HYBRIDCUA: LEARNING TO ORCHESTRATE GUI AND CLI FOR COMPUTER-USE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Computer use agents (CUAs) have demonstrated strong capabilities in completing digital tasks. However, existing CUAs either rely solely on graphical user interface (GUI) interactions, which are often inefficient and error prone, or augment GUI interactions with application specific APIs or tools, which require substantial engineering effort and are difficult to scale across applications. We argue that the next generation of CUAs should combine GUI interactions with the command line interface (CLI), leveraging the generality of the GUI and the efficiency of shell commands. A critical challenge, however, is that current models do not know when or how to use the CLI during task execution. To address this challenge, we develop a data construction pipeline that produces three types of trajectories: GUI only, CLI only, and interleaved GUI and CLI trajectories. This pipeline results in **HybridCUA-8K**, containing 5K hybrid trajectories and 3K verified RLVR tasks. Building on these data, we propose a training framework with two stages: supervised fine tuning on the constructed trajectories, followed by reinforcement learning with our CLI aware rewards that encourages agents to use the CLI selectively and reliably. Experiments show that **HybridCUA-9B** achieves **53.6%** accuracy on OSWorld, improving over the base model by **14.8** percentage points, and improves performance on WindowsAgentArena by **4.0** percentage points. These results demonstrate the effectiveness and cross platform generalizability of the hybrid GUI and CLI paradigm for computer use agents.

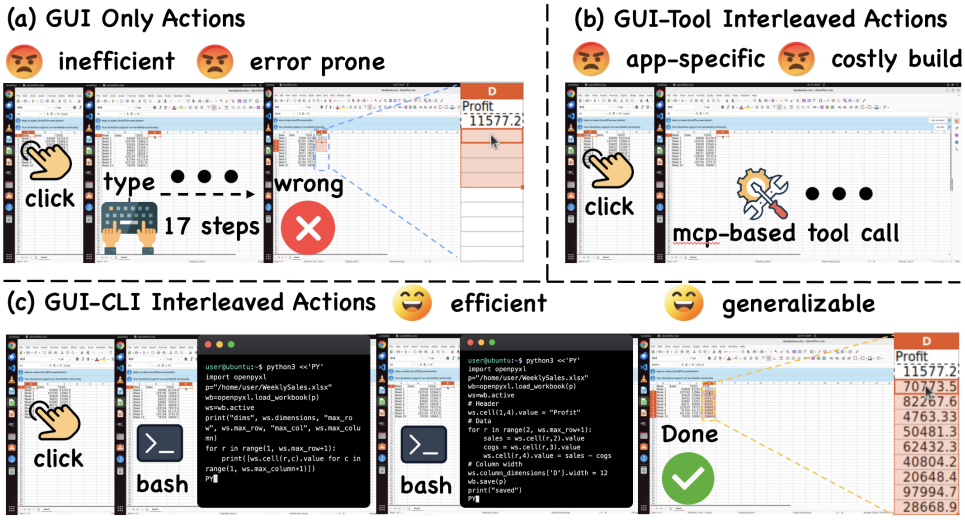


Figure 1: **Comparison of GUI only, GUI–tool, and GUI–CLI interleaved actions.** GUI only agents use low level clicks and keystrokes; GUI–tool agents invoke application specific tools; and GUI–CLI agents combine GUI interaction with general purpose shell commands.

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.

1 INTRODUCTION

Computer-use agents (CUAs) built on multimodal large language models (MLLMs) typically act through clicks and keystrokes on graphical user interfaces (GUIs)—general, but slow and prone to cascading errors over long action sequences (Qin et al., 2025; Wang et al., 2025b; Xue et al., 2026; Lu et al., 2026b). Application-specific APIs and tools speed things up, but only by sacrificing generality, as each must be built per application (Jia et al., 2025; Yang et al., 2025; Hu et al., 2026). Coding agents (OpenClaw Contributors, 2026; Claude Code Team, 2026; OpenAI, 2026) suggest a way out: the command-line interface (CLI), which ships with the operating system and can collapse a long GUI sequence into a single command. We therefore argue that next-generation CUAs should be *hybrid*, using the GUI for visual interaction and the shell for programmable operations (Figure 1). Neither interface suffices on its own, but together they pair the generality of the GUI with the efficiency of code.

Hybrid interaction, however, does not come for free: simply handing a capable MLLM a shell does not help, but in fact hurts. As shown in Figure 2a, once the CLI is exposed, all four representative agents *lose* 2.5 to 11.5 percentage points on OSWorld, indicating that they do not know *how* to turn shell access into effective actions. Some do not even recognize *when* to use it: Qwen3.5-27B (Qwen Team, 2026a) and EvoCUA-32B (Xue et al., 2026) route only 15.0% and 0.15% of their steps through the CLI (Figure 2b), grinding through long GUI sequences that a single command could replace. The bottleneck, in short, is not access to the shell but knowing when and how to use it.

We trace this deficit to two gaps in how current CUAs are trained. **First, the data never shows what hybrid behavior looks like.** GUI, shell, and code corpora are each abundant but siloed: CUA datasets contain almost exclusively GUI actions (Wu et al., 2024), whereas terminal and code datasets omit the GUI context in which commands are issued. **Second, the supervision is blind to interface choice.** Step-level imitation favors locally plausible actions, while outcome rewards merely check task completion (Lai et al., 2025); neither can tell a well-chosen interface from a successful but inefficient path.

We close both gaps with **HybridCUA**, a unified data and training framework for reliable GUI–CLI orchestration. Our key observation is that hybrid supervision need not be collected from scratch, since many GUI action sequences admit equivalent shell commands. Leveraging this, we construct a scalable pipeline spanning three execution schemas: GUI only trajectories converted from open-source datasets, CLI only trajectories sampled from strong models with an execution harness, and interleaved trajectories obtained by both free-form interface switching and GUI-to-CLI rewriting. The resulting 5,000 trajectories ground interface selection and command execution in GUI context. Beyond hybrid data construction, we further make interface choice an explicit training signal.

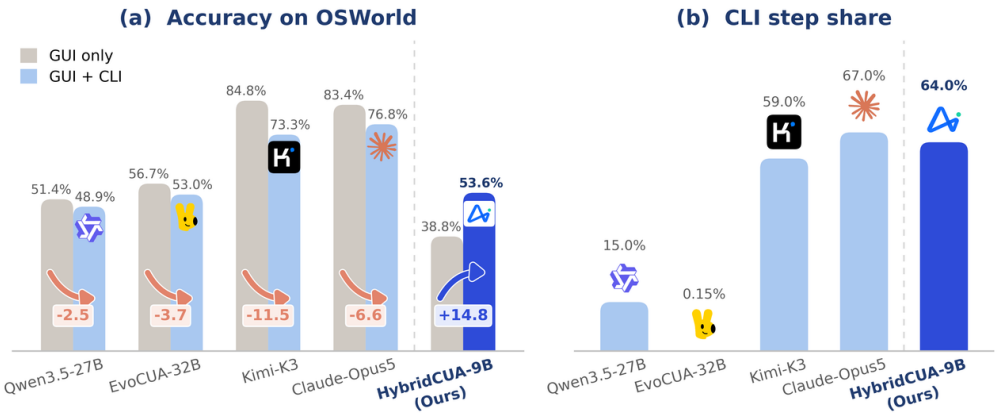


Figure 2: **The GUI–CLI orchestration problem.** **(a)** Adding CLI interface to existing agents reduces OSWorld accuracy, whereas the trained HybridCUA-9B improves over its GUI only counterpart. **(b)** CLI step share varies substantially across existing agents; HybridCUA learns to use the CLI selectively rather than merely maximizing command usage.

Following supervised fine-tuning (SFT), we perform reinforcement learning with verifiable rewards (RLVR) in a live dual-interface environment, with two CLI-aware rewards mirroring the two missing abilities: a task-level reward R_{CLI} that teaches *when* to invoke the CLI, and an action-level reward R_{exec} that teaches *how*, using terminal feedback to improve command reliability.

The resulting **HybridCUA-9B** reaches 53.6% on OSWorld, 14.8 percentage points above its base model and the best among comparably sized models. Notably, it uses the CLI for 64.0% of its steps, on par with the most CLI-heavy baseline, yet *gains* where that baseline loses (Figure 2), confirming that the improvement comes from knowing when and how to use the shell rather than from using it more. The gains carry over to Windows: on WindowsAgentArena, HybridCUA-9B reaches 36.0%, a 4.0-point improvement over the base model.

Our main contributions are summarized as follows:

- We develop a scalable data generation pipeline and construct **HybridCUA-8K**, comprising 5,000 GUI only, CLI only, and interleaved trajectories together with 3,000 verified RLVR tasks, each labeled by whether CLI use offers a clear execution advantage.
- We propose a two-stage training framework that performs SFT on the 5,000 trajectories, followed by RLVR on the 3,000 verified tasks with CLI-aware rewards that separately target routing and execution errors.
- We train **HybridCUA-9B**, which achieves 53.6% on OSWorld, improving over the base model by 14.8 percentage points. We will release the trajectories, RLVR tasks, data generation and training pipelines, and the HybridCUA-9B model.

2 Related Work

2.1 GUI-Based Computer-Use Agents

Recent computer-use agents leverage large scale GUI dataset, and interaction trajectories to jointly improve visual understanding, visual grounding, task planning, and reasoning (Wu et al., 2024; Lu et al., 2026a; Qin et al., 2025; Wang et al., 2025b; Lu et al., 2025). These advances enable a single policy to operate across diverse applications and platforms through screenshots and human-like mouse and keyboard actions. More recently, CUA-Gym and ScaleCUA scale reinforcement learning with verifiable rewards through automatic task generation, scalable environments, and efficient online training (Wang et al., 2026; Lv et al., 2026). However, these agents still operate primarily through atomic actions such as clicking, typing, and scrolling. Tasks involving file manipulation, text editing, or repeated operations may therefore require long GUI sequences even when concise programmatic solutions exist, reducing efficiency and increasing the risk of cascading errors.

2.2 Hybrid Computer Use

Recent work extends computer-use agents beyond GUI only interaction. MCPWorld and OSWorld-MCP benchmark coordination between GUI actions and APIs or MCP tools, while ComputerRL, UltraCUA, ToolCUA, and UI-TARS-2 learn policies over related hybrid action spaces (Yan et al., 2025; Jia et al., 2025; Lai et al., 2025; Yang et al., 2025; Hu et al., 2026; Wang et al., 2025a). Such tools can compress repetitive low level interactions, but their coverage and portability are bounded by predefined, application-specific interfaces.

The shell escapes this constraint, and recent work has begun to explore GUI–CLI workflows through benchmarks, environments, and agent systems (Li et al., 2026; Yang et al., 2026; Zhou et al., 2026). The two closest efforts differ mainly in how they supervise hybrid behavior. CUA-Universe synthesizes hybrid tasks at scale and collects guided trajectories, but scores them with a VLM rather than task-specific programmatic verification, leaving feedback too coarse for online training (Shi et al., 2026). RecreationWorld does offer verifiable application-recreation environments, yet supervises task outcomes alone, so interface selection remains implicit in trajectory imitation (Bai et al., 2026). In neither case is the *choice* of interface part of the learning signal. HybridCUA closes this gap by pairing executable task verifiers with CLI-aware rewards that directly supervise interface routing and command reliability.

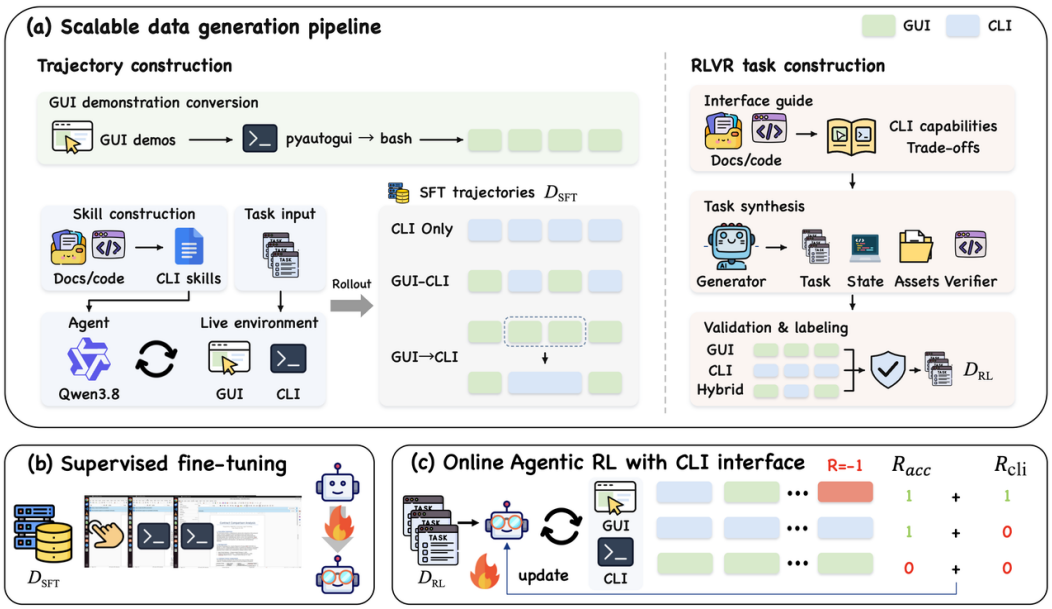


Figure 3: **The HybridCUA data and training pipeline. (a)** Scalable generation of GUI only, CLI only, and interleaved GUI–CLI trajectories, together with annotated RL tasks. **(b)** Supervised fine tuning on $\mathcal{D}_{\text{SFT}}$. **(c)** Online agentic reinforcement learning with CLI aware reward signals.

3 HybridCUA

3.1 Hybrid Computer-Use Formulation

Environment and observation. We formulate computer use as a partially observable Markov decision process $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \Omega, O, \gamma \rangle$. The latent state $s_t \in \mathcal{S}$ contains the complete computer state, whereas the agent observes only

$$o_t = (I_t, \tilde{y}_{t-1}) \sim O(\cdot \mid s_t), \tag{1}$$

where I_t is the current screenshot and $\tilde{y}_{t-1}$ is the stdout/stderr of the preceding CLI action, or empty otherwise. Each action returns one post action screenshot, paired with its CLI output when available.

Unified GUI–CLI action space. Rather than exposing separate GUI and CLI, all executable interactions use the same `bash` action. We define the unified executable action space as

$$\mathcal{A} = \{\text{bash}(c, \delta) \mid c \in \mathcal{C}_{\text{CLI}} \cup \mathcal{C}_{\text{GUI}}\} \cup \{\text{wait}, \text{terminate}, \text{answer}\}, \tag{2}$$

where c is the command and δ is an optional timeout. $\mathcal{C}_{\text{CLI}}$ contains direct shell commands and $\mathcal{C}_{\text{GUI}}$ contains quoted Python heredocs with one or more `pyautogui` code. The complete action schema is provided in Appendix Table 5.

3.2 Scalable Hybrid Data Generation Pipeline

Accordingly, our scalable pipeline in Figure 3(a) constructs SFT trajectories across all three interface modes and synthesizes RLVR tasks labeled by whether CLI use offers a clear execution advantage.

Trajectory construction across interface modes. **GUI only.** We convert open source UI-MOPD trajectories (Lian et al., 2026), re-expressing each GUI action as equivalent `pyautogui` code in our unified action format. **CLI only.** For each application, we distill its documentation and reference code into an application-specific CLI skill listing the available CLI interfaces, common commands, and representative usage patterns, used only for this collection. Equipped with these skills, a Claude Code harness, and the CLI interface alone, Qwen3.8-27B (Qwen Team, 2026b) solves tasks sampled from CUA-Gym (Wang et al., 2026), and we retain its successful rollouts. **Interleaved GUI–CLI.**

Two complementary routes yield trajectories spanning both interfaces. In the first, Qwen3.8-27B accesses both and chooses between them during execution. In the second, we rewrite suitable segments of its GUI only trajectories into equivalent CLI commands, then replay the result and keep only successful replays. Appendix C.1 details all three routes.

RLVR task construction and CLI advantage labeling. Building on the CUA-Gym pipeline, we construct 3,000 verified tasks and label whether CLI use provides a clear execution advantage. For each application, an interface guide built from its documentation and reference code lists the available CLI interfaces, the operations where CLI is more efficient or reliable, and those that require or favor the GUI, letting the task generator locate the capability boundary between the two interfaces and compose hybrid tasks. The pipeline then synthesizes the task, its environment states, assets, and verifier, keeping those with an executable verifier and a reachable target state. To label a retained task, we sample 16 rollouts under each of GUI only, CLI only, and GUI–CLI and rank the three modes by success rate, breaking near-ties by step count. We assign $b^\star = 1$ to tasks where CLI use offers a clear execution advantage, and $b^\star = 0$ otherwise. All tasks are retained. Appendix C.2 reports thresholds and label counts.

3.3 Two Stage Training Paradigm

We train HybridCUA in two stages (Figure 3(b), (c)). Supervised fine-tuning teaches GUI–CLI execution in the unified action format and interface switching, while reinforcement learning with verifiable rewards improves interface selection and CLI execution through online interaction.

Stage I: supervised warm-up. We combine the three trajectory types into the supervised training set $\mathcal{D}_{\mathrm{SFT}}$. Each trajectory is serialized as a task instruction followed by observations and actions in the unified action format. We optimize the next token prediction loss over agent generated responses:

$$\mathcal{L}_{\mathrm{SFT}} = -\mathbb{E}_{\tau\sim\mathcal{D}_{\mathrm{SFT}}}\left[\sum_{t=0}^{T}\log\pi_\theta(a_t \mid x, h_t)\right], \tag{3}$$

where x is the instruction and $h_t = (o_0, a_0, \ldots, o_t)$ is the interaction history. The three trajectory types provide complementary supervision for GUI control, shell command, and interface routing.

Stage II: CLI aware online agentic reinforcement learning. We next optimize the model through online interaction with a live GUI-CLI environment. Standard agentic RL typically uses a final accuracy reward R_{acc}, which encourage task completion but provide limited supervision for interface selection and CLI reliability. Successful rollouts receive the same accuracy reward despite unnecessary interface switches, while failed CLI commands go unpenalized if the task is eventually completed. This weak credit assignment makes it difficult to learn when and how to use the CLI.

To address these issues, we introduce a CLI aware reward consisting of a task level CLI use signal and a step level command execution signal. Each of the 3,000 verified tasks has the binary annotation $b^\star$ defined above. For a rollout τ, let $b(\tau) = \mathbb{I}[\tau \text{ contains at least one direct CLI command}]$. We define the CLI reward as

$$R_{\mathrm{CLI}} = \mathbb{I}[\mathrm{Success}(\tau)]\,\mathbb{I}[b(\tau) = b^\star]\,. \tag{4}$$

By rewarding successful rollouts whose CLI usage matches the task-specific label, R_{CLI} encourages the model to select the CLI when it offers a clear execution advantage and avoid unnecessary CLI use otherwise. It thereby teaches task dependent interface selection. For command execution, terminal feedback provides localized supervision. We define the step-level execution reward as

$$r_t^{\mathrm{exec}} = \begin{cases} -1, & \text{if the command incurs a shell-level execution failure,} \\ 0, & \text{otherwise,} \end{cases} \tag{5}$$

and set $r_t^{\mathrm{exec}} = 0$ at non-CLI steps. The two CLI-aware signals operate at complementary granularities. Because R_{CLI} evaluates whether the rollout selects an appropriate interface strategy, we incorporate it into the trajectory level reward:

$$R(\tau) = R_{\mathrm{acc}} + \lambda_{\mathrm{CLI}} R_{\mathrm{CLI}}. \tag{6}$$

Table 1: Main results on OSWorld. "Action space" denotes the executable interfaces available to each model. Accuracy is reported in percentage. Lower average steps indicate shorter trajectories.

Model	Action space	Acc.↑	Avg. Steps↓
General Models			
Qwen3.5-27B (Qwen Team, 2026a)	GUI	51.4	25.7
Qwen3.5-35B-A3B (Qwen Team, 2026a)	GUI	50.4	26.3
Specialized CUA Models			
OpenCUA-7B (Wang et al., 2025b)	GUI	28.1	19.8
OpenCUA-32B (Wang et al., 2025b)	GUI	34.1	18.7
EvoCUA-8B (Xue et al., 2026)	GUI	46.1	31.1
EvoCUA-32B (Xue et al., 2026)	GUI	56.7	25.0
ToolCUA-8B (Hu et al., 2026)	GUI+API	46.8	14.9
AutoGLM-OS-9B (Lai et al., 2025)	GUI+API	48.9	–
UltraCUA-32B (Yang et al., 2025)	GUI+API	43.7	–
Ours			
Qwen3.5-9B (Qwen Team, 2026a)	GUI	38.8	31.6
+ SFT	GUI	44.2 +5.4	26.3 −5.3
+ SFT + RL	GUI	50.4 +11.6	22.1 −9.5
Qwen3.5-9B (Qwen Team, 2026a)	GUI+CLI	18.4 −20.4	22.1 −9.5
+ SFT	GUI+CLI	46.0 +7.2	19.8 −11.8
HybridCUA-9B	**GUI+CLI**	**53.6** +14.8	**14.0** −17.6

In contrast, r_t^{exec} attribute a shell execution failure to the responsible CLI action. For a GRPO group of G rollouts, let μ_G and σ_G denote the mean and standard deviation of their trajectory rewards. We augment the normalized trajectory advantage only for tokens belonging to action a_t:

$$\widehat{A}_{t,j} = \frac{R(\tau) - \mu_G}{\sigma_G + \epsilon} + \lambda_{\text{exec}} r_t^{\text{exec}}, \qquad j \in \text{Tok}(a_t), \tag{7}$$

where ϵ ensures numerical stability.

4 Experiments

4.1 Experimental Settings

Implementation details. We initialize HybridCUA-9B from Qwen3.5-9B (Qwen Team, 2026a) and adopt a two-stage training pipeline: supervised warm-up for 2 epochs on the three trajectory types, followed by online agentic RL with GRPO using our constructed RLVR tasks, where the CLI aware reward uses $\lambda_{\text{CLI}} = 0.1$ at the trajectory level and $\lambda_{\text{exec}} = 0.3$ at the step level. We use `verl` (Sheng et al., 2025) and Megatron-LM (Shoeybi et al., 2019) for supervised training, and `slime` (THUDM, 2025) with Megatron-LM and SGLang (Zheng et al., 2024) for RL optimization and rollout. Dataset statistics and full training configurations are provided in Appendices C and D.

Baselines and benchmarks. We use OSWorld (Xie et al., 2024) as our primary benchmark and compare HybridCUA-9B against three categories of baselines: (i) GUI only agents, including Qwen3.5 (Qwen Team, 2026a), OpenCUA (Wang et al., 2025b), and EvoCUA (Xue et al., 2026); (ii) GUI–API agents, including ToolCUA (Hu et al., 2026), AutoGLM-OS-9B (Lai et al., 2025), and UltraCUA (Yang et al., 2025); and (iii) Qwen3.5-9B equipped with either the GUI only or GUI–CLI action space. Following the official evaluation protocol, we report task accuracy (**Acc.**) and average steps (**Avg. Steps**), with a maximum 50 steps per task. To evaluate out-of-distribution generalization along two complementary axes, we further test HybridCUA-9B on OSWorld-MCP and WindowsAgentArena (Bonatti et al., 2024). OSWorld-MCP measures transfer to an MCP-enabled environment, whereas WindowsAgentArena measures cross operating-system transfer.

4.2 Main Results

GUI–CLI vs. GUI only. In Table 1, HybridCUA-9B reaches **53.6%** accuracy with **14.0** average steps, the best on both metrics among comparably sized models. Merely exposing the CLI is instead harmful: steps fall by 9.5 but accuracy drops from 38.8% to 18.4%, so shorter trajectories here reflect early failure, not efficiency. Once trained to use it, the second interface pays off at both stages, 46.0% versus 44.2% after SFT and 53.6% versus 50.4% after RL, with steps cut from 26.3 to 19.8 and from 22.1 to 14.0. Both branches share the same base model, comparable SFT corpora, and identical RLVR tasks and RL steps, isolating the added interface.

GUI–CLI vs. GUI–API. HybridCUA-9B exceeds AutoGLM-OS-9B by 4.7 percentage points, ToolCUA-8B by 6.8, and the four times larger UltraCUA-32B by 9.9, with 0.9 fewer steps than ToolCUA-8B. All three rely on application-specific APIs, whereas the shell requires no per-application construction.

4.3 Ablation Study

We ablate the two training stages in turn: first the composition of the supervised corpus, then the contribution of online RL and its CLI-aware reward.

Effect of SFT data composition. Table 2 shows that **mixing all three trajectory types yields higher accuracy than any single type SFT**. The mixed corpus achieves 46.0% accuracy, outperforming GUI only, CLI only, and hybrid only SFT by 2.8, 14.3, and 5.0 percentage points, respectively. It averages 19.8 steps, only 0.7 more than CLI only SFT. Notably, interleaved trajectories alone do not match the mixed corpus, suggesting that single interface demonstrations and hybrid trajectories provide complementary supervision.

Table 2: SFT data ablation on OSWorld.

Configuration	Acc.↑	Avg. Steps↓
Qwen3.5-9B	38.8	31.6
w/ CLI only	31.7	19.1
w/ GUI only	43.2	29.5
w/ Hybrid only	41.0	22.6
HybridCUA-9B-SFT	**46.0**	**19.8**

Effect of RL. Starting from the mixed-data SFT checkpoint, RL increases accuracy from 46.0% to **53.6%** and reduces average steps from 19.8 to **14.0** (Tables 2 and 1). This yields a **7.6** percentage-point accuracy gain and saves **5.8** steps per task on average, a **29.3%** reduction relative to SFT. Thus, RL improves task completion while further shortening trajectories beyond supervised warm-up.

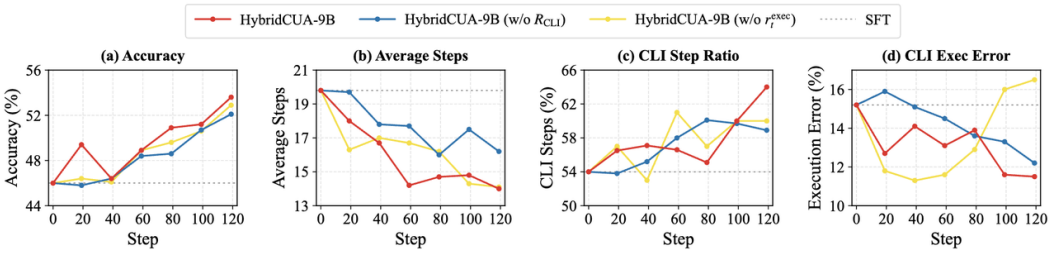


Figure 4: RL ablation curves on OSWorld. The four panels report task accuracy, average environment steps, the percentage of executable steps issued through the CLI, and CLI execution error rate.

To isolate the contribution of each CLI-aware signal, we compare three RL variants in Figure 4, all initialized from the same SFT checkpoint under an identical GRPO budget: the full HybridCUA-9B, a variant without the task-level R_{CLI}, and a variant without the step-level r_t^{exec}, with the SFT checkpoint shown as a reference line. The two signals govern different quantities. Removing R_{CLI} costs little accuracy but stalls the efficiency gain: CLI usage reaches only 58.9% instead of 64.0%, and trajectories shorten by 18.2% rather than 29.3% relative to SFT. Removing r_t^{exec} leaves accuracy and steps nearly intact, yet execution errors climb back to 16.5% by step 120, above the SFT level, while the full model settles at 11.5%. The task-level reward therefore decides *when* to use the CLI, whereas the step-level reward governs *how reliably* commands execute.

4.4 Out-of-Distribution Generalization

Table 3: Out-of-distribution generalization on OSWorld-MCP and WindowsAgentArena.

Model	Action space	OSWorld-MCP	WindowsAgentArena
Qwen3-VL-8B-Instruct (Bai et al., 2025)	GUI	28.2	26.4
Qwen3-VL-235B-A22B (Bai et al., 2025)	GUI	38.1	32.1
Qwen3.5-9B (Qwen Team, 2026a)	GUI	38.0	32.0
ToolCUA-8B (Hu et al., 2026)	GUI + API	46.8	33.8
HybridCUA-9B (ours)	**GUI + CLI**	**47.1**	**36.0**

As shown in Table 3, HybridCUA-9B achieves **47.1%** accuracy on OSWorld-MCP, exceeding Qwen3.5-9B by **9.1** percentage points and performing comparably to ToolCUA-8B. This suggests that learned GUI–CLI orchestration transfers to an MCP-enabled environment while remaining competitive with application-specific GUI–API integration. For cross operating-system transfer, HybridCUA-9B reaches **36.0%** on WindowsAgentArena, outperforming Qwen3.5-9B and ToolCUA-8B by **4.0** and **2.2** percentage points, respectively. Notably, it issues PowerShell commands despite training only on Linux shells, suggesting that what transfers is *when* to delegate to the shell rather than memorized commands. Together, these gains across both environments support our motivation to learn *when* and *how* to combine GUI and CLI as a transferable alternative to application-specific tool integration.

4.5 Analysis

Interface selection. Figure 5 suggests that HybridCUA selects interfaces according to domain specific interaction needs: CLI dominates OS tasks (84%), where shell commands provide direct access to files and system settings, whereas GUI dominates Chrome tasks (74%), where interaction centers on web pages. The breakdown in Figure 6 links these preferences to operation requirements: CLI dominates content editing (85.8%) and result verification (98.4%), which suit direct data manipulation and state inspection, while GUI is favored for spatial adjustment (58.4%), where visual feedback guides positioning and layout.

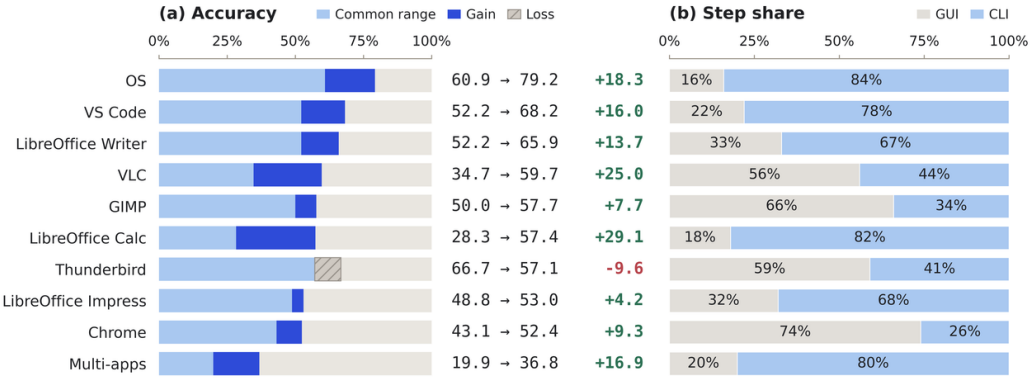


Figure 5: Domain-level accuracy and GUI/CLI step shares on OSWorld.

Complementary GUI–CLI cooperation. Figure 7 illustrates **two levels of GUI–CLI cooperation** in an essay-formatting task. **Across the trajectory**, GUI steps focus the Writer window and open its menu (step 7), while CLI steps use python-docx to apply 12 point text and single, double, and one-and-a-half line spacing to the introduction, body, and conclusion, respectively, before saving the file (step 10). These steps combine application interaction with direct document editing to advance the same task. **Within a hybrid step**, step 13 first dismisses the menu using pyautogui, then reopens the saved file through the CLI to check font size, line spacing, and spacing rules. Both operations execute sequentially in a single bash call, combining GUI state handling with file level verification within one step.

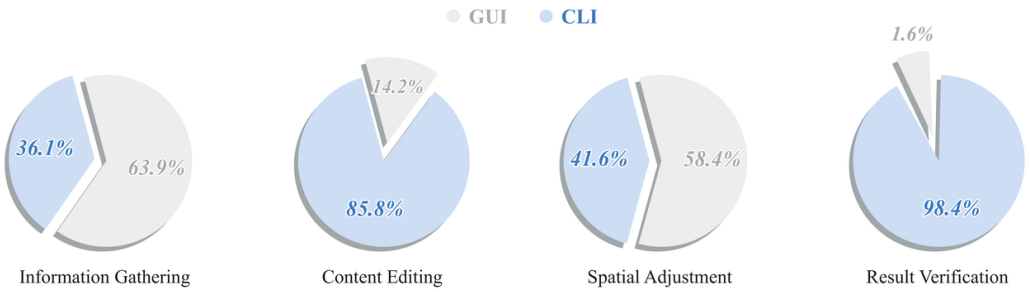


Figure 6: GUI/CLI step shares by operation category for HybridCUA-9B.

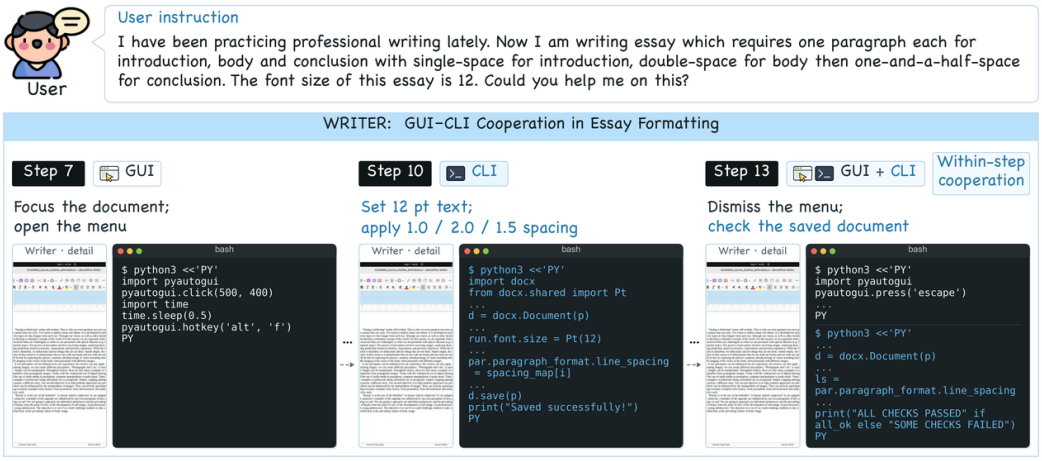


Figure 7: GUI–CLI cooperation in essay formatting. Code is excerpted and screenshots are cropped.

Unified action schema matters. We isolate the effect of the action schema by fine-tuning two variants from the same Qwen3.5-9B base and trajectories under matched settings. The separate-tool schema keeps the two interfaces in distinct tools, `computer_use` for GUI actions and `cli` for shell actions, whereas the unified schema expresses both through a single `bash` action. As shown in Table 4, the unified format achieves **46.0%** accuracy with **19.8** average steps, outperforming the separate-tool schema by **7.2** percentage points at the cost of **3.0** additional steps. It also improves over the GUI only base, indicating that a shared action grammar improves task completion while remaining more efficient than the base.

Table 4: Effect of action schema on OSWorld.

Configuration	Acc.↑	Avg. Steps↓
Qwen3.5-9B	38.8	31.6
Separate GUI/CLI tools	38.8	16.8
Unified `bash` action (ours)	**46.0**	**19.8**

5 Conclusion

We introduced **HybridCUA**, a framework that teaches computer-use agents *when* and *how* to combine GUI and CLI actions. HybridCUA constructs three types of trajectories together with verified RLVR tasks, and trains a unified policy model through supervised fine-tuning and CLI-aware online reinforcement learning. Experiments across multiple computer-use benchmarks show that HybridCUA improves both task completion and interaction efficiency over its base model, while generalizing across interfaces and operating systems. These results demonstrate that the GUI and CLI provide complementary capabilities and that learning to orchestrate them offers a promising direction for accurate, efficient, and generalizable computer-use agents.

AI USE STATEMENT

We used AI assistants in a limited supporting role for language polishing, LATEX formatting, coding, and debugging. Large language models are also part of our methodology, as our data construction pipeline samples CLI only and interleaved trajectories from strong models through an execution harness (Section 3.2). The authors reviewed AI-assisted outputs before incorporating them into the paper. The authors made the final decisions regarding the methodology, experiments, analyses, and presentation, and take full responsibility for the content and results of this work.

ETHICS STATEMENT

This work studies hybrid GUI and CLI orchestration for computer-use agents in benchmark computer environments. The experiments do not involve human participants or the collection of private or personally identifiable information. All trajectory collection, training, and evaluation are carried out in sandboxed virtual machines that are reset between episodes and have no access to real user accounts or credentials. Because shell access amplifies what an agent can do in a single action, behavior and safety in real-world deployments are beyond the scope of this study; such deployments would require further evaluation, safeguards, and human oversight.

REPRODUCIBILITY STATEMENT

Section 3 describes the hybrid formulation, the data generation pipeline, and the two-stage training paradigm with the CLI-aware reward. Appendix B specifies the action space and interaction protocol, and Appendix C details trajectory construction, RLVR task labeling, and dataset statistics. Appendix D reports the supervised and RL hyperparameters and compute setup, while Appendix E documents the benchmark settings, baselines, and metric definitions, and Appendix G provides the system prompts used for evaluation and data construction. We will release the trajectories, RLVR tasks, data generation and training pipelines, and the HybridCUA-9B model.

REFERENCES

Shuai Bai, Yuxuan Cai, Ruizhe Chen, et al. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025.

Shuai Bai, Jiayong Deng, Sicheng Fan, et al. RecreationWorld: Scalable and verifiable environments for hybrid computer-use agents. *arXiv preprint arXiv:2609.22000*, 2026.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, et al. Windows agent arena: Evaluating multi-modal os agents at scale. *arXiv preprint arXiv:2409.08264*, 2024.

Claude Code Team. Claude Code. GitHub repository, 2026. URL `https://github.com/anthropics/claude-code`.

Xuhao Hu, Xi Zhang, Haiyang Xu, et al. ToolCUA: Towards optimal gui-tool path orchestration for computer use agents. *arXiv preprint arXiv:2605.12481*, 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.

Hanyu Lai, Xiao Liu, Yanxiao Zhao, et al. ComputerRL: Scaling end-to-end online reinforcement learning for computer use agents. *arXiv preprint arXiv:2508.14040*, 2025.

Wanli Li, Bowen Zhou, Yunyao Yu, Zhou Xu, Yifan Yang, Dongsheng Li, and Caihua Shan. WeaveBench: A long-horizon, real-world benchmark for computer-use agents with hybrid interfaces. *arXiv preprint arXiv:2606.09426*, 2026.

Niu Lian, Tongbo Chen, Zhehao Yu, Chengzhen Duan, Fazhan Liu, Hui Liu, Pei Fu, Jian Luan, Heng Qu, Shu-Tao Xia, and Jinpeng Wang. UI-MOPD: Multi-platform on-policy distillation for unified gui agents. *arXiv preprint arXiv:2607.04425*, 2026.

Zhengxi Lu, Jiabo Ye, Fei Tang, Yongliang Shen, Haiyang Xu, Ziwei Zheng, Weiming Lu, Ming Yan, Fei Huang, Jun Xiao, et al. Ui-s1: Advancing gui automation via semi-online reinforcement learning. *arXiv preprint arXiv:2509.11543*, 2025.

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, 2026a.

Zhengxi Lu, Fei Tang, Guangyi Liu, Jin Ma, Kaitao Song, Xu Tan, Wenqi Zhang, Weiming Lu, Jun Xiao, Yueting Zhuang, et al. Ui-copilot: Advancing long-horizon gui automation via tool-integrated policy optimization. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19741–19762, 2026b.

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

OpenAI. Codex for (almost) everything. OpenAI, April 2026. URL https://openai.com/index/codex-for-almost-everything/. Published April 16, 2026.

OpenClaw Contributors. Peekaboo: Mac automation that sees the screen and does the clicks. GitHub repository, 2026. URL https://github.com/openclaw/Peekaboo.

Yujia Qin, Yining Ye, Junjie Fang, 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. Qwen Blog, 2026a. URL https://qwen.ai/blog?id=qwen3.5. Published February 15, 2026.

Qwen Team. Qwen3.8. Qwen Blog, 2026b. URL https://qwen.ai/blog?id=qwen3.8. Qwen3.8-27B open-weight release, August 14, 2026.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*, 2025.

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.

THUDM. slime: An LLM post-training framework for RL scaling. https://github.com/THUDM/slime, 2025.

Bowen Wang, Dunjie Lu, Junli Wang, Tianyi Bai, Shixuan Liu, Zhipeng Zhang, Haiquan Wang, Hao Hu, Tianbao Xie, Shuai Bai, Dayiheng Liu, Que Shen, Junyang Lin, and Tao Yu. CUA-Gym: Scaling verifiable training environments and tasks for computer-use agents. *arXiv preprint arXiv:2605.25624*, 2026.

Haoming Wang, Haoyang Zou, Huatong Song, et al. UI-TARS-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *arXiv preprint arXiv:2509.02544*, 2025a.

Xinyuan Wang, Bowen Wang, Dunjie Lu, et al. OpenCUA: Open foundations for computer-use agents. *arXiv preprint arXiv:2508.09123*, 2025b.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, et al. OS-ATLAS: A foundation action model for generalist gui agents. *arXiv preprint arXiv:2410.23218*, 2024.

Tianbao Xie, Danyang Zhang, Jixuan Chen, et al. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *arXiv preprint arXiv:2404.07972*, 2024.

Taofeng Xue, Chong Peng, Mianqiu Huang, Linsen Guo, Tiancheng Han, Haozhe Wang, Jianing Wang, Xiaocheng Zhang, Xin Yang, Dengchang Zhao, Jinrui Ding, Xiandi Ma, Yuchen Xie, Peng Pei, Xunliang Cai, and Xipeng Qiu. EvoCUA: Evolving computer use agents via learning from scalable synthetic experience. *arXiv preprint arXiv:2601.15876*, 2026.

Yunhe Yan, Shihe Wang, Jiajun Du, et al. MCPWorld: A unified benchmarking testbed for api, gui, and hybrid computer use agents. *arXiv preprint arXiv:2506.07672*, 2025.

Yuhao Yang, Zhen Yang, Zi-Yi Dou, et al. UltraCUA: A foundation model for computer use agents with hybrid action. *arXiv preprint arXiv:2510.17790*, 2025.

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

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Hanzhang Zhou, Panrong Tong, Xu Zhang, Quyu Kong, Chenglin Cai, Tianyu Xia, Gongjie Zhang, Jianan Zhang, Long Li, Long Chen, Lei Wang, Gaole Dai, Pengxiang Li, Liangyu Chen, Yue Wang, and Steven Hoi. Qwen-UI-Agent technical report: Toward next-generation real-world centric foundation GUI agents. *arXiv preprint arXiv:2607.28227*, 2026.

A LIMITATIONS

HybridCUA has several limitations. First, its benefits depend on the availability and stability of command-line interfaces. Applications without usable CLIs, restricted-shell environments, or operating-system-specific command semantics may require substantially different interface-routing behavior. Although we evaluate transfer on OSWorld-MCP and WindowsAgentArena, these benchmarks do not cover the full diversity of applications, operating systems, permission settings, and long-horizon workflows. Second, our data construction currently focuses primarily on the applications included in OSWorld. Although these applications cover common desktop workflows, they do not fully represent the diversity of domain-specific applications or out-of-distribution environments encountered in real-world use. Future work should extend the data construction pipeline to a broader range of specialized applications and synthesize more diverse OOD environments for HybridCUA training, thereby improving its generalization to unseen applications, workflows, and interface configurations.

B ACTION SPACE AND INTERACTION PROTOCOL

Table 5 specifies the executable and control actions. The shared `bash` interface is an action representation: using this wrapper does not make a PyAutoGUI interaction a CLI operation. GUI actions manipulate the visible interface through PyAutoGUI, whereas direct CLI commands operate through the shell and the programs it invokes.

Table 5: Action space in HybridCUA.

Action	Implementation	Definition
Executable Action		
`bash`	`bash(command, timeout)`	Executes either a GUI or CLI command through a single shell interface.

	`python3 <<'PY'`
	`import pyautogui`
	`...`	

GUI	`PY`	Executes one or more `pyautogui` calls in a quoted Python heredoc.
	`pyautogui.click(x, y)`	Left-clicks at (x, y).
	`pyautogui.doubleClick(x, y)`	Double-clicks at (x, y).
	`pyautogui.tripleClick(x, y)`	Triple-clicks at (x, y).
	`pyautogui.rightClick(x, y)`	Right-clicks at (x, y).
	`pyautogui.middleClick(x, y)`	Middle-clicks at (x, y).
	`pyautogui.moveTo(x, y)`	Moves the cursor to (x, y).
	`pyautogui.dragTo(x, y, duration=0.5)`	Drags from the current position to (x, y).
	`pyautogui.mouseDown()`	Presses and holds the mouse button.
	`pyautogui.mouseUp()`	Releases the mouse button.
	`pyautogui.scroll(-5)`	Scrolls downward; a positive value scrolls upward.
	`pyautogui.press('enter')`	Presses a single key.
	`pyautogui.hotkey('ctrl', 's')`	Presses a keyboard shortcut.
	`pyautogui.typewrite('text', interval=0.02)`	Types the specified text.
	`pyautogui.keyDown('shift')`	Presses and holds a modifier key.
	`pyautogui.keyUp('shift')`	Releases a modifier key.
CLI	Direct shell command	Executes a command in the shell; its stdout/stderr is paired with the subsequent screenshot.
Interaction and Control Actions		
`wait`	`wait(time)`	Waits for the interface to stabilize.
`terminate`	`terminate(status)`	Ends the task with a status of success or failure.
`answer`	`answer(text)`	Submits a textual answer and finishes the task.

At decision step t, the agent receives the current screenshot I_t and, when the preceding action is a direct CLI command, its stdout/stderr $\tilde{y}_{t-1}$. The task instruction and preceding interaction history provide the context for generating the next action. After execution, the environment returns a post-action screenshot together with CLI output when available. Thus, CLI only trajectories restrict the *action interface*, not the observation modality: they can still contain visual observations.

The executable action `bash(command, timeout)` accepts either a direct shell command or a quoted Python heredoc containing PyAutoGUI operations. The control actions `wait`, `terminate`, and `answer` respectively allow the interface to settle, end an episode with a status, or submit a

final textual response. A model-generated completion declaration is distinct from the task verifier's assessment of the resulting state.

C Dataset Construction and Statistics

HybridCUA-8K comprises two different data units: a corpus of 5,023 supervised trajectories and a pool of 3,000 verified RL tasks. A trajectory records one execution, while an RL task specifies an environment and a verifiable objective from which multiple rollouts can be sampled.

C.1 Supervised Fine-Tuning Data Construction

The SFT corpus contains 5,023 trajectories spanning 11 application domains and three complementary interaction modes: GUI only, CLI only, and hybrid GUI–CLI execution. LibreOffice Impress, Writer, and Calc contribute 866, 818, and 811 trajectories, respectively, while the multi-application split contributes 786. Trajectories contain 10.4 logical steps on average, with a median of 7, a 90th percentile of 22, and a maximum of 50. Figure 8 summarizes the domain and modality composition together with the trajectory-length distribution. We construct each trajectory type as follows.

GUI only trajectories. The GUI only split is derived from UI-MOPD trajectories (Lian et al., 2026). Figure 9 shows selected steps from a representative trajectory that changes a presentation slide background from the default color to red. To preserve the original GUI supervision while matching the unified action space, each source operation, including clicking, dragging, scrolling, keyboard shortcuts, and text entry, is translated into an equivalent PyAutoGUI call. For coordinate-based operations, the model performs visual grounding in a 1000×1000 coordinate space shared by its visual input and predicted GUI locations; the predicted coordinates are then mapped to the environment display coordinates at execution time. The converted calls are serialized as quoted Python heredocs inside the same `bash(command, timeout)` wrapper used by the other trajectory types. For example, a source click action becomes a `pyautogui.click(x, y)` call inside this wrapper. The wrapper standardizes action serialization and execution but does not change the interface category: PyAutoGUI-based interface manipulation remains a GUI action, whereas only a command that directly invokes a shell program or an application CLI is classified as a CLI action.

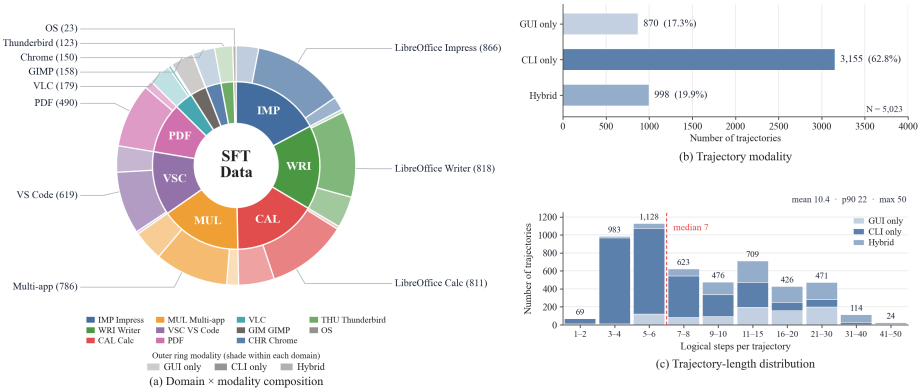


Figure 8: **Composition of the supervised fine-tuning corpus. (a)** Application-domain totals, with the outer-ring shade indicating GUI only, CLI only, or hybrid interaction within each domain. **(b)** Overall trajectory counts by modality. **(c)** Distribution of logical steps per trajectory, stacked by modality.

```
<tool_call>
<function=computer_use>
<parameter=action>
left_click
</parameter>
<parameter=coordinate>
[906, 287]
</parameter>
</function>
</tool_call>
```

⇒

```
<tool_call>
<function=computer_use>
<parameter=action>
bash
</parameter>
<parameter=command>
python3 <<'PY'
import pyautogui
pyautogui.click(906, 287)
PY
</parameter>
</function>
</tool_call>
```

```
<tool_call>
<function=computer_use>
<parameter=action>
left_click_drag
</parameter>
<parameter=coordinate>
[593, 401]
</parameter>
<parameter=start_coordinate>
[456, 401]
</parameter>
</function>
</tool_call>
```

⇒

```
<tool_call>
<function=computer_use>
<parameter=action>
bash
</parameter>
<parameter=command>
python3 <<'PY'
import pyautogui
pyautogui.moveTo(456, 401)
pyautogui.dragTo(593, 401, duration=0.5)
PY
</parameter>
</function>
</tool_call>
```

```
<tool_call>
<function=computer_use>
<parameter=action>
left_click
</parameter>
<parameter=coordinate>
[202, 223]
</parameter>
</function>
</tool_call>
<tool_call>
<function=computer_use>
<parameter=action>
key_down
</parameter>
<parameter=keys>
["shift"]
</parameter>
</function>
</tool_call>
<tool_call>
<function=computer_use>
<parameter=action>
left_click
</parameter>
<parameter=coordinate>
[202, 464]
</parameter>
</function>
</tool_call>
<tool_call>
<function=computer_use>
<parameter=action>
key_up
</parameter>
<parameter=keys>
["shift"]
</parameter>
</function>
</tool_call>
```

⇒

```
<tool_call>
<function=computer_use>
<parameter=action>
bash
</parameter>
<parameter=command>
python3 <<'PY'
import pyautogui
pyautogui.click(202, 223)
pyautogui.keyDown('shift')
pyautogui.click(202, 464)
pyautogui.keyUp('shift')
PY
</parameter>
</function>
</tool_call>
```

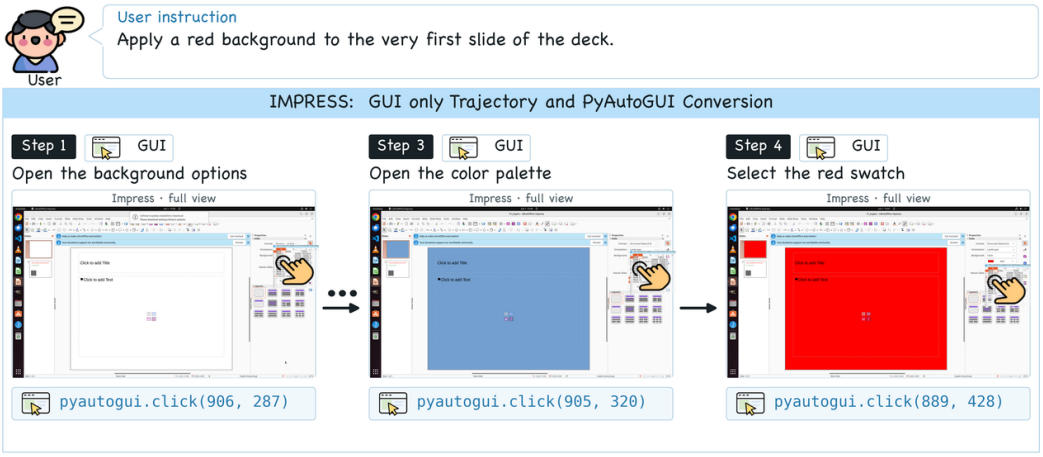


Figure 9: A representative GUI only trajectory paired with its direct PyAutoGUI representation.

CLI only trajectories. We construct application-specific CLI skills from online application documentation and reference implementations, and use Qwen3.8-27B (Qwen Team, 2026b) equipped with these skills and a Claude Code harness to sample trajectories in CUA-Gym (Wang et al., 2026). Figure 10 shows selected steps from an Impress example. Using Python/UNO, the agent identifies slides 1 and 5 from their speaker-photo placeholders, changes their backgrounds from white to pale yellow (`#FFFFCC`), saves the presentation, and cross-checks the live UNO state against the saved XML while confirming that the other four slides remain unchanged.

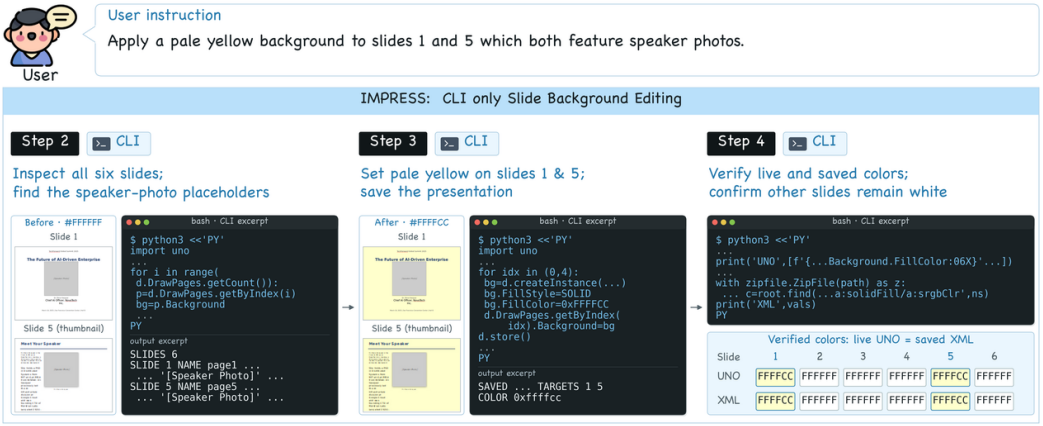


Figure 10: **A representative CLI only trajectory.**

Interleaved GUI–CLI trajectories. Two construction routes produce trajectories containing both interfaces. In the first, Qwen3.8-27B has access to GUI and CLI actions and chooses between them during execution. In the second, we apply a semantics-preserving action-level transformation to GUI only trajectories generated by Qwen3.8-27B. We first identify typing actions that enter commands in a terminal, rather than ordinary text in an application. Because one command may be split across many actions, we reconstruct the complete command before replacement. For example, typing `mkdir -p /tmp/proj` and then pressing Enter becomes one direct CLI action; likewise, a heredoc entered line by line is folded into one multiline command. We rewrite both the executable action and its tool call, remove terminal-opening shortcuts, submission keystrokes, and folded content lines, and preserve all genuine GUI interactions and non-command text input. Thus, the transformation may fold multiple GUI actions into one CLI action, but never splits one action into several. The converted trajectories are replayed, and only successful replays are retained,

since a shorter command sequence need not preserve the application state expected by subsequent GUI actions. Figure 11 illustrates this interaction. The CLI first installs the Night Owl extension, making it available to VS Code; GUI actions then open the theme picker, select and preview the exact theme, and confirm the choice. Finally, the trajectory returns to the CLI and reads back the persisted `workbench.colorTheme` setting, verifying that the GUI selection was saved. Thus, direct commands efficiently handle setup and state inspection, while the GUI is retained for the visually grounded, stateful selection step.

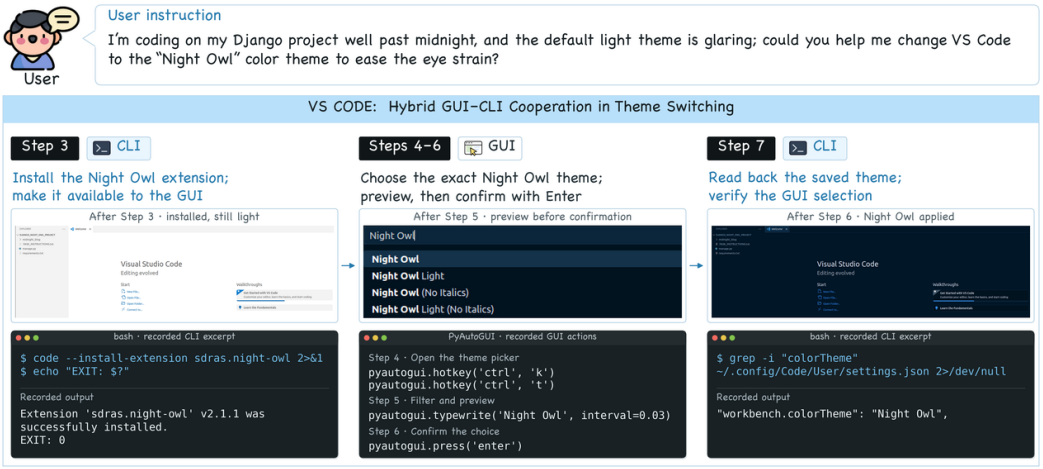


Figure 11: **A representative interleaved GUI–CLI trajectory.** CLI actions install the requested extension and verify the persisted setting, while GUI actions select, preview, and confirm the exact theme.

C.2 RLVR Task Construction and CLI Advantage Labeling

RLVR task construction. Building on CUA-Gym (Wang et al., 2026), we use application-specific interface guides to synthesize task instructions, initial environment states, required assets, and executable verifiers. We retain tasks with executable verifiers and reachable target states, yielding 3,000 verified tasks across 11 domains (Table 6).

Table 6: Domain distribution of the full 3,000-task RLVR pool, before sampling the 1,000 tasks used for online RL.

Domain	Share
`libreoffice_calc`	26.8%
`libreoffice_writer`	14.4%
`libreoffice_impress`	13.3%
`multi_apps`	12.9%
`os`	8.9%
`chrome`	5.4%
`vs_code`	4.4%
`pdf`	4.3%
`vlc`	3.6%
`gimp`	3.1%
`thunderbird`	2.9%
Total	100.0%

Labeling procedure. For each of the 3,000 verified tasks, we sample 16 rollouts with Qwen3.8-27B under each of GUI only, CLI only, and GUI–CLI action spaces. Success rates use all 16 rollouts per mode, without filtering by CLI step share. We rank modes by success rate, breaking ties within 5 percentage points by the median step count of successful rollouts; residual ties favor

a single-interface mode, then GUI only. We set $b^\star = 1$ if the top-ranked mode is CLI only, or if it is GUI–CLI and more than half of its successful rollouts contain at least one direct CLI command; otherwise, $b^\star = 0$. No task is removed during labeling.

D Training Details

D.1 Supervised Fine-Tuning Configuration

The training corpus consists of 5,023 successful computer-use trajectories collected by driving a graphical desktop, each paired with a natural-language task instruction. Expanding each trajectory at every decision step yields 52,227 step-level samples. Each sample is a demonstration prefix for a target action a_t, with context comprising the task instruction and preceding observations and actions. Filtering these samples to at most 12,000 tokens yields 46,876 step-level samples. Each prefix retains at most the three most recent screenshots at different resolutions. The current observation, on which the target action operates, is kept at higher resolution with a pixel budget of at most 2,088,960 pixels, costing 2,040 visual tokens. The two preceding screenshots are downsampled by a factor of two along each dimension, with a configured pixel budget of 548,800 pixels and a cost of 510 visual tokens each. Visual input therefore contributes at most $2{,}040 + 2 \times 510 = 3{,}060$ tokens per sample. The higher-resolution current frame preserves the spatial detail needed for coordinate-level actions, while downsampled history supplies semantic context. Older screenshots are replaced with the placeholder text `This screenshot has been collapsed.`, and the action history is limited to the 30 most recent steps. Across the corpus, 78.4% of samples contain three live screenshots, 10.8% contain two, and 10.7% contain one, corresponding to an average of 2,895 visual tokens per sample and 45.1% of all processed tokens. With three screenshots, the 12,000-token context budget leaves at most 8,940 tokens for the text prompt and target response; the longest text prompt observed across the corpus contains 8,945 tokens. Only the final assistant turn of each prefix contributes to the supervised loss; all preceding assistant turns remain in the context but are masked. The supervised target preserves the model's reasoning block.

The model is initialized from Qwen3.5-9B and trained with `verl` and Megatron-LM for 2 epochs at a global batch size of 256 step-level samples, yielding $\lfloor 46{,}876/256 \rfloor \times 2 = 366$ optimizer updates. Sequences are padded to the right, batched dynamically with a 12,000-token budget per GPU, and truncated from the left if they exceed the context limit. Training uses 2 nodes with 8 GPUs each; TP, PP, and DP denote tensor, pipeline, and data parallelism, respectively. Validation is not used for model selection, and the checkpoint at the final update is reported. Table 7 lists the full training configuration.

D.2 Online Reinforcement Learning Configuration

Tasks and training setup. We construct tasks with custom definitions, initial states, and success criteria on an OSWorld-compatible desktop framework, rather than reuse public GUI benchmark tasks. Starting from the SFT checkpoint, we train with GRPO on 1,000 tasks sampled from 3,000 verified tasks, using `slime` with Megatron-LM for optimization and SGLang for rollout. Table 8 lists the configuration; its sampling settings apply only to training rollouts, not benchmark evaluation.

Reward design and group filtering. The task evaluator provides a binary success reward R_{acc}, augmented by a success-gated modality-preference reward:

$$R_{\mathrm{CLI}}(\tau) = \mathbb{I}[\mathrm{Success}(\tau)]\,\mathbb{I}[b(\tau) = b^\star]\,. \tag{8}$$

Here $b(\tau) \in \{0, 1\}$ indicates whether τ issues at least one direct CLI command and $b^\star \in \{0, 1\}$ is the task's calibrated preference from Appendix C.2. Every task in the pool carries a label, so this term is defined for all training tasks. We normalize the combined trajectory reward $R(\tau) = R_{\mathrm{acc}} + \lambda_{\mathrm{CLI}} R_{\mathrm{CLI}}(\tau)$ within each GRPO group, with $\lambda_{\mathrm{CLI}} = 0.1$. The execution term r_t^{exec} in Eq. (5) is added *after* normalization with $\lambda_{\mathrm{exec}} = 0.3$, as in Eq. (7). Before normalization, we discard groups with $\max_{\tau \in \mathcal{G}} R(\tau) - \min_{\tau \in \mathcal{G}} R(\tau) \le 10^{-12}$, retaining at least one group to avoid empty batches. This criterion uses the combined trajectory reward, not the binary outcome alone, and excludes the later step-level execution term. Filtering removes groups without a group-relative learning signal and makes the effective batch size variable.

Table 7: Supervised fine-tuning hyperparameters.

Category	Hyperparameter	Value
Model and Training Data		
Base model	Initialization	Qwen3.5-9B
Training data	Samples / trajectories	46,876 / 5,023
Context and Visual Input		
Context length	Tokens / truncation	12,000 / left
Screenshots	Current / history frames	1 / 2
Visual tokens	Current / history frame	2,040 / 510
Action history	Steps	30
Batch Size and Training Duration		
Global batch size	Step-level samples	256
Training duration	Epochs / updates	2 / 366
Infrastructure and Parallelism		
Cluster	Nodes / GPUs per node	2 / 8
Total GPUs	–	16
Parallelism	TP / PP / DP	2 / 1 / 8
Hardware	GPU	NVIDIA H20
Optimization and Systems		
Learning rate	Initial / schedule / warm-up	1×10^{-5} / cosine / 10%
Optimizer	Type / β_1 / β_2	AdamW / 0.9 / 0.999
Regularization	Weight decay / gradient clip	0.01 / 1.0
Precision	–	bfloat16
Attention backend	–	FlashAttention
Activation recomputation	Mode / layers	Full, uniform / 1
Optimizer state	Distribution / offload	Distributed / CPU
Dynamic batching	Tokens per GPU	12,000
Evaluation and Reproducibility		
Validation / checkpoint	–	None / final update
Random seed	–	1

Context construction and step-level credit assignment. Dynamic history expands each trajectory into step-level samples, each concatenating the context visible at that step with the policy-generated response. At most three screenshots are retained, including the current frame; historical frames are downsampled and older images are replaced by text placeholders. Only the latest three steps are rendered in full, with earlier steps reduced to one-line action summaries. Pixel budgets, output truncation, and step limits are given in Table 8. Only the current response contributes to the loss; all context tokens are masked, and samples without trainable tokens are skipped. Group statistics and normalized advantages are computed after deduplicating step samples by trajectory, then each trajectory's group-relative advantage is broadcast back to its steps before adding the local execution term. This prevents long trajectories from being counted repeatedly in group normalization.

Trajectory validity and failure handling. Aborted trajectories receive zero reward and are excluded from gradient computation. Environment lifecycle failures are recorded with their stage and error description in sample metadata. In-VM execution failures returned by the executor rather than raised as exceptions do not by themselves trigger sample removal; affected steps remain eligible for training, with applicable CLI execution penalties reflected in r_t^{exec}.

Asynchronous rollout and policy staleness. Optimization and rollout are decoupled to reduce idle time during environment resets. Each step records its generating policy version, and a group's birth version v_{birth} is the minimum over all its steps. We discard groups when $v_{\text{cur}} - v_{\text{birth}} > \delta$, where v_{cur} is the current policy version and $\delta = 2$ optimizer updates.

Table 8: Online reinforcement learning hyperparameters.

Category	Hyperparameter	Value
Tasks and Infrastructure		
Training tasks	Sampled / verified	1,000 / 3,000
Training stack	Framework / optimizer / rollout	`slime` / Megatron-LM / SGLang
Cluster	Nodes / GPUs per node	3 / 8
Total GPUs	–	24
Hardware	GPU	NVIDIA H20
Parallelism and Batch Size		
Training	GPUs / TP / DP	16 / 4 / 4
Inference	GPUs / engines	8 / 8
Rollouts per prompt	–	8
Nominal batch	Groups / trajectories	8 / 64
Rewards and GRPO		
Outcome reward	R_{acc}	$\{0, 1\}$
CLI preference	λ_{CLI} / level	0.1 / trajectory
Execution term	λ_{exec} / level	0.3 / step
Clip ratio	Lower / upper	0.2 / 0.2
KL	Penalty / loss / estimator	0.001 / 0.01 / k3
Reference / entropy	–	SFT checkpoint / 0
Context and Sequence Length		
Screenshots	Maximum frames	3
Full-text history	Steps	3
Image pixels	Current / historical	2,088,960 / 548,800
CLI output limit	Characters per step	1,000
Response limit	Tokens per turn	4,096
Environment limit	Training / evaluation steps	30 / 50
Optimization		
Learning rate	Schedule / warm-up	1×10^{-6} / constant / 0
Optimizer	β_1 / β_2 / weight decay	Adam / 0.9 / 0.95 / 0.1
Gradient	Clipping / microbatch	1.0 / 1
Checkpoint interval	Updates	10
Rollout Engine		
Temperature / top-p	–	1.0 / 1.0
Thinking / KV cache	–	Disabled / 0.7
Concurrency	Requests per engine	64
Chunked prefill	Tokens	4,096
Reset / post-action wait	Seconds	60 / 0.5
Maximum policy lag	δ (updates)	2

E Evaluation Protocols and Metrics

Evaluated task sets. OSWorld (Xie et al., 2024) is the primary benchmark and we evaluate its full set of $N = 361$ tasks. For out-of-distribution transfer we evaluate the 361 tasks of OSWorld-MCP (Jia et al., 2025), which shares the OSWorld task set but exposes an MCP-enabled environment, and the 154 tasks of WindowsAgentArena (Bonatti et al., 2024), which changes the operating system. Because the two transfer settings vary different aspects of the environment, and because OSWorld-MCP is built on the same task set as OSWorld rather than on unseen tasks, they are reported separately and are not combined into a single aggregate score. Every model is given a budget of at most 50 environment steps per task, and each reported number comes from a *single* evaluation run per model and benchmark.

Reported metrics. Let $v_i \in [0, 1]$ denote the evaluator score of task i. The reported **Acc.** is the mean evaluator score over the evaluated task set,

$$\text{Acc.} = \frac{100}{N} \sum_{i=1}^{N} v_i. \tag{9}$$

Most OSWorld verifiers are binary, in which case $v_i \in \{0, 1\}$ and Eq. (9) reduces to a strict success rate; a subset of verifiers, however, returns a fractional score, and we average these scores as returned rather than thresholding them into binary outcomes. The reported values are therefore mean evaluator scores, which are an upper bound on the strict all-or-nothing success rate and should be compared only against numbers aggregated the same way. A model-generated `terminate` declaration never contributes to v_i. Let T_i further denote the number of environment interaction steps consumed by task i. The reported **Avg. Steps** is the all-task mean $\overline{T}_{\text{all}} = N^{-1} \sum_{i=1}^{N} T_i$, computed over the same task set as Eq. (9) rather than over successful tasks only. One step is one environment interaction, so `wait` and the terminating action are counted, and a task that exhausts the budget contributes $T_i = 50$.

Interface access. Interface access is part of the evaluation configuration, not solely a property of the model name: a GUI only agent, a GUI–API agent, and a GUI–CLI agent may have different executable capabilities on the same task. In particular, evaluating in an MCP-enabled environment does not by itself establish that every model receives identical MCP tools, and cross-operating-system evaluation requires an explicit description of the Windows command-execution interface rather than an assumption that a Linux shell configuration transfers unchanged.

Let N_{CLI} and N_{GUI} count executable steps assigned to the two interfaces in a specified rollout set. The pooled CLI step share is $100N_{\text{CLI}}/(N_{\text{CLI}} + N_{\text{GUI}})$; control actions are not executable GUI or CLI steps. The CLI execution error rate is $100N_{\text{CLI,err}}/N_{\text{CLI}}$, using the same failure predicate as r_t^{exec}. A rate with a zero denominator is undefined, not zero. Pooled step shares weight longer trajectories more heavily and are distinct from an unweighted mean of per-task shares.

F More Case Studies

We present three complete recorded trajectories, with 8, 9, and 8 steps, respectively. All three have a saved evaluator score of 1.0. Each row pairs **the pre-action screenshot on the left** with **the model action on the right**; the next row shows the resulting observation. Blue, teal, and lavender distinguish GUI, CLI, and control actions. The right column reproduces each `bash` command in full, or the parameters of a control action, with only typographic line wrapping. Each CLI action is followed by its recorded execution status and stdout; all recorded stderr fields contain only whitespace and are omitted. Trailing blank lines are trimmed, but diagnostic messages are retained. Reasoning is omitted. Waiting and termination steps are retained; execution status and the agent's termination declaration are distinct from the saved task-evaluator score.

Case 1 Web search to persistent Vim configuration

Recorded score: 1.0 8 steps

Task ID: `b337d106-053f-4d37-8da0-7f9c4043a66b`

Task. Search the Internet for a tutorial on absolute line numbers in Vim and make them the default for the local editor.

Hybrid pattern: Context handoff. GUI information gathering → CLI configuration.

GUI for information acquisition. Steps 1–4 search the web and inspect the displayed instructions, which identify `set number` and the `.vimrc` configuration file.

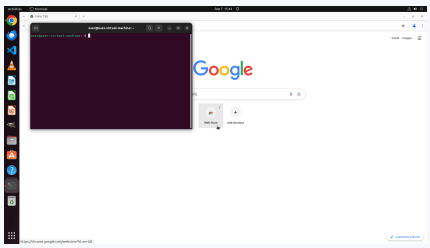
CLI for persistence and verification. Steps 6–7 append the setting to `~/.vimrc` and read back its contents.

Analysis. The handoff connects an instruction observed in the browser to a concrete local configuration change. The two interfaces address different parts of the task: GUI browsing supplies the requested online guidance, while direct file editing makes the setting persistent rather than enabling it only for one Vim session. The recorded read-back confirms `set number`, and the evaluator score is 1.0; no subsequent Vim launch is shown.

STEP 01 / 8 GUI

Case 1 | Vim line numbers

GUI OBSERVATION



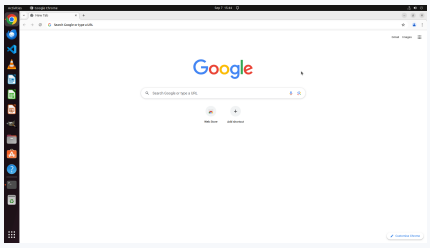
MODEL ACTION bash

```
python3 <<'PY'
import pyautogui
# Click on the Chrome browser window to bring it to
  focus
pyautogui.click(700, 275)
PY
```

STEP 02 / 8 GUI

Case 1 | Vim line numbers

GUI OBSERVATION



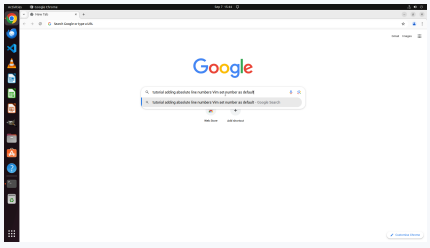
MODEL ACTION bash

```
python3 <<'PY'
import pyautogui
# Click on the Google search bar
pyautogui.click(517, 367)
# Type the search query
pyautogui.typewrite('tutorial adding absolute line
  numbers Vim set number as default', interval=0.02)
PY
```

STEP 03 / 8 GUI

Case 1 | Vim line numbers

GUI OBSERVATION

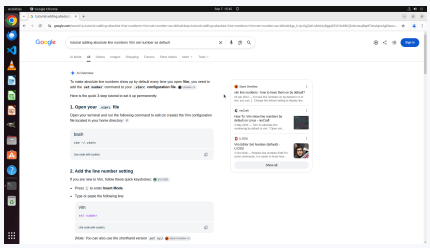


MODEL ACTION bash

```
python3 <<'PY'
import pyautogui
pyautogui.press('enter')
PY
```

STEP 04 / 8 GUI — Case 1 | Vim line numbers

GUI OBSERVATION

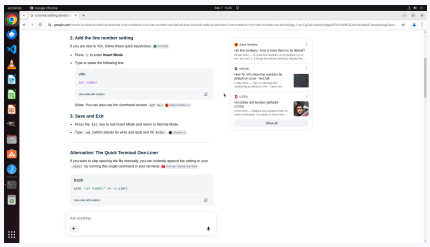


MODEL ACTION bash

```
python3 <<'PY'
import pyautogui
# Scroll down to see more of the tutorial
pyautogui.scroll(-5)
PY
```

STEP 05 / 8 GUI — Case 1 | Vim line numbers

GUI OBSERVATION

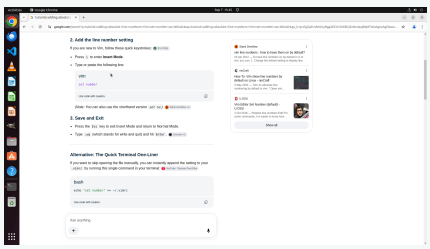


MODEL ACTION bash

```
python3 <<'PY'
import pyautogui
# Click on the terminal window to bring it to focus
pyautogui.click(250, 275)
PY
```

STEP 06 / 8 CLI — Case 1 | Vim line numbers

GUI OBSERVATION



MODEL ACTION bash

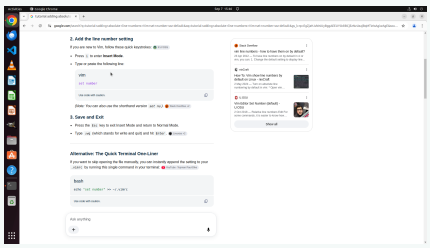
```
echo "set number" >> ~/.vimrc && echo "Added to .vimrc
  . Contents:" && cat ~/.vimrc
```

EXECUTION RESULT success

```
Added to .vimrc. Contents:
set number
```

STEP 07 / 8 CLI — Case 1 | Vim line numbers

GUI OBSERVATION



MODEL ACTION bash

```
echo "=== .vimrc location and content ===" && ls -la
  ~/.vimrc && echo "" && echo "=== Content ===" && cat
   -A ~/.vimrc | head
```

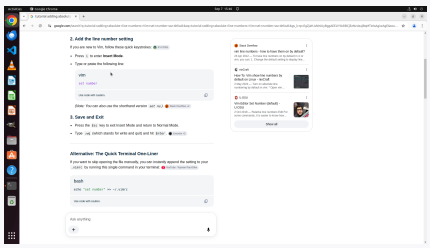
EXECUTION RESULT success

```
=== .vimrc location and content ===
-rw-r--r-- 1 user user 11 Sep  7 15:45 /home/user/.
  vimrc

=== Content ===
set number$
```

STEP 08 / 8 Control — Case 1 | Vim line numbers

GUI OBSERVATION



MODEL ACTION `terminate`

```
terminate(status="success")
```

Case 2 GUI preferences to a direct VLC configuration edit

Recorded score: 1.0 — 9 steps

Task ID: `9195653c-f4aa-453d-aa95-787f6ccfaae9`

Task. Raise VLC's maximum volume from 125% to 200%.

Hybrid pattern: Adaptive interface fallback. GUI exploration → CLI configuration repair.

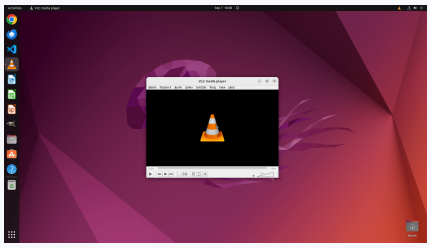
GUI for preference exploration. Steps 1–6 navigate VLC's preferences and audio settings, including a wait; this sequence does not establish the requested maximum.

CLI for targeted configuration and read-back. Step 7 locates `vlcrc` and finds `#qt-max-volume=125`. Step 8 replaces it with `qt-max-volume=200` and checks the saved value.

Analysis. Instead of continuing the same GUI search, the agent changes the level of interaction from a preferences dialog to its backing configuration file. This is a fallback from an unproductive interaction route, not evidence that VLC's GUI cannot support the setting. The CLI completes the change and exposes an explicit value for verification. The evaluator relaunches VLC and checks the configuration, returning 1.0; the rollout itself does not show the updated volume control.

STEP 01 / 9 GUI — Case 2 | VLC volume limit

GUI OBSERVATION

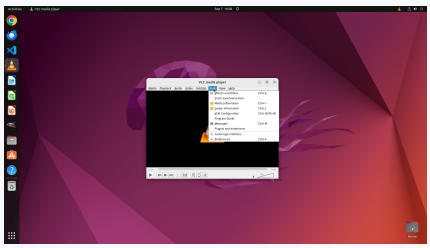


MODEL ACTION `bash`

```
python3 <<'PY'
import pyautogui
pyautogui.click(493, 344)
PY
```

STEP 02 / 9 GUI — Case 2 | VLC volume limit

GUI OBSERVATION

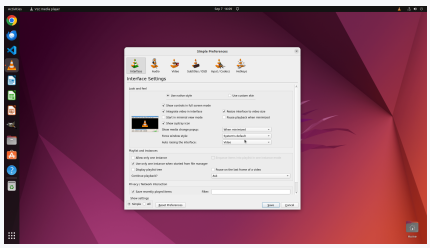


MODEL ACTION `bash`

```
python3 <<'PY'
import pyautogui
pyautogui.click(566, 557)
PY
```

STEP 03 / 9 GUI — Case 2 | VLC volume limit

GUI OBSERVATION

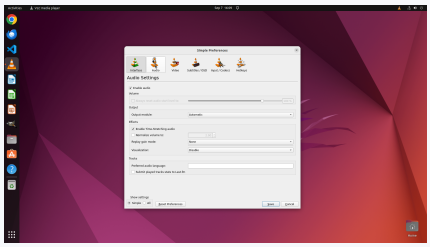


MODEL ACTION bash

```
python3 <<'PY'
import pyautogui
pyautogui.click(355, 248)
PY
```

STEP 04 / 9 GUI — Case 2 | VLC volume limit

GUI OBSERVATION

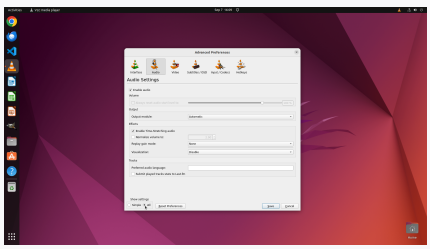


MODEL ACTION bash

```
python3 <<'PY'
import pyautogui
pyautogui.click(331, 831)
PY
```

STEP 05 / 9 Control — Case 2 | VLC volume limit

GUI OBSERVATION

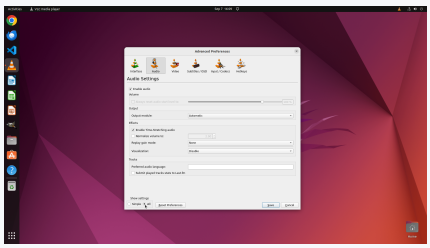


MODEL ACTION wait

```
wait(time=1)
```

STEP 06 / 9 GUI — Case 2 | VLC volume limit

GUI OBSERVATION

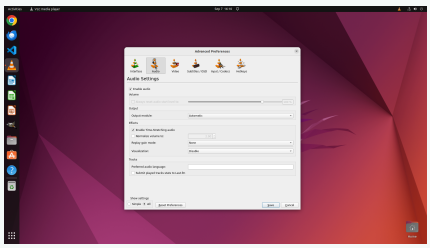


MODEL ACTION bash

```
python3 <<'PY'
import pyautogui
pyautogui.click(355, 248)
PY
```

STEP 07 / 9 CLI — Case 2 | VLC volume limit

GUI OBSERVATION



MODEL ACTION bash

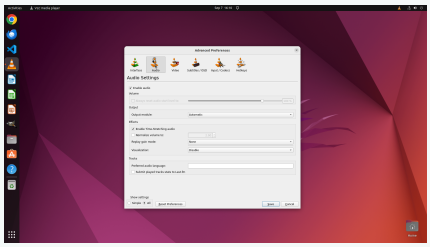
```
find / -name "vlcrc" 2>/dev/null; echo "---"; grep -n
  "max-volume" ~/.config/vlc/vlcrc 2>/dev/null || echo
   "not in home"
```

EXECUTION RESULT success

```
/home/user/.config/vlc/vlcrc
---
2650:#qt-max-volume=125
```

STEP 08 / 9 CLI — Case 2 | VLC volume limit

GUI OBSERVATION



MODEL ACTION bash

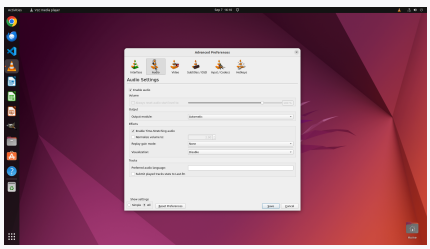
```
sed -i 's/^#qt-max-volume=125/qt-max-volume=200/' /
  home/user/.config/vlc/vlcrc && grep -n "qt-max-
  volume" /home/user/.config/vlc/vlcrc
```

EXECUTION RESULT success

```
2650:qt-max-volume=200
```

STEP 09 / 9 Control — Case 2 | VLC volume limit

GUI OBSERVATION



MODEL ACTION terminate

```
terminate(status="success")
```

Case 3 CLI setup, GUI interaction, and screenshot verification

Recorded score: 1.0 — 8 steps

Task ID: `02ce9a50-7af2-47ed-8596-af0c230501f8`

Task. Run `ls` in `/home/user` for a Linux tutorial and save a screenshot of the terminal as `ls.png` on the Desktop.

Hybrid pattern: Execution–verification loop. CLI setup → GUI execution → CLI capture and checks.

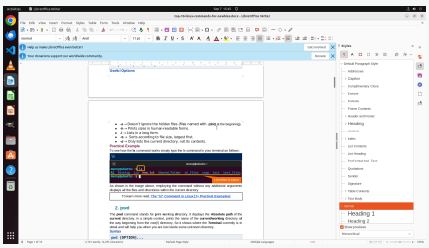
GUI for visible terminal execution. Step 3 focuses the opened terminal, types `ls /home/user`, and presses Enter so the output appears in the window to be captured.

CLI for setup, capture, and artifact inspection. Steps 1–2 inspect the environment and open the terminal; steps 4–7 save its screenshot, inspect the PNG, and open it in an image viewer.

Analysis. A command result in the execution channel alone would not provide the requested terminal image. Here, CLI setup creates the window, GUI input produces the visible content, and CLI capture turns that state into a saved artifact. The final observation displays the image opened by step 7, closing the loop from file checks back to visual inspection. Capture itself is a CLI action, not a GUI action. The recorded PNG is 876×533 and the score is 1.0; the evaluator checks for `ls` (or an OCR variant), not every directory entry.

STEP 01 / 8 CLI — Case 3 | Terminal screenshot

GUI OBSERVATION



MODEL ACTION bash

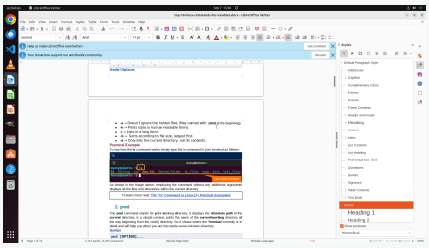
```
which gnome-terminal xdg-open; ls ~/Desktop/ 2>/dev/
  null; echo "---"; ls /home/user | head -30
```

EXECUTION RESULT success

```
/usr/bin/gnome-terminal
/usr/bin/xdg-open
top-10-linux-commands-for-newbies.docx
---
Desktop
Documents
Downloads
Music
Pictures
Public
requirement.txt
server
snap
Templates
Videos
```

STEP 02 / 8 CLI — Case 3 | Terminal screenshot

GUI OBSERVATION



MODEL ACTION bash

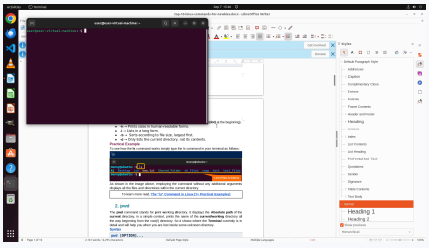
```
(DISPLAY=:0 gnome-terminal --geometry=90x24 & sleep 3;
   echo "opened")
```

EXECUTION RESULT success

```
opened
```

STEP 03 / 8 GUI — Case 3 | Terminal screenshot

GUI OBSERVATION

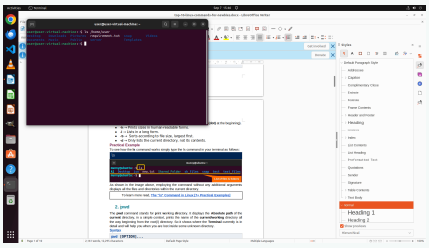


MODEL ACTION bash

```
python3 <<'PY'
import pyautogui
pyautogui.click(265, 275)
import time
time.sleep(0.5)
pyautogui.typewrite('ls /home/user', interval=0.03)
time.sleep(0.3)
pyautogui.press('enter')
PY
```

STEP 04 / 8 CLI — Case 3 | Terminal screenshot

GUI OBSERVATION



MODEL ACTION bash

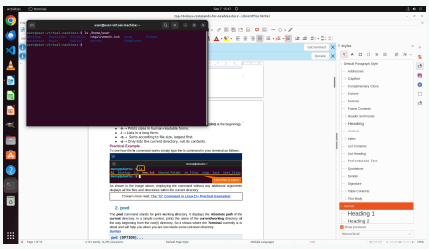
```
echo "=== windows ==="; wmctrl -lG 2>/dev/null | grep
  -iE "terminal|gnome" || echo "no wmctrl"; echo "===
  tools ==="; which import gnome-screenshot scrot 2>/
  dev/null; echo "=== try gnome-screenshot -w ===";
  DISPLAY=:0 gnome-screenshot -w -f ~/Desktop/ls.png
  2>&1; echo "exit=$?"; ls -la ~/Desktop/ls.png 2>&1
```

EXECUTION RESULT success

```
=== windows ===
no wmctrl
=== tools ===
/usr/bin/gnome-screenshot
=== try gnome-screenshot -w ===
exit=0
-rw-r--r-- 1 user user 27955 Sep  7 15:47 /home/user/
  Desktop/ls.png
```

STEP 05 / 8 CLI Case 3 | Terminal screenshot

GUI OBSERVATION



MODEL ACTION bash

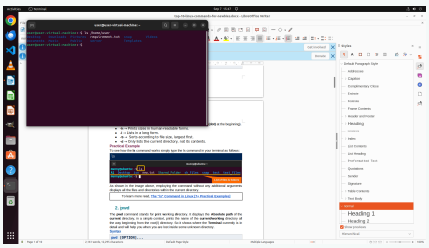
```
file ~/Desktop/ls.png; echo "=== dimensions ==="; python3 -c "from PIL import Image; im=Image.open('/home/user/Desktop/ls.png'); print('size:', im.size, 'mode:', im.mode)" 2>/dev/null || echo "PIL not available"; echo "=== check for terminal text in image ==="; python3 -c "
from PIL import Image
im=Image.open('/home/user/Desktop/ls.png')
# sample a region where terminal text should be
w,h=im.size
print('image size', w, h)
"
```

EXECUTION RESULT success

```
/home/user/Desktop/ls.png: PNG image data, 876 x 533, 8-bit/color RGBA, non-interlaced
=== dimensions ===
size: (876, 533) mode: RGBA
=== check for terminal text in image ===
image size 876 533
```

STEP 06 / 8 CLI Case 3 | Terminal screenshot

GUI OBSERVATION



MODEL ACTION bash

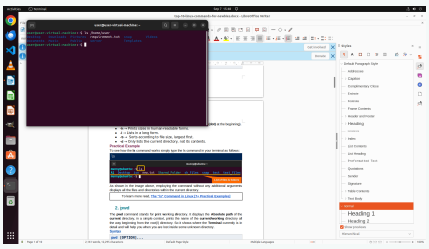
```
echo "=== terminal window geometry ==="; DISPLAY=:0 xdotool search --name "user@user-virtual-machine" getwindowgeometry %@ 2>/dev/null | head; echo "=== analyze ls.png: dark bg + text ==="; python3 <<'PY'
from PIL import Image
im = Image.open('/home/user/Desktop/ls.png').convert('RGB')
w,h = im.size
px = im.load()
# count dark pixels (background) vs light pixels (text)
dark, light, mid = 0,0,0
for y in range(0,h,2):
    for x in range(0,w,2):
        r,g,b = px[x,y]
        v = (r+g+b)/3
        if v < 60: dark+=1
        elif v > 200: light+=1
        else: mid+=1
print(f"image {w}x{h}")
print(f"dark px: {dark}, light px: {light}, mid: {mid}")
# sample a horizontal strip in the middle to see text lines
for y in [h//2, h//2+20, h//2+40]:
    vals=[]
    for x in range(0,w,4):
        v=(px[x,y][0]+px[x,y][1]+px[x,y][2])/3
        vals.append(v)
    print(f"y={y}: min={min(vals):.0f} max={max(vals):.0f} mean={sum(vals)/len(vals):.0f}")
PY
```

EXECUTION RESULT success

```
=== terminal window geometry ===
=== analyze ls.png: dark bg + text ===
image 876x533
dark px: 114584, light px: 324, mid: 2038
y=266: min=0 max=166 mean=30
y=286: min=0 max=166 mean=30
y=306: min=0 max=166 mean=30
```

STEP 07 / 8 CLI — Case 3 | Terminal screenshot

GUI OBSERVATION



MODEL ACTION `bash`

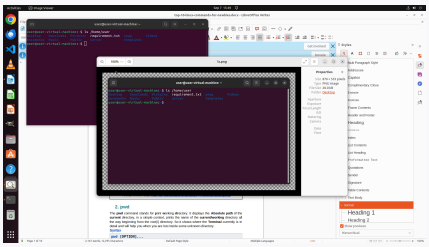
```
(DISPLAY=:0 eog ~/Desktop/ls.png >/dev/null 2>&1 &
  sleep 2; echo "opened eog")
```

EXECUTION RESULT `success`

```
opened eog
```

STEP 08 / 8 Control — Case 3 | Terminal screenshot

GUI OBSERVATION



MODEL ACTION `terminate`

```
terminate(status="success")
```

G Prompt Used in Evaluation And Training Data Construction

G.1 OSWorld Evaluation: Unified Action Schema

This is the system prompt used for our OSWorld evaluation. A single `computer_use` tool exposes `action=bash` for both direct shell commands and GUI operations via PyAutoGUI heredocs. It differs from the separate-tool `hybrid` configuration in the action-schema analysis (Table 4), whose prompt is given in Appendix G.2.

System Prompt: OSWorld Evaluation (Unified Bash)

Role

You are a multi-purpose intelligent assistant operating a computer through a bash terminal. The password of the computer is `password`.

Environment

You face a machine with a graphical desktop AND a bash terminal. You act ONLY by running one shell command per step (`action=bash`): write shell for CLI/file operations, or drive the GUI with `pyautogui` inside a quoted heredoc `python3 <<'PY' ... PY` (coordinates are 0–999). Each step you are shown the latest screenshot plus the previous command's output.

Tools

You have access to the following functions:

```
Tool type:              function
Function name:          computer_use
Parameters type:        object
Required parameters:    ["action"]
```

Control the machine through a bash terminal. One action per call. The screen is a GUI desktop: drive it with `pyautogui` via a quoted heredoc, or run shell commands / file operations directly.

Tool Parameters

- `action` *(string; required).* One of `bash`, `wait`, `terminate`, `answer`. The action descriptions are given below.
- `command` *(string).* Required by `action=bash`. A single shell command. For GUI, run `pyautogui` via a quoted heredoc: `python3 <<'PY' ... PY` (coordinates 0–999).
- `timeout` *(number).* Optional for `action=bash`: seconds to allow the command (default 60). The sandbox cuts commands short after approximately 30 s regardless.
- `time` *(number).* Optional for `action=wait`: seconds.
- `status` *(string).* One of `success`, `failure`. Required by `action=terminate`.
- `text` *(string).* Required by `action=answer`.

Action: bash

Run ONE shell command (requires `command`).

- CLI / file ops: write shell directly (`ls`, `cat`, `sed`, `grep`, `python3 - <<'PY' ... PY`, ...).
- GUI: drive the screen with `pyautogui` inside a QUOTED heredoc, e.g.

```
python3 <<'PY'
import pyautogui
pyautogui.click(500, 300)  # coordinates are 0-999
                           # screen treated as 1000x1000
pyautogui.typewrite('hello', interval=0.02)
PY
```

PyAutoGUI Functions

```
pyautogui.click(x, y)                        # left click
pyautogui.doubleClick(x, y)                  # double click
pyautogui.tripleClick(x, y)                  # triple click
pyautogui.rightClick(x, y)                   # right click
pyautogui.middleClick(x, y)                  # middle click
pyautogui.moveTo(x, y)                       # move mouse
pyautogui.dragTo(x, y, duration=0.5)         # drag to position
pyautogui.mouseDown()                        # press mouse button
pyautogui.mouseUp()                          # release mouse button
pyautogui.scroll(-5)                         # scroll down (negative=down)
pyautogui.press('enter')                     # press a key
pyautogui.hotkey('ctrl', 's')                # keyboard shortcut
pyautogui.typewrite('text', interval=0.02)   # type text
pyautogui.keyDown('shift')                   # hold key down
pyautogui.keyUp('shift')                     # release key
```

- Effects are seen via the NEXT screenshot; use `print()` for any text you need back (stdout is returned).
- One command may bundle multiple steps (several `pyautogui` lines in the heredoc; `&&` / pipes in shell).
- Optional `timeout` (seconds) for a slow command; the sandbox cuts commands short after approximately 30 s, so split anything longer or run it in the background.

Control Actions

- `wait`**:** wait for the screen to settle. Optional `time` (seconds).
- `terminate`**:** finish the task. Requires `status = success | failure`.
- `answer`**:** answer a question-type task. Requires `text`.

Function Call Format

If you choose to call a function ONLY reply in the following format with NO suffix:

```
<tool_call>
<function=example_function_name>
<parameter=example_parameter_1>
value_1
</parameter>
</function>
</tool_call>
```

Important

<IMPORTANT>

- Function calls MUST follow the specified format.
- The `action` parameter MUST be one of: `bash`, `wait`, `terminate`, `answer`.
- ALL GUI interactions MUST use `action=bash` with a `pyautogui` heredoc.
- Coordinates are 0–999 (the screen is a 1000x1000 grid).
- Heredoc delimiter MUST be quoted: `<<'PY'` (not `<<PY`).
- Observe effects via the NEXT screenshot; use `print()` for text output.
- When finished, use `action=terminate` (not bash exit commands).
- The current date is Thursday, September 24, 2026.
- Collapsed screenshots appear as: `This screenshot has been collapsed.`

</IMPORTANT>

Response Format

Every step:

1) Action: one sentence describing your next move.
2) A single <tool_call>...</tool_call> block.

Output Format Examples

GUI action (action=bash **with pyautogui heredoc**)

Action: Click the "File" menu in the top menu bar.

```
<tool_call>
<function=computer_use>
<parameter=action>
bash
</parameter>
<parameter=command>
python3 <<'PY'
import pyautogui
pyautogui.click(50, 15)
PY
</parameter>
</function>
</tool_call>
```

Shell command (action=bash **with shell**)

Action: List files in the Documents folder.

```
<tool_call>
<function=computer_use>
<parameter=action>
bash
</parameter>
<parameter=command>
ls -la ~/Documents/
</parameter>
</function>
</tool_call>
```

Wait (action=wait)

Action: Wait for the application to finish loading.

```
<tool_call>
<function=computer_use>
<parameter=action>
wait
</parameter>
<parameter=time>
3
</parameter>
</function>
</tool_call>
```

Finish (action=terminate)

Action: The task is complete.

```
<tool_call>
<function=computer_use>
<parameter=action>
terminate
</parameter>
<parameter=status>
success
</parameter>
</function>
</tool_call>
```

G.2 Action-Schema Analysis: Separate GUI and CLI Tools

The following system prompt belongs to the *Separate GUI/CLI tools* variant in Table 4 (Section 4.5). Unlike the unified OSWorld prompt above, it keeps the two interfaces in two distinct tools: `computer_use` for GUI interaction and `cli` for terminal and file operations. HybridCUA instead exposes both through the single `bash` action of Appendix G.1.

System Prompt: Separate GUI/CLI Tools

 # Role

You are a multi-purpose intelligent assistant. Based on my requests, you can use tools to help me complete various tasks. The password of the computer is `password`.

 # Tools

You have access to the following functions:

 # GUI Tool: computer_use

```
Tool type:             function
Function name:         computer_use
Parameters type:       object
Required parameters:   ["action"]
```

Use a mouse and keyboard to interact with a computer, and take screenshots.

- This is an interface to a desktop GUI. For terminal commands and file operations, use the separate `cli` tool (it runs on the same machine); use this GUI for anything that must be seen or clicked on screen. You can also launch applications from the terminal via `cli` or by clicking desktop icons.
- Some applications may take time to start or process actions, so you may need to wait and take successive screenshots to see the results of your actions.
- The screen's resolution is `1000x1000`.
- Whenever you intend to move the cursor to click on an element like an icon, you should consult a screenshot to determine the coordinates of the element before moving the cursor.
- If you tried clicking on a program or link but it failed to load, even after waiting, try adjusting your cursor position so that the tip of the cursor visually falls on the element that you want to click.
- Make sure to click any buttons, links, icons, etc with the cursor tip in the center of the element. Don't click boxes on their edges unless asked.

 # GUI Actions

The `action` parameter is a required string with the following values:

- `key`**:** Performs key down presses on the arguments passed in order, then performs key releases in reverse order.
- `type`**:** Type a string of text on the keyboard.
- `mouse_move`**:** Move the cursor to a specified (x, y) pixel coordinate on the screen.
- `left_click`**:** Click the left mouse button at a specified (x, y) pixel coordinate on the screen. Optional `text` parameter can specify modifier keys (e.g., `"ctrl"`, `"shift"`, `"ctrl+shift"`) that will be held during the click.
- `left_click_drag`**:** Click and drag the cursor to a specified (x, y) coordinate.
- `right_click`**:** Click the right mouse button at a specified (x, y) pixel coordinate on the screen. Optional `text` parameter can specify modifier keys that will be held during the click.
- `middle_click`**:** Click the middle mouse button at a specified (x, y) pixel coordinate on the screen. Optional `text` parameter can specify modifier keys that will be held during the click.

- `double_click`**:** Double-click the left mouse button at a specified (x, y) pixel coordinate on the screen. Optional `text` parameter can specify modifier keys that will be held during the click.
- `triple_click`**:** Triple-click the left mouse button at a specified (x, y) pixel coordinate on the screen (simulated as double-click since it's the closest action). Optional `text` parameter can specify modifier keys that will be held during the click.
- `scroll`**:** Performs a scroll of the mouse scroll wheel. Optional `text` parameter can specify a modifier key (e.g., `"shift"`, `"ctrl"`) that will be held during scrolling.
- `hscroll`**:** Performs a horizontal scroll (mapped to regular scroll). Optional `text` parameter can specify a modifier key that will be held during scrolling.
- `wait`**:** Wait specified seconds for the change to happen.
- `terminate`**:** Terminate the current task and report its completion status.
- `answer`**:** Answer a question.

 # GUI Tool Parameters

- `keys` *(array).* Required only by `action=key`.
- `text` *(string).* Required by `action=type` and `action=answer`. Optional for click actions (`left_click`, `right_click`, `middle_click`, `double_click`, `triple_click`) to specify modifier keys (e.g., `'ctrl'`, `'shift'`, `'ctrl+shift'`). Optional for scroll actions (`scroll`, `hscroll`) to specify a modifier key (e.g., `'shift'`, `'ctrl'`) to hold during scrolling.
- `coordinate` *(array).* (x, y) coordinates.
- `pixels` *(number).* Scroll amount.
- `time` *(number).* Seconds to wait.
- `status` *(string).* Task status for terminate. One of `success`, `failure`.

 # Terminal Tool: cli

```
Tool type:             function
Function name:         cli
Parameters type:       object
Required parameters:   ["action"]
Action type:           string
Action values:         ["bash"]
```

Run a shell command inside the machine (the same VM the GUI acts on). The only supported `action` is `bash`:

- `bash`**:** run a shell command. Requires `command`; optional `timeout` (seconds).

Output (stdout/stderr) is returned and also reflected in the next screenshot.

 # CLI Tool Parameters

- `command` *(string).* Required by `action=bash`: the shell command.
- `timeout` *(number).* Optional for `action=bash`: timeout in seconds.

 # Function Call Format

If you choose to call a function ONLY reply in the following format with NO suffix:

```
<tool_call>
<function=example_function_name>
<parameter=example_parameter_1>
value_1
</parameter>
<parameter=example_parameter_2>
This is the value for the second parameter
that can span
```

```
multiple lines
</parameter>
</function>
</tool_call>
```

 # Important

<IMPORTANT>

Reminder:

- Function calls MUST follow the specified format: an inner <function=...></function> block must be nested within <tool_call></tool_call> XML tags.
- Required parameters MUST be specified.
- You may provide optional reasoning for your function call in natural language BEFORE the function call, but NOT after.
- If there is no function call available, answer the question like normal with your current knowledge and do not tell the user about function calls.
- The current date is Thursday, September 24, 2026.
- Collapsed screenshots appear as text: `This screenshot has been collapsed.`

</IMPORTANT>

 # Response Format

Response format for every step:

1) Action: a short imperative describing the next step — a GUI interaction (`computer_use`) or a terminal/file operation (`cli`).
2) A single <tool_call>...</tool_call> block.

Rules:

- Output exactly in the order: Action, <tool_call>.
- Be brief: one sentence for Action.
- Do not output anything else outside those parts.
- If finishing, use `action=terminate` in the tool call.

G.3 Trajectory Construction: Domain CLI Skills

The following eight domain-specific CLI skills provide practical guidance for Qwen3.8-27B during trajectory collection in CUA-Gym, and are reproduced here exactly as used. Their scope is limited to teacher sampling. They are not included in the student model's supervised context, are not available during RL rollouts, and appear in no evaluation prompt; the student therefore never observes a skill at training or test time.

CLI Skill: GIMP

 # GIMP — CLI toolkit

Two paths: **Pillow** for plain raster edits (resize, crop, rotate, format convert, color tweaks) — simplest and most reliable; **GIMP batch Script-Fu** when the task needs a GIMP-specific feature (layers, `.xcf`, filters/plugins, GIMP-exact output).

 # Pillow — prefer this for ordinary image ops

```
python3 <<'PY'
from PIL import Image, ImageEnhance, ImageFilter, ImageOps
im = Image.open("/home/user/photo.jpg")
print(im.size, im.mode)
im = im.resize((800, 600))
# or im.thumbnail((800,600)) to keep ratio
```

```
im = im.rotate(90, expand=True)
im = im.crop((left, top, right, bottom))
im = ImageOps.grayscale(im)
im = ImageEnhance.Brightness(im).enhance(1.2)
im = im.filter(ImageFilter.GaussianBlur(2))
im.convert("RGB").save("/home/user/out.png")
# convert("RGB") before saving jpg
PY
```

GIMP batch (Script-Fu) — for .xcf / layers / GIMP filters

The file usually is NOT open (GIMP is heavy); if it is: `pkill -f gimp; sleep 3`. Run GIMP headless and quit at the end:

```
gimp -i -b '
  (let* ((image (car (gimp-file-load RUN-NONINTERACTIVE
                                     "/home/user/in.xcf" "in.xcf")))
         (drawable (car (gimp-image-flatten image))))
    (gimp-image-scale image 800 600)
    (file-png-save RUN-NONINTERACTIVE image drawable
                   "/home/user/out.png" "out" 0 9 1 1 1 1 1))
' -b '(gimp-quit 0)'
```

Handy Script-Fu calls: `gimp-image-flatten`, `gimp-image-scale`, `gimp-image-crop`, `gimp-item-transform-rotate-simple`, `gimp-image-convert-grayscale`, `gimp-text-fontname` (add text), `file-jpeg-save` / `file-png-save` / `gimp-xcf-save`.

Gotchas

- Flatten (`gimp-image-flatten`) before exporting to a flat format like PNG/JPEG.
- `.xcf` output must use `gimp-xcf-save`, not `file-*-save`.
- Preserve the exact output size / format / path the instruction states.
- `gimp -i` (no UI) + always end with `-b '(gimp-quit 0)'` or the process hangs.

CLI Skill: LibreOffice Calc

LibreOffice Calc — CLI

Edit on disk with openpyxl, re-read to verify, stop. Only the saved file carries the document state, not an unsaved GUI buffer; `.~lock` is advisory (doesn't block writes). Ensure lib:

```
python3 -c "import openpyxl" 2>/dev/null || pip install -q openpyxl
```

Do NOT pkill/reopen soffice, Ctrl+S in the GUI, or click a "Document Recovery" dialog — it can restore the pre-edit version and clobber your file.

Write LITERAL values, not formula strings (`ws.cell(r,4).value=sales-cogs`) — openpyxl does not evaluate formulas, so `"=B2-C2"` is stored as a string with no cached value.

```
python3 <<'PY'
import openpyxl
p="/home/user/data.xlsx"; wb=openpyxl.load_workbook(p); ws=wb.active
print([ws.cell(1,c).value for c in range(1,ws.max_column+1)])
for r in range(2,ws.max_row+1):
    ws.cell(r,4).value = ws.cell(r,2).value - ws.cell(r,3).value  # literal
wb.save(p)
v=[openpyxl.load_workbook(p).active.cell(r,4).value
   for r in range(2,ws.max_row+1)]
assert all(isinstance(x,(int,float)) for x in v), v; print("OK",v)
PY
```

- Charts: `Reference(ws,min_col,min_row,max_col,max_row)` + `add_data(titles_from_data=True)` + `set_categories(cats)`; not

chart_type/Series.values=. openpyxl quotes the sheet name in series references → strip it via series.val.numRef.f/strRef.f for a bare reference.

- Pivot: when only the aggregated values are required, aggregate in Python to a static sheet (same-width header/data rows). **A native pivotTable object cannot be built by openpyxl; use UNO** DataPilotTables.createDataPilotDescriptor**.** Lookup/threshold: bisect, not an exact dict.
- Space in sheet name → quote: ='RetailPrice'.B2 (else Err:509). 0.00 not 0,00. Colors need FF alpha (FFFF0000), exact hex. .ods→odfpy, .csv→pandas. Keep exact sheet/cell/OUTPUT path.

Recalc / Calc-only

Simplest:

```
soffice --headless --convert-to xlsx --outdir DIR FILE
```

(recomputes; never --convert-to pdf). Full engine via UNO: start

```
soffice --headless --accept="socket,host=localhost,port=2002;urp;" &
```

connect (UnoUrlResolver→ Desktop.loadComponentFromURL(...Hidden=True)), doc.calculateAll();doc.store().

CLI Skill: LibreOffice Impress

LibreOffice Impress — CLI

Edit the .pptx on disk with python-pptx, re-read to verify, stop. Only the saved file carries the document state, not an unsaved GUI buffer; .~lock is advisory. Ensure lib:

```
python3 -c "import pptx" 2>/dev/null || pip install -q python-pptx
```

Do NOT pkill/reopen soffice, Ctrl+S in the GUI, or click a "Document Recovery" dialog — it can clobber your file. Prefer python-pptx over clicking objects. Set formatting at the **run level** and read it back (an object-level change may not reach run attributes).

```
python3 <<'PY'
from pptx import Presentation
from pptx.dml.color import RGBColor
p="/home/user/deck.pptx"; prs=Presentation(p); s=prs.slides[1]
def paint(tf):
    for para in tf.paragraphs:
        for r in para.runs:
            r.font.bold=True
            r.font.color.rgb=RGBColor(0xC9,0x21,0x1E)
for sh in s.shapes:
    if sh.has_text_frame: paint(sh.text_frame)
    if sh.has_table:
        for row in sh.table.rows:
            for c in row.cells: paint(c.text_frame)
            # don't skip table cells
prs.save(p)
cols={r.font.color.rgb for sh in Presentation(p).slides[1].shapes
      if sh.has_text_frame for para in sh.text_frame.paragraphs
      for r in para.runs}
assert RGBColor(0xC9,0x21,0x1E) in cols, cols; print("OK")
PY
```

- Notes: s.notes_slide.notes_text_frame.text=.... New slide: prs.slides.add_slide(layout). Picture: s.shapes.add_picture(img,left,top,width=Cm(8)); move via shape.top/left.
- Exact hex named (Dark Red 2 = C9211E, not FF0000); watch autocorrect curly '. .odp →convert. Keep OUTPUT path.

- `font.name/size` stay `None` unless set explicitly on the run; `text=/add_paragraph` drops the placeholder's inherited `pPr` (bullet `buChar`, size). Background, strikethrough, and bullet changes can fail without raising, so write them through the underlying `<a:p>` XML via lxml and read the result back.

 # Export / Impress-only

```
soffice --headless --convert-to pdf --outdir DIR FILE
```

or UNO: start

```
soffice --headless --accept="socket,host=localhost,port=2002;urp;" &
```

connect, then

```
doc.storeToURL("file:///.../deck.pdf", (
    PropertyValue(Name="FilterName",Value="impress_pdf_Export"),))
```

CLI Skill: LibreOffice Writer

 # LibreOffice Writer — CLI

Edit the `.docx` on disk with python-docx, re-read to verify, stop. Only the saved file carries the document state, not an unsaved GUI buffer; `.~lock` is advisory. Ensure lib:

```
python3 -c "import docx" 2>/dev/null || pip install -q python-docx
```

Do NOT pkill/reopen soffice, Ctrl+S in the GUI, or click a "Document Recovery" dialog — it can clobber your file with the pre-edit version.

Edit at the **run level** (`run.text`/`run.font`), never `par.text=` (drops run formatting).

```
python3 <<'PY'
import docx
p="/home/user/report.docx"; d=docx.Document(p)
for par in d.paragraphs:
    for r in par.runs:
        if "OLD" in r.text: r.text=r.text.replace("OLD","NEW")
d.save(p)
d2=docx.Document(p)
assert not any("OLD" in r.text for par in d2.paragraphs for r in par.runs)
print("OK")
PY
```

- Run format: `run.bold=True`; `run.font.size=Pt(14)`; `run.font.color.rgb=RGBColor(0xFF,0,0)`.
- Tables: `doc.tables`→`table.rows`→`cell.paragraphs`→runs. `.odt`→odfpy. Keep exact OUTPUT path.
- Export: `soffice--headless--convert-topdf--outdirDIRFILE`. If the exported text drops soft-wrap trailing spaces, re-export via UNO with `FilterData=[UseTaggedPDF=True]`.

 # Writer-only (update TOC/fields, mail merge)

Start

```
soffice --headless --accept="socket,host=localhost,port=2002;urp;" &
```

connect (`UnoUrlResolver`→ `Desktop.loadComponentFromURL(...Hidden=True)`), e.g. `doc.getTextFields().refresh();doc.store()`.

CLI Skill: OS / Desktop

OS / desktop — CLI toolkit

Filesystem, process, permission, package, and system-setting tasks are almost entirely shell work — the GUI file manager and settings dialogs are slower and error-prone. Drive these from bash and verify by reading back state.

Files & directories

```
ls -la /home/user         # inspect permissions, sizes, hidden files
find /home/user -name '*.log' -mtime -1  # search by name/time
mkdir -p /home/user/a/b/c
cp -r src dst; mv old new; rm -rf junk
du -sh /home/user/*       # sizes; df -h for disk
```

Content edits & search

```
grep -rn "TODO" /home/user/proj
sed -i 's/old/new/g' /home/user/file.txt  # in-place edit
python3 - <<'PY'                        # structured/json edits
import json, pathlib
p = pathlib.Path("/home/user/config.json")
d = json.loads(p.read_text()); d["key"] = "value"
p.write_text(json.dumps(d, indent=2))
PY
```

Permissions & ownership

```
chmod 644 file; chmod +x script.sh; chmod -R 755 dir
sudo chown user:user file  # password is the sudo password in the prompt
```

Processes, archives, packages

```
ps aux | grep -i firefox; pkill -f soffice.bin
tar -czf out.tar.gz dir/; tar -xzf in.tar.gz -C dest/; unzip a.zip -d dest/
sudo apt-get install -y <pkg>  # proxy is exported for the bash channel
```

GNOME desktop settings (wallpaper, theme, etc.)

```
gsettings set org.gnome.desktop.background picture-uri \
    'file:///home/user/bg.png'
gsettings get org.gnome.desktop.interface gtk-theme
```

Gotchas

- `sudo` needs `-S` with the password piped, or an interactive tty; the prompt states the password. Example: `echo"$PW"|sudo-S<cmd>`.
- `rm -rf` is irreversible — double-check the path before running it.
- Verify every change (`ls -la`, `cat`, `stat`, `gsettings get`) before terminating.

CLI Skill: PDF

PDF — CLI toolkit

PDF tasks (stamp text, merge/split, rotate, extract text/images, count pages, fill forms, redact) are almost always faster and more accurate via CLI than GUI. PDFs have no persistent editing app holding a lock, so no `pkill` dance is needed — just read the input path and write the exact OUTPUT path the instruction names.

Edit / annotate with PyMuPDF (fitz)

```
python3 <<'PY'
import fitz
```

```
doc = fitz.open("/home/user/in.pdf")
print("pages:", doc.page_count)
for page in doc:
    print(page.get_text()[:200])  # extract text
    # (x,y) in points, origin top-left
    page.insert_text((72, 40), "Filed: April 1, 2026",
                     fontsize=10, fontname="times", color=(0, 0, 0))
    tw = fitz.get_text_length("Case No. 2026-CV-04521",
                              fontname="times", fontsize=10)
    page.insert_text((page.rect.width - 72 - tw, 40),
                     "Case No. 2026-CV-04521", fontsize=10,
                     fontname="times")
doc.save("/home/user/out.pdf")  # or doc.saveIncr() to edit in place
PY
```

 # Common recipes with pypdf

```
python3 <<'PY'
from pypdf import PdfReader, PdfWriter
# merge
w = PdfWriter()
for f in ["/home/user/a.pdf", "/home/user/b.pdf"]:
    for pg in PdfReader(f).pages: w.add_page(pg)
with open("/home/user/merged.pdf", "wb") as fh: w.write(fh)
# split / rotate / extract a page range
r = PdfReader("/home/user/in.pdf")
w2 = PdfWriter()
for pg in r.pages[0:3]:
    pg.rotate(90)
    w2.add_page(pg)
with open("/home/user/first3_rotated.pdf", "wb") as fh: w2.write(fh)
PY
```

 # Other tools

- `page.insert_image(rect,filename=...)` (fitz) to stamp a logo/image.
- `pdfplumber` for table extraction; `pikepdf` for encryption/metadata/repair.
- Make a PDF from a doc: `libreoffice--headless--convert-topdf--outdir/home/userX.docx`.
- Rasterize a page to check visually: `page.get_pixmap(dpi=150).save("/tmp/p0.png")`.

 # Gotchas

- fitz coordinates are POINTS from the TOP-LEFT; y grows downward.
- Use exactly the font name/size the instruction specifies; `insert_text` needs the y baseline, not the top of the glyph.
- Write to the precise OUTPUT path the instruction names, not back to the input path.

CLI Skill: VLC / Media

 # VLC / media — CLI toolkit

Media tasks split cleanly: **inspect/convert/trim/extract** → `ffmpeg`/`ffprobe`; **edit tags/metadata** → `mutagen`; **playback / playlist / snapshot** actions that must happen inside VLC → the `cvlc` command line or GUI. Only touch the VLC GUI when the task is about VLC's own state (now-playing, playlist order, a setting).

 # Inspect with ffprobe

```
ffprobe -v error -show_format -show_streams /home/user/clip.mp4  # full info
ffprobe -v error -show_entries format=duration -of csv=p=0 \
    /home/user/clip.mp4
```

Convert / trim / extract with ffmpeg

```
ffmpeg -y -i in.mp4 -c:v libx264 -c:a aac out.mkv       # transcode
ffmpeg -y -ss 00:00:10 -to 00:00:25 -i in.mp4 \
    -c copy clip.mp4                                  # trim (fast copy)
ffmpeg -y -i in.mp4 -vn -acodec libmp3lame audio.mp3   # extract audio
ffmpeg -y -i in.mp4 -ss 5 -vframes 1 frame.png         # snapshot a frame
ffmpeg -y -i in.mp4 -vf scale=1280:720 out720.mp4       # resize
```

Read / write tags with mutagen

```
python3 <<'PY'
from mutagen.easyid3 import EasyID3
from mutagen import File
print(File("/home/user/song.mp3").info.length)  # duration in seconds
a = EasyID3("/home/user/song.mp3")
a["title"] = "New Title"; a["artist"] = "Artist"; a.save()
PY
```

VLC command line (only when VLC itself must act)

```
cvlc --play-and-exit /home/user/clip.mp4  # headless play
cvlc video.mp4 --video-filter=scene --scene-path=/tmp \
    --vout=dummy vlc://quit             # snapshots
```

VLC config/state lives at `~/.config/vlc/vlcrc`; recent-media in the same dir.

Gotchas

- `-c copy` only works when not re-encoding; drop it if you change codec/resolution.
- Always pass `-y` in batch so ffmpeg overwrites without prompting.
- Match the requested container/codec/bitrate and the exact output path.

CLI Skill: VS Code

VS Code — CLI toolkit

Most VS Code tasks are really file/code tasks: create or edit source files, run them, change settings, or manage extensions. Do the file work directly in the shell/python — it is exact and fast — and use the `code` CLI for editor-level state.

Edit files directly (don't hunt through the editor UI)

```
cat > /home/user/proj/app.py <<'PY'
def add(a, b):
    return a + b
print(add(2, 3))
PY
python3 /home/user/proj/app.py  # run and check output
```

For surgical edits use python (read, replace, write) rather than retyping in the editor.

Settings, keybindings, snippets (JSON on disk)

```
~/.config/Code/User/settings.json  # e.g. {"editor.tabSize": 2}
~/.config/Code/User/keybindings.json
~/.config/Code/User/snippets/*.code-snippets
# workspace settings:
<project>/.vscode/settings.json
```

Edit these JSON files with python (`json.load` → mutate → `json.dump`) to guarantee valid JSON.

 # The code CLI

```
code --list-extensions          # what's installed
code --install-extension ms-python.python
code --uninstall-extension <id>
code -r /home/user/proj/app.py  # open a file in the running window
```

The on-disk file is the persistent state — edit on disk and stop; don't reopen/reload the editor just to make it visible (no GUI save needed).

 # Gotchas

- Extension installs need network; the VM egress proxy is already exported for bash.
- Keep JSON valid — a trailing comma breaks `settings.json` silently.
- Confirm the exact file path/name the task expects (case-sensitive).