OS-WIZARD: SCALING COMPUTER-USING AGENTS TO SPECIALIZED SOFTWARE VIA KNOWLEDGE-GUIDED EXPLORATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Computer-using agents are increasingly capable of performing complex tasks in the digital world. However, high-quality interaction trajectories remain scarce for specialized desktop applications, where limited application knowledge, cumbersome GUI workflows, and incomplete exploration hinder automated collection. We introduce OS-Wizard, an automated trajectory collection framework. It combines tutorial-guided exploration with a hybrid GUI–Code action space, providing application-specific knowledge and simplifying GUI workflows through direct execution of application-native commands. The framework also checkpoints and revisits pivotal states during tree exploration to explore alternative workflows, complementing linear rollouts for broader coverage. We collect 10.9K trajectories comprising 54.9K state–action samples. Fine-tuning open-source base models on this corpus yields a best-performing model that improves performance on a benchmark of specialized desktop workflows by 58.2 percentage points to 64.1%, surpassing reported human performance of 59.8% and establishing a new state of the art. The fine-tuned model also improves performance on general desktop workflows using GUI actions only, suggesting that the collected supervision benefits general desktop tasks as well. Comparative experiments and ablations demonstrate the efficiency of trajectory collection and the importance of code actions.

1 INTRODUCTION

Computer-using agents (CUAs) extend foundation models from understanding multimodal information to directly interacting with computers, enabling them to complete tasks through graphical user interfaces (GUIs) across web and desktop environments (Deng et al., 2023; Zhou et al., 2024; Xie et al., 2024; Yuan et al., 2026). Given a task instruction and observations such as screenshots, a CUA repeatedly reasons about the current state and executes actions such as mouse clicks or keyboard input (Hong et al., 2024; Zheng et al., 2024a; Agashe et al., 2025). As these agents are increasingly expected to handle longer and more complex workflows, their capabilities depend critically on diverse and high-quality interaction trajectories that provide supervision for both interface grounding and multi-step decision making.

However, computer-using trajectories remain substantially more difficult to obtain than conventional text or vision data, and existing datasets still rely heavily on costly human demonstrations (Wang et al., 2026; Liu et al., 2026; Qin et al., 2025). Recent work has begun to automate trajectory synthesis through model-driven exploration and task generation (Xu et al., 2025; Sun et al., 2025; Wei et al., 2026), but collecting useful data for specialized desktop applications faces three challenges. First, the scarcity of application-specific interaction data leaves general-purpose models with limited knowledge of how to use these applications correctly, hindering effective exploration. Second, even when applications expose domain-specific languages (DSLs), issuing commands through the GUI requires additional interface interactions, introducing failure points beyond those of direct command execution. Third, collection through independent linear rollouts leaves functional coverage largely to chance: unguided sampling can miss important capabilities, while valuable intermediate states are not systematically revisited to explore alternative workflows.

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.

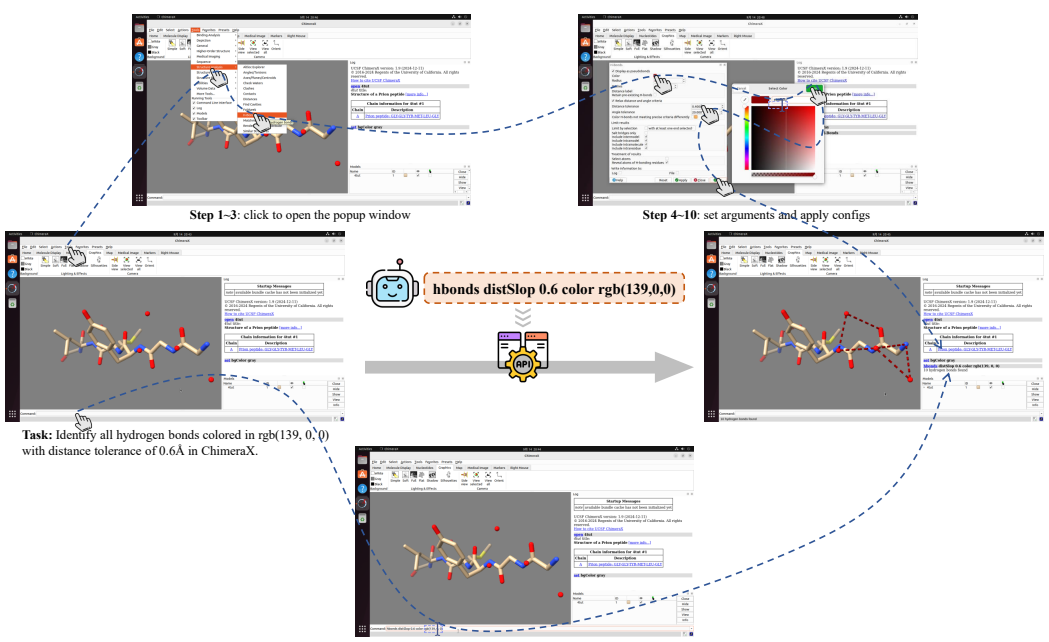


Figure 1: **Three interaction paths for the same visualization task in ChimeraX. Top:** A GUI-only workflow takes 10 actions to open the dialog, configure parameters, and apply the settings. **Middle:** Our domain-specific code interface executes the equivalent application-native command with a single action. **Bottom:** Entering the command through the GUI takes 3 actions to click the input, type the command, and press Enter.

To address these challenges, we introduce **OS-Wizard**, an automated trajectory collection framework for specialized desktop applications. OS-Wizard augments GUI interactions with domain-specific code actions that directly submit application-native commands, including DSL expressions, reducing the interface interactions required to execute expert operations. In Figure 1, this abstraction reduces a multi-step hydrogen-bond visualization workflow to a single code action. To equip the collector with the knowledge needed to use this action space, we retrieve application documentation and tutorials that explain application workflows and command usage, guiding exploration and task execution in unfamiliar software. Beyond independent linear rollouts, we introduce tree exploration organized around *pivotal states*—states expected to offer valuable branching opportunities. By identifying and checkpointing these states, the collector can repeatedly restore them and explore different continuations, reusing shared workflow prefixes to broaden functional coverage. Both collection strategies combine execution checks with stage-specific verification.

We evaluate OS-Wizard on ScienceBoard (Sun et al., 2026), a benchmark of specialized scientific applications, and OSWorld (Xie et al., 2024), a general desktop benchmark. On ScienceBoard, fine-tuning Qwen3.5-27B on our collected trajectories improves overall task success from 5.9% to 64.1%, surpassing the reported overall human performance of 59.8% and establishing a new state of the art. Under a matched 10K-sample training budget, OS-Wizard achieves 36.5% success compared with 13.0% and 18.9% for the Task-driven and Branch-driven collection baselines, demonstrating that our collection framework produces more effective supervision per sample. On OSWorld, the fine-tuned model improves success from 39.1% to 42.9% using GUI actions only, which shows that the collected supervision supports substantial gains on specialized applications while also improving general desktop task performance.

Our main contributions are:

- We introduce OS-Wizard, an automated trajectory collection framework for specialized desktop applications, using tutorials to mitigate knowledge gaps and revisiting pivotal states to explore alternative workflows beyond independent linear rollouts.

- We develop a validatable hybrid GUI–Code action space that reduces the interface interactions needed to execute application-native commands, with ablations supporting the value of code supervision.
- We collect 10,919 trajectories comprising 54,938 state–action samples, improving performance on ScienceBoard and OSWorld and outperforming alternative collection strategies under a matched training-data budget.

2 Related Works

Computer-using Agents. Existing computer-using agents primarily emulate human computer use through GUI interactions. Cradle (Tan et al., 2024), CogAgent (Hong et al., 2024), SeeClick (Cheng et al., 2024), Agent S (Agashe et al., 2025), UFO (Zhang et al., 2025c), and UI-TARS (Qin et al., 2025) use mouse and keyboard actions to perform tasks from visual observations. Recent work extends this paradigm with programmatic interfaces: CoAct-1 (Song et al., 2025) and ActionEngine (Zhong et al., 2026) introduces coding actions for executable Python or Bash programs; UltraCUA (Yang et al., 2025), UFO^2 (Zhang et al., 2025b), and Axis (Lu et al., 2025) incorporate high-level tools or native application APIs; and ToolCUA (Hu et al., 2026) and EE-MCP (He et al., 2026) study hybrid GUI–tool interaction and orchestration through MCP-based interfaces. These systems focus on general-purpose computer use, with limited dedicated support for the domain-specific knowledge and operations required by specialized desktop applications.

Data Synthesis for GUI Agents. The scarcity of high-quality interaction trajectories has motivated scalable data generation for GUI agents. Tutorial-based methods such as AgentTrek (Xu et al., 2025), TongUI (Zhang et al., 2025a), and Video2GUI (Xiong et al., 2026) convert web tutorials into executable trajectories, while task-driven approaches, including AgentSynth (Xie et al., 2026) and Insta (Trabucco et al., 2025), synthesize task intents and execute them with capable agents. Exploration-driven methods such as OS-Genesis (Sun et al., 2025), GUI-ReWalk (Lin et al., 2025), and AutoPlay (Ramrakhya et al., 2026) instead derive tasks or workflows from environment interaction. Recent systems further improve structure and scalability: ANCHOR (Wei et al., 2026) expands verified demonstrations from meaningful branch states, and SEE (Fan et al., 2026) models GUI state transitions for long-horizon synthesis. However, dedicated mechanisms for exploring domain-specific operations and collecting verified trajectories in specialized desktop applications remain underexplored.

Advancing GUI Agents with External Knowledge. Agentic retrieval augments conventional RAG with iterative planning, query refinement, and tool-based retrieval, allowing agents to adapt the retrieval process to the task (Zhang et al., 2024; Jin et al., 2025; Singh et al., 2025; Du et al., 2026; Suresh et al., 2026). Domain- and document-grounded agents apply these mechanisms to specialized knowledge, with PaperQA (Lála et al., 2023), RAGentA (Besrour et al., 2025a) and SQuAI (Besrour et al., 2025b) combining multi-agent retrieval and evidence verification for attributed and scientific QA, while LongAgent (Zhao et al., 2024), M3DocAgent (Cho et al., 2024), RAG-Anything (Guo et al., 2025), VisRAG (Yu et al., 2025), and Patho-AgenticRAG (Zhang et al., 2026) ground multimodal and long-document reasoning in external references. These methods primarily ground reasoning and answer generation in retrieved knowledge, leaving its role in application-specific actions and workflow exploration underexplored.

3 OS-Wizard

OS-Wizard is an automated framework for collecting computer-using agent trajectories in specialized applications. Its key insight is to explore professional software through a hybrid GUI–Code action space, capturing both graphical interactions and domain-specific operations. To scale data collection over this action space, OS-Wizard combines structured tree exploration with complementary linear rollouts. Figure 2 illustrates both collection strategies: tree exploration builds on cold-start navigation and pivotal-state checkpointing to support tutorial-guided parallel exploration, while linear rollouts collect independent task-oriented trajectories. Both strategies combine execution checks with stage-specific verification.

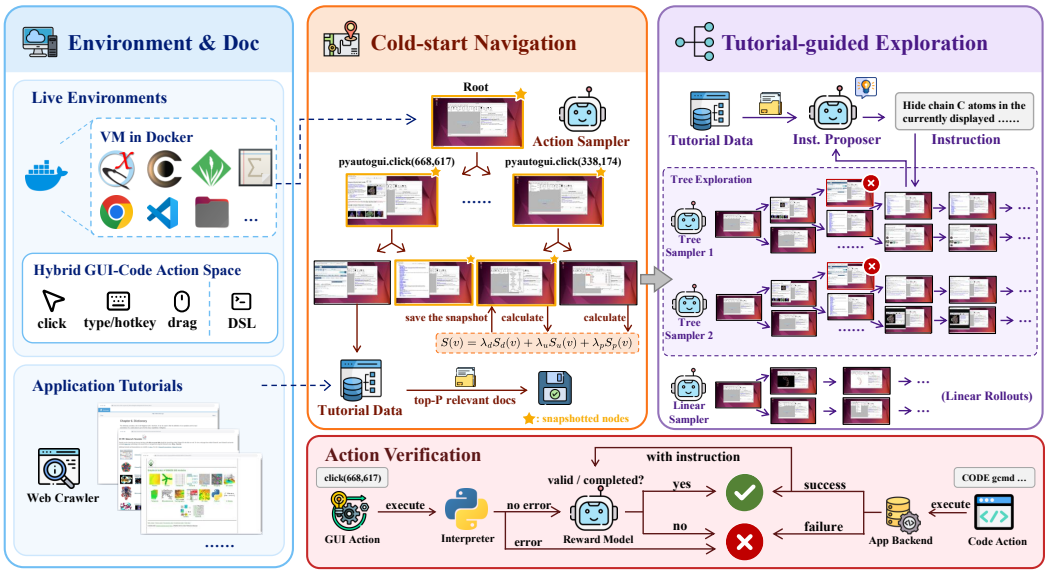


Figure 2: **Overview of trajectory collection in OS-Wizard. Left:** Tutorials guide exploration with GUI and code actions. **Center:** Cold-start navigation builds an initial tree and checkpoints pivotal states. **Right:** Parallel tree exploration and linear rollouts collect trajectories. **Bottom:** Execution checks and reward-model verification ensure trajectory quality.

We model desktop GUI exploration as a rooted exploration tree $T = (V, E)$ with maximum depth D, where each node $v = (s, x, t) \in V$ represents a distinct visited state instance, consisting of a screenshot s, accessibility information x, and a textual state description t. Each edge $e = (v, a, v') \in E$ records an action $a \in A$ executed from node v that non-deterministically produces a new child state instance v'. Linear rollouts arise as a special case of this formulation: when every non-root node is restricted to have at most one child, the exploration tree becomes a collection of independent linear trajectories sharing the same initial root state.

3.1 Pivotal State Identification

Naively expanding GUI state trees incurs space complexity of $O(N^D)$ for branching factor N, while repeatedly recovering successive states along a d-step trajectory via action replay under BFS incurs a cumulative cost of $O(d^2)$. Since GUI states differ substantially in their downstream exploration value, we define a sparse set of *pivotal states* as states expected to serve as high-value branching points for subsequent exploration. OS-Wizard allocates more exploration budget to such states and stores VM snapshots for them, reducing both redundant branching and recovery overhead.

Given adjustable weights λ_d, λ_u and λ_p and candidate set of states C, we score each state $v \in C$ by

$$S(v) = \lambda_d S_d(v) + \lambda_u S_u(v) + \lambda_p S_p(v), \tag{1}$$

where each term is min–max normalized before aggregation. $S_d(v)$ is a depth prior favoring shallow states, which expose broader functional regions and are shared by more trajectories. Specifically, the root is always treated as a pivot. $S_u(v)$ measures the magnitude of the preceding state transition based on changes in both the screenshot and the textual state description. Larger transitions such as page-level changes are more likely to introduce useful branching contexts. $S_p(v)$ optionally incorporates empirical action success rate from v when such statistics are available, favoring states with reliable interactions. In practice, we select the top $\lceil 0.2|C| \rceil$ states in C by $S(v)$ as pivots.

3.2 Two-stage Tree Exploration

The tree exploration component of OS-Wizard proceeds in two stages: *Cold-start Navigation* and *Tutorial-guided Parallel Exploration*, as is shown in Algorithm 1. Exploration follows a depth-dependent branching function $f(d) = \max\{N - 3d, 1\}$, where $N = 10$ denotes the root branching

Algorithm 1 Two-stage collection in tree exploration

Stage I: Cold-start Navigation
$V_1 \leftarrow \textsc{Expand}(v_0, N)$, $V_2 \leftarrow \bigcup_{v \in V_1} \textsc{Expand}(v, N)$
$S \leftarrow \textsc{Pivot}(V_1, V_2)$, $\textsc{Snapshot}(S)$
$\textsc{Retrieve}(v, P)\ \forall v \in V_2$
Stage II: Tutorial-guided Parallel Exploration
for all workers **in parallel do**
  $Q \leftarrow$ randomly sparsified V_2 with S preserved
  while $Q \neq \varnothing$ **do**
    $(v, d) \leftarrow \textsc{Pop}(Q)$
    if $f(d) > 1$ **then**
      $Q \leftarrow Q \cup \textsc{Verify}(\textsc{Expand}(v, f(d)))$
    else
      $\textsc{Rollout}\big(v, \textsc{Synth}(\text{tutorials, exemplars}), D\big)$

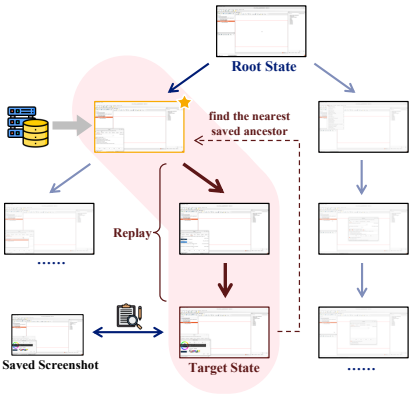


Figure 3: **Left:** Two-stage collection in the tree exploration component of OS-Wizard. **Right:** Snapshot-assisted state recovery (§3.4): find and restore the nearest saved ancestor, then replay all actions along the path to the target while checking screenshot similarity.

factor. We truncate exploration beyond the maximum depth $D = 16$. The motivation and design of this branching schedule are discussed in §5.3.

Cold-start Navigation. Starting from the root state, a single worker performs open-ended exploration, where an LLM sampler proposes diverse and non-redundant actions from the current screenshot and exploration history. The worker expands $f(0) = N$ first-level states and oversamples each of them to N second-level states. Pivotal states are then identified using the scores in §3.1: each first-level state forms a singleton candidate set and is thus retained as a pivot, while each group of ten second-level siblings forms a candidate set from which 2 pivots are selected and checkpointed. For every second-level state, we additionally retrieve the top-P relevant tutorials with $P = 5$.

Tutorial-guided Parallel Exploration. Each worker receives the same cold-start tree but independently sparsifies and expands it. Within each group of second-level siblings, we randomly drop $M = N - f(1)$ nodes while always preserving pivots, reducing the effective branching factor to $f(1)$. The remaining nodes are inserted into a BFS queue and expanded according to $f(d)$. For a pivotal state, sampling continues until obtaining up to $f(d)$ valid children, whereas a non-pivotal state is attempted at most $f(d)$ times and may therefore yield fewer children. At branching states with $f(d) > 1$, we use the same open-ended action sampling as in cold-start navigation to diversify action-level expansions. Once $f(d)$ decreases to 1, the retrieved tutorials and 5–10 OOD exemplar instructions guide the synthesis of a task instruction encompassing both the exploration prefix and the subsequent rollout. With tutorials as context, the worker follows the instruction until completion or the maximum depth is reached. The single branch is executed continuously without BFS recovery between intermediate states. Independent dropout and action sampling lead workers toward different branches, improving trajectory diversity and depth. Newly generated transitions are validated according to their action types before being added to the tree as described in §3.3.

3.3 Validatable Hybrid Action Space

The key design of OS-Wizard is its hybrid action space $A = A_{\text{GUI}} \cup A_{\text{Code}}$. Professional applications often expose command-line interfaces or domain-specific languages alongside graphical controls, making GUI-only interaction unnecessarily long or brittle for many expert operations. A_{GUI} contains PyAutoGUI-based interactions, while A_{Code} contains domain-specific command primitives. Domain-specific code actions do not expose privileged functionality, but abstract commands that could otherwise be entered and executed through the GUI. They may be provided directly by the application or benchmark, or realized by automatically entering the corresponding commands through the GUI based on accessibility information. See §A for code actions supported by each application and additional implementation details.

Quality control is integrated into action generation and post-execution verification. Each sampled action must be grounded in the current screenshot and accessibility information and, when an instruction is present, remain consistent with the task objective. At branching states, previously sampled sibling actions are provided as negative references, and duplicate or semantically redundant actions are rejected to promote intent-level diversity. Recent execution history is also included to suppress local action loops.

All actions must execute without errors. Without a task instruction, GUI actions additionally require reward-model validation of transition validity, whereas code actions are accepted if the application reports no execution error. With an instruction, both action types undergo execution checks, and the reward model judges only task completion: success terminates the rollout; otherwise, execution continues until completion or the depth limit D. Trajectories terminated solely by this limit are excluded. Lean reference-proof replay uses pre-collection compilation checks and a separate completion procedure (§D).

3.4 Snapshot-assisted State Recovery

To reduce BFS recovery cost and improve robustness to non-deterministic GUI behavior, OS-Wizard checkpoints all pivotal states in $S \subseteq V$. As illustrated in the right part of Figure 3, given a target state v, we recover from its nearest checkpointed ancestor

$$p(v) = \underset{u \in \mathrm{Anc}(v) \cap S}{\arg\max}\ d(u), \tag{2}$$

by loading the snapshot of $p(v)$ and then replaying only the actions along the path from $p(v)$ to v. Since QXL graphics state cannot be reliably restored from QEMU snapshots, we use an Xvfb-based virtual desktop instead; see Appendix §B for details.

During the replay, each recovered state is verified against its recorded screenshot via visual similarity. A failed step is retried up to $R = 10$ times before the recovery is aborted. For applications with time-varying or network-dependent content, we adopt a relaxed visual-consistency threshold to tolerate benign rendering differences.

4 Experiments

4.1 Experimental Settings

Benchmarks. We conduct experiments on ScienceBoard (Sun et al., 2026) and OSWorld (Xie et al., 2024). Both benchmarks are adapted to the Qwen3 tool-call format (Bai et al., 2025), where ScienceBoard uses the hybrid GUI–Code action space described in §3.3 for all applications except Celestia, which lacks a programmatic control interface comparable to the command-line or DSL available in other applications, while OSWorld uses GUI actions only.

Data Collection. To balance collection quality and cost, we use GPT-5.6-Sol (OpenAI, 2026b), GPT-5.5 (OpenAI, 2026a), and Kimi-K2.5 (Team et al., 2026) as instruction proposers and action samplers, GPT-5.6-Sol as the reward model for transition and completion verification, and Qwen3-Embedding-8B (Zhang et al., 2025d) for tutorial retrieval. We collect data using both OS-Wizard tree exploration and linear rollouts. The resulting corpus contains 10,919 trajectories and 54,938 state–action training samples, averaging 5.03 steps per trajectory, including terminal steps. Tree exploration contributes 32,250 samples, while linear rollouts contribute 22,688 samples. Of the full corpus, 17,442 samples (31.75%) come from general-purpose applications, while the remaining 37,496 samples come from specialized applications. Lean uses only linear rollouts with an application-specific collection procedure. Detailed collection statistics and the Lean-specific procedure are provided in §C and §D.

Baselines. For ScienceBoard, we compare against (1) base models, (2) proprietary models, (3) some related results from other open-source models, and (4) SFT models trained on human-collected, *Task-driven*, and *Branch-driven* trajectories. We implement the latter two following OS-Genesis (Sun et al., 2025) and ANCHOR (Wei et al., 2026). For OSWorld, where our goal is to assess whether OS-Wizard training preserves and improves general desktop capability, we primarily compare against the corresponding Qwen3.5-27B base model.

Model	Astron	Biochem	GIS	Alg	ATP	Doc	Overall
Human†	51.5	69.0	55.9	<u>74.2</u>	**42.3**	68.8	<u>59.8</u>
Base models							
Qwen3-VL-8B (Bai et al., 2025)	12.1	10.3	2.9	9.7	0.0	0.0	6.5
Qwen3.5-27B (Qwen Team, 2026a)	12.1	13.8	2.9	0.0	0.0	6.3	5.9
Proprietary models							
GPT-5.5 (OpenAI, 2026a)	18.2	41.4	11.8	9.7	3.8	31.3	18.3
GPT-5.6-Sol (OpenAI, 2026b)	36.4	69.0	<u>58.8</u>	41.9	3.8	37.5	42.6
Claude-Opus-4.8 (Anthropic, 2026)	24.2	65.5	8.8	35.5	0.0	37.5	27.8
Gemini-3.1-Pro (Google DeepMind, 2026)	18.2	6.9	2.9	22.6	0.0	25.0	11.8
Open-source models							
Qwen3-VL-235B-A22B (Bai et al., 2025)	9.1	31.0	0.0	9.7	0.0	6.25	9.5
Qwen3.5-397B-A17B (Qwen Team, 2026a)	9.1	17.2	5.9	6.5	0.0	6.3	7.7
Qwen3.8-27B (Qwen Team, 2026b)	36.4	65.5	41.2	61.3	0.0	43.8	42.0
Kimi-K2.5 (Team et al., 2026)	3.0	27.6	0.0	9.7	0.0	12.5	8.3
ScaleCUA-32B (Liu et al., 2026)†	12.1	10.3	0.0	9.7	0.0	0.0	5.9
SCALECUA-Qwen3.5-9B (Lv et al., 2026)†	**60.6**	62.1	35.3	67.7	19.0	<u>86.7</u>	54.0
Sampling methods							
Human Data	12.1	24.1	0.0	9.7	0.0	12.5	9.5
Task-driven (Sun et al., 2025)	15.2	13.8	14.7	19.4	0.0	12.5	13.0
Branch-driven (Wei et al., 2026)	21.2	20.7	20.6	29.0	0.0	18.8	18.9
Ours							
OS-Wizard-Qwen3-VL-8B	$36.3_{\pm 2.5}$	$\underline{71.2}_{\pm 1.6}$	$50.0_{\pm 4.2}$	$54.8_{\pm 2.6}$	$6.4_{\pm 1.8}$	$64.5_{\pm 7.8}$	$46.6_{\pm 0.6}$
OS-Wizard-Qwen3.5-27B-10K	$20.2_{\pm 1.4}$	$51.7_{\pm 0.0}$	$43.1_{\pm 2.8}$	$65.6_{\pm 4.0}$	$6.4_{\pm 1.8}$	$20.8_{\pm 3.0}$	$36.5_{\pm 0.7}$
OS-Wizard-Qwen3.5-27B	$\underline{52.5}_{\pm 1.4}$	$\mathbf{83.9}_{\pm 1.6}$	$\mathbf{68.6}_{\pm 2.8}$	$\mathbf{77.4}_{\pm 2.6}$	$\underline{21.8}_{\pm 1.8}$	$\mathbf{87.5}_{\pm 5.1}$	$\mathbf{64.1}_{\pm 0.7}$

Table 1: ScienceBoard success rates (%). † denotes results reported in the original paper; all other results are evaluated by us with a maximum step limit of 50. Results in the *Ours* block are reported as mean ± standard deviation over three evaluation runs, while results of SCALECUA-Qwen3.5-9B are reported as means. Best results are **bolded** and the second best results are <u>underlined</u>.

Training Setup. We perform full-parameter supervised fine-tuning of Qwen3-VL-8B (Bai et al., 2025) and Qwen3.5-27B (Qwen Team, 2026a) on the collected trajectories using Qwen3-style tool-call conversations. Training configurations, conversation construction, and evaluation details are provided in §E.

4.2 Main Results

ScienceBoard. Table 1 reports success rates on ScienceBoard. OS-Wizard-Qwen3.5-27B achieves state-of-the-art overall performance among the compared methods at 64.1%, surpassing the reported SCALECUA-Qwen3.5-9B result of 54.0% by 10.1 percentage points and the reported human performance of 59.8%. Across all individual domains, OS-Wizard-Qwen3.5-27B ranks first or second among the compared models and reported human results. Across both backbones, fine-tuning on OS-Wizard trajectories yields substantial improvements: Qwen3-VL-8B improves from 6.5% to 46.6%, while Qwen3.5-27B increases from 5.9% to 64.1%. Compared with the newer Qwen3.8-27B base model at 42.0%, Qwen3.5-27B starts from a much lower success rate, indicating limited task-execution capability in these specialized domains before fine-tuning. Its subsequent improvement highlights the effectiveness of the collected supervision in developing this capability. Notably, even OS-Wizard-Qwen3-VL-8B surpasses the strongest evaluated proprietary model, GPT-5.6-Sol (46.6% vs. 42.6%), showing that our supervision can train a smaller model to outperform the strongest model used for data collection. Finally, under the same 10K-sample training budget, OS-Wizard-Qwen3.5-27B-10K substantially outperforms both our Task-driven and Branch-driven baselines (36.5% vs. 13.0% and 18.9%), demonstrating the effectiveness of the overall collection pipeline under a matched data budget.

OSWorld. Figure 4(a) reports success rates on OSWorld. Fine-tuning Qwen3.5-27B on OS-Wizard trajectories improves the success rate from 39.1% to 42.9%, a gain of 3.8 percentage points. The training corpus contains 17,442 samples from general-purpose applications, accounting for 31.75% of all 54,938 samples, while the remaining samples come from specialized applications.

Model	Overall
Base models	
Qwen3-VL-8B †	33.9
Qwen3.5-27B	39.1
Ours	
OS-Wizard-Qwen3.5-27B	**42.9**

Configuration	Samples	ScienceBoard(%)
OS-Wizard	Full	**64.1**
OS-Wizard (mixed)	20K	55.0
w/o code supervision	20K	26.6
w/o linear rollouts	20K	41.4
w/o tree exploration	20K	30.7

Figure 4: **(a)** OSWorld success rates (%). OS-Wizard-Qwen3.5-27B uses a maximum step limit of 50, and † denotes results reported in the original paper. **(b)** ScienceBoard ablations on domain-specific code supervision and trajectory collection strategies using Qwen3.5-27B. The code ablation randomly samples trajectories containing no code calls throughout, yielding 20K training samples.

These results show that training on this corpus improves general desktop performance alongside the gains on ScienceBoard.

5 Analysis

5.1 Ablations on Components

As shown in Figure 4(b), we ablate domain-specific code supervision and trajectory collection strategies to understand where the gains of OS-Wizard come from. As a key design of the framework, the hybrid GUI–Code action space enables concise abstraction of expert operations in specialized applications. Under a matched budget of 20K state–action training samples, training on mixed data achieves 55.0% ScienceBoard success, whereas training on data randomly sampled from trajectories containing no code calls achieves 26.6%. This 28.4 percentage-point gap supports the value of including code supervision in the training corpus.

Tutorial guidance is particularly important for specialized applications where general-purpose models exhibit low task success. For example, GPT-5.5 achieves only 9.7% and 11.8% success rates on KAlgebra and GRASS GIS. In our no-document collection runs on these applications, 75.71% of branches reach the maximum depth $D = 16$ without completion and are discarded, substantially increasing the collection cost per usable trajectory and making data yield less reliable. This indicates that application-specific guidance is important for efficient long-horizon trajectory collection.

We further train models using 20K samples drawn exclusively from either the tree-based or the linear collection strategy. These two variants achieve 41.4% and 30.7% respectively. Both substantially outperform the corresponding base model, demonstrating that each collection strategy provides useful training supervision even with the same amount of training data. These results motivate combining the two collection strategies to construct the final training corpus.

5.2 Data and Model Size Scaling

Data Scaling. We first vary the number of state–action training samples while holding the backbone fixed. Figure 5(a) shows that additional data generally improves ScienceBoard success rates, with substantial gains at small data scales and diminishing returns toward the full corpus. All three backbones improve markedly over their base models with the initial subset of training data, indicating that even limited OS-Wizard supervision is useful for adapting to specialized applications. The benefits also extend beyond this initial adaptation: for Qwen3.5-27B, expanding the training set from 10K samples to the full corpus yields a 27.6 percentage-point improvement. The returns to additional data nevertheless vary across backbones: performance continues to improve for the 9B and 27B models, while the 4B model largely plateaus at the upper end of the explored range. Thus, the results support expanding the collected corpus, while suggesting that the marginal benefit should be assessed for the target backbone.

Model Scaling. We next compare backbone sizes under the same training-data budget. Although the base models do not exhibit a monotonic ordering by size, larger backbones consistently achieve higher success rates at every non-zero data budget in Figure 5(a). This change in ordering suggests that base-model performance alone does not fully reflect the benefits of model scale after

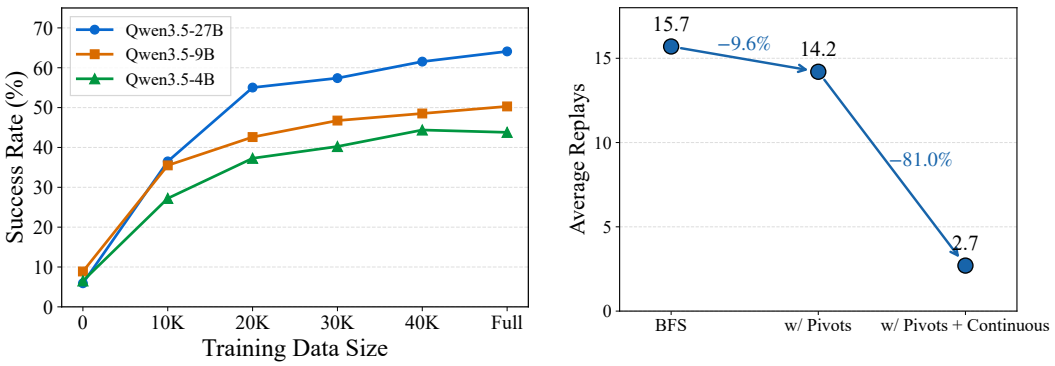


Figure 5: Scaling and recovery-efficiency analysis of OS-Wizard. **(a)** ScienceBoard success rate as a function of training data size for Qwen3.5-4B, 9B, and 27B. **(b)** Average replay overhead of standard BFS, standard BFS with pivotal-checkpoint recovery, and standard BFS with pivotal checkpoints and continuous single-branch execution.

application-specific fine-tuning. With the full corpus, scaling from 4B to 9B and from 9B to 27B improves the success rate by 6.5 and 13.8 percentage points respectively. Since these comparisons hold the amount of training data fixed, they identify backbone size as another effective axis for improving the trained agent. Within the evaluated Qwen3.5 family, a larger backbone can therefore deliver higher task success from an existing corpus, making model scale a useful consideration when the collection budget is fixed.

5.3 Complexity Control

The depth-dependent branching function $f(d) = \max\{N - \gamma d, 1\}$ balances exploration breadth and trajectory depth while controlling tree growth. Since shallow states are shared by more downstream trajectories and often expose broader functional regions, we allocate more branching budget near the root and gradually reduce it with depth. Under $N = 10$, $\gamma = 3$, and $D = 16$, the branching factors decrease as $10 \to 7 \to 4 \to 1$. With an action validation rate of 53.8% estimated from a pilot run, approximately 187.6 branches per worker are expected to reach the single-branch instruction-following stage, preserving broad early coverage without continued combinatorial growth.

OS-Wizard reduces state-recovery overhead through pivotal checkpoints and continuous single-branch execution. In our data-collection runs, standard BFS requires replaying 15.7 actions on average, while recovering from pivotal checkpoints reduces this to 14.2. Once $f(d) = 1$, branches are executed continuously without intermediate BFS recovery, further reducing the observed average to 2.7 replays. Together, these mechanisms control tree growth and the overhead of extending deep branches.

6 Conclusion

We introduced OS-Wizard, an automated trajectory collection framework for specialized desktop applications. OS-Wizard combines a validatable hybrid GUI–Code action space with tutorial-guided exploration and complementary tree-based and linear collection to generate diverse and useful interaction trajectories. Experiments on ScienceBoard and OSWorld show that training on the resulting data consistently improves computer-using agents, with particularly strong gains on specialized professional applications. These results demonstrate the potential of automated trajectory collection and richer action abstractions for scaling supervision beyond common GUI workflows. Future work may further strengthen trajectory quality control and explore more efficient collection architectures that better balance long-horizon exploration with state-storage and recovery costs. Another promising direction is to scale OS-Wizard to substantially larger data and model regimes.

AI USE STATEMENT

In this work, we used generative AI tools to draft sections of the paper and to revise and polish the writing. We also used generative AI as a core component of the synthetic data generation pipeline in OS-Wizard, including task instruction generation, GUI and DSL action sampling, and model-based verification of state transitions and task completion. The models, collection procedures, and prompts are documented in the paper and appendix.

ETHICS STATEMENT

This work does not use private, sensitive, or illicit data. The task files, application resources, and trajectory artifacts used for data collection and evaluation are curated for research purposes and do not contain personal information, confidential documents, authentication credentials, or other sensitive user data. We follow the applicable licenses and terms of use for the benchmarks, models, applications, datasets, and software used in this work. The collected trajectories are generated within controlled task environments rather than from real users' computer activity, and are used for research, training, and evaluation of computer-using agents.

Our hybrid GUI–Code action space also introduces safety considerations for real-world deployment. While code actions can reduce interaction overhead, they can increase the scope and potential impact of erroneous or unintended operations. In particular, computer-using agents deployed on real devices may access sensitive files, credentials, or other private information, and may encounter malicious or misleading content through interfaces or external files, including prompt injection. Real-world deployment should therefore employ sandboxing and least-privilege access controls, restrict code execution and access to sensitive resources, and require appropriate human confirmation and post-action validation for high-impact or irreversible operations. Execution logging and policy checks can further help monitor and constrain agent behavior. The isolated experimental setting used in this work reduces these risks during data collection, but does not eliminate the safety and privacy risks associated with deploying such agents in real-world environments.

REFERENCES

The KAlgebra handbook. `https://docs.kde.org/stable_kf6/en/kalgebra/kalgebra/index.html`.

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

Anthropic. Claude opus 4.8 system card, May 2026. URL `https://www.anthropic.com/claude-opus-4-8-system-card`.

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.

Fabrice Bellard et al. Qemu, a fast and portable dynamic translator. *Usenix ATC, Freenix Track*, 2005:41–46, 2005.

Ines Besrour, Jingbo He, Tobias Schreieder, and Michael Färber. Ragenta: Multi-agent retrieval-augmented generation for attributed question answering. *arXiv preprint arXiv:2506.16988*, 2025a.

Ines Besrour, Jingbo He, Tobias Schreieder, and Michael Färber. Squai: Scientific question-answering with multi-agent retrieval-augmented generation. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, pp. 6603–6608, 2025b.

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.

Jaemin Cho, Debanjan Mahata, Ozan Irsoy, Yujie He, and Mohit Bansal. M3docrag: Multi-modal retrieval is what you need for multi-page multi-document understanding. *arXiv preprint arXiv:2411.04952*, 2024.

Agentic Commons. Formal math autoformalization dataset, 2026. Released under CC0-1.0. Zenodo version DOI 10.5281/zenodo.22765922.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Mingxuan Du, Benfeng Xu, Chiwei Zhu, Shaohan Wang, Pengyu Wang, Xiaorui Wang, and Zhendong Mao. A-rag: Scaling agentic retrieval-augmented generation via hierarchical retrieval interfaces. *arXiv preprint arXiv:2602.03442*, 2026.

Zhuohang Fan, Beichen Zhang, Yuanfa Li, Changqiao Wu, Wei Liu, Jian Luan, and Weigang Zhang. See: Structure-aware exploring\& exploiting for long-horizon gui agent trajectory synthesis. *arXiv preprint arXiv:2607.18046*, 2026.

Google DeepMind. Gemini 3.1 pro model card, February 2026. URL `https://deepmind.google/models/model-cards/gemini-3-1-pro/`.

Zirui Guo, Xubin Ren, Lingrui Xu, Jiahao Zhang, and Chao Huang. Rag-anything: All-in-one rag framework. *arXiv preprint arXiv:2510.12323*, 2025.

Tiantian He, Yihang Chen, Keyue Jiang, Ka Yiu Lee, Kaiwen Zhou, Kun Shao, and Shuai Wang. Ee-mcp: Self-evolving mcp-gui agents via automated environment generation and experience learning. *arXiv preprint arXiv:2604.09815*, 2026.

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.

Xuhao Hu, Xi Zhang, Haiyang Xu, Kyle Qiao, Jingyi Yang, Xuanjing Huang, Jing Shao, Ming Yan, and Jieping Ye. Toolcua: Towards optimal gui-tool path orchestration for computer use agents. *arXiv preprint arXiv:2605.12481*, 2026.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Jakub Lála, Odhran O'Donoghue, Aleksandar Shtedritski, Sam Cox, Samuel G Rodriques, and Andrew D White. Paperqa: Retrieval-augmented generative agent for scientific research. *arXiv preprint arXiv:2312.07559*, 2023.

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. *arXiv preprint arXiv:2509.15738*, 2025.

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. In *International Conference on Learning Representations*, volume 2026, pp. 62707–62766, 2026.

Junting Lu, Zhiyang Zhang, Fangkai Yang, Jue Zhang, Lu Wang, Chao Du, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. Axis: Efficient human-agent-computer interaction with api-first llm-based agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7711–7743, 2025.

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.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pp. 625–635. Springer, 2021.

Markus Neteler, M Hamish Bowman, Martin Landa, and Markus Metz. Grass gis: A multi-purpose open source gis. *Environmental Modelling & Software*, 31:124–130, 2012.

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

OpenAI. GPT-5.6 System Card. https://deploymentsafety.openai.com/gpt-5-6, July 2026b.

Eric F Pettersen, Thomas D Goddard, Conrad C Huang, Elaine C Meng, Gregory S Couch, Tristan I Croll, John H Morris, and Thomas E Ferrin. Ucsf chimerax: Structure visualization for researchers, educators, and developers. *Protein science*, 30(1):70–82, 2021.

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.

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

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026b. URL https://qwen.ai/blog?id=qwen3.8.

Ram Ramrakhya, Andrew Szot, Omar Attia, Bogdan Mazoure, Anh Nguyen, Yuhao Yang, Zhe Gan, Harsh Agrawal, and Alexander Toshev. Scaling synthetic task generation for agents via exploration. In *International Conference on Learning Representations*, volume 2026, pp. 123630–123660, 2026.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 3505–3506, 2020.

Aditi Singh, Abul Ehtesham, Saket Kumar, Tala Talaei Khoei, and Athanasios V Vasilakos. Agentic retrieval-augmented generation: A survey on agentic rag. *arXiv preprint arXiv:2501.09136*, 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.

Qiushi Sun, Zhoumianze Liu, Chang Ma, Zichen Ding, Fangzhi Xu, Zhangyue Yin, Haiteng Zhao, Zhenyu Wu, Kanzhi Cheng, Zhaoyang Liu, et al. Scienceboard: Evaluating multimodal autonomous agents in realistic scientific workflows. In *International Conference on Learning Representations*, volume 2026, pp. 75694–75731, 2026.

Susheel Suresh, Hazel Mak, Shangpo Chou, Fred Kroon, and Sahil Bhatnagar. Agenticrag: Agentic retrieval for enterprise knowledge bases. *arXiv preprint arXiv:2605.05538*, 2026.

Weihao Tan, Wentao Zhang, Xinrun Xu, Haochong Xia, Ziluo Ding, Boyu Li, Bohan Zhou, Junpeng Yue, Jiechuan Jiang, Yewen Li, et al. Cradle: Empowering foundation agents towards general computer control. *arXiv preprint arXiv:2403.03186*, 2024.

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.

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

Jinbiao Wei, Yilun Zhao, Kangqi Ni, and Arman Cohan. Anchor: Branch-point data generation for gui agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17031–17047, 2026.

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, 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.

Weimin Xiong, Shuhao Gu, Bowen Ye, Zihao Yue, Lei Li, Feifan Song, Sujian Li, and Hao Tian. Video2gui: Synthesizing large-scale interaction trajectories for generalized gui agent pretraining. *arXiv preprint arXiv:2605.14747*, 2026.

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.

Yuhao Yang, Zhen Yang, Zi-Yi Dou, Anh Nguyen, Keen You, Omar Attia, Andrew Szot, Michael Feng, Ram Ramrakhya, Alexander Toshev, et al. Ultracua: A foundation model for computer use agents with hybrid action. *arXiv preprint arXiv:2510.17790*, 2025.

Shi Yu, Chaoyue Tang, Bokai Xu, Junbo Cui, Junhao Ran, Yukun Yan, Zhenghao Liu, Shuo Wang, Xu Han, Zhiyuan Liu, et al. Visrag: Vision-based retrieval-augmented generation on multi-modality documents. In *International Conference on Learning Representations*, volume 2025, pp. 21074–21098, 2025.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, et al. Osworld2. 0: Benchmarking computer use agents on long-horizon real-world tasks. *arXiv preprint arXiv:2606.29537*, 2026.

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 multi-modal web tutorials. *arXiv e-prints*, pp. arXiv–2504, 2025a.

Chaoyun Zhang, He Huang, Chiming Ni, Jian Mu, Si Qin, Shilin He, Lu Wang, Fangkai Yang, Pu Zhao, Chao Du, et al. Ufo2: The desktop agentos. *arXiv preprint arXiv:2504.14603*, 2025b.

Chaoyun Zhang, Liqun Li, Shilin He, Xu Zhang, Bo Qiao, Si Qin, Minghua Ma, Yu Kang, Qingwei Lin, Saravan Rajmohan, et al. Ufo: A ui-focused agent for windows os interaction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 597–622, 2025c.

Weinan Zhang, Junwei Liao, Ning Li, Kounianhua Du, and Jianghao Lin. Agentic information retrieval. *arXiv preprint arXiv:2410.09713*, 2024.

Wenchuan Zhang, Jingru Guo, Hengzhe Zhang, Penghao Zhang, Jie Chen, Shuwan Zhang, Zhang Zhang, Yuhao Yi, and Hong Bu. Patho-agenticrag: towards multimodal agentic retrieval-augmented generation for pathology vlms via reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29921–29929, 2026.

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*, 2025d.

Jun Zhao, Can Zu, Xu Hao, Yi Lu, Wei He, Yiwen Ding, Tao Gui, Qi Zhang, and Xuan-Jing Huang. Longagent: achieving question answering for 128k-token-long documents through multi-agent collaboration. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 16310–16324, 2024.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. Gpt-4v (ision) is a generalist web agent, if grounded. *arXiv preprint arXiv:2401.01614*, 2024a.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 3: system demonstrations)*, pp. 400–410, 2024b.

Hongbin Zhong, Fazle Faisal, Luis França, Tanakorn Leesatapornwongsa, Adriana Szekeres, Kexin Rong, and Suman Nath. Actionengine: From reactive to programmatic gui agents via state machine memory. *arXiv preprint arXiv:2602.20502*, 2026.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

A Action Space and Code Definitions

Collection and evaluation share an OSWorld-style execution interface: GUI operations are expressed as PyAutoGUI commands, supplemented by application-specific `CODE` commands and task-control tokens. When constructing training trajectories, we converts collected actions to Qwen-style JSON tool calls; the trained model emits this same format at inference. The evaluation adapter then translates these calls back into executable environment actions.

In the model representation, the agent interacts through a single function, `computer_use`, whose JSON arguments contain an `action` field and the corresponding parameters in Table 2. The interface exposes 14 GUI and task-control actions, plus an application-specific `code` action when enabled. ScienceBoard uses this hybrid interface except for Celestia[1], which uses GUI actions only; OSWorld also uses GUI actions only.

Table 2: Model-facing action space in Qwen tool-call format and its environment execution semantics. Parameters are fields in the arguments of `computer_use`; `coordinate` is an (x, y) pair.

Action	Parameters	Execution semantics
`key`	`keys` (list)	Press a single key or a key combination, releasing the keys afterwards.
`key_down`	`key` (string)	Hold a key without releasing it.
`key_up`	`key` (string)	Release a held key.
`type`	`text` (string)	Enter text using `pyautogui.typewrite`.
`mouse_move`	`coordinate`	Move the cursor to the specified location.
`left_click`	`coordinate`	Click the left mouse button.
`right_click`	`coordinate`	Click the right mouse button.
`middle_click`	`coordinate`	Click the middle mouse button.
`double_click`	`coordinate`	Double-click the left mouse button.
`left_click_drag`	`coordinate`	Drag from the current cursor location to the target; default duration is 0.5 seconds.
`scroll`	`pixels` (number)	Pass the value to `pyautogui.scroll`; positive values scroll up and negative values scroll down.
`hscroll`	`pixels` (number)	Execute the same ordinary scroll operation as `scroll`.
`wait`	`time` (number)	Emit the environment's `WAIT` control token.
`terminate`	`status`	Emit `DONE` for `success` or `FAIL` for `failure`.
`answer`	`text` (string)	Submit an answer as `ANS <text>`.
`code`	`cmd` (string)	Dispatch `CODE <cmd>` to the application adapter.

The default coordinate convention is a normalized 1000×1000 grid. For a native screenshot of width W and height H, the adapter maps (x, y) to $(\lfloor xW/1000 \rfloor, \lfloor yH/1000 \rfloor)$. Click actions without coordinates operate at the current cursor position. Although the schema names the scroll argument `pixels`, the adapter forwards it directly to PyAutoGUI as a scroll amount without pixel conversion. Likewise, `wait` exposes a `time` argument but emits only `WAIT`, leaving the waiting duration to the environment. The `terminate` status is the agent's completion declaration, not an independently computed task-success score.

CODE actions provide application-specific command and file-editing primitives within the hybrid action space defined in §3.3. Table 3 lists their syntax and semantics, following the application interfaces. In a tool call, `cmd` contains the command without the `CODE` prefix; the evaluation adapter adds this prefix when dispatching the action. For example, arguments {`"action":"code","cmd":"open 251d"`} produce `CODE open 251d` for ChimeraX.

For file-editing commands, paths cannot contain spaces, and only the final argument may contain spaces or newlines, except for `replace`, whose source and destination strings can each be enclosed in double quotes. Line indices start at 1. The `insert` operation strips leading and trailing whitespace from its text argument unless that argument is enclosed in double quotes, which preserves indentation. These quotes are argument delimiters and do not support escape sequences.

[1] `https://celestiaproject.space/`

Table 3: CODE action definitions. Angle brackets denote arguments supplied by the agent. File operations are shared by Lean 4 and TeXstudio; TeXstudio additionally supports forwarding commands to a running ChimeraX or KAlgebra instance.

Application	Syntax	Operation
ChimeraX	`CODE <command>`	Execute a native ChimeraX command.
GRASS GIS	`CODE gcmd <cmdline>`	Execute a console command and display its output in the Tool window.
	`CODE switch <location> <mapset>`	Switch to the specified location and mapset under `/home/user/grassdata`.
	`CODE load <layer>`	Load the named layer from the current mapset.
KAlgebra	`CODE calc <expr>`	Submit an expression to the Calculator.
	`CODE 2d <expr>`	Add a two-dimensional curve.
	`CODE 3d <expr>`	Set the displayed three-dimensional surface.
Editors	`CODE read <path>`	Read the complete file and return its text in the next message.
	`CODE overwrite <path> <text>`	Replace the entire file contents with the supplied text.
	`CODE insert <path> <index> <line>`	Insert text at the specified 1-based line position.
	`CODE replace <path> <src> <dst>`	Replace the first occurrence of the source string with the destination string.
	`CODE delete <path> <index>`	Delete the line at the specified 1-based position.
TeXstudio	`CODE chimerax <command>`	Forward a native command to ChimeraX.
	`CODE kalgebra <operation> <arg>`	Forward a `tab`, `calc`, `2d`, or `3d` operation to KAlgebra with the semantics above.

The `overwrite` operation accepts multiline text, while `replace` changes only the first matching occurrence.

ChimeraX. We use the built-in RESTful remote-control interface of ChimeraX (Pettersen et al., 2021). The adapter submits the native command following the `CODE` prefix to this interface, allowing the agent to issue commands such as `CODE open 251d` directly to the running application.

GRASS GIS. We extend the wxGUI logic of GRASS GIS (Neteler et al., 2012) with an interface for executing console commands. ScienceBoard (Sun et al., 2026) directly invokes `RunCommand` to execute instructions silently. Our implementation instead uses Python to simulate entering the command into the console and pressing the execution button, so that the command and its output appear on screen in the Tool window. This makes the execution output available in subsequent screenshots. The adapter also exposes operations for switching the active location and mapset and loading a named layer.

KAlgebra. KAlgebra provides calculator and 2D/3D plotting functionality (kde). We extend the internal server introduced by ScienceBoard for KAlgebra. In addition to the existing 2D and 3D graph operations, we add `calc` to submit expressions to the Calculator. The adapter routes calculation and graph commands to the corresponding server endpoints, updating the running application.

Lean 4 and TeXstudio. We add file read/write interfaces for Lean 4 (Moura & Ullrich, 2021) and TeXstudio[2] without modifying either application. Both adapters support reading, overwriting, inserting, replacing, and deleting file contents, enabling direct edits to Lean source files and LaTeX documents. TeXstudio reuses the same file-operation implementation as Lean and additionally

[2] `https://www.texstudio.org/`

forwards prefixed ChimeraX and KAlgebra commands to their respective adapters when those applications are running. Both applications monitor file changes in real time and automatically reflect them in their respective editors.

B IMPLEMENTATION OF VM

Each environment runs QEMU (Bellard et al., 2005) (`qemu-system-x86_64`) inside an Ubuntu 22.04 Docker container with access to the host's `/dev/kvm` for hardware acceleration. The guest also uses Ubuntu 22.04, installed with legacy BIOS boot. The default configuration allocates four virtual CPUs and 8 GiB of RAM, with a 64 GiB virtual disk in qcow2 format. QEMU uses the `pc-q35-4.2` machine type, the host CPU model, and VirtIO disk and network interfaces. The container entrypoint launches QEMU under `tini`; mounted directories `/storage` and `/qmp` hold disk images and the QEMU Machine Protocol (QMP) socket, respectively. The controller uses QMP to manage VM execution and checkpointing. The guest-side interaction server is accessible through forwarded port 5000, while port 5900 provides remote desktop access.

QXL with SPICE is available for initial configuration and debugging. For trajectory collection and state recovery, we launch QEMU with `-display none -vga none`, since QXL graphics state could not be reliably restored in our snapshot workflow. Inside the guest, Xvfb provides an X11 display at `:1` with a resolution of 1920×1080 and 24-bit color depth. GNOME Flashback with the Metacity window manager runs on this display, and `x11vnc` exposes it on port 5900. The desktop session is enabled through the `xtop` user service. This configuration places the virtual framebuffer and desktop processes in guest memory, allowing them to be captured together with application state without relying on QXL restoration. We enable toolkit accessibility and configure the guest for unattended interaction by disabling screen blanking, screen locking, and automatic sleep.

A checkpoint is a pair $C = (Q, M)$: Q is a preserved qcow2 disk image, including its backing chain, and M is a QEMU migration stream containing guest memory and execution state, including CPU and device state. Thus, the runtime snapshot is stored in a separate migration file. By default, disk writes use a copy-on-write overlay over a base image. Once an image becomes part of a checkpoint, subsequent execution writes to a new child overlay, preserving the checkpoint's disk state. A writable overlay belongs to a single running VM, while preserved backing images can be reused across restorations. The launcher also supports direct writes and temporary QEMU snapshot mode, but our checkpoint workflow uses explicit qcow2 overlays paired with migration files.

The VM lifecycle implements the state recovery in §3.4 as follows:

1. **Initial launch.** Create a writable overlay backed by the base qcow2 image, boot the guest using this overlay, and establish the QMP connection.
2. **Save.** Pause the guest with QMP `stop` and export its runtime state to migration file M. After the export completes, terminate the current QEMU process and close the QMP connection. Preserve the current disk state as Q by retaining its overlay and backing chain, producing checkpoint (Q, M). Create a new writable overlay O backed by Q, restart QEMU with O, and import M to continue from the saved state.
3. **Load.** Terminate the current VM and close its QMP connection, then discard its working overlay. For the requested checkpoint (Q, M), create a fresh overlay O' backed by Q and start QEMU with the matching migration file M. The launcher imports this file through QEMU's `-incoming` option. Subsequent execution modifies only O', allowing repeated exploration from the same checkpoint.
4. **Final exit.** Terminate the VM, close the QMP connection, and discard its remaining working overlay. Retain saved checkpoint images, their backing chains, and migration files for later restoration.

Pairing the preserved disk image with its corresponding migration file ensures that restored memory and disk state refer to the same checkpoint. After loading the nearest pivotal checkpoint, the collector replays the remaining actions and verifies recovered screenshots as described in §3.4.

Whenever a new VM is loaded, the controller registers cleanup handlers for normal program exit and termination signals such as `SIGINT`. A background polling loop additionally detects abandoned containers. Every `ACTIVE_TIME` interval, it updates the current timestamp in

`/var/log/heartbeat` inside the containers it manages and checks this file in containers matching the `os-aks` naming convention. Containers whose heartbeat age exceeds `INACTIVE_TIME` are forcibly stopped. This complements exit handlers when a collector terminates without completing normal cleanup.

C DATASET STATISTICS

The collected corpus contains 54,938 state–action training samples from 10,919 trajectories across 13 applications. Table 4 reports the application-level breakdown. We count each serialized training record as one sample and each terminal record as the end of one trajectory. Trajectory lengths include the terminal step. Counts refer to the corpus before deduplication, including 173 exactly duplicated records. There are 9,862 distinct instruction strings, deduplicated within each application, and 36,226 distinct referenced screenshot paths; different trajectories may share instructions or screenshots.

Table 4: Application-level dataset statistics. Samples and mean trajectory lengths include terminal steps. The overall mean is weighted by trajectory count.

Application	Trajectories	Training samples	Mean steps
Celestia	1,120	7,401	6.61
ChimeraX	1,411	6,458	4.58
GRASS GIS	1,561	5,487	3.52
KAlgebra	1,636	7,841	4.79
Lean 4	1,486	7,546	5.08
TeXstudio	591	2,763	4.68
LibreOffice Calc	428	2,676	6.25
Chrome	420	2,286	5.44
GIMP	409	2,279	5.57
LibreOffice Impress	455	2,541	5.58
Nautilus	514	2,522	4.91
VS Code	412	2,459	5.97
LibreOffice Writer	476	2,679	5.63
Total	10,919	54,938	5.03

Tree exploration contributes 5,739 trajectories and 32,250 training samples, while linear rollouts contribute 5,180 trajectories and 22,688 samples, including the Lean proof trajectories described in §D. The six specialized applications contribute 37,496 samples. The remaining 17,442 samples (31.75%) come from these general-purpose applications: LibreOffice, Chrome, GIMP, Nautilus, and VS Code.

Ordinary tree exploration and linear rollouts use a depth limit of $D = 16$; Lean reference-proof replay is exempt. During conversion, paths with leaf depth greater than 40 are discarded before terminal training records are appended. In the resulting corpus, trajectory lengths, including terminal steps, have a mean of 5.03 and a median of 4; the 90th and 95th percentiles are 9 and 12, respectively, and the observed maximum is 40. Each training sample contains the current screenshot and up to four historical screenshots, with a mean of 3.15 images per sample. These repeated context images are counted separately from the distinct screenshot paths reported above.

Action statistics count only the target response of each training sample, excluding tool calls repeated in its conversation history. The corpus contains 55,471 target tool calls: 30,444 GUI operations, 13,037 code operations, and 11,990 task-control operations (`wait`, `answer`, or `terminate`). The processing pipeline also accepts responses with multiple tool calls, so call counts can exceed sample counts. Domain-specific code appears in 23.73% of training samples and accounts for 29.98% of GUI and code operations combined, excluding task-control operations from this denominator.

D TRAJECTORIES COLLECTION FOR LEAN 4

Lean 4 poses additional challenges for automatic trajectory collection. Limited familiarity with Lean 4 syntax, tactics, and library conventions makes correct proof generation difficult for the collection models, and entering and editing proofs through a GUI adds further interaction overhead. Generating theorem statements from scratch also provides no assurance that the resulting tasks admit proofs within the available interaction budget. To efficiently build supervision for automated theorem proving (ATP), we therefore construct desktop interaction trajectories from existing Lean proofs. We obtain 1,608 problems with proof code from the Formal Math Autoformalization Dataset (Commons, 2026) and use their proofs to guide deterministic linear rollouts.

Before collection, we compile each theorem together with its reference proof and retain only those that pass compilation. For each retained problem, we preserve the imports, declarations, and target theorem statement, and replace the proof body with an indented `sorry` placeholder. The theorem statement with this unfinished proof serves as the task instruction. The collector launches a VM with VS Code open on the Lean project and stages the problem in `/home/user/sci/Sci/Basic.lean`, recording the initial screenshot. This setup fixes both the task and its reference solution before collection, while the recorded trajectory shows how to construct that solution through the same file-editing code interface used during evaluation.

As a concrete example, the following shows a reference proof retained for replay after preprocessing:

Lean 4 Reference Proof

```
import Mathlib

theorem div_add_div_pos_ge_two (a b : ℝ) (ha : 0 < a) (hb : 0 < b) :
    a / b + b / a ≥ 2 :=
by
  rw [ge_iff_le, ← sub_nonneg]
  have hab : (0 : ℝ) < a * b := mul_pos ha hb
  have key : a / b + b / a - 2 = (a - b) ^ 2 / (a * b) := by
    field_simp
    ring
  rw [key]
  exact div_nonneg (sq_nonneg _) hab.le
```

The collector replays the nonempty proof lines in order. For the example above, its first action is `CODE replace <path> sorry <first-line>`; subsequent actions use `CODE insert <path> <index> <line>` at the corresponding 1-based line positions. Subsequent lines retain their indentation, including nested tactic blocks. In the example above, the first action replaces `sorry` with `rw [ge_iff_le, ← sub_nonneg]`. Later actions insert the positivity fact, the auxiliary algebraic identity and its nested proof, and the concluding tactics. Each action adds one source line, which need not constitute a complete tactic block or a complete proof on its own. After each edit, the collector waits four seconds for the editor to update, saves a screenshot, and records the command and waiting time as a transition. This captures intermediate desktop states throughout proof construction rather than only the completed source file.

After replay, we compile the resulting Lean file; only trajectories that pass compilation are marked complete and retained. Intermediate edits may leave proofs incomplete, so compilation is not required at every step. The retained paths are converted into Qwen-format training conversations using the procedure in §E. For Lean, the converter also reconstructs the source file after each edit and includes its current contents in the corresponding observation, allowing the model to condition on both the screenshot and the partially constructed proof. The proof recorder leaves action descriptions and reasoning fields empty, so these demonstrations primarily supervise the file-editing tool calls. After compilation checks, replay, and trajectory filtering, the final dataset contains 1,486 proof trajectories and 7,546 step-level training samples, including terminal steps, from the original 1,608 problems.

Table 5: Training settings for the main Qwen3.5-27B experiment on the full corpus. Other experiments use similar training settings with minor adjustments. The effective global batch size is the per-device batch size multiplied by the gradient accumulation steps and the number of data-parallel workers.

Setting	Qwen3.5-27B
Fine-tuning method	Full-parameter SFT
DeepSpeed ZeRO stage	2
Hardware	16× H200
Precision	BF16
Maximum sequence length	32,768
Per-device batch size	1
Gradient accumulation steps	8
Effective global batch size	128
Learning rate	1×10^{-5}
Learning-rate schedule	Cosine
Warmup steps	100
Training epochs	1
Validation split	None
Checkpoint saving	End of epoch

E TRAINING AND EVALUATION DETAILS

We fully fine-tune Qwen3-VL-8B and Qwen3.5-27B on $8\times$ and $16\times$ H200 GPUs, respectively, using Qwen3-style tool-call conversations. For ScienceBoard applications with code support, domain-specific code actions are represented as {`"action":"code", "cmd":"..."`} arguments to `computer_use`.

Training conversations are constructed from the collected exploration paths using `traj.py`, following the representation conversion in §A. The converter parses supported PyAutoGUI calls through Python's abstract syntax tree and maps their function names and arguments to `computer_use` actions. Native screenshot coordinates (u, v) are normalized to $(\lfloor 1000u/W \rfloor, \lfloor 1000v/H \rfloor)$ for screenshot dimensions (W, H); the optional relative-coordinate input mode instead scales unit coordinates by 1000. For example, `pyautogui.click(960,540)` on a 1920×1080 screenshot becomes {`"action":"left_click","coordinate":[500,500]`}. The `CODE` and `ANS` prefixes become the `code` and `answer` actions with their payloads preserved. Each action is serialized as a JSON call to `computer_use` inside `<tool_call>` tags.

The converter selects paths whose task is marked complete and whose leaf depth does not exceed 40, omits actions that cannot be translated, and appends a successful `terminate` action paired with the final screenshot. Each step yields a conversation containing the task instruction, the current screenshot, and up to four recent screenshot–response pairs; older steps are summarized by their action descriptions. Application-specific responses include the recorded `Thought`, `Action`, and tool call, whereas the generic desktop mode uses only `Action` and the tool call. The resulting conversations are written to JSONL with their image references for supervised fine-tuning. At evaluation time, `qwen.py` translates model-generated calls back to the execution interface using coordinate quantization and the action-specific conventions in Table 2.

We report the training configuration for the main Qwen3.5-27B experiment using the full corpus. We perform full-parameter supervised fine-tuning with LLaMA-Factory (Zheng et al., 2024b) using the state–action conversations described above. The run concatenates the selected training datasets and uses the `qwen3_vl` conversation template with the `qwen` tool format. The template option `enable_thinking` is set to `false`; the supervised response content follows the conversation construction described above. Training uses DeepSpeed (Rasley et al., 2020) and BF16 precision, with a maximum sequence length of 32,768 tokens. Table 5 summarizes the settings for this main experiment.

This run uses DeepSpeed ZeRO stage 2, with partitioned all-gather, reduce-scatter, communication overlap, contiguous gradients, and round-robin gradient handling enabled. The all-gather and re-

duction bucket sizes are each set to 2×10^9 elements. Batch sizes, gradient accumulation, gradient clipping, and BF16 enablement are configured as `auto` for resolution by the training framework; the configuration does not specify optimizer or parameter offloading. The activation-checkpointing configuration enables activation partitioning and contiguous-memory optimization, disables CPU checkpointing and checkpoint-boundary synchronization, and leaves the checkpoint count unspecified. Activation-checkpoint profiling and wall-clock breakdown are disabled.

The main Qwen3.5-27B run starts from its pretrained backbone without resuming a training checkpoint. We train for one epoch, save model weights at the end of the epoch, and use the resulting checkpoint for downstream evaluation. No validation split is held out, and evaluation during training is disabled. With one sample per device and eight gradient accumulation steps, the 16-GPU run has an effective global batch size of 128.

We evaluate on ScienceBoard and OSWorld using the Qwen3-style tool-call interfaces described in the main text. ScienceBoard exposes the hybrid GUI–Code action space, with domain-specific code operations defined in §A; Celestia uses GUI actions only. OSWorld uses GUI actions only. We report task success rates on both benchmarks.

At each interaction step, the evaluation adapter receives the task instruction and current desktop screenshot. Its default history window contains the four most recent observation–response pairs, including their screenshots and full model responses. Earlier steps are represented by a numbered list of their action descriptions, without their screenshots or full responses. The current screenshot is appended to this context, together with the latest execution output when supplied by the environment. For Lean, the current contents of `/home/user/sci/Sci/Basic.lean` are additionally included as authoritative source text when available in the observation. Historical execution outputs and historical Lean source text are not separately replayed in the conversation. The adapter's reset routine clears its stored history between tasks.

Screenshots are resized before being encoded as PNG images. The resizing routine approximately preserves the aspect ratio while rounding each dimension to a multiple of 32. It uses a maximum image area of $16 \times 16 \times 4 \times 12{,}800 = 13{,}107{,}200$ pixels and first limits the longest side to 8,192 pixels. Mouse coordinates are mapped back to the original desktop resolution using the normalized convention in §A.

The system prompt identifies the application, states its operating constraints, and supplies the `computer_use` schema. These constraints include preserving the active ChimeraX session, using a single GRASS GIS map display, avoiding parametric 2D plots in KAlgebra, preserving the Lean proof prefix and prohibiting `sorry`/`admit`, and avoiding unrelated edits in TeXstudio. The model is prompted to return a brief `Thought`, an `Action` description, and one JSON function call enclosed in `<tool_call>` tags. The adapter parses the call and converts it into PyAutoGUI operations or environment control commands, including application code commands. Relative to the original Qwen adapter, this implementation adds application-specific prompting and code dispatch, held-key operations, explicit answer submission and failure termination, and execution-output and Lean-source feedback.

F Prompts for Sampling

The following templates are used for instruction proposal, action sampling, and transition/completion verification. Braced names denote runtime substitutions; doubled braces in the Python source are rendered as literal JSON braces here. Screenshots are attached as images alongside the user messages: one current screenshot for proposal and action sampling, and the before/after pair for verification. The optional `reference` field contains reference instructions, and `examples` supplies a JSON array with `count` entries. The `primitive` field supplies domain-specific code definitions. The `output` field appends the latest command output when available. Missing optional inputs follow the empty-string or `<ABSENT>` conventions in the implementation.

All actions must execute without errors. Without an instruction, the reward model checks GUI transition validity, while code actions require only error-free application execution. The reward model's `succ` field indicates GUI transition validity in this mode. With an instruction, it indicates task completion for both action types; `succ=false` means the rollout continues until completion or the depth limit.

F.1 Instruction Proposal

Instruction Proposal: System Prompt

```
You are an autonomous goal-proposal agent operating on a Ubuntu
    desktop using a screenshot and optional documents.
Given a screenshot, history instruction and some documents (if
    provided), propose {count} plausible user goals (inst) that a
    human might want to achieve next.
All proposed goals should be based on what is visible in the
    screenshot or explicitly stated in the documents or instruction.
    Do not assume hidden state, background tasks, or unstated intent.

Goal rules
Each goal must be actionable, realistic, and meaningfully different
    from the others.
Do not invent UI elements, files, or workflows not supported by the
    inputs.
Newly generated goals must not be identical to, or trivial rephrasings
     of, any previously used history instruction.
Use clear references instead of vague ones, e.g. we prefer 'draw a
    curve of y=x^2+1' rather than 'draw any of quadratic curves'.

{reference}
Output format
You must output exactly the following JSON wrapped in backticks. Do
    not include any extra text outside the array.
```json
{examples}
```
```

Instruction Proposal: User Prompt

```
Current observation:

[SCREENSHOT]
The provided image is a screenshot of the current Ubuntu desktop state

[DOCUMENTATION]
{document}

[HISTORY INSTRUCTION]
{instruction}
```

F.2 Action Sampling

Action Sampling: System Prompt

```
You are an autonomous GUI interaction agent operating on a Ubuntu
    desktop via visual observation and accessibility information.

Your task is to propose ONE plausible next user action that a real
    human user might reasonably perform next, based on the information
     provided:
- a screenshot of the current desktop,
- an accessibility (a11y) tree describing visible UI elements,
- (optional) a software-related document,
- (optional) an instruction,
```

```
- (optional) a list of reference commands, representing commands that
    have already been checked.
- (optional) a list of history commands with description, representing
     commands that have already been executed.

You are not executing the action, only proposing it. Your proposal
    must be grounded strictly in what is visible or explicitly
    provided. Do not assume hidden UI state, background tasks, or an
    implicit goal unless it is clearly implied by the document or
    instruction.

Document & Instruction Handling:
If instruction is provided, the proposed action MUST follow
    instruction.
If instruction is not provided but doc is provided, the proposed
    action SHOULD be guided by and consistent with doc.
If neither doc nor instruction is provided, choose a conservative,
    broadly applicable exploratory action based only on the screenshot
     and a11y tree, favoring actions that expand future interaction
    possibilities (e.g., opening a menu, revealing additional panels
    or controls, focusing the main window, scrolling a primary panel).
Never invent steps, commands, UI elements, or workflows that are not
    supported by the screenshot, a11y tree, doc, or instruction.

Action Form & Primitives:
The action must be either a single pyautogui sentence or a single
    predefined code primitive. Do not chain actions, do not combine
    primitives, and do not include conditionals or explanations.
CODE primitives are first-class actions and should be strongly
    preferred when the document or instruction references command-line
     usage, system-level operations, or scripted workflows.
CODE primitives should also be preferred when a UI interaction would
    be fragile, slow, or ambiguous, or when a primitive better matches
     the intent expressed in doc or instruction.
The following describe the available code primitives specified to
    current desktop environments:
{primitive}

Environment & Timing Constraints:
The computer is slow and UI responses may take a long time, so acting
    too quickly is risky while waiting longer than necessary is
    acceptable.
Wait times must always be multiples of 15 seconds such as 15, 30, 45,
    or 60. Average safe waiting time is approximately 30 seconds.

Reference Command Rule:
A reference command entry includes the full command, its associated
    wait time, and whether it was accepted or rejected.
You MUST NOT output a command that is identical to any reference
    command, nor one that is only a trivial variation of it (e.g.,
    same action with slightly different coordinates, timing, wait
    values, or other inconsequential parameter changes). Changing only
     coordinates, timing, wait, or selecting the same UI element in a
    slightly different way is not considered a different intent.
Your proposed command must represent a meaningfully different intent
    or interaction compared to all reference command entries; if no
    such meaningful action exists, choose a different UI element or
    interaction direction instead of repeating or slightly modifying a
     previous one.
Attention: some rejected reference commands may have failed only
    because the wait time was too short. If the action involves
    complex UI operations, loading, or data downloads, repeating the
    same command with a longer wait time is allowed in this specific
```

```
    case, but it should be used sparingly and only when there is
    strong reason to believe insufficient waiting was the cause of
    failure.

History Command Rules:
A history command entry includes all commands executed from the moment
     the software was launched up to the current state, and is
    provided strictly for reference.
Attention, history command represents previously executed commands,
    while reference command represents potential commands that can be
    executed from the current state and have not yet been executed.
You MUST carefully review recently executed commands and avoid
    reissuing actions that have already been performed.
If the same command has been repeatedly executed in the most recent
    steps, you MUST NOT execute it again. Instead, choose an
    alternative interaction method, a different UI element, or a
    different operational direction to achieve the goal.

Robustness & Safety:
Prefer reversible and non-destructive actions.
Avoid closing applications, deleting data, or confirming irreversible
    dialogs unless explicitly required by doc or instruction.
Avoid actions that rely on pixel-perfect coordinates, rapid sequences,
     or precise timing.
Avoid stating your references or source in the 'reason' section, such
    as 'According to the document, ...'.

Output Format (STRICT WITHOUT EXTRA TEXT):
You must output EXACTLY ONE JSON block without duplicating any given
    reference commands.
You must output EXACTLY the following JSON wrapped in backticks.
```json
{
  "cmd": "<CODE primitive OR pyautogui sentence>",
  "wait": <integer_seconds>,
  "action": "<brief natural language description for cmd>",
  "reason": "<reasoning & thinking for the action proposed>"
}
```
```

Action Sampling: User Prompt

```
Current observation:

[SCREENSHOT]
The provided image is a screenshot of the current Ubuntu desktop state

[ACCESSIBILITY TREE]
{a11y_tree}

[DOCUMENTATION]
{document}

[REFERENCE COMMANDS]
{ref_cmd}

[HISTORY COMMANDS]
{exe_cmd}{output}

[INSTRUCTION]
{inst}
```

F.3 Transition and Completion Verification

Transition and Completion Verification: System Prompt

```
You are a GUI state observer and execution verifier operating on a
    Ubuntu desktop.

You are given:
- a previous screenshot and a current screenshot
- the executed command with the wait time after execution
- an optional instruction.
Your task is to produce two things: a clear, concise, human-readable
    description of the current visible UI state, and a boolean
    judgment of whether the last command should be considered
    successful.

Role of Instruction:
If an instruction is provided, judge whether the current UI state
    satisfies it as a completed outcome; intermediate progress is
    insufficient.
If no instruction is provided, you must judge whether the command
    produced a meaningful, effective interaction, and in this mode you
     should be more aggressive about declaring failure when no real
    progress is visible.

Success & Failure Semantics When Instruction is Absent:
Success means that the current UI shows clear evidence that the
    command produced a real and meaningful interaction, such as
    opening a window, changing focus, selecting a control, revealing
    new content, or otherwise making observable progress that a
    reasonable user would interpret as the command having taken effect
    .
Failure means that visible error dialogs, crash messages, alerts, or
    obviously incorrect UI transitions appear, or that the command
    interacted with no meaningful UI element and resulted in no
    observable change or progress. Also, lack of visible progress,
    lack of interaction with real UI components, or failure to reach a
     plausible micro-goal implied by the command should be judged as
    failure rather than treated as acceptable ambiguity.

Success & Failure Semantics When Instruction is Present:
Success means that the current UI state satisfies the instruction as a
     completed outcome, meaning that the goal described by the
    instruction is already achieved and no further UI interaction
    would be required by a reasonable user to fulfill it.
Failure means that, based on the currently provided visual information
    , the instruction cannot be verified as successfully completed,
    including cases where the UI shows partial progress, intermediate
    states, or being \on the right path" but the explicit goal is not
    yet clearly achieved, as well as cases where visible error dialogs
    , crashes, or contradictory UI states appear.
If the current UI shows no visible change or interaction trace toward
    fulfilling the instruction, it must be considered failure.

Output Format (STRICT WITHOUT EXTRA TEXT):
You must output exactly the following JSON wrapped in backticks.
```json
{
  "obs": "<string describing the current UI state>",
  "succ": <true or false>
}
```
```

Transition and Completion Verification: User Prompt

```
Current observation:

[SCREENSHOT]
Two images successively provided are screenshots of the Ubuntu desktop
     state before & after executing the command below.

[PREVIOUS COMMAND]
{cmd}
and system has been waited for {wait}s.

[INSTRUCTION]
{inst}
```