14296*, 2026.

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. *arXiv preprint arXiv:2505.13227*, 2025a.

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

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.

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

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. *arXiv preprint arXiv:2507.05791*, 2025b.

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

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

Ziyun Zhang, Zezhou Wang, Xiaoyi Zhang, Zongyu Guo, Jiahao Li, Bin Li, and Yan Lu. Infiniteweb: Scalable web environment synthesis for gui agent training. *arXiv preprint arXiv:2601.04126*, 2026.

A APPENDIX

Roadmap: Section A.1 complements with more details of **VISTA** implementations. Section A.2 shows the complete statistics during the synthesis and verification process of **VISTA**. Section A.3 introduces more details of the benchmarks we use for evaluation. Section A.4 presents the hyperparameters and prompts we use for post-training and inference. Section A.6 introduces the computing resources to use to reproduce our experiments. Sections **??**, **??** and discusses the limitations, societal impacts.

A.1 IMPLEMENTATION DETAILS OF **VISTA**

We design and implement **VISTA** on top of the OSGym infrastructure (Qin et al., 2025b). To enable automatic task and verifier synthesis, we refactor the setup and evaluator representations inherited from OSWorld (Xie et al., 2024) into Python-style function signatures, making them easier for the model to call, validate, and compose. Below, we present the prompts used for the task creator and the verifier's semantic-checking module.

Prompt for Task Creator

You generate complete OS task examples: instruction, setup config, and evaluator (`postconfig` + `eval`) in one pass. Task and verifier are designed together: every task must include a verifier that checks what the instruction asks for.

```
## Tasks
- Tasks must produce concrete, persistent state changes. Avoid pure
    data entry,
  read-only tasks, or tasks requiring external assets.
- Try to propose diverse tasks. Here are some axes to vary along:
  1. Spread across the app's menu tree, preference panels, and
    document/object
     properties, such as author and page setup, rather than only
    body content.
  2. For the same feature, try different concrete options each
    time, such as
     bold vs. italic vs. font color vs. highlight, rather than
    repeating the
     same option.
  3. Prefer edits that are not too simple, such as content-rich
    rows,
     paragraphs, slides, mixed types, or formatting.

## Setup helpers
The tasks will be run on a Ubuntu VM booting from a clean snapshot
    without
pre-existing user files. Therefore, every file the task touches
    must be created
in `config` before being opened and used.

Here are some helpers to create files; more comprehensive ones will
    be provided:
- `_create_calc_file_setup()` -- .xlsx with optional cell values.
- `_create_writer_file_setup()` -- .docx with optional paragraphs.
- `_create_impress_file_setup()` -- .pptx with optional slides.
- `_create_gimp_image_setup()` -- blank raster (.png/.jpg/.bmp;
    format from extension).
- `_command_setup()` -- general-purpose shell fallback
  (`bash -c`, `printf`, `python -c`, ...).

## Evaluator
The `eval` is one Python expression that:
1. Calls getters (`getter(env, config={...})`) to extract VM state.
2. Calls metrics to compare extracted state against expected state.
```

```
3. Uses `get_rule(env, config={'rules': ...})` to feed expected
    values into a metric.
4. Combines multiple checks with `and` / `or` when the task spans
    several conditions.

Some requirements for the `eval`:
- The `eval` must be directly related to the instruction.
- The `eval` MUST inspect the substantive change described by the
    instruction,
  NOT a superficial proxy. The instruction's verb is the test
    target.
  - "Apply filter X to image and save" -> verify the saved image
    content reflects
    filter X, such as pixels, dimensions, perceptual hash, or color
    histogram.
    Do NOT check only file size, existence, or mtime.
  - "Set bold on paragraph 2" -> verify that paragraph's run has
    `bold=True`.
    Do NOT check only that the file was modified or grew.
  - "Rename column to 'foo'" -> verify the column header reads
    'foo'.
    Do NOT check only mtime or row count.
  - "Add a sheet named 'Report'" -> verify the sheet exists AND its
    name is
    'Report'. Do NOT check only the workbook's overall size.

## Rules
1. The instruction must be concrete and achievable on Ubuntu.
2. The `eval` must check the substantive content/state implied by the
   instruction's verb. Never use only a file path, file size, or
    mtime when the
   task names a content change.
3. No network downloads (`_download_setup` disabled); only use real,
    verifiable
   URLs if unavoidable.
4. Every name in `config`, `evaluator.postconfig`, and
    `evaluator.eval` must
   appear verbatim in the provided function catalog OR be declared
    in
   `extra_functions` with a runnable implementation. No invented
    variants and
   no stdlib calls inside `eval`; wrap them in a custom getter or
    metric.
5. Read each function's schema carefully and pass every required
    key with the
   correct name, type, and shape.
6. Return valid JSON only. Do not use markdown fences.

## Output schema
Each task is a JSON object:
- "snapshot": app snapshot name
- "instruction": one-sentence description
- "config": list of setup function call strings
- "related_apps": list of app names
- "evaluator": {
    "postconfig": [...],
    "eval": "<single Python expression>"
  }
  Both `postconfig` and `eval` are REQUIRED.
- "extra_functions": REQUIRED whenever any call name is not in the
    catalog;
  omit otherwise. Each entry should follow:
  {
```

```
    "name": "...",
    "kind": "setup|getter|metric",
    "description": "...",
    "implementation": "<runnable Python>"
  }

Do not include other fields in the JSON object, such as "id",
    "domain", or "source".
```

Prompt for Verifying Task–Evaluator Relevance

You audit synthesized OS task examples and decide whether the example's evaluator actually checks what the instruction asks for. You are the last line of defense before the example is run on a VM, so be strict.

```
You will receive:
- The natural-language instruction the agent must satisfy.
- The evaluator's postconfig, i.e., setup-call strings.
- The evaluator's eval, i.e., a single Python expression composed of
  getter/metric calls.

## What "relevant" means
The eval must inspect the SUBSTANTIVE state described by the
    instruction's
verb / target. A relevant evaluator examines the same artifact and
    the same
property the instruction names.

## Common irrelevance patterns to flag
- Instruction asks for an image content/filter change, such as
    applying Edge
  Detect, adding a stroke border, or blurring an image, but eval
    only checks
  check_image_size, file existence, mtime, or file size.
- Instruction asks for a content change in a document, such as
    setting bold on
  a paragraph, renaming a column header, or adding a sheet named X,
    but eval
  only checks that the file exists or that mtime changed.
- Instruction names a specific value, such as a particular color,
    font, label,
  or number, but eval only checks a generic shape, such as any
    non-empty string
  or any numeric value.
- Instruction names an output path that differs from the path the
    evaluator
  inspects.
- Instruction calls for a transformation, such as resize, crop,
    rotate, or
  recolor, but eval only verifies the source artifact still exists
    with its
  original attributes.
- The evaluator inspects a different file/object than the one the
    instruction
  names.

An evaluator can also be irrelevant if the postconfig opens an
    unrelated
resource, or if the metric/getter combination cannot logically
    observe the
property the instruction names.
```

```
## What is NOT a reason to mark irrelevant
- The check is partial but on the right artifact and property. For
    example,
  verifying width and height of a resized image when the
    instruction says
  "resize to 400x300" is relevant because width/height is the
    substantive
  property.
- The check uses a stricter or stronger metric than strictly
    necessary.
- You personally would have written the eval differently, but it
    still pins
  down what the instruction asked for.

## Output schema
Return ONLY a JSON object, no markdown fence, no commentary:
{
  "relevant": yes | no,
  "reason": "<one sentence explaining the verdict, naming the
    specific
            instruction target and the specific evaluator
    behavior>"
}
```

Memory Block. To reduce near-duplicate task generation, we maintain a vector store of previously generated tasks using Qwen3-Embedding-4B (Zhang et al., 2025). During synthesis, each new task candidate is embedded and compared against stored task embeddings to identify semantically similar examples. The most relevant recent duplicates are then fed back into the task-generation prompt as negative context, encouraging the model to explore different task goals, software features, and state changes rather than repeatedly proposing similar tasks.

Verifier's Static Check. Here we provide more details of the verifiers introduced in about the static validation part. As is shown in Figure 5, the static verification workflow filters each synthesized task before it is sent to VM execution. The input is a complete synthetic example, including the task instruction, setup scripts and evaluators. The workflow then checks whether the generated setup and evaluator are syntactically valid, compatible with the catalog, and likely executable. The AST-based validator checks every function call in the setup functions and evaluators as a Python AST expression and checks whether the arguments match the real implementation signature. If the example declares "extra_functions", the verifier will check whether its structure is legal, including a valid name, kind, and implementation. The verifier also checks the implementation of the newly proposed evaluator by parsing it for safe top-level structure, compiled, executed in an isolated namespace, and checked to ensure that it defines a callable with the declared name and an inspectable signature.

A.2 Complete Statistics of Synthetic Tasks and Evaluators

The full synthesis statistics in Table 10 and Table 11 provide a more detailed view of the synthesis behavior of **VISTA** on different VLMs. To illustrate the resulting data, Figure 8, Figure 6, and Figure 7 present representative synthesized tasks from the Multi_apps, LibreOffice Calc, and Thunderbird domains, respectively. Each example shows the task instruction, the initial state, the executable evaluator, and a golden solution. Beyond the core operation, these tasks often impose additional constraints, such as leaving the source file and other settings unchanged, preserving the original file format, or selecting the correct message among distractors, which are explicitly checked by the evaluator.

A.3 Details of Benchmarks

- **OSWorld-Verified:** OSWorld (Xie et al., 2024) originally curated 369 human-crafted tasks covering a wide range of applications, each accompanied by its own environment setup and evaluation

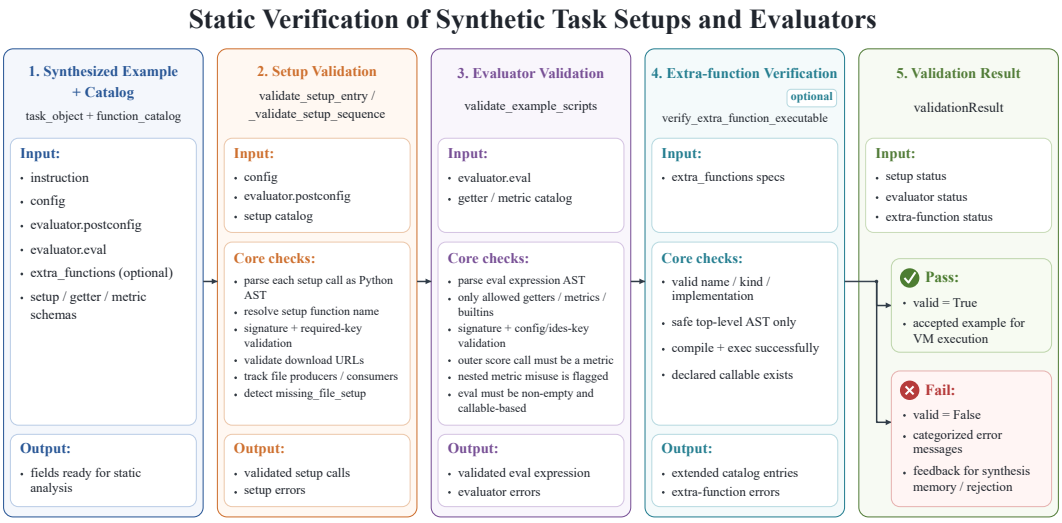
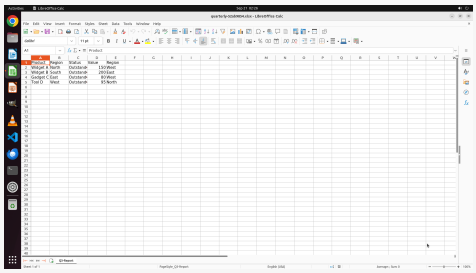


Figure 5: Illustration of the static verification workflow of synthetic examples.

Task instruction Domain: `libreoffice_calc` Apps: LibreOffice Calc

In the prepared workbook, every cell that says 'Outstanding' should say 'Completed', the sheet should be called 'Final Summary', and every other cell stays as it is; save the workbook.

Initial state



`quarterly-52ab9854.xlsx` is open with a single sheet `Q3-Report`; column C holds four 'Outstanding' cells.

Evaluator

- Saved file has exactly one sheet, named `Final Summary`, of size 5×5.
- Cells `C2–C5` = `Completed`.
- All other cells in `A1:E5` keep their original values.

Golden solution

1. Press `Ctrl+H`; set *Find* to `Outstanding` and *Replace* to `Completed`; click *Replace All*, then close the dialogs.
2. Double-click the sheet tab `Q3-Report`, type `Final Summary`, and click *OK*.
3. Press `Ctrl+S` and choose *Use Excel 2007-365!* to keep the `.xlsx` format.

Figure 6: An example LibreOffice Calc task synthesized by OSGEN. The agent must perform a targeted find-and-replace and rename the sheet, while preserving all other cells and the file format.

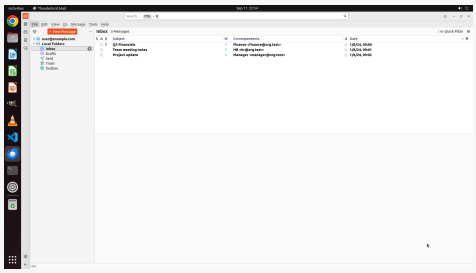
script. The OSWorld team has now verified every task, fixing those that were infeasible to test because of outdated dependencies, evaluation errors, or ambiguous instructions, and has released the improved benchmark as OSWorld-Verified (Xie et al., 2025b)

- **OSWorld-G:** OSWorld-G (Xie et al., 2025a) has 564 samples that systematically cover text matching, element recognition, layout understanding and fine-grained manipulation, with annotations for the element types required to solve each task.
- **WindowsAgentArena-V2:** WindowsAgentArena (Bonatti et al., 2025) evaluates computer-use agents in a real Windows OS environment, covering tasks in office documents, web browsing, system settings, coding, media playback, and Windows utilities, each accompanied by an automatic environment setup and a hand-written evaluation script. WindowsAgentArena-V2 (**?**) refines it into 141 tasks across 11 widely used Windows applications, resetting the VM state between

Task instruction Domain: `thunderbird` Apps: Thunderbird

The file ‘summary.csv’ attached to the message ‘Q3 Financials’ in Thunderbird’s Inbox should be saved into the Desktop folder under its original name, with its contents unchanged.

Initial state



Thunderbird is open on Local Folders; the Inbox holds three unread messages, and only ‘Q3 Financials’ carries an attachment.

Evaluator

- File `/home/user/Desktop/summary.csv` exists.
- Its content contains `Q3 revenue is high`.

Golden solution

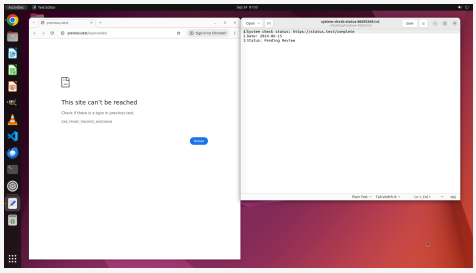
1. Click the message *Q3 Financials* in the Inbox to open it in the preview pane.
2. In the attachment bar, click *Save* next to `summary.csv`.
3. In the file dialog, select the *Desktop* folder, keep the name `summary.csv`, and click *Save*.

Figure 7: An example Thunderbird task synthesized by OSGEN. The agent must locate the correct message among distractors and save its attachment to a specified location.

Task instruction Domain: `multi_apps` Apps: Chrome, OS

The file `system-check-status-86d92369.txt` in the Desktop folder `handover-86d92369` records the system check status the team agreed on. Set Chrome’s page to open on startup to the address recorded in that file, and leave every other Chrome setting as it is. Do not change the file itself.

Initial state



Chrome starts on the stale page `previous.test/superseded`; the handover file is open in the text editor.

Evaluator

- Chrome `Preferences`: `session.restore_on_startup` = `4` `session.startup_urls` = `["https://status.test/complete"]`
- Source file unchanged: text still equals the original three lines.

Golden solution

1. Read the URL `https://status.test/complete` from the open text file.
2. In Chrome, open the three-dot menu → *Settings* → *On startup*.
3. Keep *Open a specific page or set of pages* selected; click the three-dot button next to `previous.test/superseded` → *Edit*.
4. Replace the URL with `https://status.test/complete` and click *Save*.
5. Leave the text file untouched and finish.

Figure 8: An example task synthesized by OSGEN. The agent must read a value from a text file and transfer it into Chrome’s startup settings, while leaving the source file and all other settings unchanged.

Table 8: Hyperparameter settings for training. Group size applies only to GRPO.

Hyperparameter	Qwen3-VL-8B-Instruct		Qwen3.5-9B	
	SFT w/ AgentNet	GRPO w/ VISTA	SFT w/ AgentNet	GRPO w/ VISTA
Total steps	350	350	350	350
Max prompt length	8192	8192	8192	8192
Max response length	1024	1024	1024	1024
Learning rate	2×10^{-6}	2×10^{-6}	2×10^{-6}	1.5×10^{-6}
Global batch size	32	128	32	128
Weight decay	0.1	0.1	0.1	0.1
Group size per prompt	–	16	–	16

Table 9: Task completion steps across different software domains on OSWorld-verified. Bold shows the lowest average completion steps in each domain (lower is better).

Model	Chrome	GIMP	Calc	Impress	Writer	Multi Apps	OS	Thunderbird	VLC	VS Code	Overall
Qwen3-VL-8B-Instruct	22.6	29.7	27.9	28.6	27.8	29.6	18.9	20.4	**16.8**	24.2	24.7
SFT w/ AgentNet	23.4	27.6	28.4	**26.1**	**25.6**	27.9	19.7	**18.3**	17.5	21.7	23.6
RL w/ **VISTA**-synthetic-data	**19.2**	**26.1**	**24.7**	27.3	26.7	**26.2**	**17.8**	19.5	17.2	**20.4**	**22.5**
Qwen3.5-9B	21.4	27.8	26.6	26.9	25.8	28.4	18.1	19.0	**16.2**	23.5	23.4
SFT w/ AgentNet	20.1	28.5	24.1	**25.1**	26.6	25.8	19.0	**17.4**	16.9	21.2	22.5
RL w/ **VISTA**-synthetic-data	**18.3**	**25.4**	**23.2**	26.0	**24.7**	**24.6**	**17.5**	18.2	16.6	**19.8**	**21.4**

tasks to remove cross-task evaluation dependency, removing infeasible tasks to prevent infeasible hacking, stabilizing the VM initial state, and fixing flawed evaluators.

- **Screenspot-Pro:** Screenspot-Pro (Li et al., 2025a) focuses on high-resolution desktop environments, especially in professional settings. UIVision includes fine-to-course grained tasks to evaluate model's performance on understanding professional software, spatial reasoning, and complex actions in desktop environments.

A.4 Training and Inference Details

Table 8 summarizes the training settings used for all baselines and our method. To ensure a fair comparison, we keep the total number of optimization steps and the context-length budget identical across runs for each base model, so that differences in performance can be attributed to the training data rather than to the training budget. We use unified prompt templates to standardize agent behavior during both training and evaluation. For OSWorld (online), the agent has access to the full action space and, at each step, must output an explicit action description followed by a single structured tool call. For OSWorld-G (offline), we adopt the benchmark's restricted mouse-only interface, constraining the agent to `mouse_move` and `left_click` actions with coordinate arguments and requiring it to emit a `FAIL` action when the task is infeasible.

A.5 Task Completion Efficiency

Table 9 further shows that **VISTA** improves not only task success, but also execution efficiency. With **VISTA**-synthetic data, the average completion steps decrease from 24.7 to 22.5 for Qwen3-VL-8B-Instruct and from 23.4 to 21.4 for Qwen3.5-9B. The reduction is especially clear in domains that require structured exploration of software layouts, such as such as GIMP, VSCode, and specific operation patterns, such as Chrome and Calc. This suggests that the synthesized tasks do not merely teach the model to reach the correct final state, but also help it learn more direct action strategies for navigating software interfaces.

A.6 Computing Resources

We conduct our experiments on an Ubuntu system equipped with 8 A100 GPUs, and use one EC2 instance with 8 vCPUs and 192 GB RAM for environment execution. To reproduce our full synthesis and training pipeline, we recommend using 8 A100 GPUs. For inference-only evaluation, a single A100 GPU is sufficient.

Table 10: Synthesis statistics of **VISTA** with Qwen3-8B across domains.

Domain	# Proposed Tasks	# Valid Tasks	Task Acceptance Rate	# Proposed Evaluators	# Accepted Evaluators	Evaluators Reuse Rate	Estimated Avg. Steps
Chrome	203	116	57.1%	63	62	98.4%	13.16
GIMP	791	263	33.2%	303	191	63.0%	20.51
LibreOffice Calc	417	235	56.4%	279	178	63.8%	20.98
LibreOffice Writer	234	130	55.6%	191	183	95.8%	16.41
LibreOffice Impress	144	77	53.5%	115	103	89.6%	15.53
Thunderbird	135	50	37.0%	15	1	6.7%	10.51
OS	663	229	34.5%	75	41	54.7%	12.31
VLC	89	33	37.1%	43	30	69.8%	17.46
Multi Apps	340	171	50.3%	158	142	89.9%	22.74
VSCode	333	203	61.0%	57	6	10.5%	12.99
Total	3,349	1,507	45.0%	1,299	937	72.1%	17.00

Table 11: Synthesis statistics of **VISTA** with Qwen3.5-9B across domains.

Domain	# Proposed Tasks	# Valid Tasks	Task Acceptance Rate	# Proposed Verifiers	# Accepted Verifiers	Verifier Reuse Rate	Estimated Avg. Steps
Chrome	239	143	59.8%	108	101	93.5%	11.10
GIMP	501	179	35.7%	87	58	66.7%	16.09
LibreOffice Calc	739	291	39.4%	451	312	69.2%	18.50
LibreOffice Writer	315	186	59.0%	492	456	92.7%	11.94
LibreOffice Impress	590	236	40.0%	219	206	94.1%	12.93
Thunderbird	500	279	55.8%	17	3	17.6%	9.39
OS	500	274	54.8%	86	74	86.0%	9.93
VLC	290	157	54.1%	91	73	80.2%	11.88
Multi Apps	614	266	43.3%	275	273	99.3%	21.69
VSCode	501	287	57.3%	69	9	13.0%	11.16
Total	4,789	2,298	48.0%	1,895	1,565	82.6%	13.62

System prompt for training and inference on OSWorld

You are a helpful assistant that can understand screenshots in the images and take actions in a computer environment to achieve the task.

```
## Tools
You are provided with function signatures within <tools></tools>
    XML tags:
<tools>
# tools_def omitted for brevity (action space specification)
</tools>
You may call the tools defined below to assist with the given task.
...
```

Here are some tips for using the tools: - Use a mouse and keyboard to interact with a computer, and take screenshots. - This is an interface to a desktop GUI. You do not have access to a terminal or applications menu. You must click on desktop icons to start applications. - Some applications may take time to start or process actions, so you may need to wait and take successive screenshots to see the results of your actions. E.g., if you click on Firefox and a window doesn't open, try wait and taking another screenshot. - If you tried clicking on a program or link but it failed to load even after waiting, try adjusting your cursor position so that the tip of the cursor visually falls on the element that you want to click.

```
## Response format
For each step, you must produce exactly two components in the
    following order:

1. An action description:
   - A single sentence that begins with "Action: "
   - It should briefly and clearly describe what you intend to do
    in the UI.

2. A <tool_call> block:
```

```
   - This block must contain only one JSON object with the
    structure:
     {"name": "<function-name>", "arguments": <args-json-object>}
   - Wrap this JSON inside <tool_call>...</tool_call>.
   - Do not include any extra text or explanation inside the block.

## Action Rules:
- The system is running on a x86_64 ubuntu system.
- Chrome is the default browser that has been installed for you to
    use.
- The current working directory is /home/user.
- The password for the user is "password". Use it when you need to
    authenticate or use sudo commands.
- The current date is {DATE}.
- To invoke a tool that requires `element_description`, provide a
    textual description of the element (e.g., "the top-left corner
    of the search bar").
- Do not output anything else outside the action and tool call
    blocks.
- Leave all windows and applications open after completing the task.
- If finishing, invoke DONE if the task completes or FAIL if the
    task is infeasible.
"""
```

Prompt for inference on OSWorld-G/Screenspot-Pro

You may call one or more functions to assist with the user query. You are provided with function signatures within <tools> ...</tools> XML tags:

```
<tools>
{
  "name": "computer_use",
  "description": "Use a mouse to interact with a computer. The
    screen's resolution is <display_width_px>x<display_height_px>.",
  "notes": "Click with the cursor tip centered on targets; avoid
    edges unless asked. Do not use other tools (type, key, scroll,
    left_click_drag). Only left_click and mouse_move are allowed. If
    you can't find the element, terminate and report failure.",
  "parameters": {
    "type": "object",
    "required": ["action"],
    "properties": {
      "action": {
        "type": "string",
        "enum": ["mouse_move", "left_click"],
        "description": "The action to perform."
      },
      "coordinate": {
        "type": "array",
        "description": "(x, y): pixels from left/top. Required for
    action=mouse_move and action=left_click."
      }
    }
  }
}
</tools>
```

For each function call, return a JSON object with function name and arguments within <tool_call> ...</tool_call> XML tags:

```
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>
```

Additionally, if you think the task is infeasible (e.g., the task is not related to the image), return:

```
<tool_call>
{"name": "computer_use", "arguments": {"action": "FAIL"}}
</tool_call>
```