OS-HELIX: UNIFYING GUI INTERACTION AND CODE EXECUTION IN NATIVE COMPUTER-USING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Real desktop automation is neither purely visual nor purely programmatic. Agents must understand user-facing interfaces while efficiently manipulating files, scripts, and system state. However, native GUI/code coordination cannot be obtained by prompt-level tool composition. In our preliminary experiments, naively adding a code tool degrades a GUI agent, revealing that GUI/code routing must be internalized through training on high-quality hybrid data. We present OS-Helix, an end-to-end trained multimodal computer-using agent that unifies GUI actions and code execution in one action space. To construct the required supervision, we introduce application–artifact composition, pairing desktop applications with authentic file artifacts to instantiate executable Ubuntu states, and a decoupled instruction/evaluator generation pipeline that produces environment-grounded tasks whose final states can be verified regardless of the solution modality. We cold-start the policy with SFT on filtered hybrid data and further improve it through online RL in live desktop environments. For RL, we propose Step-balanced GSPO, which aggregates importance weight by action step rather than raw assistant tokens to prevent long code actions from dominating heterogeneous-action optimization. On OSWorld-Verified, OS-Helix-27B and OS-Helix-35B-A3B achieve relative improvements of 35.0% and 33.7% over their matched Qwen3.5 hybrid baselines, while OS-Helix-27B-RL further improves by 42.9% to reach 66.44% success. Ablations provide evidence consistent with contributions from learned GUI/code coordination, scalable verifiable supervision, and action step balanced optimization, while demonstrating cross-platform transfer on WindowsAgentArena. We will release all research artifacts.

1 INTRODUCTION

The rapid progress of vision-language models (VLMs) (Team, 2026; Wang et al., 2025b; Team et al., 2026a; Anthropic, 2025; Zeng et al., 2026; OpenAI, 2025; Bai et al., 2026; Zou et al., 2026) has accelerated the development of computer-using agents (CUAs), which aim to execute open-ended tasks in human-facing digital environments. Most existing CUAs formulate digital automation as graphical interaction, where agents perceive screenshots, reason over interface states, and act through mouse and keyboard operations.

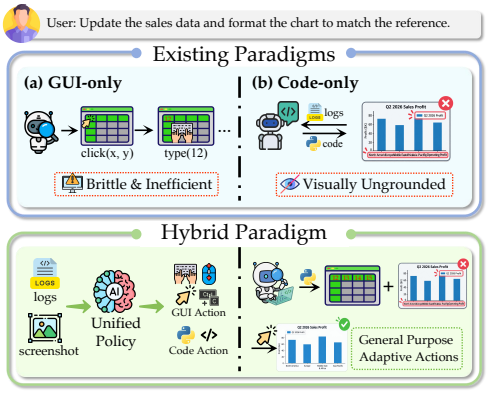

Figure 1: Overview of the GUI/code hybrid setting targeted by OS-Helix.

However, GUI-centric CUAs (Liu et al., 2026; Xue et al., 2026; Wang et al., 2026b; Zhang et al., 2026a; Luo et al., 2025; Cheng et al., 2024; Feizi et al., 2026) still face practical bottlenecks in realistic desktop workflows, where visual grounding can fail on small or ambiguous interface elements and fine-grained interaction can make structured state manipulation cumbersome even with batched actions. A complementary interface is programmatic control, through which command-line and code agents can inspect files, access system state, and execute compound operations directly through the

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.

shell (Yang et al., 2026e; 2024; Wang et al., 2025c). Yet code execution alone cannot fully address the long tail of desktop task scenarios that depends on visual layout, application-specific UI state, or software behavior exposed only through the interface. As illustrated in Figure 1, these complementary strengths motivate a single agent that coordinates GUI and code, where GUI provides general visual access to arbitrary applications while code provides efficient and precise manipulation of programmatic state.

Despite this complementarity, building a native hybrid GUI/code CUA remains an open problem. Simply attaching a code tool to a GUI model does not teach it when or why to use either modality. In our preliminary experiments, as shown in Table 14, Qwen3-VL-32B-Instruct (Bai et al., 2025) achieves over $\sim$34% success on OSWorld-Verified (Xie et al., 2025) with GUI actions alone, but falls to roughly $\sim$32% when code execution is naively exposed. Prompt-level composition can therefore disrupt an existing GUI policy rather than create coordinated behavior, making it essential to internalize hybrid coordination capabilities into the model through training on high-quality hybrid data. Several existing approaches have explored hybrid execution paradigms that combine GUI interaction with code-like actions. Framework-centric systems (Gonzalez-Pumariega et al., 2025; Hou et al., 2024; Yang et al., 2026a) rely on external harnesses to coordinate GUI, code, memory, and recovery; tool-call-based approaches (Jia et al., 2025; Yang et al., 2025b; Shi et al., 2026) expose programmatic operations through predefined application-specific interfaces; and model-fusion approaches (Wang et al., 2025a; Qin et al., 2025) combine complementary capabilities through predefined designs. While these systems expand the scope of hybrid computer use, they leave open how a single policy can learn task-grounded routing between GUI and code actions through hybrid training.

To address these challenges, we present OS-Helix, an end-to-end trained computer-using agent that unifies GUI interaction and code execution within a single policy.

On the infrastructure side, we introduce application–artifact composition, which combines versioned applications with authentic artifacts to instantiate diverse and reproducible desktop states. An **Exploration Agent** proposes instructions grounded in these states, while an independent **Evaluator Agent** constructs task-specific, primarily rule-based evaluators. Together, this pipeline supports scalable synthesis, filtering, and collection of high-quality GUI/code trajectories for model training, thereby providing the data foundation necessary to internalize hybrid coordination capabilities into the model.

On the training side, we first use filtered hybrid trajectories to internalize basic action semantics and modality routing, and then optimize the policy through online RL in executable desktop environments. In early RL experiments, we observe a steady rise in the proportion of code actions accompanied by a degradation in performance. We attribute this to the token-count asymmetry between short GUI actions and substantially longer code actions, which causes the optimization signal to be dominated by code steps. To address this, we propose **Step-balanced GSPO**, which constructs the sample-level importance ratio over action steps rather than raw assistant-token counts, enabling more balanced optimization of the two action modalities. These training components directly answer the open question of how a single policy can natively learn to coordinate GUI and code actions through hybrid training, yielding an agent that performs strongly and reliably across diverse desktop tasks.

In summary, our contributions are:

1) We formulate native GUI/code hybrid computer use as an end-to-end policy-learning problem and show through controlled experiments that merely exposing a GUI policy to code execution fails to yield reliable routing, motivating the need to internalize coordination through hybrid training.

2) We introduce application–artifact composition and a decoupled instruction/evaluator synthesis pipeline to construct diverse, reproducible, executable desktop states and generate high-quality hybrid data for internalizing GUI and code execution capabilities.

3) We propose Step-balanced GSPO for the agentic RL stage of native hybrid policy training, which prevents long code actions from dominating optimization, and enables a single policy to learn GUI and code coordination.

4) We will release the OS-Helix model family and research artifacts, including the desktop data-synthesis and RL environments, task and trajectory collection pipelines, approximately 9K generated

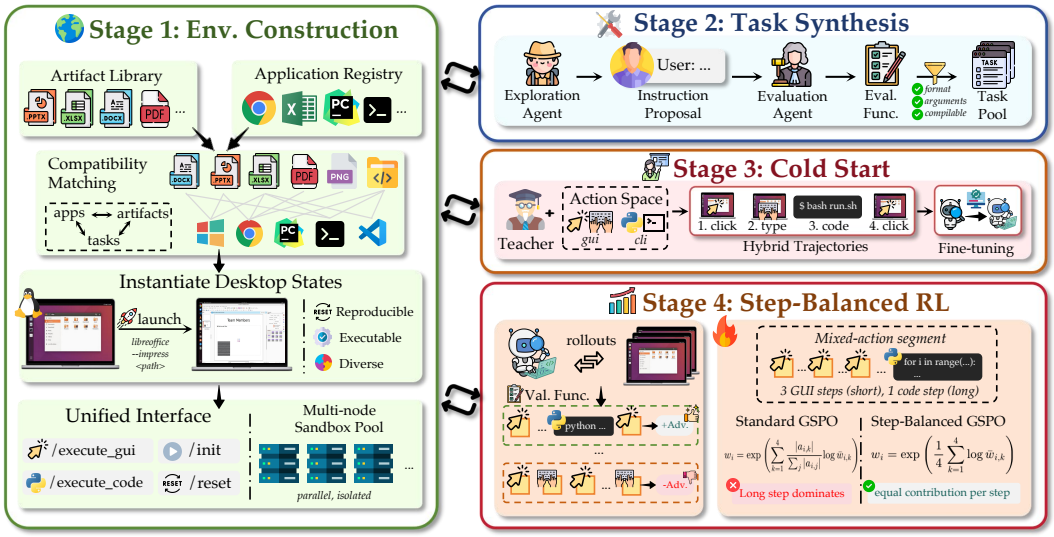

Figure 2: Overview of the OS-Helix pipeline. **Stage 1** constructs executable desktop environments by composing applications and artifacts. **Stage 2** explores each environment to synthesize instructions and evaluators, yielding verifiable tasks. **Stage 3** uses teacher-generated GUI/code trajectories for SFT cold start. **Stage 4** performs online agentic RL in live sandboxes.

instructions, 83K unfiltered SFT samples, and about 31K successful filtered SFT samples in the unified GUI/code action space.

2 OS-Helix

2.1 Overview

An overview of the OS-Helix pipeline is shown in Figure 2. OS-Helix is built around a simple premise: a real computer-using agent should treat GUI interaction and code execution as two complementary ways of changing the same digital state. Rather than attaching a shell tool to a GUI model at inference time, we construct a unified pipeline in which environments, instructions, trajectories, and reinforcement signals are all designed for hybrid GUI/code execution.

Formally, let q denote the user instruction, h_i the history context before step i, and o_i the current screen observation. The policy selects an action from a unified hybrid action space:

$$a_i \sim \pi_\theta(\cdot \mid q, h_i, o_i), \qquad a_i \in \mathcal{A}_{\text{hybrid}} = \underbrace{\mathcal{A}_{\text{GUI}} \cup \mathcal{A}_{\text{meta}}}_{\mathcal{A}_{\text{native}}} \cup \mathcal{A}_{\text{code}}.$$

Here, $\mathcal{A}_{\text{GUI}}$ denotes GUI actions such as clicking and typing; $\mathcal{A}_{\text{meta}}$ denotes episode-level meta-actions such as task termination; actions in $\mathcal{A}_{\text{code}}$ execute either Python or Bash directly in the operating system. Each code action runs in a fresh process, the stdout and stderr of code execution are appended to the current-turn context as a tool response for subsequent decision-making.

2.2 Data Infrastructure

The scale of computer-using agent training is ultimately bounded by the diversity of executable states from which meaningful tasks can arise. This is particularly important on desktops, where workflows are shaped jointly by applications, user artifacts, and persistent local state. We view authentic artifacts as a natural axis for expanding this state space, with artifact types and file formats providing a direct link to the applications that can operate on them. Based on this principle, we construct diverse executable desktop environments and synthesize grounded, verifiable tasks from the resulting states.

2.2.1 APPLICATION–ARTIFACT COMPOSITION FOR DESKTOP ENVIRONMENTS

Real-world desktop workflows are commonly grounded in existing artifacts that are read, edited, or transformed through applications. We therefore factor scalable desktop-environment construction into two reusable components: applications with fixed versions, launch procedures, and diverse initialization configurations, and artifacts carrying authentic content and persistent state. Their compatibility-aware composition produces executable desktop states from which realistic tasks can arise.

To instantiate such states at scale, we organize artifacts and applications into an *application–artifact compatibility graph*, whose edges are determined by the artifact types and file formats that applications can consume. The underlying application configuration records these supported input types, together with the version, launch commands, and initialization settings required for reproducible execution. For example, a `.pptx` artifact can be connected to LibreOffice Impress and WPS PowerPoint directly, whose application configurations specify how each application is launched with the file and which application-specific initialization settings are required. The resulting states are instantiated on reproducible Ubuntu virtual machines for task synthesis and trajectory collection.

2.2.2 ENVIRONMENT-GROUNDED TASK SYNTHESIS

Given the diverse desktop environments constructed above, we next translate environment diversity into training-task diversity. Each synthesis episode first samples a target application and then selects compatible artifacts via the *application–artifact compatibility graph*, thereby instantiating a coherent and executable task context.

Relying primarily on model priors tends to produce generic, repetitive instruction patterns and underutilize the state diversity of instantiated environments at scale. We therefore ground instruction generation in active exploration of the current environment. To avoid repeatedly covering familiar functionality, we maintain a global application memory that records functions previously exercised for each application and exposes this history during subsequent exploration, encouraging broader coverage over time. The **Exploration Agent** receives this memory together with application metadata, exact file paths, launch configuration, tutorial, and initial screenshot, and performs a short non-destructive exploration restricted to opening, clicking, scrolling, and terminating. Based on the observed artifact content and application state, it proposes a realistic user instruction together with its initialization configuration and observable success conditions, without prescribing a GUI/code solution or revealing answer-critical information.

To make synthesized tasks consistently verifiable, we adopt an *everything-is-a-file* abstraction in which terminal conditions are mapped, whenever possible, to inspectable file representations, and evaluator synthesis selects rule-based, VLM-based, or hybrid checks according to their observability. Verification specifies the desired terminal state without constraining how it is reached, allowing the same task to be completed through GUI interaction, code execution, or their combination.

We decouple the instruction proposal from evaluator synthesis. Once an instruction proposal is accepted, its instruction and success conditions are frozen and passed to an independent **Evaluator Agent**, which compiles the corresponding verification logic. Programmatically observable conditions are evaluated primarily with rule-based checks over VM files and command-line state, while visual conditions that cannot be reliably recovered programmatically are handled with VLM-based evaluation. Instructions and evaluators are validated separately in the pipeline. Instructions are checked for reachability, unambiguity, realism, and sufficient verification conditions, while evaluators must compile and assign zero reward to the initial state. Invalid evaluators are repaired when possible. Each validated instruction and evaluator then constitutes a task, after which semantically near-duplicate tasks are removed. Together, environment-grounded synthesis and file-backed verification produce a scalable pool of verifiable tasks shared by SFT trajectory collection and online RL. Data infrastructure details are provided in Appendix C.

2.3 TRAINING PARADIGM

2.3.1 SFT COLD START

Most open-source CUAs used as base models have previously been trained primarily for GUI-only interaction. They may possess visual understanding, coordinate grounding, and low-level control,

but have not learned the new decision boundary created when code execution becomes an equally valid action. Merely adding a code tool expands the action space without supplying a routing prior for when to remain in the GUI, when to switch to code, and when to return to the interface for verification; it can even disrupt already learned GUI behavior. SFT therefore serves as a cold start in our pipeline: it establishes the action semantics of the unified tool schema and distills from teacher trajectories a coarse but usable GUI/code routing policy. We collect SFT trajectories on tasks produced by the data infrastructure described in Section 2.2, using Claude-Sonnet-4.6 (Anthropic, 2026) as the teacher model with both Computer Use and Bash Tool enabled. The resulting trajectories are then postprocessed and normalized into the OS-Helix action schema. Each trajectory is scored by its evaluator, using the teacher model itself when visual judgment is required, and the resulting score serves as the primary signal for rejection sampling. We further apply a filtering pipeline to remove malformed tool calls, schema-mismatched actions, and trajectories that fail evaluator or quality checks, yielding a set of high-quality SFT samples. Training hyperparameters and data-mixture details are provided in Appendix D.1.

2.3.2 Agentic RL with Step-Balanced GSPO

We first build a cross-node environment management system that monitors and manages over one hundred Ubuntu sandbox environments across multiple standalone machines. The system exposes a unified port to the training side and internally dispatches requests to the environments. We then perform online RL on carefully filtered generated tasks with rule-based evaluators only, further improving model performance. Each rollout interacts with a live desktop environment, and the paired evaluator score serves as the sole outcome reward signal. If an assistant turn contains an unparsable tool call or any other invalid action-format error, that assistant response is isolated as an invalid sample and assigned a reward of 0, even if the remaining trajectory could receive a positive evaluator score.

We use three efficiency mechanisms to make online RL feasible under costly desktop interaction. First, we adopt an asynchronous RL pipeline that allows the rollout policy to lag behind the latest policy by at most one version, enabling environment interaction and policy optimization to run in parallel. Second, we maintain per-task success-rate estimates and bias sampling toward tasks near the current capability frontier, down-weighting tasks that are already too easy or remain too difficult. Third, we apply bounded sliding-window visual-context segmentation to control context length and sample growth. Rollouts are split into segments and each segment can contain multiple assistant actions that contribute to loss. We apply the same sliding-window segmentation logic during rollout inference to maintain consistency between training and inference.

In our preliminary RL experiments, standard GSPO exhibited a systematic drift toward code-intensive behavior as shown in Figure 4. Specifically, as training progresses, the model increasingly favors code-related actions, ultimately resulting in a slight performance degradation relative to the initial SFT checkpoint. Within a mixed GUI/code training sample, a code action typically contains substantially more assistant tokens than a GUI click or keystroke. Standard GSPO averages token-level log-importance weights over all valid assistant tokens in the sample. Consequently, even when a long code action constitutes only one of several GUI and code steps, it can dominate both the sample-level importance ratio and the clipping decision. This motivates an aggregation in which the weight assigned to each action step in the sample-level log-ratio is no longer determined by its token count.

Step-balanced GSPO modifies only the aggregation of the sample-level ratio. For training sample i, let $a_{i,k}$ be the set of assistant-token positions that contribute to the loss for action step k, and let $\tau_{i,k}$ denote its action type. Define the token-level importance weight $w_{i,t}(\theta) = \pi_\theta(y_{i,t} \mid h_{i,t})/\pi_{\text{old}}(y_{i,t} \mid h_{i,t})$ and its log-importance weight $\delta_{i,t} = \log w_{i,t}(\theta)$. We first compute the mean token-level log-importance weight for each action step

$$\bar{\delta}_{i,k} = \frac{1}{|a_{i,k}|} \sum_{t \in a_{i,k}} \delta_{i,t},$$

and then define the step-balanced sample-level ratio as

$$w_i^{\text{step}}(\theta) = \exp\left(\frac{\sum_{k=1}^{K_i} \alpha_{\tau_{i,k}} \bar{\delta}_{i,k}}{\sum_{k=1}^{K_i} \alpha_{\tau_{i,k}}} \right).$$

Method	Setting	Success Rate (%)					
		OS	Office	Daily	Prof.	Work	Avg.
Agentic Frameworks							
UiPath w/ GPT-5 (2025)	GUI	73.91	49.52	62.12	71.43	37.30	53.69
CoAct-1 w/ GPT-5 (2026)	Hybrid	75.00	62.93	57.94	71.43	47.87	59.93
GTA1 w/ GPT-5 (2026d)	GUI	79.17	63.91	62.56	79.59	50.91	63.41
Agent S3 w/ GPT-5 (2025)	Hybrid	77.50	66.46	61.23	69.80	51.37	62.63
OS-Symphony w/ GPT-5 (2026a)	Hybrid	79.17	65.73	67.76	69.23	57.98	65.84
General Models & Specialist Models							
Claude-Sonnet-4.5 (2025)	GUI	70.83	72.59	61.35	63.27	49.54	62.88
OpenCUA-72B (2026b)	GUI	61.13	44.73	49.95	72.58	22.16	44.91
UI-TARS (2025)	GUI	41.67	50.42	55.69	51.02	14.66	41.85
UI-TARS-2 (2025a)	GUI	41.67	61.11	62.12	61.22	34.13	53.10
DeepMiner-Mano-72B (2025)	GUI	66.67	63.22	52.51	83.67	24.41	53.91
EvoCUA-32B (2026)	GUI	78.26	59.79	64.55	81.63	27.89	56.73
EvoCUA-1.5-32B (2026)	GUI	–	–	–	–	–	63.20
Kimi K2.5 (2026b)	GUI	73.91	69.20	66.42	73.47	46.11	63.34
Kimi K2.6 (2026)	GUI	79.17	80.03	77.11	81.63	54.97	73.07
Qwen3.7-Plus (2026b)	GUI	79.17	82.91	74.03	73.47	59.25	73.33
Backbone Baselines							
Qwen3.5-27B (2026a)♣	GUI	66.67	39.60	57.50	52.00	36.06	46.61
Qwen3.5-27B (2026a)♣	Hybrid	66.67	39.26	58.79	49.96	38.29	46.50
Qwen3.5-35B-A3B (2026a)♣	GUI	37.50	46.92	44.57	38.77	25.65	39.20
Qwen3.5-35B-A3B (2026a)♣	Hybrid	69.57	40.94	46.21	52.22	31.01	42.84
OS-Helix-35B-A3B	Hybrid	60.87 (↓8.7)	70.00 (↑29.1)	52.19 (↑6.0)	56.31 (↑4.1)	45.14 (↑14.1)	57.27 (↑14.4)
OS-Helix-27B	Hybrid	75.00 (↑8.3)	72.60 (↑33.3)	58.32 (↓0.5)	56.49 (↑6.5)	54.28 (↑16.0)	62.75 (↑16.3)
OS-Helix-27B-RL	Hybrid	82.61 (↑15.9)	69.14 (↑29.9)	65.01 (↑6.2)	79.59 (↑29.6)	53.32 (↑15.0)	66.44 (↑19.9)

Table 1: Main results of OS-Helix on OSWorld. ♣ represents the result reproduced by us, and others are sourced from the official leaderboard. OS contains system-level OS tasks; Office contains LibreOffice Calc, Impress, and Writer; Daily contains Chrome, Thunderbird, and VLC; Prof. contains GIMP and VS Code; and Work contains multi-application tasks.

In our setting, GUI and code steps use equal weights, so every action step contributes equally to the sample-level log-ratio regardless of its token count.

The policy objective follows the standard GSPO formulation and changes only the aggregation used to form the sample-level importance ratio, replacing the original sequence-level aggregation with $w_i^{\text{step}}(\theta)$. This substitution does not, in general, equalize full parameter-gradient norms or variances across action lengths. Appendix F provides the full derivation.

3 Experiments

3.1 Experimental Setup

Evaluation Benchmarks. Our evaluation centers on desktop environments, employing OSWorld[1] (Xie et al., 2024; 2025) as our primary benchmark, which comprises 369 real-world tasks across five domains in an Ubuntu environment. Following common practice (Xie et al., 2024), we exclude the 8 Google Drive tasks, yielding a final set of 361 tasks. We report mean success rate as the primary metric. Beyond the Ubuntu environment, we additionally report results on WindowsAgentArena (Bonatti et al., 2024) to evaluate cross-platform transfer to Windows.

Models and Baselines. We instantiate OS-Helix from Qwen3.5-27B and Qwen3.5-35B-A3B (Qwen Team, 2026a). For broader context, we also report representative agentic frameworks, general-purpose models, specialist models, and matched backbone baselines.

Implementation Details. For task synthesis, we use Claude-Sonnet-4.6 (Anthropic, 2026) for the Exploration and Evaluator Agent, and GPT-5.5 (OpenAI, 2026) for independent quality review agent. For SFT, we train for one epoch on 31K successful filtered desktop samples synthesized by our pipeline and 40K general-purpose multimodal samples spanning code, reasoning, and related domains. For RL, we initialize from the dense OS-Helix-27B checkpoint and conduct 1,000-step

[1]In this paper, "OSWorld" refers to "OSWorld-Verified".

training run using rule-based evaluators as reward signals. Complete data details are provided in Appendix C, while training details are provided in Appendix D.

Unless otherwise specified, each evaluation episode is capped at 50 interaction steps and retains up to 8 historical screenshots (resolution 1504x832) together with the full textual interaction history. All baseline models are evaluated using the same harness and environment configuration.

3.2 Main Results

Table 1 shows that hybrid-policy training, rather than code-tool availability alone, yields substantial gains under matched-backbone comparisons. OS-Helix-27B and OS-Helix-35B-A3B improve over their corresponding Hybrid baselines by 16.25 and 14.43 absolute points, respectively. The largest gains occur in Office (+33.3 and +29.1 points) and Workflow (+16.0 and +14.1 points). Office covers Calc, Impress, and Writer, where tasks often combine visually grounded interaction with rendered document states and precise transformations of structured artifacts. Workflow comprises multi-application tasks that require task-relevant information to be handed off across applications and action modalities. These interpretations are consistent with the agent behavior study and representative trajectory audit in Appendix G.

Model	**Succ. (%)**
Agent S3 w/ GPT-5 (2025)	54.10
OS-Symphony w/ GPT-5 (2026a)	63.45
Qwen3.5-27B (2026a)♣	56.41
OS-Helix-27B	57.95
OS-Helix-27B-RL	63.87

Table 2: Main results of OS-Helix on WindowsAgentArena. ♣ represents the result reproduced by us. All entries use 50-step budget and hybrid action space.

Starting from the SFT policy, OS-Helix-27B-RL further raises the average success rate from 62.75% to 66.44%. Its incremental gains are concentrated in Professional (+23.10 points), OS (+7.61 points), and Daily (+6.69 points), whereas Office and Workflow do not improve further over their already strong SFT results. This profile shows that RL provides additional gains in specific task domains rather than a uniform increase across the benchmark. The paired-bootstrap analysis of the aggregate difference is reported in Table 16.

In broader, non-controlled comparisons, OS-Helix-27B-RL is competitive with GPT-5-based agentic frameworks such as Agent-S3 (Gonzalez-Pumariega et al., 2025) and OS-Symphony (Yang et al., 2026a), while newer and substantially larger general models remain stronger overall. Table 2 further shows that the gains transfer to WindowsAgentArena, where OS-Helix-27B-RL improves over the Qwen3.5-27B GUI baseline by 23.05 points. This provides evidence of cross-platform transfer on WindowsAgentArena, with detailed results deferred to Appendix E.1. Notably, the RL checkpoint also exhibits a larger improvement over the SFT checkpoint on WAA, suggesting that our online RL training further enhances the model's generalization capability.

3.3 Ablation Study

We conduct three groups of ablations: data robustness, training-stage and RL-objective attribution, and the unified GUI/code paradigm. Unless otherwise specified, all variants use Qwen3.5-27B and are evaluated on OSWorld with the same harness configuration as the main experiments.

Data robustness ablation. Table 3 tests whether the SFT gains require a strong trajectory teacher or data from OSWorld-related application domains. Panel (a) compares Kimi K2.5 (Team et al., 2026b) and Claude Sonnet 4.6 (Anthropic, 2026) trajectories collected from the same 800 synthesized tasks. Panel (b) removes all samples from the nine OSWorld application domains, reducing the desktop SFT set from approximately 31K to 17K samples. Both teacher-derived datasets outperform the unadapted Hybrid backbone, while the stronger teacher performs better. Removing the nine related application domains shifts success from 62.75% to 62.31%. The retained performance after domain removal is consistent with capabilities that transfer beyond the evaluated application domains.

(a) Trajectory teacher.

Teacher	Success (%)	Δ vs. base
Kimi K2.5	54.03	+7.53
Claude Sonnet 4.6	**58.77**	**+12.27**

(b) OSWorld-domain removal.

Desktop data	Samples	Succ. (%)
Full synthesized pool	∼31K	**62.75**
Nine domains removed	∼17K	62.31

Table 3: Data robustness ablations.

(a) Training stages.

Variant	SFT	RL	Succ. (%)
Hybrid Backbone	No	No	46.50
RL only	No	Yes	48.20
SFT only	Yes	No	62.75
SFT → RL	Yes	Yes	**66.44**

(b) RL algorithm.

RL objective	Succ. (%)	Δ vs. SFT	Act.(%)
Standard GSPO	60.47	-2.28	40.0
Step-balanced GSPO	**66.44**	**+3.69**	26.1

Table 4: Training-stage and RL algorithm ablations. *Act.* reports the fraction of actions assigned to code execution.

Training-stage and algorithm ablation. Table 4 separates SFT, RL, and the RL objective. Panel (a) shows that SFT provides the main improvement over the unadapted Hybrid policy, RL alone changes little (+1.70 points), and applying RL after SFT yields the highest success rate of 66.44%. Panel (b) isolates the RL objective under the same SFT initialization, data, sampling strategy, reward signal, and optimization budget. Standard GSPO lowers success to 60.47% and increases the code-action rate from 27.9% under the SFT policy to 40.0%, whereas Step-balanced GSPO raises success while maintaining a 26.1% code-action rate close to the SFT policy. Together, these results suggest that action-step balancing improves evaluated performance while keeping the GUI/code action balance close to the SFT policy rather than drifting toward code-heavy behavior.

(a) Code access at inference.

Policy	Succ. (%)	Acc.	Act.(%)
Qwen3.5-27B	42.59	No	–
OS-Helix-27B	52.13	No	–
OS-Helix-27B	62.75	Yes	27.9

(b) SFT modality at fixed scale.

SFT data	Samp.	Succ. (%)	Acc.	Act.(%)
GUI only	10K	57.56	Yes	11.7
Hybrid GUI/code	10K	60.61	Yes	37.1

Table 5: Unified GUI/Code Paradigm ablations. *Acc.* denotes whether code execution access is enabled at inference; *Act.* reports the fraction of actions assigned to code execution.

Unified GUI/code paradigm ablation. Table 5 disentangles code-tool availability from learned coordination. Panel (a) evaluates a fixed OS-Helix-27B checkpoint with and without code access. Panel (b) holds the backbone, SFT configuration, 10K-sample budget, and Hybrid evaluation harness fixed, changing only whether the SFT trajectories are GUI-only or hybrid GUI/code. Disabling code for the same policy still yields 52.13% success; enabling it adds 10.62 points, demonstrating retained GUI competence and code complementarity. With inference-time code access and the 10K-sample budget fixed, hybrid SFT adds 3.05 points over GUI-only SFT, indicating that training on hybrid trajectories enables a more effective use of shared GUI/code actions rather than relying on GUI-only supervision to generalize to code actions at inference.

4 Discussion

Accuracy versus efficiency. In Figure 3, evaluator scores above 0.5 are counted as successes and scores below 0.5 as failures. Panel (a) reports mean interaction steps, All tasks and Own successes use each run's complete and successful task sets, whereas Joint successes and Joint failures retain only shared tasks on which all three policies have the same binary outcome, controlling for task-composition differences; panel (b) reports cumulative Success@K on OSWorld. Across the full set, successful episodes, and outcome-matched subsets, OS-Helix-27B uses fewer interactions while covering more successes under small step budgets. Learned modality routing therefore improves both execution efficiency and task coverage, an effect not reproduced by enabling code only at inference time.

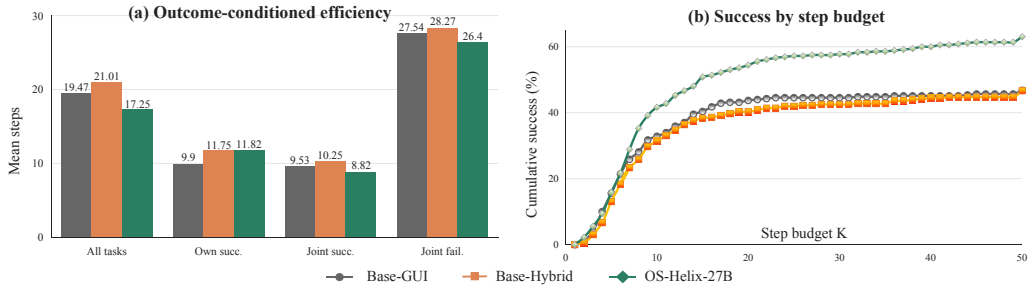

Figure 3: Execution efficiency on OSWorld.

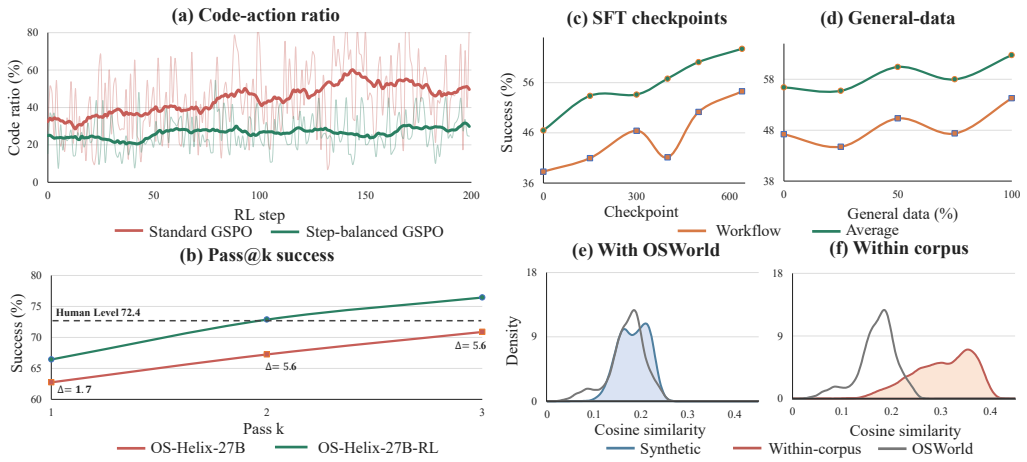

Figure 4: Training dynamics, scaling, and coverage analysis.

Why step balance matters. Panel (a) of Figure 4 uses the fraction of actions invoking code at each online RL update as a behavioral diagnostic. Under the same training setup, standard GSPO progressively drifts toward code-intensive behavior, whereas Step-balanced GSPO remains lower and more stable over the shared observed horizon. Panel (b) reports repeated-sampling coverage. Compared with the SFT policy, OS-Helix-27B-RL improves Pass@1 from 62.75% to 66.44% (+3.69 points) and Pass@3 from 70.88% to 76.44% (+5.56 points), with the latter supported by a paired-bootstrap confidence interval of 95% [2.24,8.97]. The larger gain under Pass@3 indicates that RL does more than strengthen a single preferred trajectory and increases the probability of discovering distinct successful trajectories across independent attempts.

Training scaling and data distributions. Panels (c)–(f) of Figure 4 further examine the performance scaling and instruction distributions. Panel (c) reports Workflow-domain and overall-average success across checkpoints from the first SFT epoch. Panel (d) holds the desktop data fixed and varies the fraction of the 40K general-purpose multimodal pool from 0% to 100%. Both sets of curves improve overall as the supervision grows, indicating complementary contributions from hybrid trajectories and general-purpose data. Panels (e)–(f) show kernel-density estimates of these per-task statistics. We encode each instruction with `text-embedding-3-large` (OpenAI, 2024), L2-normalize the embedding, and compute cosine similarity by vector dot product. The cross-corpus similarity to OSWorld is comparable to OSWorld's internal-instruction similarity, indicating limited instruction-level overlap. Together with domain-removal ablation in Table 3, this provides complementary evidence against direct instruction overlap or reliance on data from the evaluated domains. Meanwhile, the synthesized corpus exhibits a higher but still broadly spread internal-similarity distribution, suggesting that it captures shared, transferable desktop-task structure rather than collapsing to a narrow instruction template family.

5 Conclusion

We present OS-Helix, a native hybrid computer-using agent trained end-to-end through scalable executable desktop task synthesis, hybrid SFT, and Step-balanced GSPO online RL, designed to natively coordinate heterogeneous GUI and code actions within a single policy. On OSWorld-Verified, OS-Helix-27B and OS-Helix-35B-A3B achieve relative improvements of 35.0% and 33.7% over their matched Qwen3.5 hybrid baselines, while OS-Helix-27B-RL further improves by 42.9% to reach 66.44% success. OS-Helix-27B-RL also transfers to WindowsAgentArena with 63.87% success, surpassing GPT-5-based framework methods on both benchmarks. Ablations and behavioral analyses show that SFT provides the main gain, hybrid supervision outperforms GUI-only supervision at a fixed scale, and learned routing improves both execution efficiency and task coverage beyond code access alone. Under matched RL settings, Step-balanced GSPO outperforms standard GSPO and mitigates its drift toward code-intensive behavior, while RL increases Pass@3 from 70.88% to 76.44%, indicating broader discovery of successful trajectories across independent attempts. These findings establish native GUI/code coordination as a learnable hybrid paradigm and provide a reproducible foundation for further research on hybrid computer-using agents.

AI USE STATEMENT

In the preparation of this manuscript, we utilized LLMs to generate synthetic datasets, to refine the writing (including grammar and spelling corrections), to draft sections of the paper, and to assist with the preliminary derivation of mathematical formulas in Appendix F. The authors reviewed and take responsibility for the research design, experiments, analysis, claims, and final presentation.

ETHICS STATEMENT

This work does not involve human-subject studies or the collection of private data from real users. We study training methods for general-purpose computer-using agents, and conduct all trajectory collection, reinforcement learning, and evaluation in controlled, isolated Ubuntu virtual machines. The VMs use fixed application versions and a curated local file library, and are separated from personal host files and credentials. This setup reduces the risk that exploratory actions during research inadvertently modify host state or access unauthorized resources.

Although the experiments themselves present no substantial direct ethical risk, deploying agents with combined GUI interaction and code execution on real devices raises important safety and privacy concerns. Such agents can continuously observe screen content, modify local files, and execute Python or Bash, and may therefore encounter personally identifiable information, confidential documents, or authentication credentials. They may also be exposed to prompt injection through interface content or external files. Code actions further increase the potential impact of erroneous behavior or malicious workflows. The isolated experimental setting used in this work does not resolve these deployment-time risks.

Real-world deployment should follow least-privilege and safety-by-design principles. These include granular permissions over applications, directories, networks, and code execution; explicit human confirmation before irreversible, credential-bearing, or otherwise high-impact operations; runtime policy checks, sandboxed execution, audit logs, and post-action verification; and data minimization and sanitization for screenshots, trajectories, and tool outputs. Users should retain the ability to define and revoke the agent's authority, with sensitive applications such as banking, private communication, and credential management inaccessible by default. Because general-purpose automation is inherently dual use, future CUA research should advance capabilities together with systematic red teaming, interpretable action auditing, and reliable human takeover mechanisms.

REFERENCES

Pranjal Aggarwal, Graham Neubig, and Sean Welleck. Gym-anything: Turn any software into an agent environment. *arXiv preprint arXiv:2604.06126*, 2026.

Moonshot AI. Kimi k2. 6: Advancing open-source coding. *URL https://www. kimi. com/blog/kimi-k2-6*, 2026.

Anthropic. Claude code overview. https://docs.anthropic.com/en/docs/claude-code/overview, 2025.

Anthropic. Introducing claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, Sep 2025. Accessed: 2025-09-30.

Anthropic. Introducing claude sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6, Feb 2026. Accessed: 2026-02-17.

Lei Bai, Jiaqi Cao, Chiyu Chen, Guanzhou Chen, Kai Chen, Guangran Cheng, Erfei Cui, Xuanlang Dai, Shengyuan Ding, Shangheng Du, et al. Intern-s2-preview: Scientific agentic foundation model. *arXiv preprint arXiv:2608.13505*, 2026.

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

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, et al. Windows agent arena: Evaluating multi-modal os agents at scale. *arXiv preprint arXiv:2409.08264*, 2024.

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

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9313–9332, 2024.

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

Horia Cristescu, Charles Park, Trong Canh Nguyen, Sergiu Talmacel, Alexandru-Gabriel Ilie, and Stefan Adam. Ui-cube: Enterprise-grade computer use agent benchmarking beyond task accuracy to operational reliability. *arXiv preprint arXiv:2511.17131*, 2025.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

Zihan Ding, Longxu Dou, Qi Gao, Xiangwu Guo, Shengchao Hu, Zilong Huang, Zihang Jiang, Lei Ke, Mengcheng Lan, Weixian Lei, et al. Ui-mate: Advancing open-weight foundation gui agents with in-context demonstrations. *arXiv preprint arXiv:2608.15930*, 2026.

Aarash Feizi, Shravan Nayak, Xiangru Jian, Kevin Qinghong Lin, Kaixin Li, Rabiul Awal, Xing Han Lù, Johan S Obando Ceron, Juan A Rodriguez, Nicolas Chapados, et al. Grounding computer use agents on human demonstrations. In *International Conference on Learning Representations*, volume 2026, pp. 5181–5215, 2026.

Tianyu Fu, Anyang Su, Chenxu Zhao, Hanning Wang, Minghui Wu, Zhe Yu, Fei Hu, Mingjia Shi, Wei Dong, Jiayao Wang, et al. Mano technical report. *arXiv preprint arXiv:2509.17336*, 2025.

Yikun Fu, Bowen Fu, Zhenyu Wu, Shuang Cheng, Xiaowei Sun, Bowen Yang, Zehao Li, Yibo Zhao, Zichen Ding, Zhoumianze Liu, et al. Macagentbench: Benchmarking ai agents on real-world macos desktop. *arXiv preprint arXiv:2606.22557*, 2026.

Heyang Gao, Zexu Sun, Erxue Min, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Xu Chen. Solving the granularity mismatch: Hierarchical preference learning for long-horizon llm agents. In *International Conference on Learning Representations*, volume 2026, pp. 67081–67110, 2026.

Gonzalo Gonzalez-Pumariega, Vincent Tu, Chih-Lun Lee, Jiachen Yang, Ang Li, and Xin Eric Wang. Scaling agents for computer use. *arXiv preprint arXiv:2510.02250*, 2025.

Xinming Hou, Mingming Yang, Wenxiang Jiao, Xing Wang, Zhaopeng Tu, and Wayne Xin Zhao. Coact: A global-local hierarchy for autonomous agent collaboration. *arXiv preprint arXiv:2406.13381*, 2024.

Mianqiu Huang, Taofeng Xue, Chong Peng, Jinrui Ding, Sicheng Fan, Jiale Hong, Yufei Gao, Xiaocheng Zhang, Linsen Guo, Xin Yang, et al. EvoCUA-1.5: Online reinforcement learning for multi-turn computer-use agents. *arXiv preprint arXiv:2607.09773*, 2026.

Hongrui Jia, Jitong Liao, Xi Zhang, Haiyang Xu, Tianbao Xie, Chaoya Jiang, Ming Yan, Si Liu, Wei Ye, and Fei Huang. Osworld-mcp: Benchmarking mcp tool invocation in computer-use agents. *arXiv preprint arXiv:2510.24563*, 2025.

Deyang Jiang, Jing Huang, Xuanle Zhao, Lei Chen, Liming Zheng, Fanfan Liu, Haibo Qiu, Peng Shi, and Zhixiong Zeng. TreeCUA: Efficiently scaling GUI automation with tree-structured verifiable evolution. *arXiv preprint arXiv:2602.09662*, 2026.

Hanyu Lai, Xiao Liu, Yanxiao Zhao, Han Xu, Hanchen Zhang, Bohao Jing, Yanyu Ren, Shuntian Yao, Yuxiao Dong, and Jie Tang. Computerrl: Scaling end-to-end online reinforcement learning for computer use agents. In *International Conference on Learning Representations*, volume 2026, pp. 23553–23591, 2026.

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.

Dunjie Lu, Shuai Bai, Tianyi Bai, Sicheng Fan, Chang Gao, Jian Guan, Feng Hu, Mianqiu Huang, Xingyang Huang, Yizhen Jiang, et al. Qwen-cua: Native computer use for (almost) everything. *arXiv preprint arXiv:2608.02352*, 2026a.

Zhengxi Lu, Yuxiang Chai, Yaxuan Guo, Xi Yin, Liang Liu, Hao Wang, Han Xiao, Shuai Ren, Pengxiang Zhao, Guangyi Liu, et al. Ui-r1: Enhancing efficient action prediction of gui agents by reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 17608–17616, 2026b.

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

Zijun Min, Bingshuai Liu, Ante Wang, Long Zhang, Anxiang Zeng, Haibo Zhang, and Jinsong Su. Orchestrating tokens and sequences: Dynamic hybrid policy optimization for rlvr. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 37782–37795, 2026.

OpenAI. New embedding models and API updates. https://openai.com/index/new-embedding-models-and-api-updates/, January 2024. Accessed: 2026-09-09.

OpenAI. Openai gpt-5 system card. https://cdn.openai.com/gpt-5-system-card.pdf, August 2025. Accessed: 2025-08-07.

OpenAI. Introducing GPT-5.5. https://openai.com/index/introducing-gpt-5-5/, 2026. Accessed: 2026-04-23.

OpenClaw. Openclaw. https://github.com/openclaw/openclaw, 2026. Open-source personal AI assistant, version 2026.3.8, accessed 2026-03-09.

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

Tejal Patwardhan, Rachel Dias, Elizabeth Proehl, Grace Kim, Michele Wang, Olivia Watkins, Simon Fishman, Marwan Aljubeh, Phoebe Thacker, Laurance Fauconnet, et al. Gdpval: Evaluating ai model performance on real-world economically valuable tasks. In *International Conference on Learning Representations*, volume 2026, pp. 24005–24040, 2026.

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. Qwen3.7: The agent frontier, May 2026b. URL https://qwen.ai/blog?id=qwen3.7.

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.

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

Haoting Shi, Wenhao Wang, Weicheng Fang, Yaozhong Liang, Tian Jin, Pengxiang Zhao, Guangyi Liu, Siheng Chen, and Yanfeng Wang. Cua-universe: A scalable and dynamic environment for hybrid gui+ cli agents. *arXiv preprint arXiv:2609.05374*, 2026.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Zeyuan Chen, Jieyu Zhao, Ran Xu, et al. Coact-1: Computer-using multi-agent system with coding actions. In *International Conference on Learning Representations*, volume 2026, pp. 127091–127107, 2026.

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, Kanzhi Cheng, Yian Wang, Bowen Yang, Hang Yan, Liheng Chen, Fangzhi Xu, Zichen Ding, Nuo Chen, Jialin Cao, et al. Osreward: Instituting standardized evaluation for cross-platform computer-use reward models. *arXiv preprint arXiv:2607.28609*, 2026a.

Yiyou Sun, Xinyang Han, Weichen Zhang, Yuanbo Pang, Tianyu Wang, Yuhan Cao, Yixiao Huang, Chris Duroiu, Haoyun Zhang, Jeffrey Lin, et al. Agents' last exam. *arXiv preprint arXiv:2606.05405*, 2026b.

Fei Tang, Zhangxuan Gu, Zhengxi Lu, Xuyang Liu, Shuheng Shen, Changhua Meng, Wen Wang, Wenqi Zhang, Yongliang Shen, Weiming Lu, et al. Gui-g^2: Gaussian reward modeling for gui grounding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 33214–33222, 2026a.

Zhengyang Tang, Yuxuan Liu, Xin Lai, Junyi Li, Pengyuan Lyu, Yiduo Guo, Zhengyao Fang, Yang Ding, Yi Zhang, Weinong Wang, et al. Phoneworld: Scaling phone-use agent environments. *arXiv preprint arXiv:2605.29486*, 2026b.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026a.

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*, 2026b.

Qwen Team. Qwen3.5-Omni technical report. *arXiv preprint arXiv:2604.15804*, 2026.

Bowen Wang, Dunjie Lu, Junli Wang, Tianyi Bai, Shixuan Liu, Zhipeng Zhang, Haiquan Wang, Hao Hu, Tianbao Xie, Shuai Bai, et al. Cua-gym: Scaling verifiable training environments and tasks for computer-use agents. *arXiv preprint arXiv:2605.25624*, 2026a.

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

Weiyun Wang, Zhangwei Gao, Lixin Gu, Hengjun Pu, Long Cui, Xingguang Wei, Zhaoyang Liu, Linglin Jing, Shenglong Ye, Jie Shao, et al. InternVL3.5: Advancing open-source multimodal models in versatility, reasoning, and efficiency. *arXiv preprint arXiv:2508.18265*, 2025b.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025c.

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

Xuehui Wang, Zhenyu Wu, JingJing Xie, Zichen Ding, Bowen Yang, Zehao Li, Zhaoyang Liu, Qingyun Li, Xuan Dong, Zhe Chen, et al. Mmbench-gui: Hierarchical multi-platform evaluation framework for gui agents. *arXiv preprint arXiv:2507.19478*, 2025d.

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

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.

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

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, 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 2025. URL https://xlang.ai/blog/osworld-verified.

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

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.

Bowen Yang, Kaiming Jin, Zhenyu Wu, Zhaoyang Liu, Qiushi Sun, Zehao Li, JingJing Xie, Zhoumianze Liu, Fangzhi Xu, Kanzhi Cheng, et al. Os-symphony: A holistic framework for robust and generalist computer-using agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22300–22330, 2026a.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026b.

Kun Yang, Yanmeng Wang, Zhigen Li, Ning Cheng, Shaojun Wang, Jing Xiao, et al. Sspo: Subsentence-level policy optimization. *arXiv preprint arXiv:2511.04256*, 2025a.

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

Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, january 2024. *URL https://github. com/fla-org/flash-linear-attention*, 22, 2024.

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

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*, 2025b.

Yuhao Yang, Tianyu Fan, and Chao Huang. Cli-anything: Towards agent-native computer use. *arXiv preprint arXiv:2606.03854*, 2026e.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2026.

Xinbin Yuan, Jian Zhang, Kaixin Li, Zhuoxuan Cai, Lujian Yao, Jie Chen, Enguang Wang, Qibin Hou, Jinwei Chen, Peng-Tao Jiang, et al. Enhancing visual grounding for gui agents via self-evolutionary reinforcement learning. *arXiv preprint arXiv:2505.12370*, 2025.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Bofei Zhang, Zirui Shang, Zhi Gao, Wang Zhang, Rui Xie, Xiaojian Ma, Tao Yuan, Xinxiao Wu, Song-Chun Zhu, and Qing Li. TongUI: Internet-scale trajectories from multimodal web tutorials for generalized GUI agents. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 12367–12375, 2026a.

Yuheng Zhang, Chenlu Ye, Shuowei Jin, Changlong Yu, Wei Xiong, Saurabh Sahu, and Nan Jiang. Rethinking importance sampling in llm policy optimization: A cumulative token perspective. *arXiv preprint arXiv:2605.07331*, 2026b.

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

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

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Hanzhang Zhou, Panrong Tong, Xu Zhang, Quyu Kong, Chenglin Cai, Tianyu Xia, Gongjie Zhang, Jianan Zhang, Long Li, Long Chen, et al. Qwen-ui-agent technical report: Toward next-generation real-world centric foundation gui agents. *arXiv preprint arXiv:2607.28227*, 2026a.

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

Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An llm post-training framework for rl scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.

Yicheng Zou, Dongsheng Zhu, Lin Zhu, Tong Zhu, Yunhua Zhou, Peiheng Zhou, Xinyu Zhou, Dongzhan Zhou, Zhiwang Zhou, Yuhao Zhou, et al. Intern-s1-pro: Scientific multimodal foundation model at trillion scale. *arXiv preprint arXiv:2603.25040*, 2026.

A Related Work

Hybrid GUI/Code Computer-Using Agents. Traditional GUI-native agents learn screenshot-to-action policies over visual interface states (Qin et al., 2025; Wang et al., 2026b; Wu et al., 2025; Liu et al., 2026; Wang et al., 2025d), whereas command-line and coding agents operate over files, shells, and programmatic state (Anthropic, 2025; OpenClaw, 2026; Yang et al., 2026b; 2024). Their complementary strengths motivate hybrid computer use. Framework-centric systems (Gonzalez-Pumariega et al., 2025; Hou et al., 2024; Yang et al., 2026a) coordinate GUI, code, memory, and recovery through external harnesses, often using strong proprietary models. Their performance is coupled to costly inference and hand-engineered orchestration, making the model's own hybrid coordination difficult to isolate. Tool-call-based approaches (Jia et al., 2025; Yang et al., 2025b; Shi et al., 2026) expose programmatic operations through predefined, application-specific interfaces, limiting coverage and transfer to available integrations and tool schemas. Some approaches integrate GUI, code, and other capabilities through model fusion (Wang et al., 2025a; Qin et al., 2025), relying on predefined fusion rather than coordination learned within a single policy. Qwen-UI-Agent (Zhou et al., 2026a) also explores training with mixed GUI and code-like action capabilities, demonstrating the practical potential of native hybrid agents. Existing studies, however, describe data construction, executable environments, supervision, and training configurations at different levels of granularity, making controlled reproduction and comparison of learned coordination mechanisms difficult. Our work complements these efforts by studying executable application–artifact state construction, decoupled instruction–evaluator synthesis, and more stable optimization for training a single GUI/code policy.

Environment Scaling and Task Synthesis for Computer-Using Agents. Realistic long-horizon tasks and interactive environments are essential for training CUAs, yet manual task design and trajectory annotation are expensive and difficult to scale. This makes automated acquisition of high-quality instructions and demonstrations a central problem. One line of work crawls web tutorials and documentation and decomposes them into executable task instructions (Xu et al., 2025; Zhang et al., 2026a). Another line starts from the target environment itself and infers tasks from what the agent can observe or manipulate: Explorer (Pahuja et al., 2025) extracts and refines task intents from web entry pages, while OS-Genesis (Sun et al., 2025), GUI-ReWalk (Lin et al., 2025), TreeCUA (Jiang et al., 2026), and AUTO PLAY (Ramrakhya et al., 2026) synthesize tasks through exploratory interaction with the environment. These methods primarily scale task and trajectory synthesis, but they do not substantially scale the underlying interactive worlds, which constrains the diversity of behaviors learned by the agent. Unlike more constrained interaction domains such as web and mobile (Cao et al., 2026; Zhang et al., 2026c; Wu et al., 2026; Tang et al., 2026b; Cheng et al., 2026), PC environments, such as Ubuntu, Windows and MacOS, provide a natural substrate for scalable CUA training through system-level code execution, mature application ecosystems, persistent file systems, and package managers, enabling verifiable tasks grounded in concrete system state (Sun et al., 2026a; Fu et al., 2026). Recent work has begun to use virtual machines to expand desktop environments and synthesize tasks (Wang et al., 2026a; Lv et al., 2026; Huang et al., 2026), exploring different parts of desktop-state construction and task generation. Table 6 summarizes the reported coverage of existing task-synthesis pipelines. Existing efforts differ in the extent to which they scale applications, authentic artifacts, executable initial states, and verifiable terminal conditions. Our pipeline combines these components in a unified setting, while cross-application synthesis remains a preliminary component.

Training Paradigms for Computer-Using Agents. CUA training has broadly evolved from imitation-style supervised fine-tuning toward reinforcement learning in interactive environments. SFT-based systems learn action formats, visual grounding, and task execution from human demonstrations, synthetic trajectories, web tutorials, or teacher-model rollouts (Cheng et al., 2024; Qin et al., 2025; Wang et al., 2026b; Zhang et al., 2026a; Sun et al., 2025; Liu et al., 2026). Between SFT and online interaction, offline preference-optimization methods such as DPO (Gao et al., 2026) and offline GRPO (Yang et al., 2026c) improve policies from static preference data, but remain limited by the coverage of collected trajectories and cannot directly optimize recovery under environment feedback. Online RL methods for CUAs further split into grounding RL and agentic RL. Grounding RL optimizes low-level GUI localization or action selection with specialized visual rewards (Lu et al., 2026b; Yuan et al., 2025; Zhou et al., 2026b; Tang et al., 2026a; Feizi et al., 2026). Agentic RL instead optimizes complete trajectories in executable environments with rule-based or verifiable rewards. Recent CUA work has begun to tailor the optimization algorithm to this setting:

ComputerRL (Lai et al., 2026) introduces Entropulse, which alternates online RL with SFT to mitigate entropy collapse; CUA-Gym (Wang et al., 2026a) trains CUA checkpoints with GSPO (Zheng et al., 2025); and ScaleCUA (Lv et al., 2026) adopts GRPO-style online RL (Shao et al., 2024; Yu et al., 2026) together with frontier sampling and visual-context segmentation. EvoCUA-1.5 (Huang et al., 2026) proposes Step-Level Policy Optimization for decomposed multi-turn samples. Nevertheless, how to systematically train a single policy to route between native GUI and code actions while maintaining stable optimization across heterogeneous action granularities remains underexplored. Our work addresses this setting with hybrid SFT and online agentic RL with Step-balanced GSPO algorithm.

B Limitations and Future Work

OS-Helix is an initial step toward native GUI/code hybrid computer use, with remaining limitations centered primarily on computational and API resource constraints.

First, desktop environments are difficult to scale comprehensively: even an application–artifact desktop infrastructure spanning diverse applications cannot cover the full diversity of real user machines, private artifacts, application versions, and personalized configurations. Future work should further expand the application and artifact ecosystem and study broader randomization of system states and user configurations while preserving reproducibility.

Second, automatic verification remains imperfect. Unlike swe/code tasks, where gold final states can be obtained deterministically, automated CUA task synthesis under limited resources cannot reliably capture ground-truth terminal states—tasks may therefore be subject to feasibility risks, and the resulting evaluators carry intrinsic brittleness. Concretely, rule-based evaluators reliably cover many file-backed tasks but require careful construction and often depend on specific application versions or export formats; more fundamentally, even when an agent's execution is semantically near-correct, inconsistent strictness in file-attribute checks can reduce the score to zero. Rubric-based partial scoring, adopted in several recent works including ours, partially alleviates this issue but does not resolve it entirely. VLM-based evaluators extend coverage to textual outputs and visual outcomes, yet may miss fine-grained discrepancies; more critically, due to their inherent brittleness, existing RL engineering practice typically excludes VLM-based scoring from the reward signal altogether, concentrating construction effort on rule-based evaluators—which in turn constrains the types and diversity of tasks that can be incorporated into training. Future work should develop stronger evaluator-generation, repair, and validation pipelines, and systematically study cross-evaluator agreement as well as robustness to reward hacking.

Third, our agentic RL experiments are subject to certain limitations imposed by available computational resources. RL training was conducted only on the dense 27B model with relatively conservative hyperparameter settings; the total number of training steps was likewise constrained, precluding a systematic investigation of the scaling behavior of agentic RL. For OS-Helix-35B-A3B, despite its strong performance in SFT experiments, the Mixture of Experts (MoE) architecture introduces non-trivial additional challenges in training-inference consistency and optimization stability during agentic RL, making it difficult to support this training pipeline reliably under current resource conditions. We regard a more thorough exploration of RL training dynamics and scaling laws—together with stably extending Step-balanced GSPO to larger MoE computer-using agents as one of the most important directions for future work.

Beyond scaling, we believe the community is now at an inflection point where recursive self-improvement (RSI) deserves serious consideration. As CUA models attain a meaningful level of general computer-use capability, the model itself can serve as its own teacher: rolling out interaction trajectories on existing tasks, providing the training signal for the next iteration, and progressively improving through this self-bootstrapping loop. Rather than relying solely on human demonstration or externally curated data, such a paradigm allows the agent to continuously generate its own training trajectories with growing quality as capability improves and feed them back into subsequent training rounds. We view this self-evolving, teacher-free training paradigm as one of the most promising long-term directions for advancing CUA agents.

Finally, many existing desktop benchmarks were originally designed for GUI-only agents, with task design, reference solutions, and evaluation functions all implicitly centered on visual interaction

Method	Synthesis Grounding	Application Scaling	Real Artifact Scaling	Cross-App Tasks	Rule-Based Evaluator
OS-Genesis (2025)	E	×	×	×	×
AUTO PLAY (2026)	K/E/C	×	×	×	×
TreeCUA (2026)	K/E/C	×	×	×	×
EvoCUA-1.5 (2026)	K	×	✓	×	✓
Gym-Anything (2026)	K	✓	✓	×	×
CUA-Gym (2026a)	K/C	✓	×	✓	✓
OpenComputer (2026)	K/C	✓	×	×	✓
ScaleCUA (2026)	K	×	×	×	✓
Qwen-CUA (2026a)	K	✓	✓	✓	✓
Qwen-UI-Agent (2026a)	K/C	×	×	✓	✓
OS-Helix	K/E/C	✓	✓	✓	✓

Table 6: Comparison of recent CUA task- and environment-synthesis pipelines. Synthesis grounding distinguishes external knowledge or application priors (K), task-specific environment or file inspection before the final instruction is proposed (E), and cross-instance coverage or capability memory (C). Application scaling denotes support for adding and configuring applications beyond an inherited benchmark; real artifact scaling denotes authentic or realistic file-backed inputs; rule-based evaluators programmatically inspect terminal states. A green check denotes explicit support, a red cross denotes no explicit support, and an orange check denotes preliminary but not yet systematic support.

paths. Although their tasks can often be completed with hybrid strategies, existing evaluators may not capture every valid GUI/code solution. Furthermore, using success rate as the sole metric is overly coarse: it neither distinguishes whether the agent selected the more efficient modality nor captures potential safety risks inherent to hybrid execution, such as unintended side effects of code execution or inadvertent system-level operations.

A particularly promising complementary direction is the migration and adaptation of existing general-purpose agent benchmarks such as GDPEval (Patwardhan et al., 2026) and Agent Last Exam (Sun et al., 2026b) into GUI environments. These benchmarks are inherently designed around authentic artifact manipulation and complex reasoning tasks, and thus naturally encompass diverse environment states and authentic artifacts. Porting them into desktop GUI settings not only amortizes the high cost of designing tasks from scratch, but also enables a unified evaluation framework that simultaneously exercises GUI interaction, code execution, and their hybrid coordination, providing a more ecologically valid testbed for deeply probing the GUI/Code hybrid-driving paradigm. Future benchmarks should explicitly support hybrid computer use from the task-design stage onward, providing multi-path reference solutions and jointly evaluating terminal-state correctness, execution efficiency, modality-selection appropriateness, and safety, thereby offering a more comprehensive and equitable evaluation framework for next-generation hybrid agents.

C Data Details

C.1 Task Synthesis Comparison

Table 6 compares recent CUA task- and environment-synthesis pipelines across five dimensions: task grounding, application scaling, authentic artifact integration, cross-application task support, and executable evaluation. The comparison is organized along the axis of synthesis coverage rather than data scale or maturity. The broader contribution of OS-Helix is to extend exploration-grounded task generation toward artifact-grounded executable desktop-state construction: the pipeline jointly addresses K/E/C grounding, application scaling, authentic artifact-backed state initialization, and executable rule-based evaluation within a single coherent framework. Concretely, an application–artifact compatibility graph organizes valid pairings of application configurations and supported artifact types or file formats, enabling reproducible initial states to be instantiated systematically. The evaluator pipeline verifies persistent terminal outcomes rather than constraining the action sequence, supporting diverse task objectives and evaluation criteria across application domains and providing the infrastructure for subsequent hybrid-policy training.

Cross-application task synthesis is currently realized as a limited prototype, reflected by the partial symbol in the table; extending this component into a larger and more systematically characterized

Category	Count	Applications
Web, communication, and knowledge work	5	Chrome, Thunderbird, Notion, Zotero, Foliate
Office, documents, and publishing	9	Libreoffice Writer, Libreoffice Calc, Libreoffice Impress, WPS Presentation, WPS Writer, WPS Spreadsheets, Pdfarranger, Texstudio, Text Editor
Programming, databases, and version control	5	Pycharm, Vscode, Dbeaver, Gitkraken, Meld
Graphics, diagrams, CAD, and photography	6	Gimp, Inkscape, Krita, Darktable, Drawio, Librecad
Audio, video, music, and 3D media	9	Blender, Audacity, Handbrake, Mixxx, Obs Studio, Musescore, Spotify, Shotcut, Vlc
Scientific, engineering, geospatial, and network tools	8	Scilab, Wireshark, Kicad, Grass, Chimerax, Kalgebra, Celestia, Google Earth
Personal organization and finance	2	Calendar, Homebank
Operating-system operations	1	OS
Total	**45**	**45 applications**

Table 7: Functional classification of all 45 applications in the Ubuntu desktop infrastructure.

File group	Types
Documents, publishing, and office files	.doc, .docx, .pptx, .xlsx, .pdf, .epub, .txt, .tex
Code, projects, and structured data	Project Folders, .py, .sql, .csv, .sci, .xhb
Web, geospatial, and network artifacts	url, .html, .kmz, .pcapng
Images, vectors, diagrams, and design files	.png, .jpeg, .svg, .drawio, .kra
Audio, video, and music files	.mp3, .wav, .mp4, .mscz
CAD and 3D assets	.blend, .dxf

Table 8: File and artifact types covered by the complete inventory of 45 applications.

collection remains an explicit direction for future work. Works such as Qwen-CUA and Qwen-UI-Agent have demonstrated more extensive coverage of long-horizon and cross-application workflows at scale, and those dimensions represent clear avenues for subsequent expansion of OS-Helix.

C.2 Applications and Artifacts Details

Our complete desktop application inventory comprises 45 applications, consisting of 44 named desktop applications spanning diverse functional categories and one dedicated OS application for system-level operations. Table 7 enumerates all applications organized by functional category, Table 8 lists the artifact types and file formats covered by the pipeline, and Table 9 reports the primary sources of commonly used artifact pools.

Table 9 focuses on the major artifact pools consumed by the application–artifact sampler. Long-tail artifact types not individually itemized in the table are handled through lightweight alternatives including small curated example libraries, application-bundled demo assets, simple format conversions, and manually constructed seed files rather than dedicated large-scale collection efforts, reflecting practical constraints on human resource availability.

C.3 Final State Materialization

Our *everything-is-a-file* abstraction treats the file system as a common inspection interface rather than as a restriction on where a task state must originate. A terminal condition may initially reside in a downloaded web page, an in-memory application state, an unsaved layout, or a persistent user artifact. We attempt to materialize the relevant condition into a VM-resident file representation whenever this can preserve the semantics needed for evaluation: a web state may be downloaded or serialized locally, an in-memory result may be exported, and an answer may be written to a structured or plain-text file. This externalization separates the location of a state from its verifiable representation and allows heterogeneous desktop objectives to share a common evaluation interface.

The materialization step is not uniformly lossless. Some conditions, such as exact document structure, metadata, or serialized application state, admit faithful programmatic inspection; others depend on rendered layout, canvas appearance, transient UI state, or content for which serialization does

Type	High-level group	Primary source
`.docx`	Office documents	superdoc-dev/docx-corpus
`.pptx`	Office documents	ForceLess/PPTAgent_parsed_data
`.xlsx/.csv`	Office documents	Multilingual-Multimodal-NLP/TableBench
`.pdf`	Documents	lmms-lab/DocVQA converted to PDF format
`.png/.jpeg`	Images	moonshotai/WorldVQA
`.mp3`	Audio	thang101020/musical_audio_cache
`.mp4`	Video	raushan-testing-hf/videos-test; 2e8konjak/youtube_videos_12
`.wav`	Audio	Sakshi113/MMAU
`.tex`	LaTeX source	TIGER-Lab/arxiv-latex-5T
`.svg`	Vector graphics	lucide-icons/lucide
`.blend`	3D assets	Poly Haven[1]; DietCoke4671/BlenderBench
`.dxf`	CAD files	LibreCAD examples; tobygerlach/dxf_dataset
`.py`	Code files	Common algorithmic problems collected from programming problem sets
`.txt`	Plain text	Plain-text files converted from other file types
`url`	Web artifacts	OpenGVLab/ScaleCUA Web URLs
`Project folder`	Project files	SWE-bench/SWE-bench

[1] https://polyhaven.com/

Table 9: Primary sources of the file library by file type. Public upstream sources are written in repository-style form, including GitHub and dataset repositories where applicable. When an upstream source contains more files than needed for local deployment, we randomly sample a subset from that source. File types outside this table are covered through lightweight collection from small example libraries, application demo assets, simple conversions, or manually constructed seed files rather than large-scale harvesting.

not preserve the relevant evidence. Evaluator selection therefore reflects an observability trade-off: rule-based checks are used when a deterministic file or command representation captures the required semantics, VLM-based checks are used when visual or otherwise non-serializable evidence is essential, and hybrid evaluators combine the two when they provide complementary evidence. The resulting evaluator is agnostic to whether the agent reached the terminal condition through GUI actions, code execution, or their combination.

C.4 Instruction and Evaluator Synthesis Details

The primary task-synthesis pipeline instantiates states through the application–artifact compatibility graph. The pipeline then follows a one-way information flow: application and artifact configurations determine the initialization context; the **Exploration Agent** produces a structured instruction proposal and textual success conditions; the independent **Evaluator Agent** compiles the frozen conditions into rule-based or visual checks; and each independently validated instruction-evaluator pair enters the task pool.

Application-centered initialization. For each application, we record a canonical name, version, supported artifact types and file formats, launch commands, and initialization templates, and use these fields to build the application–artifact compatibility graph. Each synthesis episode first restores a controlled VM snapshot and samples a primary application, then selects zero or more compatible artifacts through the graph according to the application's initialization templates. For every sampled pre-existing artifact, initialization creates a VM-resident, evaluator-only golden copy before task execution while leaving the original path as the active task target; this copy is bound to the task configuration and may serve as the expected reference in subsequent content or structural comparisons. The system then provides the Exploration Agent with application metadata, exact file paths, launch configuration, an application tutorial, global application memory, and an initial screenshot.

The following partial PyCharm configuration illustrates how application metadata connects application initialization with artifact-library sampling.

The `version` field fixes the application version assumed by task synthesis, trajectory collection, and evaluator implementation, while `type` declares the artifact or project types that the application can directly consume. The two entries in `commands` are alternative initialization templates: the first launches an empty application, and the second launches the application with a sampled artifact. The position-aligned values in `random` are normalized into initialization probabilities; hence `[10, 90]` assigns approximately 10% probability to an empty PyCharm launch and 90% to a file- or

project-backed launch. When the second template is selected, the system randomly samples an absolute file path whose type is compatible with `type` from the artifact library and substitutes it for `PATH` before executing the command. The `related_app` and `related_type` fields retain compatibility relations among applications and artifact types for selecting conventional companion applications and for the graph-guided cross-application prototype.

This declarative configuration is what makes application scaling reproducible rather than a collection of ad hoc launch scripts. Adding an application requires specifying its version, launch procedures, supported artifact types, initialization probabilities, and optional relations to companion applications. The compatibility graph then filters valid application–artifact pairings before an episode is instantiated in a fixed VM, allowing new applications and artifact types to expand the executable state space while preserving initialization and sampling semantics.

Example Application Configuration: `Pycharm`

```
"pycharm": {
  "version": "Pycharm 2025.3.3",
  "type": [
    "py",
    "project_folder"
  ],
  "commands": [
    [
      [["pycharm-community"]]
    ],
    [
      [["pycharm-community", "PATH"]]
    ]
  ],
  "random": [10, 90],
  "related_app": [
    "vscode",
    "gitkraken"
  ],
  "related_type": [
    "py",
    "project_folder"
  ]
}
```

Grounded instruction proposal. To avoid repeatedly covering familiar functionality, we maintain global application memory that records functions previously exercised for each application and expose it during subsequent exploration. When additional interface information is needed, the Exploration Agent may perform a short non-destructive inspection using only `open`, `click`, `scroll`, and `done`. Keyboard and free-form text input are unavailable to avoid mutating the known initial state. It then proposes a goal-oriented instruction and fine-grained success conditions for evaluator synthesis. The resulting instruction record contains fields such as `snapshot`, versioned `related_apps`, `instruction`, `config`, `complexity`, `category`, `success_criteria`, and `evaluation_requirements_text`. The `instruction` field is the core user-facing task specification. The `snapshot`, `related_apps`, and `config` fields are determined during initialization and record the reproducible environment context. The `category` field is generated by the Exploration Agent together with the instruction and is divided into three groups. `app_only` denotes tasks centered on application or system functionality, such as changing software or user settings. `file_only` denotes tasks centered on processing input files whose final deliverables are file artifacts. `mixed` denotes tasks that involve both. The `complexity` field estimates the length of a pure-GUI solution. We label tasks with 5–15 steps as `easy`, tasks with 15–40 steps as `medium`, and tasks with more than 40 steps as `complex`. The `success_criteria` field lists the terminal conditions that must hold after successful task completion. The `evaluation_requirements_text` field translates those conditions into concrete checks for evaluator synthesis, such as which file, slide, field, format, or application state should be inspected. The instruction states non-answer constraints such as locations, scope, ordering, quantity, format, and filenames. Exact observed values are retained only in these hidden success and evaluation fields.

Evaluator synthesis. Evaluator synthesis begins only after an instruction proposal has been accepted and is performed by an independent Evaluator Agent in a separate context. Its inputs are the frozen instruction fields, initialization configuration, success criteria, and textual evaluation requirements. The Evaluator Agent may generate only the verification specification and cannot revise the instruction or proposal metadata. A rule-based evaluator comprises one or more self-contained Python judge functions. Each function declares its required imports locally, can run independently, accepts `result` and `expected` as inputs, and typically evaluates one well-defined subgoal. Either input may be supplied by `vm_file` as a VM-resident file or by `vm_command_line` as command-observable state. When initialization creates a local golden copy, `expected` may refer to it for content or structural comparison. When the target can be encoded directly in the rule, `expected` may be empty. Each judge accumulates a rubric-based score over fine-grained completion items with assigned weights. It returns a value in $[0, 1]$, with zero for no satisfied items and one for complete satisfaction; if execution raises an exception, it returns the score accumulated before the exception. The evaluator relies only on VM-resident artifacts and deterministic rules, without external or cloud-hosted golden assets, and checks the final artifact's content, structure, formatting, metadata, or configuration state.

When deterministic rules fully express the success conditions, the evaluator sets `need_rule_judge=true` without enabling visual judgment. When visual layout, canvas state, unparseable application-internal state, or dynamic web content is indispensable for verification, it may set `need_vlm_judge=true`. In this case, the VLM judge receives the agent's execution trajectory as input, and `vlm_desc` specifies the visual objects, attributes, and spatial relations to inspect in that trajectory.

Rule-based and VLM-based judgments may both be used for task verification, but their scores are recorded separately. For a task with multiple rule judges, the rule-based score is the mean of the individual rule-judge scores, whereas the VLM-based score is produced independently by visual judgment. During SFT filtering, the two scores are thresholded separately, and a task is retained only when both its rule-based score and VLM-based score meet their respective thresholds (both 0.8). As described in Section 3.1, RL uses only tasks with rule-based evaluators to provide reward signals. This design avoids the brittleness and computational cost of VLM judgment and prevents the calibration mismatch that could arise from averaging the two score types directly. Although both scores lie in $[0, 1]$, rule-based scores are derived from fine-grained, explicitly weighted completion items, whereas VLM scores generally reflect a coarser and more subjective judgment scale. The two scores therefore do not have interchangeable numerical semantics. Consequently, VLM judgment is used primarily to cover programmatic observability gaps during SFT-stage filtering.

Instruction and evaluator filtering. Filtering first checks whether the JSON schema, getter arguments, and Python code can be parsed and compiled statically. The system then initializes the environment from the initialization configuration and executes the evaluator before any action. The initial-state score must be zero, as a nonzero score indicates reward leakage. An independent review model next scores instruction quality and evaluator quality separately. The former assesses whether the instruction is unambiguous, reachable, and realistic, while the latter assesses whether the evaluator covers all success conditions, is sufficiently strict, and resists reward hacking. An instruction below the quality threshold is removed directly. If the instruction is valid but its evaluator fails an automated check or quality review, the system uses the previous evaluator and its failure information as revision context, regenerates the evaluator, and repeats the full validation and scoring procedure. The associated proposal is removed only after multiple repair attempts remain unsuccessful.

After correctness filtering, we use embedding similarity to remove near-duplicate instructions generated across batches, producing the task pool used for training stages. The RL task pool is then built by rolling out trajectories with the SFT-trained model on this pool and keeping tasks of moderate difficulty.

Prototype cross-application extension. Beyond the application–system compositions in the primary task pool, we implemented and began collecting a more structured form of cross-application task synthesis. The prototype uses the `related_app` field in the application configurations to construct an application graph. Applications serve as nodes, while edges encode compositional relations induced by shared artifact types or file formats, convertible intermediate artifacts, or complementary workflows. This graph provides a structured prior for composing applications rather than treating

all application combinations as equally plausible. For each task, the system first specifies a target number of applications, such as three, randomly samples a primary application, and then samples subsequent applications from its neighboring nodes. After each selection, the neighbors of the selected nodes are added to the candidate frontier, allowing the application set to expand along graph structure. To prevent the graph from overly constraining the composition space, each sampling step retains a small-probability jump mechanism that ignores graph connectivity and samples an application uniformly from the broader inventory. Artifact selection follows a corresponding compositional procedure and may introduce zero or more related artifacts. This setting expands the exploration space and requires the Exploration Agent to additionally infer cross-application dependencies, including the execution order and the transfer of artifacts or intermediate results, before proposing a long-horizon instruction.

We have already generated a small number of such prototype tasks and included them in the task pool. Their current scale is insufficient for a systematic evaluation of cross-application capabilities, so future work will expand the collection and study its effect on long-horizon execution across applications.

C.5 Instruction and Action Distributions

Figure 5 summarizes the synthesized tasks and desktop SFT data from four complementary perspectives. Panel (a) groups the 8,880 tasks generated before filtering by application domain. Panel (b) reports the task-level evaluator composition over all 8,880 generated tasks, distinguishing rule-only, VLM-only, and hybrid rule–VLM evaluation. Panel (c) partitions the full pool of 83,701 desktop SFT samples, collected from trajectories for 5,416 tasks after filtering, including successful samples, failed samples, and three invalid-sample categories covering tool-call format or schema errors, content-quality issues, and other issues. Panel (d) shows the distribution of 40,729 tool actions in the successful SFT samples.

C.6 General Data Mixture

The SFT data mixture follows a commonly used roughly 1:1 mixing ratio, consisting of approximately 31K filtered distilled desktop samples and 40K general-purpose samples. The general-purpose data covers GUI understanding, coding, mathematics, OCR, chart interpretation, and dialogue, helping stabilize model training and reduce over-specialization to desktop trajectories.

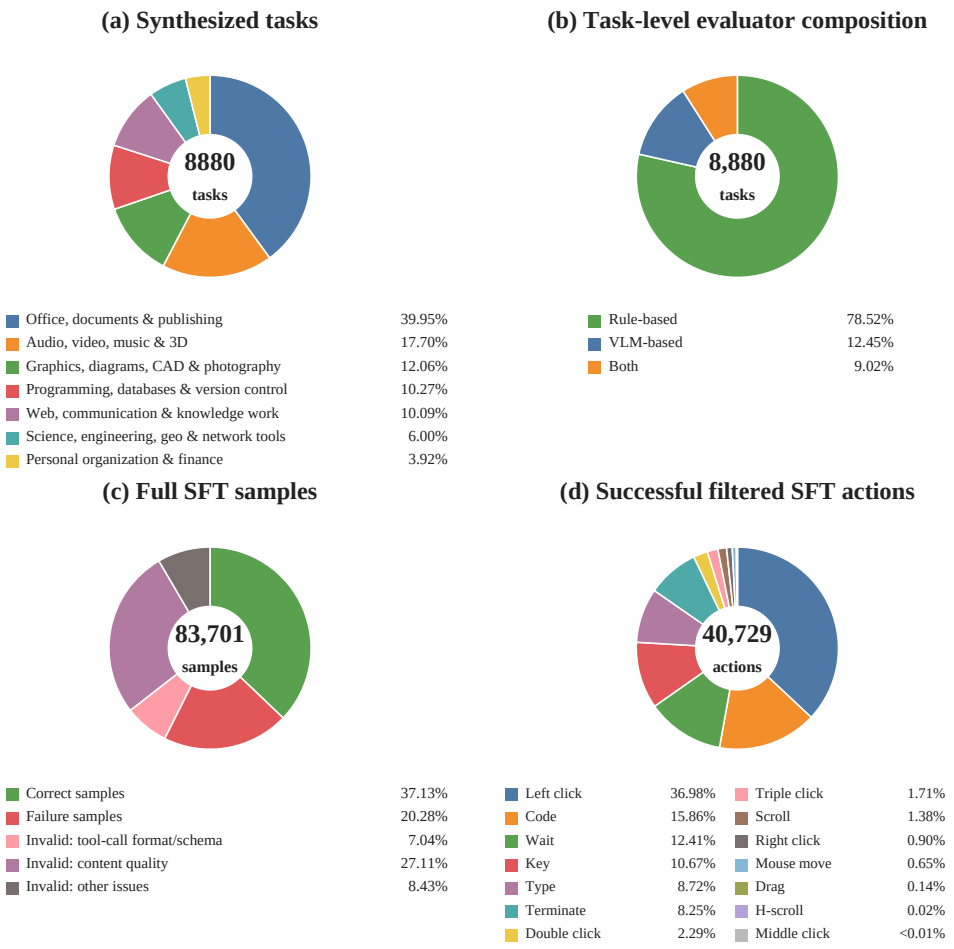

Figure 5: Distributions of synthesized instructions, tool actions in successful filtered SFT samples, task-level evaluator composition, and outcomes of the full desktop SFT sample pool. Percentages are computed from the raw counts for each panel and displayed to two decimal places.

D Training Details

D.1 SFT Configuration

We perform full-parameter SFT with Megatron-SWIFT (Zhao et al., 2025). Training uses tensor parallelism of 4, pipeline parallelism of 1, and bfloat16 precision. We set the global and micro batch sizes to 16 and 1, respectively, train for one epoch, use a maximum sequence length of 32,768, and enable sample packing. The learning rate is 1×10^{-5}, with a minimum learning rate of 1×10^{-6} and a warmup fraction of 0.05. In addition to the settings in Table 10, we enable full recomputation, FlashAttention (Dao et al., 2022), FlashLinearAttention (Yang & Zhang, 2024), and sequence parallelism for efficient and stable long-context full-parameter training.

D.2 GSPO RL Configuration

Our RL system is built on the asynchronous Slime framework (Zhu et al., 2025), augmented with custom training logic for online desktop interaction. Each rollout collection contains 8 prompts and samples 8 trajectories per prompt, yielding 64 rollout samples per collection. We use a maximum of 50 GUI interaction steps, a rollout temperature of 0.5, and a maximum response length of 10,240. Tasks are dynamically sampled according to an exponential moving average of their success rates.

Parameter	Value
Training machine	8× Nvidia H200
Training framework	Megatron-SWIFT
Training type	Full-parameter fine-tuning
Tensor parallel size	4
Pipeline parallel size	1
Precision	bfloat16
Global batch size	16
Micro batch size	1
Packing	Enabled
Training epochs	1
Maximum sequence length	32K
Optimizer	Adam
Adam β_1/β_2	0.9 / 0.95
Learning rate	1×10^{-5}
Minimum learning rate	1×10^{-6}
Warmup fraction	0.05
Enable Thinking	False

Table 10: OS-Helix SFT training parameters.

SGLang (Zheng et al., 2024) serves as the rollout backend, while Megatron (Shoeybi et al., 2019) is used for policy training. The RL objective is Step-balanced GSPO, with disabled KL divergence and equal weights of 1.0 for GUI and code steps. The full configuration is listed in Table 11.

D.3 Online RL Sampling, Segmentation, and Rewards

Uniform task sampling is inefficient for online CUA training because many rollouts are spent on tasks that are either already solved or still out of reach. Following the frontier-sampling intuition in prior ScaleCUA (Lv et al., 2026), we maintain a success-rate estimate for each task and bias sampling toward tasks near the current capability frontier.

For each task i, we initialize $\hat{p}_i = 0.5$. After each update window, if n_i new rollouts are available for task i, we update its success-rate estimate by

$$\hat{p}_i \leftarrow \alpha \hat{p}_i + (1-\alpha) \cdot \frac{1}{n_i} \sum_{j=1}^{n_i} \mathbf{1}[r_{i,j} > 0],$$

where α is the EMA retention coefficient, consistent with the task-sampling EMA parameter in Table 11. The task sampling weight is then computed with a Gaussian kernel,

$$w_i = \exp\left(-\frac{(\hat{p}_i - \mu)^2}{2\sigma^2}\right),$$

We set the target success rate to $\mu = 0.5$ and the bandwidth to $\sigma = 0.25$. Thus tasks with very high or very low estimated success rates receive lower weights, while tasks near the target success rate are sampled more often.

To control visual-context length, we use a bounded sliding-window strategy. Each rollout maintains a history of 5–10 screenshots. When the history exceeds 10 screenshots, we save the current stream as a training segment, discard the five oldest screenshots, retain the five most recent screenshots, and resume collecting observations. Only assistant responses generated while the retained screenshot history lies in this 5–10 range contribute to the loss. Earlier context tokens are retained only as conditioning information with loss mask 0.

Rewards come only from the paired environment evaluator, `env.evaluate`. We do not use auxiliary format rewards or handcrafted step-level rewards. For valid trajectories and segments, evaluator scores are real values in $[0, 1]$. If an assistant response contains an unparsable tool call, an action outside the allowed schema, or another invalid action-format failure, that response is extracted as a separate invalid sample and assigned reward 0, even if the remaining trajectory could otherwise obtain a positive task-level score. This prevents malformed actions from receiving accidental positive credit through later recovery.

Parameter	Value
Basic Configuration	
Base reinforcement learning framework	Slime
Total GPUs	32× Nvidia H200
Training GPUs	16× Nvidia H200
Rollout GPUs	16× Nvidia H200
Training steps	1,000
Data	
Task sampling	Success-rate EMA
Task-sampling EMA α	0.9
Task-sampling min weight / β_{sample}	0.1 / 1.0
Rollout	
Rollout backend	SGLang
Rollout GPUs per engine	1
Rollout batch size	8
Samples per prompt	8
Steps per rollout update	2
Rollout temperature	0.5
Maximum GUI steps	50
Maximum response length	10,240
Enable Thinking	False
Image history length	5–10 screenshots; retain newest 5 on overflow
Training	
Training Backend	Megatron
Actor GPUs per engine	4 (tp=4)
Frozen parameters	Vision parameters
Dynamic batch size	Enabled, max 30,000 tokens/GPU
Optimizer	Adam
Adam β_1/β_2	0.9 / 0.95
Learning rate	1×10^{-6}
Learning-rate schedule	Constant
Weight decay	0.1
GSPO Objective	
RL objective	Step-balanced GSPO
GUI/code step weights	1.0 / 1.0
Lower policy clip ϵ_{low}	0.2
Upper policy clip ϵ_{high}	0.28
Dual-clip coefficient c	3.0
KL loss	Disabled

Table 11: RL training parameters.

E Extra Experiments

E.1 Transfer to WindowsAgentArena

Table 12 reports additional results for the 27B backbone and its corresponding OS-Helix variants on WindowsAgentArena, providing evidence of transfer from the Ubuntu-based training setting to this Windows benchmark. We retain the GUI and Hybrid baselines from the main experiments to preserve the same comparison between action-space access and learned hybrid behavior. Of particular note, since the operating system environment is Windows, we restrict Code actions to Python only, excluding the use of Bash. When Cmd/PowerShell commands would otherwise be required, system calls are instead issued directly within Python; this adaptation is implemented via prompt revision.

Despite the absence of Windows-specific training data, OS-Helix improves on this benchmark. The SFT-only OS-Helix-27B raises the matched GUI backbone from 40.82% to 57.95%, a gain of 17.13 absolute points, and the RL variant reaches 63.87%, exceeding the GUI baseline by 23.05 points and the Hybrid baseline by 7.46 points. The unadapted Hybrid baseline also improves substantially over GUI (56.41% versus 40.82%), while OS-Helix-27B adds a further 1.54 points. These

Method	Setting	Max Steps	Success Rate (%)						
			Office	Web	Sys.	Code	Media	Util.	Avg.
Agentic Frameworks									
Agent S3 w/ GPT-5 (2025)	Hybrid	50	–	–	–	–	–	–	54.10
Agent S3 w/ GPT-5 (2025)	Hybrid	100	–	–	–	–	–	–	56.60
OS-Symphony w/ GPT-5-Mini (2026a)	Hybrid	50	42.86	73.00	79.17	68.42	48.66	66.67	62.15
OS-Symphony w/ GPT-5 (2026a)	Hybrid	50	54.76	73.00	75.00	42.11	70.09	75.00	63.45
General Models & Specialist Models									
Qwen3-VL-32B-Thinking (2025)	GUI	50	17.07	56.33	68.18	64.29	29.33	63.64	45.95
UI-TARS-1.5-7B (2025)	GUI	50	–	–	–	–	–	–	42.10
UI-TARS-2 (2025a)	GUI	50	–	–	–	–	–	–	50.60
UI-Mate-27B (2026)	GUI	–	–	–	–	–	–	–	66.20
EvoCUA-32B (2026)	GUI	–	–	–	–	–	–	–	56.48
EvoCUA-1.5-32B (2026)	GUI	–	–	–	–	–	–	–	62.10
Backbone Baselines									
Qwen3.5-27B (2026a)♣	GUI	50	30.23	36.33	62.50	50.00	28.41	50.00	40.82
Qwen3.5-27B (2026a)♣	Hybrid	50	41.86	63.00	83.33	54.17	42.70	66.67	56.41
OS-Helix-27B	Hybrid	50	39.53	66.33	91.67	54.17	39.75	75.00	57.95
OS-Helix-27B-RL	Hybrid	50	39.53	76.33	75.00	73.91	65.81	75.00	63.87

Table 12: Additional results on WindowsAgentArena. Office includes Writer and Calc; Web includes Edge and Chrome; Sys. denotes Windows system tasks; Code includes VS Code; Media includes VLC; Util. includes Windows utilities. ♣ marks baselines reproduced by us.

comparisons provide benchmark-specific evidence that learned GUI/code routing transfers to WindowsAgentArena under the reported Python-only prompt adaptation.

The remaining gap is also diagnostic. Manual inspection of failed WindowsAgentArena trajectories shows that OS-Helix-27B often identifies the correct high-level strategy but can fail in the concrete system-command layer, mixing Bash-like patterns into Windows execution or producing brittle PowerShell-style logic despite the Python-only prompt adaptation. This pattern is compatible with the absence of Windows-specific command data. In contextual comparisons reported on the same benchmark, OS-Helix-27B-RL slightly exceeds OS-Symphony w/ GPT-5 and Agent S3 w/ GPT-5 under the shared 50-step budget. These results motivate scaling the synthesis pipeline along the operating-system dimension and evaluating on broader Windows and macOS task distributions.

We further analyze infeasible tasks as a complementary limitation of the current data pipeline. Both OSWorld and WindowsAgentArena contain a non-negligible fraction of infeasible tasks, roughly 10% of their task sets. These tasks mix at least two phenomena: genuinely impossible instructions whose target condition cannot be satisfied, and GUI-infeasible tasks where the visual route may be blocked while a programmatic route could still modify the underlying state. Our synthesis pipeline is designed around executable goal states and therefore does not naturally generate infeasible training tasks. As a result, the model can under-learn when to stop and declare infeasibility, often continuing until the maximum interaction budget is exhausted. Conversely, for tasks that are infeasible only under GUI interaction, a hybrid agent may legitimately solve them through code, which can still be judged as incorrect if the benchmark expects infeasibility recognition. As a diagnostic, after removing infeasible tasks from the evaluation set, the average success rate on OSWorld and WindowsAgentArena increases by up to **3–5 absolute points**. This suggests that future task design and data synthesis should explicitly model infeasibility, distinguishing true task impossibility from modality-specific infeasibility so that agents can learn both effective execution and calibrated refusal.

E.2 Additional Analysis

The diagnostic experiments in this subsection are conducted with Qwen3-series models (Bai et al., 2025), which differ from the Qwen3.5-series base models used in the experiments above. The training data and data recipe, however, are kept exactly the same, so the comparison isolates how the same OS-Helix training signal behaves on a different backbone family.

Model	OSWorld-G	ScreenSpot-V2
Qwen3-VL-32B-Instruct	65.10	95.80
Qwen3-VL-32B-Instruct♣	61.35	95.05
OS-Helix-32B	57.21	93.47

Table 13: Grounding benchmark results on OSWorld-G and ScreenSpot-V2. ♣ represents the result reproduced by us.

All experiments in this subsection use supervised fine-tuning (SFT) only and do not include reinforcement learning (RL). Accordingly, the analyses below study the transferability of the learned GUI/code behavior and the information content of the SFT data, rather than the contribution of online policy optimization.

E.2.1 Grounding Ability Check

Table 13 reports grounding benchmark performance on Qwen3-VL-32B-Instruct and an OS-Helix-32B variant with the same data recipe as our Qwen3.5-series models. The reproduced base model obtains 61.35 on OSWorld-G (Xie et al., 2026) and 95.05 on ScreenSpot-V2 (Wu et al., 2025), while OS-Helix-32B obtains 57.21 and 93.47, respectively.

This decrease is expected rather than contradictory to our main results. The training objective does not only optimize pixel grounding; it also teaches the model to format tool calls, reason over files, invoke code when programmatic execution is more reliable, and recover from multi-step environment feedback. In this sense, the model reallocates some capacity away from isolated grounding benchmarks toward hybrid desktop execution. The key observation is that even when pure grounding slightly decreases, code-capable action selection can compensate for this loss in end-to-end tasks by avoiding brittle pixel-level interactions and solving precise file or data operations through the operating system.

E.2.2 Backbone Replacement Analysis

To test whether the training signal transfers beyond the original backbone, we repeat the SFT experiment with Qwen3-VL-32B-Instruct and Qwen3-VL-32B-Thinking. Table 14 reports the complete OSWorld breakdown; arrows in the OS-Helix rows denote absolute-point changes relative to the Qwen3-VL baseline under the same Setting. With the Instruct backbone, adding Code without training a hybrid policy slightly decreases the baseline from 34.17% (GUI) to 32.68% (Hybrid). After SFT, however, OS-Helix-32B reaches 50.56%, a +17.88-point improvement over the matched Hybrid baseline. With the Thinking backbone, the unadapted Hybrid baseline is somewhat better than GUI (42.88% versus 39.08%), but OS-Helix-32B-Thinking still improves it by +8.79 points to 51.67%. Thus, the result isolates two effects: Code availability is not itself sufficient for reliable hybrid execution, while the SFT training signal transfers across both backbones and converts the available action space into a substantially stronger policy.

The domain-level changes clarify what is learned. For Instruct, OS-Helix-32B improves all five domains over the matched Hybrid baseline. The corresponding gains in OS, Office, Daily, Prof., and Work are 16.67, 19.66, 16.83, 12.24, and 19.76 points, respectively. The gains are therefore broad rather than being explained by a single application family, with the largest improvements in Office and multi-application Work tasks. This pattern is consistent with learning cross-step state management and modality routing, rather than simply improving visual grounding on isolated screens. For Thinking, the gains remain broad in OS (+8.33), Office (+11.94), Daily (+11.40), and Work (+9.40), but Prof. decreases by 2.83 points. The single negative domain is a useful qualification: backbone replacement does not erase model-specific priors, and SFT can redistribute capability unevenly across application families. Nevertheless, the positive average and the gains across system, office, daily, and workflow tasks show that the central hybrid behavior is not tied to one backbone's pretrained tool-use bias.

These results also strengthen the distinction between action-space access and policy learning in the main claim. The Instruct baseline provides a controlled counterexample in which the same model becomes slightly worse when Code is merely exposed at inference time. In contrast, the

Model	Setting	Step	OS	Office	Daily	Prof.	Work	Avg.
Qwen3-VL-32B-Instruct♣	GUI	50	41.67	31.54	51.23	44.90	15.57	34.17
Qwen3-VL-32B-Instruct♣	Hybrid	50	50.00	33.25	39.31	38.78	18.78	32.68
Qwen3-VL-32B-Thinking♣	GUI	50	58.33	35.84	54.44	55.10	18.71	39.08
Qwen3-VL-32B-Thinking♣	Hybrid	50	62.50	40.97	49.81	53.85	27.95	42.88
OS-Helix-32B	Hybrid	50	66.67(↑16.7)	52.91(↑19.7)	56.14(↑16.8)	51.02(↑12.2)	38.54(↑19.8)	50.56(↑17.9)
OS-Helix-32B-Thinking	Hybrid	50	70.83(↑8.3)	52.91(↑11.9)	61.21(↑11.4)	51.02(↓2.8)	37.35(↑9.4)	51.67(↑8.8)

Table 14: Backbone replacement analysis on OSWorld. ♣ marks baselines reproduced by us.

Training Recipe	Inference Context	OS	Office	Daily	Prof.	Work.	Avg.
Normal	Normal	66.67	52.91	56.14	51.02	38.54	50.56
Normal	− CoT	54.17(↓12.5)	28.98(↓23.9)	53.58(↓2.6)	38.77(↓12.3)	31.43(↓7.1)	37.82(↓12.7)
− CoT	Normal	50.00(↓16.7)	35.55(↓17.4)	46.99(↓9.2)	38.77(↓12.3)	28.87(↓9.7)	37.74(↓12.8)
− History CoT	Normal	60.87(↓5.8)	43.51(↓9.4)	54.19(↓2.0)	39.80(↓11.2)	31.30(↓7.2)	43.22(↓7.3)
+ Failure Data	Normal	54.17(↓12.5)	48.67(↓4.2)	54.80(↓1.3)	50.28(↓0.7)	35.16(↓3.4)	47.06(↓3.5)

Table 15: Training-recipe ablations on OSWorld with Qwen3-VL-32B-Instruct. "− CoT" denotes the removal of all CoT text while retaining action text only; "− History CoT" denotes the removal of historical CoT while retaining the current-step CoT and action text; "+ Failure Data" denotes augmenting the SFT samples with partial rejection-sampled failure samples.

trained model gains most strongly on domains where successful execution requires coordinating GUI observations, executable operations, and persistent task state. The improvement should therefore be interpreted as evidence for learned coordination and routing, not as an artifact of giving the model an additional tool.

E.2.3 Training Recipe and Context Analysis

We next isolate how the structure of demonstrations and test-time history affects performance, using Qwen3-VL-32B-Instruct with OSWorld. The normal recipe reaches 50.56%. Removing CoT from training and retaining actions only reduces performance to 37.74% (-12.82 points), while keeping the training data unchanged but removing CoT from the inference context yields 37.82% (-12.74 points). The near-identical degradation is important: the reasoning trace is not merely an auxiliary training target. It also functions as an execution-time state representation that helps the model maintain intent, resolve intermediate observations, and select between GUI and code actions across multi-step execution.

Removing only historical CoT is less damaging, but still lowers the average to 43.22% (-7.34 points). This partial recovery indicates that the current-step decision can retain useful local information, yet removing the accumulated reasoning history weakens temporal credit assignment and state tracking. The effect is especially visible in Work., where success falls from 38.54% to 28.87% without training CoT, 31.43% without inference CoT, and 31.30% without historical CoT. Multi-application tasks therefore expose the central role of structured context most clearly: their difficulty lies not only in individual grounding decisions, but in preserving a coherent plan while the active application and tool modality change.

Finally, mixing in trajectories rejected during filtering reduces the score to 47.06% (-3.50 points). This smaller but consistent decline suggests that data quality is not interchangeable with data quantity. Rejected failures may preserve superficial action patterns, but they also introduce inconsistent or non-executable supervision that blurs the association between observations, tool calls, and successful state transitions. Taken together, the ablations support a precise interpretation of the SFT recipe: CoT and history provide the structured latent state needed for hybrid execution, while rejection filtering protects the reliability of the behavior-to-outcome mapping. The full comparison is given in Table 15.

Metric	Comparison	Δ (pp)	95% CI
Primary score (Pass@1)	SFT − Hybrid baseline	+16.21	[+10.75, +21.66]
Primary score (Pass@1)	RL − Hybrid baseline	+19.89	[+14.24, +25.59]
Primary score (Pass@1)	RL − SFT	+3.68	[+0.93, +8.33]
Pass@3	RL − SFT	+5.56	[+2.24, +8.97]

Table 16: Paired-bootstrap confidence intervals on OSWorld. We resample the 361 tasks with replacement for 100,000 replicates and recompute the paired mean-score difference.

F STEP-BALANCED GSPO DETAIL

During reinforcement learning, a training sample may contain multiple assistant responses that contribute to the policy loss, with each response corresponding to one action step. The optimization unit of a training sample therefore need not coincide with an individual environment action step. In particular, the assistant span associated with a code action step is often substantially longer than that associated with a GUI step. The standard GSPO objective computes one sample-level importance ratio by aggregating token-level log-importance weights over all valid assistant tokens in the training sample (Zheng et al., 2025). Although this prevents a sample from receiving a larger sample-level weight solely because it contains more tokens, it does not equalize the contributions of different action steps within the same sample. As a result, a long code action can dominate both the sample-level ratio and the clipping decision even when it is only one of several actions in the training sample.

This section formalizes the effect and derives a step-balanced aggregation for the GSPO ratio. Our goal is to preserve GSPO's token-level importance weights and clipping structure while changing their sample-level aggregation to remove the deterministic dependence of an action step's aggregate gradient coefficient on its token count. Recent work has also studied the granularity of importance ratios for LLM policy optimization, including sequence-level GSPO (Zheng et al., 2025), subsentence-level ratios (Yang et al., 2025a), cumulative token ratios (Zhang et al., 2026b), and hybrid token/sequence ratios (Min et al., 2026). These methods differ from our setting at the objective level. SSPO assigns separate ratios to text fragments, but its published objective aggregates fragments with token-length normalization and does not use environment-action boundaries or action-type weights. CTPO and DHPO use token-level (or token/response-hybrid) objectives, while GSPO and GSPO-token use one response-level ratio. None of these objectives replaces token-count weights by normalized weights over labeled GUI/code action steps within a multi-action training sample.

Notation. Let i denote a training sample. Training sample i contains K_i action steps. We denote by $a_{i,k}$ the set of assistant-token positions that contribute to the loss for action step k, and by $|a_{i,k}|$ its cardinality. Thus, $a_{i,k}$ may include reasoning tokens, tool-format wrapper tokens, and action-content tokens. Let $m_{i,t} \in \{0, 1\}$ indicate whether token position t contributes to the loss. For each loss-participating assistant token, define the token-level importance weight and its logarithm as

$$w_{i,t}(\theta) = \frac{\pi_\theta(y_{i,t} \mid h_{i,t})}{\pi_{\text{old}}(y_{i,t} \mid h_{i,t})}, \qquad \delta_{i,t} = \log w_{i,t}(\theta) = \log \pi_\theta(y_{i,t} \mid h_{i,t}) - \log \pi_{\text{old}}(y_{i,t} \mid h_{i,t}).$$

Here, $y_{i,t}$ is the sampled token and $h_{i,t}$ is its preceding causal context. The total number of valid assistant tokens in sample i is

$$N_i = \sum_t m_{i,t} = \sum_{k=1}^{K_i} |a_{i,k}|.$$

For each action step, we define the mean token-level log-importance weight as

$$\bar{\delta}_{i,k} = \frac{1}{|a_{i,k}|} \sum_{t \in a_{i,k}} \delta_{i,t}.$$

Let $\tau_{i,k}$ denote the action type of step k, e.g., $\tau_{i,k} \in \{\text{GUI}, \text{code}\}$. We assign each action type a nonnegative balancing coefficient $\alpha_{\tau_{i,k}}$. At least one action type in each sample must have a positive coefficient so that the normalizer below is nonzero.

F.1 GSPO

The standard GSPO sample-level importance ratio for sample i is

$$w_i^{\mathrm{GSPO}}(\theta) = \exp\left(\frac{1}{N_i}\sum_t m_{i,t}\delta_{i,t}\right).$$

Grouping the loss-participating tokens by action step gives

$$\log w_i^{\mathrm{GSPO}}(\theta) = \sum_{k=1}^{K_i} \frac{|a_{i,k}|}{N_i}\bar{\delta}_{i,k}.$$

Therefore, in standard GSPO, the implicit weight of action step k in the sample-level log-ratio is

$$\omega_{i,k}^{\mathrm{GSPO}} = \frac{|a_{i,k}|}{\sum_j |a_{i,j}|}.$$

This weight is determined by token count rather than by action semantics. If a sample contains two short GUI actions and one long code action, the code action can dominate $\log w_i^{\mathrm{GSPO}}(\theta)$ simply because it has more valid assistant tokens. The same ratio is then used for all tokens in the sample, so the long action also has disproportionate influence over whether the whole sample is clipped.

F.2 Step-Balanced GSPO

We replace token-count-based action weighting with explicit action-level weighting. Define

$$Z_i = \sum_{k=1}^{K_i} \alpha_{\tau_{i,k}}, \qquad \omega_{i,k} = \frac{\alpha_{\tau_{i,k}}}{Z_i}.$$

The step-balanced GSPO sample-level importance ratio is

$$\boxed{w_i^{\mathrm{step}}(\theta) = \exp\left(\sum_{k=1}^{K_i} \omega_{i,k}\bar{\delta}_{i,k}\right)}.$$

Equivalently, the ratio can be written directly in terms of policy log-probabilities as

$$\boxed{w_i^{\mathrm{step}}(\theta) = \exp\left[\frac{\sum_{k=1}^{K_i} \alpha_{\tau_{i,k}} \left(\frac{1}{|a_{i,k}|}\sum_{t\in a_{i,k}} \log \frac{\pi_\theta(y_{i,t}|h_{i,t})}{\pi_{\mathrm{old}}(y_{i,t}|h_{i,t})}\right)}{\sum_{k=1}^{K_i} \alpha_{\tau_{i,k}}}\right]}.$$

This formulation differs from standard GSPO only in the aggregation weights used to form the sample-level log-ratio. Standard GSPO weights each action step in proportion to its number of valid assistant tokens, whereas Step-balanced GSPO uses explicit action-level weights. When these weights are set proportional to token counts, the formulation recovers the original GSPO ratio as a special case.

The policy objective retains the standard GSPO structure. Given the sample-level advantage A_i, the unclipped objective is $J_i(\theta) = A_i w_i^{\mathrm{step}}(\theta)$, and the clipped objective is

$$J_i^{\mathrm{clip}}(\theta) = \min\left(A_i w_i^{\mathrm{step}}(\theta), A_i \,\mathrm{clip}\left(w_i^{\mathrm{step}}(\theta), 1-\epsilon_{\mathrm{low}}, 1+\epsilon_{\mathrm{high}}\right)\right).$$

Any additional dual-clipping rule for negative advantages is applied in the same way as in standard GSPO, after replacing its sample-level ratio with $w_i^{\mathrm{step}}(\theta)$. In the reported experiments, the KL loss is disabled (Table 11), so the objective contains no additional token-level KL term. If a KL penalty is enabled in future variants, applying corresponding step-level weighting would avoid reintroducing token-count-dependent regularization. The per-sample training loss is $L_i(\theta) = -J_i^{\mathrm{clip}}(\theta)$, and the batch loss over B samples is

$$L(\theta) = \frac{1}{B}\sum_{i=1}^{B} L_i(\theta).$$

F.3 Gradient Contribution

We now show that the proposed ratio removes the deterministic linear dependence between an action step's aggregate scalar gradient coefficient and its token count. Consider the unclipped branch, where $L_i(\theta) = -A_i w_i^{\text{step}}(\theta)$. For any token $t \in a_{i,k}$, only $\bar{\delta}_{i,k}$ depends on the corresponding current-policy log-probability. Therefore,

$$\frac{\partial \log w_i^{\text{step}}(\theta)}{\partial \log \pi_\theta(y_{i,t} \mid h_{i,t})} = \omega_{i,k} \frac{1}{|a_{i,k}|}.$$

Since $\partial w_i^{\text{step}} / \partial \log w_i^{\text{step}} = w_i^{\text{step}}$, we have

$$\frac{\partial L_i}{\partial \log \pi_\theta(y_{i,t} \mid h_{i,t})} = -A_i w_i^{\text{step}}(\theta) \frac{\omega_{i,k}}{|a_{i,k}|}.$$

Summing over all tokens in action step k gives

$$\sum_{t \in a_{i,k}} \frac{\partial L_i}{\partial \log \pi_\theta(y_{i,t} \mid h_{i,t})} = -A_i w_i^{\text{step}}(\theta) \omega_{i,k}.$$

Thus, the aggregate scalar coefficient of action step k is independent of $|a_{i,k}|$. This is the sense in which Step-balanced GSPO changes the optimization signal: it leaves the token-level importance weights $w_{i,t}$ intact, but changes how their log values are aggregated into the sample-level importance ratio. Under standard GSPO, a long serialized action occupies a larger share of the sample-level log-ratio, so a successful code step can receive more aggregate scalar credit simply because it contains more assistant tokens. This does not mean that the routing token selecting the code tool is directly multiplied by the number of subsequent code tokens. Rather, the longer serialized action contributes more token-level gradient terms under the same sample-level advantage and can therefore exert disproportionate influence on the update direction. Step-balanced GSPO removes this deterministic length-based multiplier from the action-step aggregate coefficient, reinforcing a successful action as a step rather than in proportion to its serialized length. The full parameter gradient still depends on token-specific gradient directions, clipping activation, and reward correlations, so this analysis explains a source of token-length-driven drift toward code actions rather than providing a formal guarantee of correct modality routing.

G Case Study

This section presents two filtered synthesis tasks covering an office/code workflow and a media workflow. The first is grounded in a spreadsheet, whereas the second constructs and transfers persistent application configuration between a recording tool and a video editor. Together, they illustrate application–artifact composition, environment-grounded task specification, and component-wise terminal verification in executable task records. Both use reviewed rule-based evaluators that assign partial terminal credit across independently verifiable task components. We reproduce the task metadata, complete instructions, and selected rule code to make these success conditions concrete.

G.1 Synthesis Task Case Study 1: Office and Code Workflow

The first task spans LibreOffice Calc and VSCode. It starts from a spreadsheet containing vehicle records, requires row-level categorization in the spreadsheet, and then asks for a runnable Python audit script over the saved artifact. The task is labeled complex and estimated to require 30 interaction steps.

Reviewed Synthesized Task 1

Task ID: `86350ffd-56e5-4fd9-90da-416688b5b182`
Applications: LibreOffice Calc 7.3.7.2; VSCode 1.85.1
Asset source: Multilingual-Multimodal-NLP/TableBench spreadsheet library
Category: `mixed` **Complexity:** complex **Estimated steps:** 30
Initialization: Open `/home/user/Desktop/test_files/xlsx/tablebench_0a53ffc65168d29f6a0784ec6741ffb0.xlsx` in LibreOffice Calc.
Instruction: The spreadsheet contains car data with columns `Model`, `Fuel Type`, `US gallons (mpg)`, `L/100km`, and `MZ Rating`. Add a new column F with the header `Efficiency Category`. For each car, assign `High` when `L/100km` is below 5.0, `Medium` when it is at least 5.0 and below 6.0, and `Low` otherwise. Save the spreadsheet. Then create `~/Desktop/categorize_check.py`, which reads the saved spreadsheet and prints the category counts as `High: <N>`, `Medium: <N>`, and `Low: <N>`, one line per category.
Rule inputs: modified source spreadsheet; `categorize_check.py`; stdout from executing the script.

The evaluator averages three terminal checks. The first gives weighted credit for the header, correct per-row labels, and the absence of invalid labels. The second checks that the script reads an XLSX file and implements data-derived counting rather than only hard-coding an answer. The third runs the script and scores each category count independently before checking the strict output order and format.

This case illustrates component-wise terminal scoring for a workflow that crosses an office application and a code editor. The reward is assigned only after the episode, but it distinguishes spreadsheet correctness, the construction of a data-derived program, and the program's executed output rather than collapsing those outcomes into a single pass–fail check.

Rule-Based Evaluator for Task 1

```
def call_rule_judge_1(result, expected, **options) -> float:
    """Check the spreadsheet header and per-row efficiency labels."""
    try:
        import openpyxl
        wb = openpyxl.load_workbook(result, data_only=True)
        ws = wb['Sheet1'] if 'Sheet1' in wb.sheetnames else wb.worksheets[0]

        f1_val = ws.cell(row=1, column=6).value
        header_score = 1.0 if f1_val is not None and str(f1_val).strip() == 'Efficiency Category' else 0.0

        data_rows = []
        for row in range(2, ws.max_row + 1):
            d_val = ws.cell(row=row, column=4).value
            f_val = ws.cell(row=row, column=6).value
            if d_val is not None or f_val is not None:
                data_rows.append((d_val, f_val))
        if not data_rows:
            return 0.0

        valid_values = {'High', 'Medium', 'Low'}
        correct = 0
        invalid = 0
        for d_val, f_val in data_rows:
            try:
                l100 = float(d_val)
                value = str(f_val).strip()
            except Exception:
                invalid += 1
                continue
            if value not in valid_values:
                invalid += 1
                continue
            expected_value = 'High' if l100 < 5.0 else ('Medium' if l100 < 6.0 else 'Low')
            if value == expected_value:
                correct += 1
            else:
                invalid += 1

        total = len(data_rows)
        correctness_score = correct / total
        no_invalid_score = max(0.0, 1.0 - invalid / total)
        return round(0.2 * header_score + 0.5 * correctness_score + 0.3 * no_invalid_score, 4)
    except Exception:
        return 0.0


def call_rule_judge_2(result, expected, **options) -> float:
    """Check that the audit script derives category counts from the spreadsheet."""
    try:
        import os
```

Rule-Based Evaluator for Task 1 (continued)

```
      import re
      if not os.path.isfile(result):
         return 0.0
      with open(result, 'r', encoding='utf-8', errors='replace') as stream:
         content = stream.read()
      if not content.strip():
         return 0.0

      lower = content.lower()
      score = 0.0
      reads_xlsx = any(package in lower for package in ('openpyxl', 'pandas', 'xlrd'))
      references_file = ('tablebench_0a53ffc65168d29f6a0784ec6741ffb0' in content or
                  ('.xlsx' in content and any(token in lower for token in ('read_excel', 'load_workbook'
                        , 'open'))))
      has_conditionals = bool(re.search(r'\bif\b.*\belse\b|\belif\b', content))
      has_threshold = bool(re.search(r'5\.0|5,0|[<>=]=?\s*5|[<>=]=?\s*6', content))
      has_labels = all(re.search(label, content, re.IGNORECASE) for label in ('high', 'medium', 'low'))
      has_loop = bool(re.search(r'\bfor\b|\bwhile\b|iterrows|itertuples', content))
      has_count_logic = bool(re.search(r'count|sum|len|value_counts|groupby', lower))

      if reads_xlsx:
         score += 0.25
      if references_file:
         score += 0.25
      if has_conditionals or has_threshold:
         score += 0.15
      if has_labels:
         score += 0.20
      if 'print' in content:
         score += 0.15
      if not reads_xlsx or (not has_loop and not has_count_logic):
         score *= 0.5
      return round(min(score, 1.0), 4)
   except Exception:
      return 0.0


def call_rule_judge_3(result, expected, **options) -> float:
   """Run the script output against counts derived from the saved spreadsheet."""
   try:
      import re
      import openpyxl
      lines = [line.strip() for line in (result or '').splitlines() if line.strip()]
      pattern = re.compile(r'^(High|Medium|Low):\s*(\d+)$')
      parsed = {}
      for line in lines:
         match = pattern.match(line)
         if match:
            parsed[match.group(1)] = int(match.group(2))
      if not all(label in parsed for label in ('High', 'Medium', 'Low')):
         return 0.0

      wb = openpyxl.load_workbook(expected, data_only=True)
      ws = wb['Sheet1'] if 'Sheet1' in wb.sheetnames else wb.worksheets[0]
      counts = {'High': 0, 'Medium': 0, 'Low': 0}
      for row in range(2, ws.max_row + 1):
         value = ws.cell(row=row, column=4).value
         if value is None:
            continue
         l100 = float(value)
         counts['High' if l100 < 5.0 else ('Medium' if l100 < 6.0 else 'Low')] += 1

      count_score = sum(parsed[label] == counts[label] for label in counts) / 3.0
      order_ok = len(lines) == 3 and [pattern.match(line).group(1) for line in lines] == ['High', 'Medium'
          , 'Low']
      return round(0.7 * count_score + 0.3 * float(order_ok), 4)
   except Exception:
      return 0.0
```

G.2 Synthesis Task Case Study 2: Media Workflow

The second task connects OBS Studio and Shotcut. It configures a recording pipeline in OBS and then transfers the resulting resolution and frame-rate requirements to a compatible editing project in Shotcut. The task is labeled medium and estimated to require 22 interaction steps.

Reviewed Synthesized Task 2

Task ID: `df728e7b-240a-49f8-972e-27171051f3ab`
Applications: OBS Studio 27.2.3+dfsg1-1; Shotcut 22.01.30
Category: `app_only` **Complexity:** medium **Estimated steps:** 22
Initialization: Launch OBS Studio.
Instruction: Configure OBS Studio with recording path `/home/user/Videos`, recording format `mkv`, output resolution `1280x720`, and common frame rate `30`. Then create a Shotcut project whose video mode is `HD 720p 30 fps` so that it matches the OBS output configuration. Save the project as `~/Videos/obs_match_project.mlt`. Finally, create `~/Desktop/video_config.txt` with exactly two lines: `Resolution: 1280x720` and `FPS: 30`.
Rule inputs: persisted OBS profile configuration; saved Shotcut `.mlt` project; final text record.

The evaluator averages configuration checks for the two applications and the final text record. OBS receives partial credit according to the number of correct recording-path, format, resolution, and frame-rate fields. Shotcut receives full credit for a matching 720p/30 project profile and partial credit when two of its three core fields match. The text record is scored independently by line.

This task has an explicit cross-application handoff: OBS establishes the recording specification, and Shotcut must instantiate a project that is compatible with it. The evaluator checks persistent application state and project metadata rather than replaying a prescribed action sequence, while the partial terminal score distinguishes incomplete configuration from a fully compatible production setup.

Rule-Based Evaluator for Task 2

```
def call_rule_judge_1(result, expected, **options) -> float:
   """Check persistent OBS recording and video settings."""
   try:
      import configparser
      parser = configparser.RawConfigParser()
      parser.read_string(result or '')

      def get_key(key):
         for section in parser.sections():
            for name, value in parser.items(section):
               if name.lower() == key:
                  return value.strip()
         return None

      recording_path = get_key('recfilepath') or get_key('filepath')
      recording_format = get_key('recformat2') or get_key('recformat')
      output_x = get_key('outputcx') or get_key('basecx')
      output_y = get_key('outputcy') or get_key('basecy')
      fps = get_key('fpscommon')
      checks = [
         recording_path == '/home/user/Videos',
         recording_format is not None and recording_format.lower() == 'mkv',
         output_x == '1280',
         output_y == '720',
         fps == '30',
      ]
      passed = sum(checks)
      return 1.0 if passed == 5 else (round(passed / 5.0, 4) if passed >= 3 else 0.0)
   except Exception:
      return 0.0


def call_rule_judge_2(result, expected, **options) -> float:
   """Check that the Shotcut project matches the OBS 720p/30 configuration."""
   try:
      import xml.etree.ElementTree as ET
      root = ET.fromstring(result)
      profile = root if root.tag == 'profile' else next((node for node in root.iter('profile')), None)
      if profile is None:
         return 0.0

      width_ok = profile.get('width', '').strip() == '1280'
      height_ok = profile.get('height', '').strip() == '720'
      try:
         fps = float(profile.get('frame_rate_num', '')) / float(profile.get('frame_rate_den', '1'))
         fps_ok = abs(fps - 30.0) < 0.05
      except Exception:
         fps_ok = False
      passed = sum((width_ok, height_ok, fps_ok))
      return 1.0 if passed == 3 else (0.5 if passed == 2 else 0.0)
   except Exception:
      return 0.0
```

Rule-Based Evaluator for Task 2 (continued)

```
def call_rule_judge_3(result, expected, **options) -> float:
    """Check the two-line handoff record."""
    try:
        lines = (result or '').splitlines()
        while lines and not lines[-1].strip():
            lines.pop()
        if len(lines) != 2:
            return 0.0
        resolution_ok = lines[0] == 'Resolution: 1280x720'
        fps_ok = lines[1] == 'FPS: 30'
        return 1.0 if resolution_ok and fps_ok else (0.5 if resolution_ok or fps_ok else 0.0)
    except Exception:
        return 0.0
```

G.3 Agent Behavior Case Study: GUI–Code Coordination

Figure 6 shows two successful trajectories in which OS-Helix coordinates GUI perception with code execution. These cases are intended to make the learned hybrid behavior concrete: the GUI is used where the task depends on the current application state or a user-facing interface, while code is used where the same goal contains structured extraction, precise inspection, or repetitive operations.

In the first case, the agent starts from the browser/PDF interface and uses the opened course material to infer the lecture-file URL pattern. Once this pattern is grounded by GUI observation, the agent switches to code and downloads the remaining weekly PDFs in a loop. The important behavior is not simply that code is faster for downloading files, but that the code action is conditioned on information first obtained from the live GUI state. This illustrates a complementary route in which GUI interaction resolves task-specific context and code then executes the scalable part of the workflow.

In the second case, the agent inspects a payment spreadsheet with code to identify students whose tuition has not been paid, then returns to the GUI to fill the e-mail recipients in the mail client. Here, code provides reliable structured filtering over tabular records, while GUI interaction completes the user-facing communication step. Together, the two examples show that the unified action space allows the policy to decompose a task across modalities instead of committing to a single interface throughout the trajectory.

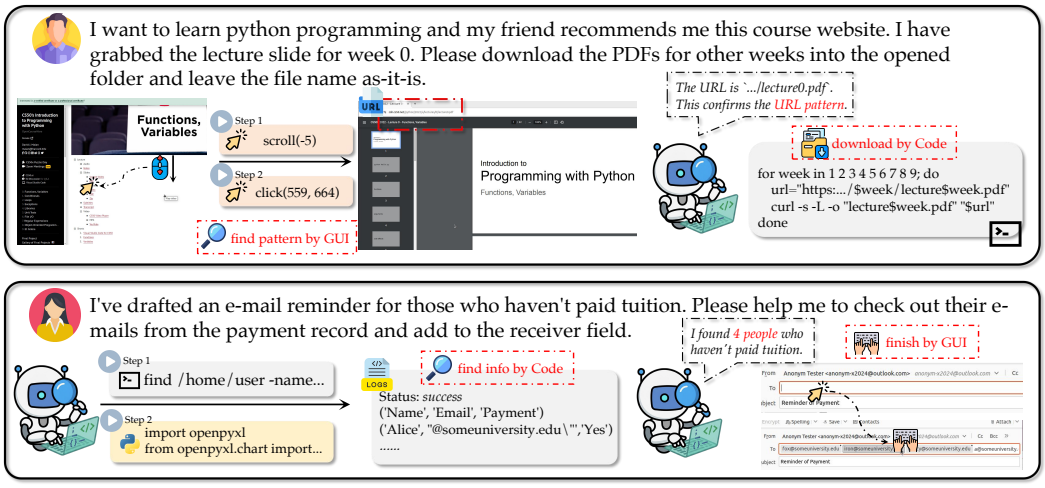

Figure 6: Successful GUI–code coordination cases. The top case first uses GUI observation to infer the URL pattern from an opened lecture slide, then uses code to download the remaining weekly PDFs. The bottom case uses code to extract payment records from a spreadsheet and then returns to the GUI to fill the e-mail recipients.

G.4 Agent Behavior Case Study: Cross-Modality Verification

Figure 7 presents two successful trajectories where one modality solves the task and the other verifies or repairs the result. This behavior is qualitatively different from merely choosing between GUI

and code at the beginning of a task: the agent can use the second modality as an intermediate feedback channel, detect a mismatch between the intended and realized artifact, and then continue the trajectory with a corrective action.

In the spreadsheet-chart case, the agent first creates the requested clustered column chart with code. The subsequent GUI observation reveals that the produced chart only contains the Sales series and misses COGS, which would be difficult to notice from the code path alone without explicitly checking the rendered artifact. The agent then repairs the chart through code, producing the required two-series visualization. This trace shows how GUI feedback can expose visual or application-level errors in code-generated artifacts and turn them into actionable corrections.

In the slide-editing case, the agent performs the visible edits through GUI actions: it adds the note on the second slide and bolds the title. It then switches to code-based inspection of the saved presentation file to verify that the persistent artifact matches the instruction, including both the note text and the title formatting. This reverse direction, GUI for editing and code for validation, highlights that the benefit of the hybrid policy is bidirectional. Across the four successful traces in Appendix G.3 and Appendix G.4, the learned policy uses GUI and code as mutually checking and mutually enabling tools, supporting the claim that a unified GUI/code action space yields complementary problem solving rather than replacing one modality with the other.

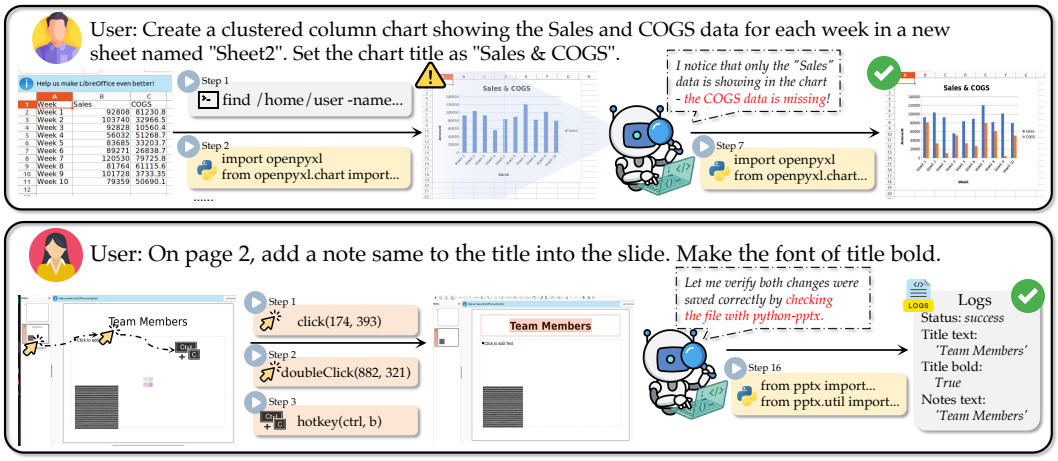

Figure 7: Successful cross-modality solve-and-verify cases. The top case creates a spreadsheet chart with code and then detects from the GUI that one data series is missing before repairing it. The bottom case completes slide editing through GUI actions and then verifies the saved file with code.

G.5 Failure Analysis: Evaluation Coverage and Cross-Modality Handoffs

Figure 8 presents two representative failures that expose a gap between task specification and evaluation coverage, as well as a state-handoff problem at the GUI–code boundary. Rather than indicating a deficiency of either modality in isolation, these traces identify systematic challenges for a hybrid policy in interpreting visual intent and maintaining persistent state across modalities.

The top trace illustrates a specification–evaluation coverage gap. "Preserve the original formatting" defines a visual equivalence condition involving row banding, numeric display, borders, and other rendered attributes. When the instruction and validation signal do not fully characterize these attributes, or do not cover admissible solution paths equivalently, they create an underspecified proxy objective that can favor one modality. After code-based conversion, the agent detects a mismatch through the GUI but attributes it to missing currency symbols, adding dollar signs rather than restoring the missing alternating row shading. A natural GUI solution would select the Calc table and paste it into Writer, preserving its rendered appearance; serialization through code instead requires the visual formatting to be reconstructed explicitly. This case motivates aligning user intent, rendering semantics, and verification coverage so that the benchmark recognizes the full set of valid solutions without presupposing an action modality.

The bottom trace exposes a cross-modality state-handoff failure. The agent first completes the first subgoal correctly through the GUI. When code subsequently edits and saves the second subgoal, it writes only a local disk-side state and does not incorporate the preceding modification that remains in the GUI-resident state. The GUI then mistakes the code-side save for a global commit and selects "Don't Save", discarding the correct update for the first subgoal. This is not a planning failure on either subgoal in isolation; it is a loss of persistence and commit semantics at the modality transition. Reliable hybrid execution requires reconciling GUI-resident and code-side artifact states before handoff, while maintaining a consistent representation of save scope and commit status.

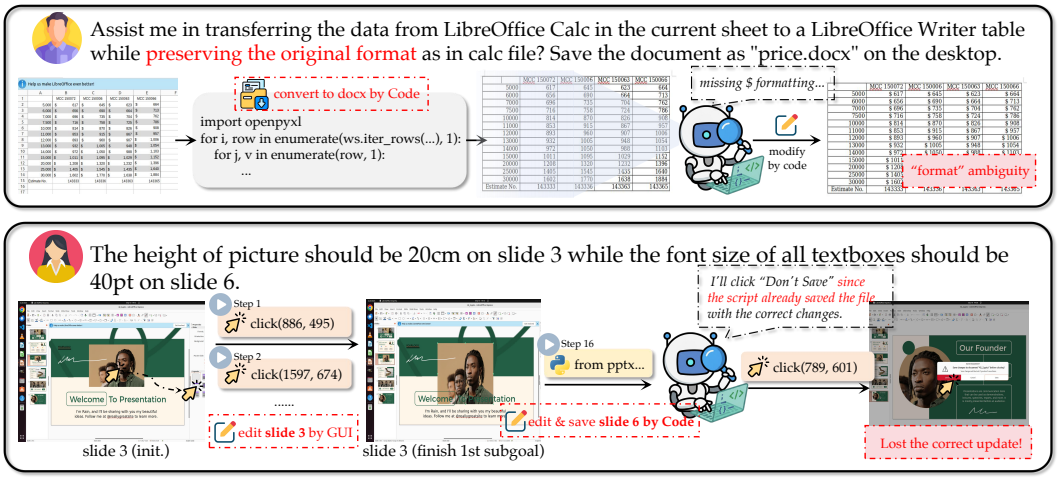

Figure 8: Representative failure cases. **Top:** In a Calc-to-Writer transfer task requiring preservation of the original formatting, code-based conversion fails to retain the alternating row shading. The subsequent GUI observation identifies a mismatch but misattributes it to missing currency symbols, so the agent adds dollar signs instead of restoring the table style. **Bottom:** After correctly completing the first PowerPoint subgoal through GUI actions, the agent edits and saves the second with code. The code-side save does not include the unsaved GUI state, and selecting "Don't Save" subsequently discards the earlier update.

G.6 Trajectory Audit

We manually annotated 30 representative trajectories from the multi-step, cross-application Workflow subset to examine whether modality transitions reflect task-grounded coordination. For each trajectory, we recorded the final outcome, a compressed flow of the key GUI and code stages involved in coordination, and the purpose of any state or information handoff. The reported flow retains the principal stages that produce, consume, verify, or repair the reported handoff, while omitting repeated within-modality actions and transitions unrelated to that handoff. As summarized in Table 17, 27/30 trajectories contain a grounded GUI/code handoff, including 22/30 in which one modality verifies or repairs state produced by the other. The audited set contains 21 successful, 4 partially successful, and 5 failed trajectories. In four of the five failed trajectories, cross-modal state is transferred correctly and failure arises later from task incompletion or application/tool errors. The remaining trajectory exposes a failed handoff: the PDF source observed in the GUI is replaced by a different, unverified URL when code writes the spreadsheet. As the audit covers only a representative task subset, the observed behavioral frequencies may not reflect the overall trends across the full benchmark. Moreover, the audited trajectories do not establish that the observed routing decisions are optimal. Even with these limitations, the examples directly show that OS-Helix performs task-grounded information handoffs and cross-modal verification and repair, using GUI and code as mutually enabling and checking mechanisms. This process-level evidence complements aggregate task scores, which capture outcomes rather than coordination behavior, and further supports our central claim that the model learns substantive cross-modal coordination.

Task ID	Outcome	Key Modality Flow	Handoff Purpose	Label
2c9fc0de	Success	Code only	Command-line completion; no handoff	A
e135df7c	Success	Code only	Programmatic conversion; no handoff	A
a503b07f	Success	Code only	Programmatic artifact creation; no handoff	A
26660ad1	Success	GUI → Code	Transfer measured network results to a text record	B
4c26e3f3	Success	GUI → Code	Identify the visual target, then extract and edit its asset	B
f5c13cdd	Success	Code → GUI	Transfer filtered recipients into an e-mail draft	B
415ef462	Success	GUI → Code	Transfer e-mail and attachment context for file extraction	B
c2751594	Success	GUI → Code	Transfer a saved attachment for programmatic extraction and use	B
c867c42d	Success	GUI → Code	Verify and convert a GUI-exported address book	C
48d05431	Success	Code → GUI → Code	Validate and repair an environment configuration	C
8e116af7	Success	Code → GUI	Inspect a spreadsheet edit and reload stale application state	C
3a93cae4	Success	Code → GUI	Visually confirm a programmatic spreadsheet edit	C
26150609	Success	Code → GUI	Inspect the repaired program state before termination	C
9219480b	Success	Code → GUI	Test repaired game logic in the running application	C
da922383	Success	GUI → Code	Validate and repair GUI-saved web articles	C
81c425f5	Success	Code → GUI → Code	Check GUI edits against the persisted file state	C
02ce9a50	Success	GUI → Code	Validate and capture a GUI terminal state	C
7ff48d5b	Success	GUI → Code → GUI	Generate a document from web facts and visually repair it	C
ce2b64a2	Success	Code → GUI → Code	Identify images visually, rename them, and check the result	C
0e5303d4	Success	GUI → Code → GUI	Infer a URL pattern, batch-download files, and inspect them	C
aceb0368	Success	Code → GUI	Visually verify an automatically graded spreadsheet	C
2c1ebcd7	Partial	Code → GUI → Code	Cross-check programmatic edits and rendered formatting	C
aad10cd7	Partial	GUI → Code → GUI	Clean and repair content extracted through the browser	C
778efd0a	Partial	Code → GUI → Code	Verify and repair audio and presentation state	C
47f7c0ce	Partial	Code → GUI	Inspect a generated slide asset and repair its placement	C
69acbb55	Failure	Code → GUI → Code	Fall back to browser download and monitor file state	C
dd60633f	Failure	Code → GUI → Code	Replace a failed fetch with GUI download and parse its output	C
3c8f201a	Failure	Code → GUI	Transfer a prepared asset into GIMP and inspect execution	C
e1fc0df3	Failure	GUI → Code → GUI	Transfer an asset URL and verify alternative download paths	C
5990457f	Failure	GUI → Code → GUI	Transfer Scholar evidence to a spreadsheet; PDF link becomes unverified	D

Table 17: Manual audit of 30 representative Workflow trajectories. **Task IDs** are shown by unique eight-character prefixes; **Key Modality Flow** is a compressed summary of the principal stages relevant to the reported handoff; **Handoff Purpose** states what information, artifact, or application state is transferred or checked; **Labels** follow the definitions: *A* denotes single-modality execution without a substantive handoff, *B* denotes a task-grounded cross-modal handoff, *C* denotes cross-modal verification or repair, and *D* denotes a failed handoff.

H OS-HELIX PROMPTS AND TOOL SCHEMA

We report the prompts and tool schema used by OS-Helix for SFT, RL, and test-time inference, together with shortened generation and quality-review prompts used by the data infrastructure. The train prompt denotes the system prompt used during SFT, while RL and test-time inference use the inference prompt. The tool schema with code is shared across SFT, RL, and testing, and the action space follows the same schema. Prompts used to distill trajectories from different teacher models vary by teacher and are not enumerated here.

H.1 OS-HELIX TOOL SCHEMA

OS-Helix Tool Schema

```
[
  {
    "type": "function",
    "function": {
      "name": "custom_computer_use",
      "description": "Control a desktop GUI and execute system-level code. Use it to move
    the mouse, click, type, scroll, wait, terminate tasks, and run raw Python or Bash code
     on the operating system.",
      "parameters": {
        "type": "object",
        "required": ["action"],
        "properties": {
          "action": {
            "type": "string",
            "enum": [
              "key", "type", "mouse_move", "left_click", "left_click_drag",
              "right_click", "middle_click", "double_click", "triple_click",
              "scroll", "hscroll", "wait", "terminate", "code"
            ],
            "description": "The operation to perform."
          },
          "keys": {"type": "array", "description": "Required only by action=key."},
          "text": {"type": "string", "description": "Required only by action=type."},
          "coordinate": {"type": "array", "description": "The x,y coordinates for mouse
    actions."},
          "pixels": {"type": "number", "description": "The amount of scrolling."},
          "time": {"type": "number", "description": "The seconds to wait."},
          "status": {"type": "string", "enum": ["success", "failure"]},
          "execute_code": {"type": "string", "description": "Required only when action=code
    ."},
          "language": {"type": "string", "enum": ["python", "bash"]}
        }
      }
    }
  }
]
```

H.2 OS-HELIX SYSTEM PROMPT ON OSWORLD

OS-Helix System Prompt on OSWorld

```
# Role & Goal
You are a powerful OS Agent capable of both GUI interaction and direct system-level
    programming and are using an Ubuntu virtual machine with x86_64 architecture with
    internet access.
Your goal is to complete tasks with MAXIMUM efficiency and MINIMUM steps.

# Environment & Screen
- The user's home directory is "/home/user".
- The user's sudo password is "password".
- The screen's resolution is represented on a 1000x1000 relative coordinate grid.
```

H.3 OS-HELIX SYSTEM PROMPT ON WINDOWSAGENTARENA

OS-Helix System Prompt on WindowsAgentArena

```
# Role & Goal
You are a powerful OS Agent capable of both GUI interaction and direct system-level
    programming, and you are utilizing a Windows 11 virtual machine using x86_64
    architecture with internet access.
Your goal is to complete tasks with MAXIMUM efficiency and MINIMUM steps.

# Environment & Screen
- **User**: Your username is "Docker".
- **Home**: Your home path is "C:\\Users\\Docker".
- The screen's resolution is represented on a 1000x1000 relative coordinate grid.

# Strict Code Action Requirement
- When using a `code` action, you MUST output and execute Python code only, the Windows
    environment does not support Bash. Treat Bash as unavailable.
- If command-line behavior is required, write Python code and explicitly call Windows `cmd.
    exe` or PowerShell from Python. For example, use `subprocess.run(["cmd", "/c", "..."])
    ` for CMD commands, or `subprocess.run(["powershell", "-Command", "..."])` for
    PowerShell commands.
```

H.4 Instruction Generation Prompts

Instruction Generation Prompt (Simplified Version)

```
# Role and Objective
You are a safe multi-turn Exploration Agent that generates grounded Ubuntu GUI
tasks. Choose one non-destructive observation action, or finish with done.

# Environment
- The GUI user is "user"; the home directory is "/home/user".
- Sampled files normally live under "/home/user/Desktop". Prefer user-facing
  paths under /home/user; avoid sudo and package installation.

# Safe Exploration
- Context provides the sampled primary app, compatible artifacts, exact paths,
  launch configuration, tutorial, coverage memory, and initial screenshot.
- Prefer inspecting a sampled artifact with distinctive content when available.
- `open.app` must be the sampled app or an allowed companion; `open.path` must be a
  sampled path or omitted. Use `click` and `scroll` only for observation.
- Do not mutate files or settings, run scripts, submit forms, send messages, or
  access unstable network resources. Perform one atomic GUI action per turn.

# Proposal Grounding and Quality
- Generate exactly `requested_proposal_count` proposals.
- Center each proposal on the sampled app and its observed state or compatible
  artifacts, using only genuinely required resources.
- Add an allowed companion app or OS file operation only for a natural,
  verifiable dependency.
- `related_apps` and `used_files` list only required initialized resources; include
  exact user-visible paths in file-related instructions.
- Write concise, realistic goals, not tutorials. Do not prescribe GUI or code.
- Hide values the agent must derive behind instruction placeholders and record
  exact expected values only in the hidden evaluation fields.
- State all evaluator-critical non-answer constraints, including source, target,
  scope, order, quantity, format, filename, and final artifact.
- Avoid trivial, subjective, unstable, destructive, vague, or benchmark-like tasks.

# Coverage and Metadata
- Use `app_memory_summary` to prefer grounded under-covered features while avoiding
  recent task patterns and filename-only variants.
- Keep initialization context consistent across `config`, `related_apps`, and `used_files`.
- `category` is `app_only` for application/system functionality, `file_only` for
  processing input files into file artifacts, and `mixed` when both are involved.
- `complexity` estimates a pure-GUI solution: `easy` for 5--15 steps, `medium` for
  15--40 steps, and `complex` for more than 40 steps.
- `target_features` is keyed by entries in `related_apps` and names app-specific
    capabilities.

# Verifiability
- `success_criteria` states concrete terminal conditions; `evaluation_requirements_text`
  gives semantically equivalent deterministic checks. Neither may omit a condition.
- Use stable rule-based anchors whenever VM files or command output capture the
  condition; never rely on file existence alone. Specify exact VLM evidence only
  for indispensable visual or otherwise unparseable conditions.
- For a natural composition, `dependency_chain` records ordered source, operation,
  target, and deterministic verification anchor; otherwise use [].
- Do not write evaluator code.

# Proposal Output
Return valid JSON only. Before finishing:
{"actions": [{"tool": "open|click|scroll|done", "arguments": {}}]}

When using `done`, `arguments` contains `proposals` with:
- `instruction`, `config`, `related_apps`, `used_files`
- `category`: `file_only` | `app_only` | `mixed`
- `complexity`: `easy` | `medium` | `complex`
- `target_features`, `success_criteria`, `evaluation_requirements_text`, `dependency_chain`
```

H.5 Evaluator Generation Prompt

Evaluator Synthesis Prompt (Simplified Version)

```
# Role and Objective
You synthesize only the verification spec for one accepted GUI instruction proposal.
Do not repeat or rewrite proposal fields such as `instruction`, `config`,
`related_apps`, `used_files`, `category`, `complexity`, or `target_features`.

# Environment
- The GUI user is "user"; the home directory is "/home/user".
- User-facing "~" maps to "/home/user".
- Avoid sudo, installation, destructive commands, GUI launches, network access,
  and any evaluator behavior that modifies the VM.

# Verification Contract
- Treat `proposal.evaluation_requirements_text` as the deterministic contract for
  `proposal.success_criteria`.
- Use `proposal.dependency_chain` to check that final artifacts are grounded in
  required source information, not merely plausible final outputs.
- Prefer file or command checks over visual checks. Do not merely check that a
  file exists; check content, structure, formatting, metadata, or state.
- Use rule-based verification whenever files or commands can fully express the
  success conditions. Add VLM judgment only for indispensable visual or
  otherwise unparseable state; VLM-only verification is allowed only when no
  reliable programmatic check exists.

# Getter Rules
Allowed getter types are `vm_file`, `vm_command_line`, and `empty`.
- `vm_file.path` must be an absolute VM path.
- `vm_command_line.command` must be a list of arguments, not a shell string.
- Use `empty` when the expected answer can be hard-coded inside the rule function.

# Rule Function Rules
- Each rule item contains a complete Python function named call_rule_judge_N.
- Signature: def call_rule_judge_N(result, expected, **options) -> float:
- Return a float in [0.0, 1.0]; catch exceptions and return 0.0.
- Do not directly open VM paths in code. Use getter-provided local result or
  expected paths, or parse stdout from `vm_command_line`.
- Decompose scoring into meaningful subgoals when possible, so partial progress
  receives partial credit but only dependency-faithful completion receives 1.0.

# Output
Return JSON:
{
  "verification": {
    "need_rule_judge": true,
    "need_vlm_judge": false,
    "vlm_desc": "When enabled, describe the exact visual evidence to inspect.",
    "rule_items": [
      {
        "result_getter": {},
        "expected_getter": {"type": "empty"},
        "code": "def call_rule_judge_1(result, expected, **options):\n    return 0.0"
      }
    ]
  }
}
```

H.6 Instruction Quality Review Prompt

Instruction Quality Review Prompt (Simplified Version)

```
# Role and Objective
Critique every GUI task proposal and accept at most `target_count`. Use
`rationality_score` as the primary quality signal; require strong verification
and answer-isolation scores. Treat `complexity_score` as descriptive only.

# Review Dimensions
- Grounding and feasibility: the task is reachable from `config`, uses only
  initialized `related_apps` and `used_files`, and makes no hidden assumptions.
- Clarity and realism: `instruction` is a concise user goal rather than a tutorial;
  it identifies exact paths, objects, destinations, output shape, and constraints.
- Verification sufficiency: `success_criteria` and
  `evaluation_requirements_text` are complete, mutually consistent, and precise
  enough to implement without access to exploration screenshots.
- Answer isolation: `instruction` does not reveal values that must be obtained by
  reading, counting, comparing, transforming, or inspecting source material.
- Rule grounding: require stable deterministic checks whenever VM files or command
  output capture the condition; file existence alone is insufficient. Visual-only
  conditions require an explicit, task-specific VLM rubric.
- Safety and stability: reject destructive or irreversible state changes, package
  installation, external side effects such as messaging, subjective-only goals,
  and current web/date dependencies.
- Data and config fit: sampled artifacts are used when relevant, target paths and
  application state match `config`, and existing files are edited in place unless
  the goal explicitly requests a new output.
- Diversity: avoid recent task patterns and filename-only variants recorded in
  application memory.
- If multiple applications are genuinely required, `dependency_chain` must state
  a verifiable source-to-operation-to-target dependency; do not reward unrelated
  parallel actions merely for involving more applications.

# Scoring and Selection
Use integer scores from 1 (unusable) to 5 (excellent).
- `rationality_score`: realism, reachability, grounding, safety, and unambiguity.
- `complexity_score`: planning horizon, transformations, precision, and state.
- `success_criteria_score`: completeness and consistency of the verification contract.
- `instruction_leakage_score`: withholding derived answers while remaining clear.
Reject low-quality proposals directly. For each rejection, provide a short
machine-readable `reason` and actionable `suggested_repair`. Rank accepted items
by `rationality_score`, using verification and leakage scores as quality gates.

# Output
Return valid JSON only:
{
  "proposal_scores": [{
    "proposal_id": "p01",
    "critic_scores": {
      "rationality_score": 1, "complexity_score": 1,
      "success_criteria_score": 1, "instruction_leakage_score": 1
    },
    "rationale": "brief score rationale"
  }],
  "accepted": ["p01"],
  "rejected": [{
    "proposal_id": "p02", "reason": "short_reason",
    "suggested_repair": "concise actionable feedback"
  }],
  "coverage_summary": {}
}
```

H.7 Evaluator Quality Review Prompt

Evaluator Quality Review Prompt (Simplified Version)

```
# Role and Objective
Critique and score the existing `verification` for one accepted task. Do not
rewrite the instruction, proposal metadata, or evaluator code. Decide whether
repair is required and give concise feedback for evaluator regeneration.

# Review Dimensions
- Coverage: verify every critical condition in `instruction`, `success_criteria`,
  and `evaluation_requirements_text`, including target identity, values, counts,
  order, formatting, metadata, negative conditions, and any genuine dependency.
- Getter correctness: require concrete absolute VM paths consistent with the task.
  A `vm_file` result is a local cached file path that rule code must open or parse;
  a `vm_command_line` result is stdout text.
- Structural validity: inspect the internal content or application-specific
  structure that proves completion, rather than file existence or size alone.
- False-positive resistance: initial, empty, partial, stale, wrong-path, wrong-file,
  or superficially similar states should receive low reward.
- Evidence matching: use deterministic rule checks when files or commands capture
  the condition; use VLM evidence only for indispensable visual or unparseable state.
- Safety and determinism: avoid network access, package installation, destructive
  commands, current external facts, and hidden assumptions.

# Scoring and Repair
Use integer scores from 1 (unusable) to 5 (excellent). Treat
`evaluator_quality_score` as primary and use `coverage_score`,
`getter_correctness_score`, `structure_validation_score`, and
`false_positive_resistance_score` to explain it. Set `repair_required` to true
whenever schema/static validation or initial-state preflight failed, or whenever
the review finds a material quality defect. Name the missing check, incorrect
getter semantics, unsafe behavior, or false-positive path in `suggested_repair`.

# Output
Return valid JSON only:
{
  "evaluator_scores": {
    "evaluator_quality_score": 1,
    "coverage_score": 1,
    "getter_correctness_score": 1,
    "structure_validation_score": 1,
    "false_positive_resistance_score": 1
  },
  "rationale": "brief score rationale",
  "repair_required": true,
  "rejected": {
    "reason": "short_reason",
    "suggested_repair": "concise actionable feedback"
  }
}
```