

Mage-VL-Cowork: A Full Training Recipe to Learn Hybrid Computer Use From Evolved Harnesses

Microsoft Mage Team

Effective computer-use agents (CUAs) should seamlessly combine visual GUI interaction with code and terminal execution within a single policy. We present Mage-VL-Cowork, a 4B hybrid CUA trained from a general multimodally aligned checkpoint through an end-to-end, progressive five-stage curriculum across midtraining, SFT, and RL. To power this curriculum at scale, Mage-CUA-Forge replaces per-task agent construction with reusable, skill-conditioned generation workflows and application-specific preparers, synthesizing 293,806 desktop and web tasks across 6,915 skills while reducing recurring per-task expenditure by 82.6%. Rather than directly distilling a default teacher, we evolve the teacher harness and instantiate specialized CUA harnesses to yield high-quality trajectories alongside stage-specific supervision. Mage-VL-Cowork-4B finally climbs to 66.0% on OSWorld-Verified through different training stages with more efficient reasoning cost, achieving state-of-the-art performance at its parameter size and pushing forward the frontier of on-device intelligence.

Project Page: <https://microsoft.github.io/Mage>

Code: <https://github.com/microsoft/Mage>

Model: <https://huggingface.co/collections/microsoft/mage>

Date: October, 2026

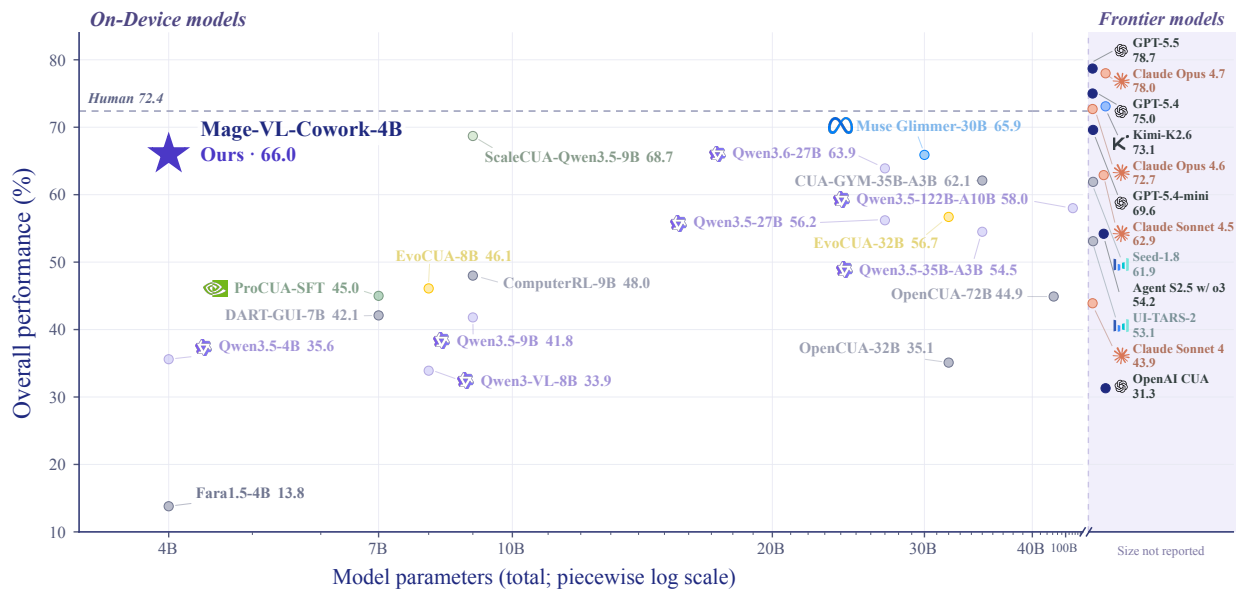


Figure 1 OSWorld-Verified overall score versus model size. Mage-VL-Cowork-4B is compared with compact and larger open-source agents; frontier models without reported parameter counts are shown in the right strip. The dashed line marks human performance (72.4%).

Contents

1	Introduction	3
2	Related Work	4
3	Synthesizeded Data Generation for Hybrid Computer Use	5
3.1	Evolution of the Hybrid Computer-Use Harness	5
3.2	Workflow-First Task and Environment Generation	6
3.3	Iterative Skill Taxonomy Expansion and Refinement	7
3.4	Dataset Statistics	8
4	Training Pipeline of Mage-VL-Cowork	8
4.1	Stage 1: GUI-Oriented Midtraining	9
4.2	Stage 2: GUI Agent SFT	10
4.3	Stage 3: Coding & Terminal SFT	10
4.4	Stage 4: Hybrid CUA SFT	10
4.5	Stage 5: Online Hybrid RL	10
5	Reinforcement Learning Infrastructure	11
5.1	Workload Characteristics and Design Requirements	11
5.2	System Architecture and Asynchronous Execution	11
6	Experiments	12
6.1	Benchmarks and Evaluation Setup	12
6.2	Main Results on OSWorld-Verified	13
6.3	Task-Generation Expenditure	13
6.4	Training Progression	15
6.5	Execution Efficiency and Token Usage	15
6.6	Browser Task Performance	16
6.7	Effect of the Evolved Harness on Student Performance	17
7	Conclusion	17
A	Web Replica Environments and Data Counting Definitions	19
A.1	High-Fidelity Web Replicas	19
A.2	Web Task Synthesis and Verifiable Contracts	20
A.3	Dataset Counting Definitions	21
B	Midtraining Explorations, Context Design, and Compute Accounting	21
B.1	GUI Midtraining Design Trade-Offs	21
B.2	Single Step Cowork Bench	22
B.3	Context Window Design and Step-Level Reasoning Synthesis	23
B.4	Dataset Volumes and Compute Accounting	23
C	RL Infrastructure Details and Measurements	24
C.1	Rollout-as-a-Service Architecture	24
C.2	Resource Management and Sandbox Execution	24
C.3	Asynchronous Rollouts and Trajectory Preparation	25
C.4	Deployment Scale and Runtime Profile	25
C.5	Workload and Transport Accounting	26
C.6	Policy-Lag Aggregation	26

1 Introduction

Computer-use agents (CUAs) for productivity perceive graphical desktops through screenshots and act through keyboard and mouse commands. This GUI channel is universal, but it is a modeling choice rather than a property of the domain, and an expensive one: a typical screenshot can require thousands of tokens, increasing the cost of processing high-resolution visual input and grounding interface elements. A person fluent with a computer does not confine themselves to the mouse: they open a terminal when a terminal is faster and read a file to confirm what a dialog claims to have saved. Restricting the policy to pixels forces it to re-derive through the GUI what a single command would accomplish, and removes the most reliable channel it has for checking its own work. We therefore train a *hybrid* CUA that acts through two action channels: the GUI channel, with visual observation and coordinate control, and the terminal channel, a persistent shell environment in which it may run code, invoke command-line utilities, and call application libraries directly.

Hybrid access does not by itself produce hybrid competence, even for frontier teachers. Prior work shows that access to application tools does not ensure their effective use, and that the same tool set can affect different models differently [1, 2]. The policy must learn which channel is appropriate for the current operation and at what level of abstraction to act. Effective hybrid execution routes each operation to the channel where it fits best: coding and terminal execution for structured files (such as updating a spreadsheet with one library call rather than dozens of clicks), and GUI interaction for visual or live application state (such as browser settings that a running browser caches in memory and overwrites on exit). Because even frontier models exhibit suboptimal channel routing under a naive setup, distilling their default rollouts passes those same limits on to the student. The harness is not a neutral container: its tool interfaces, context handling, and guidance directly shape how a fixed teacher model acts. Evolving the harness before large-scale rollout elicits stronger hybrid execution than the teacher achieves by default and carries those improvements across the entire training corpus.

Moreover, whereas existing pipelines such as ProCUA and ScaleCUA [3, 4] post-train models that already have strong grounding, coding, and agent capabilities, we present an end-to-end, progressive five-stage curriculum starting from a general multimodally aligned checkpoint: GUI-oriented midtraining, GUI agent SFT, coding and terminal SFT, hybrid CUA SFT, and online hybrid RL. Sustaining such a multi-stage curriculum requires massive, diverse training environments and trajectories, for which we propose the Mage-CUA-Forge environment synthesis pipeline. Unlike prior agent-first approaches that rely on open-ended agent execution to construct each task from scratch, Mage-CUA-Forge uses an authoring agent to construct application-specific preparers and reusable generation workflows tailored to each application and its atomic functionalities, scaling task and environment synthesis with high consistency and low recurring cost.

We introduce Mage-VL-Cowork, a hybrid CUA trained on 318,837 teacher trajectories over 293,806 synthesized tasks and environments that cover 6,915 atomic skills in an agent-constructed skill taxonomy. Three design decisions define the pipeline:

1. **Learning from an evolved hybrid CUA harness.** To collect high-quality demonstrations even when a frontier teacher’s default channel routing is imperfect, we evolve the harness including its scaffold, tool interfaces, prompts, and terminal execution environment, before large-scale rollout. Evolution here is as much pruning as growth: unlike skill libraries that continuously accumulate tools [5–7], we remove overlapping or ill-suited tools so the teacher produces clean hybrid trajectories that yield much larger gains when distilled into the student (§3.1, §6.7). Moreover, harness evolution produces specialized configurations with different tool sets—including a GUI harness, a terminal harness, and the hybrid CUA harness—allowing us to roll out the same synthesized tasks under multiple harnesses to collect trajectories emphasizing distinct capabilities across our training stages (§4).

2. **A single policy that learns the hybrid execution policy.** Mage-VL-Cowork observes full-desktop screenshots and emits coordinate mouse and keyboard actions, and in the same episode may execute code and terminal commands on the machine it is operating. The two channels are complementary: pixels are the only channel that reaches spatial, canvas, and rendering-dependent state, while code edits at the abstraction the instruction is written in and lets the agent check its work against ground truth rather than against a screenshot of a dialog. Unlike systems that route between a GUI operator and a coding operator at the orchestrator level [8, 9], the choice is internal to the policy and is therefore something the training data must teach (§3.1).
3. **Workflow-first task and environment synthesis.** Agent effort is spent once per *skill* rather than once per task. For each atomic skill, an authoring agent constructs and debugs a reusable generation workflow that combines LLM nodes with application-specific *preparers*, which materialize input artifacts such as a Calc workbook or a Thunderbird profile with a populated mail store. Combined workflows produce a task package of instruction, environment setup script, and rubric, and gate nodes check its consistency before the task is accepted. Shared workflows and preparers propagate each improvement across all subsequently generated task instances. Task synthesis, including skill-based generation, has also been studied in terminal settings, where the environment is a container filesystem [10, 11]; on a full desktop the environment is application state, which is what the preparer exists to produce (§3.2).

With 4B parameters, Mage-VL-Cowork achieves 66.0% on OSWorld-Verified under a 50-step budget, outperforming larger open-source agents including Muse Glimmer-30B (65.9%), Qwen3.6-27B (63.9%), and EvoCUA-32B (56.7%). Across our curriculum, performance climbs by 20.44 percentage points from GUI SFT (45.56%) through Hybrid SFT (55.84%) to final RL (66.0%), pushing forward practical on-device intelligence for autonomous desktop workflows.

2 Related Work

Computer-use agents and evaluation. GUI grounding methods such as SeeClick [12] learn to locate interface elements from language instructions. OSWorld [13] and WebVoyager [14] evaluate the additional requirements of multi-step interaction with desktop and browser environments. OpenCUA [15] provides open training foundations for computer-use agents. Hybrid approaches extend visual interaction with coding and terminal actions: CoAct-1 [8] uses an orchestrator to coordinate GUI and coding agents, whereas our student selects GUI and terminal actions within a single policy. Tool availability alone does not determine effective channel selection, as also examined in OSWorld-MCP and controlled screenshot/tool comparisons [1, 2].

Synthesized Environment. OS-Genesis [16] constructs GUI trajectories through reverse task synthesis, while CUA-Gym [17] co-generates task instructions, environment states, and reward functions through an adversarial generator–discriminator loop, and RecreationWorld [18] builds verifiable multi-platform recreation environments for hybrid computer-use agents. For browser environments, InfiniteWeb [19] synthesizes functional websites from specifications, and WebHarbor [20] reconstructs self-hosted website replicas for controllable evaluation and training. Rather than repeating an open-ended agent construction loop for every task, Mage-CUA-Forge uses an authoring agent to build reusable application-specific preparers and skill-conditioned workflows that construct files, application state, and deterministic web replicas at scale.

CUA Posttraining. ProCUA [3] and ScaleCUA [4] study scalable computer-use post-training, but start from checkpoints that already possess strong GUI grounding, coding, and agentic capabilities. MolmoWeb [21] also documents an end-to-end training pipeline starting from a multimodally

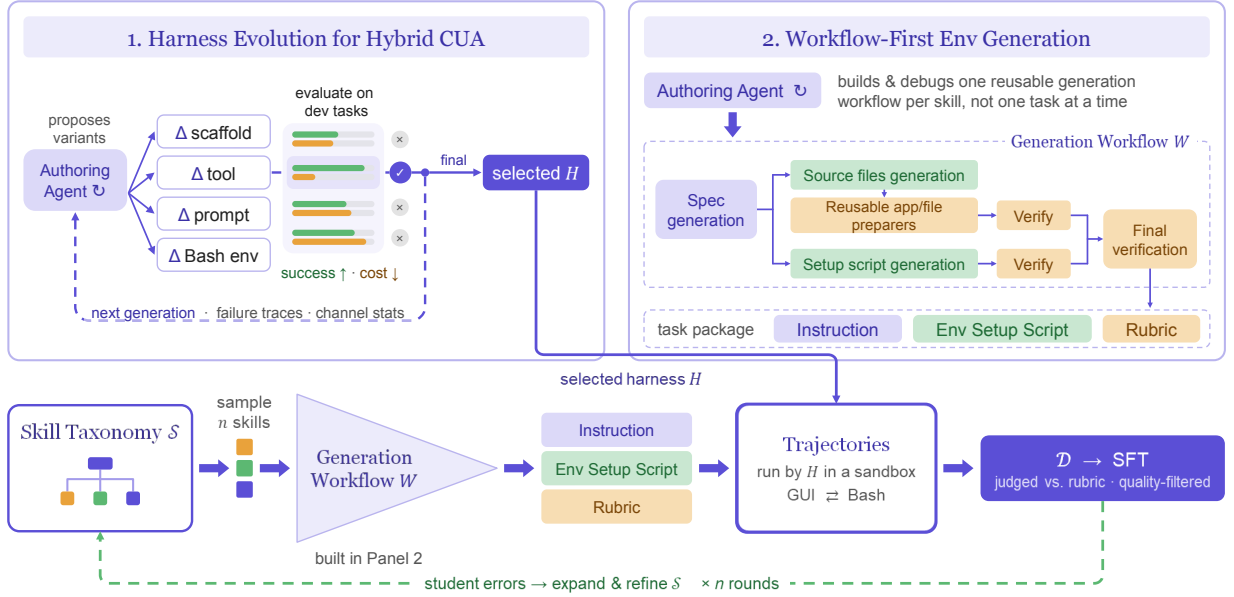


Figure 2 Schematic overview of the data generation pipeline. (1) An authoring agent evolves the hybrid computer-use harness by proposing scaffold, tool-interface, prompt, and terminal-environment variants and selecting the harness H with the best success-cost trade-off on development tasks. (2) An authoring agent builds one reusable generation workflow W per skill. A shared specification coordinates source-file and setup-script branches, whose preparers and gates produce a task package of instruction, environment setup script, and rubric. Bottom: skills sampled from the taxonomy $\mathcal{S}$ are instantiated through W , the teacher agent with H executes the tasks in a sandbox through GUI and terminal actions, and rubric-judged, quality-filtered trajectories form $\mathcal{D}$ for SFT. Student performance feedback guides adaptive expansion and refinement of $\mathcal{S}$ across generation rounds.

aligned checkpoint, yet its scope is restricted to browser-only web agents and lacks coverage of general desktop applications and hybrid GUI-terminal execution. We instead present a progressive five-stage training curriculum starting from a general multimodally aligned checkpoint across full desktop and browser workflows.

3 Synthesized Data Generation for Hybrid Computer Use

We construct supervised fine-tuning data through a pipeline that combines a skill taxonomy $\mathcal{S}$, reusable generation workflows W , and an evolved hybrid computer-use harness H . Each skill describes an atomic capability in a specific application. We sample several skills to form a composite capability and combine their workflows to generate a task instruction, an environment setup script, and an evaluation rubric. Executing the setup script prepares a sandbox with the preconditions needed to attempt the task. A teacher agent, whose harness is optimized before large-scale trajectory collection, then executes the task. Rubric-based LLM judging and trajectory-quality filtering produce the retained trajectory dataset $\mathcal{D}$, which is used to train the student model M . Student performance feedback guides adaptive expansion and refinement of the skill taxonomy across generation rounds. Figure 2 gives an overview of the pipeline. We first describe teacher-agent evolution, then workflow-first generation, and finally the taxonomy feedback loop.

3.1 Evolution of the Hybrid Computer-Use Harness

Collecting high-quality hybrid trajectories requires more than a strong frontier model: the teacher must also select, sequence, and combine GUI actions and terminal commands effectively, a strategy we call the *hybrid execution policy*. Under a standard setup, even frontier models show suboptimal

channel habits—such as clicking through repetitive file edits or modifying files on disk that a running application caches in memory. Distilling directly from such rollouts passes the teacher’s default limitations on to the student.

We improve the teacher’s hybrid execution by evolving its external harness while keeping the underlying model weights fixed. We write the teacher agent as

$$A_T = (M_T, H), \quad H = (F, \mathcal{T}, P, \mathcal{E}), \quad (1)$$

where M_T is the teacher model and the harness H consists of four components. The scaffold F organizes the execution loop and context handling. The tool interface $\mathcal{T}$ exposes the actions of each action channel—GUI mouse and keyboard actions and terminal commands—together with their argument schemas. The prompt P , i.e., the system prompt, guides tool use and the hybrid execution policy. The terminal execution environment $\mathcal{E}$ comprises the command-line utilities and application libraries preinstalled in the Docker sandbox and accessible through the terminal channel. Operational deployments require explicit authorization and sandbox isolation for terminal execution.

Harness evolution optimizes F , $\mathcal{T}$, P , and $\mathcal{E}$ on a shared development set. An authoring agent uses rubric-based task success, per-task token cost, execution feedback, and application-level channel statistics to select revisions that improve the success–cost trade-off.

Harness evolution includes removing redundant tools as well as adding useful ones. Removing `xdotool` in favor of native screenshot-grounded GUI actions avoids duplicate mouse/keyboard routes and fragile window-focus handling that would split supervision across unequal choices. Conversely, providing domain libraries such as `python-pptx` in the terminal environment lets the teacher operate directly on slides, shapes, and text runs rather than manually editing unpacked OOXML files.

The GUI-and-coding variant achieves higher task success with fewer execution steps than its AX-augmented counterpart; as shown in Section 6.7, this advantage widens into a 12.5-percentage-point gain when distilled into the 4B student. This comparison guides joint optimization of the tool interface and prompt for effective hybrid execution. Figure 3 presents selected teacher–harness configurations across channel, scaffold, teacher-model, and prompt refinements.

Once selected, the harness remains essentially fixed across subsequent data-generation rounds. We use `gpt-5.6-terra` as the rubric judge to verify task completion, while a trajectory-quality filter retains demonstrations with clean channel routing and tool use. Optimizing the harness before large-scale rollout ensures that $\mathcal{D}$ captures strong hybrid execution for training M (Section 4).

3.2 Workflow-First Task and Environment Generation

A computer-use task package combines an instruction, an executable environment setup, and an evaluation rubric. Workflow-first generation reuses the construction procedure for these three coordinated components across task instances.

Mage-CUA-Forge builds a reusable generation workflow for each atomic skill. Each workflow combines LLM nodes, file preparers, and setup operations. For a composite capability, the workflows share a unified specification of the requested outcome, required files, and application preconditions, producing a coordinated task package.

The source-file and setup-script branches share the task specification. Branch-specific checks feed a final consistency gate over the complete task package.

The workflow routes each required file to a preparer appropriate for its type, reusing application-specific preparation logic across skill workflows. These preparers materialize the input artifacts needed by the task. Other workflow nodes produce the remaining setup code and combine it with shared setup utilities, such as application launching and desktop-state adjustment. The result is an *environment setup script* (*env setup script*), whose purpose is to establish the required task

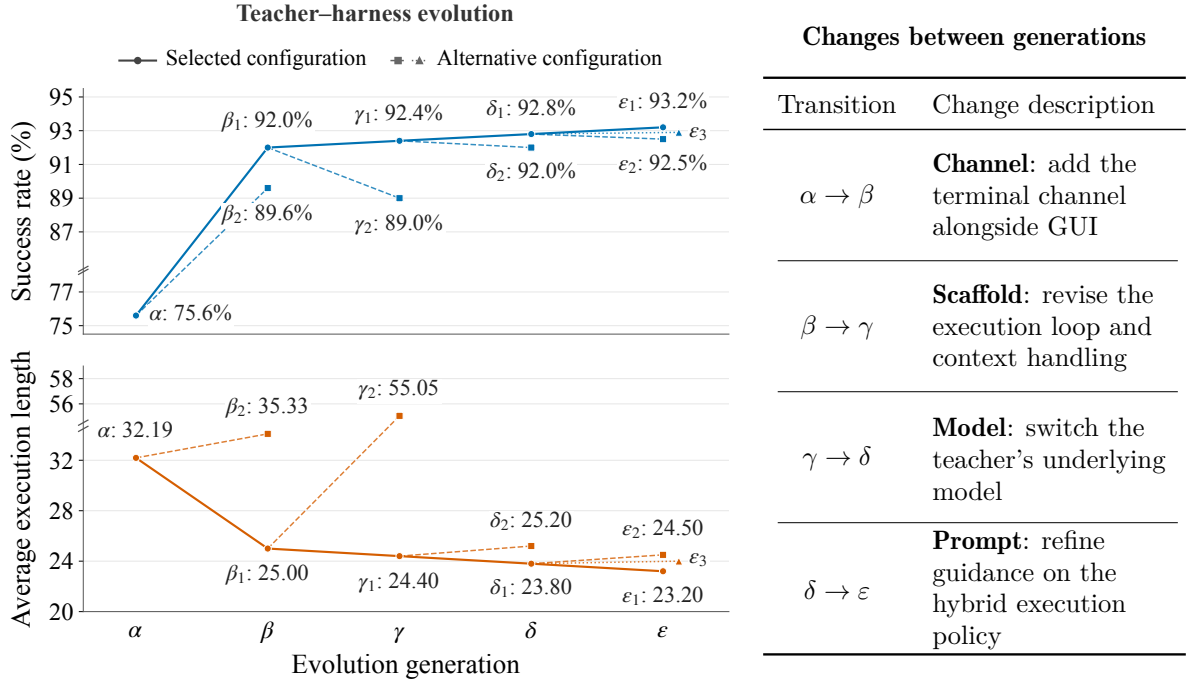


Figure 3 Teacher-harness evolution across channel, scaffold, model, and prompt refinements. Left: task success and execution length for selected configurations. Right: the refinement dimension at each transition. Execution length is measured in model turns for α and β_2 , and in tool calls for γ_2 . Channel, scaffold, and prompt refinements modify H ; model transitions update M_T .

preconditions and initial desktop state inside the sandbox. Workflow-specific generation logic is thus separated from reusable file and application preparation, as well as from generic setup operations that are not tied to an individual skill.

Dedicated LLM nodes generate the user instruction and its evaluation rubric from the shared specification. Task completion is judged by an LLM against this rubric. Application setup details remain in the setup script, while the instruction describes the user’s intended outcome.

Validation operates at two levels. File-level gates check artifacts against the shared specification, and a package-level gate checks consistency among the instruction, setup requirements, and rubric. Gate feedback improves the reusable workflow and shared preparers. Subsequent trajectory evaluation assesses the teacher’s execution of accepted tasks.

Workflow reuse separates one-time construction from recurring generation. LLM nodes supply task-specific content within an established procedure, while shared preparers provide reusable file and application setup logic across skills and task instances.

For browser and cross-application workflows, such as retrieving structured records from a website and compiling them into a spreadsheet or document, live websites introduce anti-bot blocks, content drift, and non-resettable backend state. We therefore pair desktop application preparers with self-hosted, deterministic web environments combining synthesized InfiniteWeb [19] sites and high-fidelity website replicas built upon and extending WebHarbor [20], where each replica resets from a local seed database per episode to support verifiable web and cross-application task generation (Appendix A.1).

3.3 Iterative Skill Taxonomy Expansion and Refinement

An exploration agent builds the initial skill taxonomy $\mathcal{S}$ by traversing desktop applications and recording their atomic skills. Rather than keeping $\mathcal{S}$ fixed, we refine and expand it across generation rounds using feedback from student evaluations, adjusting skill granularity, enriching descriptions, and adding application-specific variations where needed.

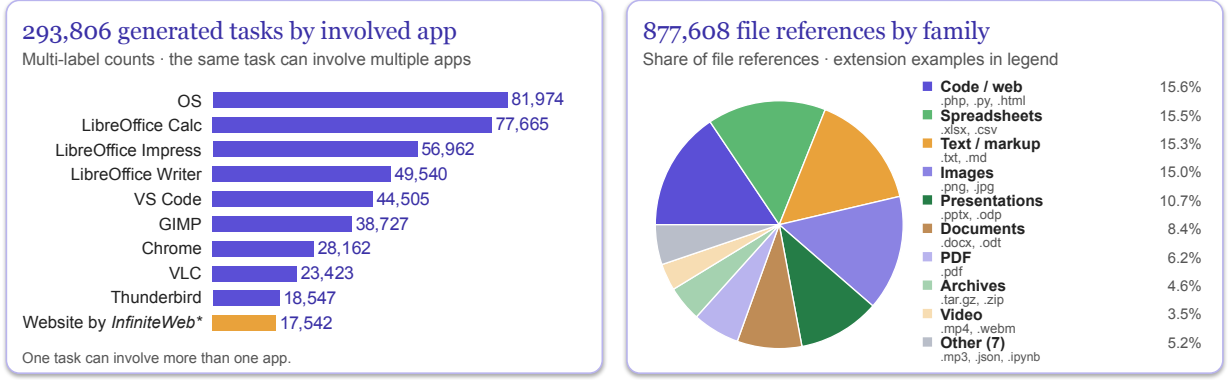


Figure 4 Application coverage and task-file reference families. Application counts measure multi-label application-task incidences. The 17,542 InfiniteWeb [19] tasks are reported separately. File-family shares are computed over 877,608 references.

The first generation round balances task coverage across atomic skills. Composite capabilities instantiate these skills through their workflows; teacher collection and filtering produce the supervision used to train M .

Student evaluation identifies capability areas that need denser coverage in the next round. We add or refine the corresponding skill entries in $\mathcal{S}$, update their workflows, and generate additional tasks and teacher trajectories. Keeping the evolved harness fixed while expanding $\mathcal{S}$ across three rounds cleanly separates improving *how the teacher executes tasks* from improving *which tasks the pipeline supplies*.

3.4 Dataset Statistics

Mage-CUA-Forge produces 293,806 distinct tasks covering 6,915 atomic skills. Single-application and multi-application tasks account for 61.76% and 38.24%, respectively. Teacher execution and quality filtering yield 318,837 retained trajectories.

The task inventory contains 877,608 task-file references, counted as distinct extracted paths or filenames within each task. Selected reference families include coding (136,736), spreadsheets (135,805), text or Markdown (134,050), documents (74,052), PDFs (53,996), and presentations (93,784). File references occur in 266,030 tasks (90.5%), averaging 2.99 references per task. The inventory spans 278 suffix categories: 277 named suffixes and one no-extension category. Figure 4 summarizes application involvement and reference families; Appendix A defines the counting rules.

These synthesized tasks are rolled out under different harness configurations, including the GUI harness, the terminal harness, and the evolved hybrid CUA harness, to collect trajectory data that solve the tasks through different channels and emphasize distinct capabilities for the subsequent training stages described in Section 4. Moreover, we also synthesize the reasoning content before tool call with a multi-stage annotation by GPT-5.4-mini. Within the retained hybrid CUA dataset $\mathcal{D}$, supervision covers both execution channels: GUI and terminal actions account for 68.19% and 31.81% of teacher tool calls, respectively.

4 Training Pipeline of Mage-VL-Cowork

Unlike pipelines that post-train checkpoints already specialized for GUI grounding or coding, we present an end-to-end, progressive five-stage training curriculum starting from an intermediate 4B Mage-VL checkpoint: (1) GUI-Oriented Midtraining, (2) GUI Agent SFT, (3) Coding & Terminal SFT, (4) Hybrid CUA SFT, and (5) Online Hybrid RL. This specialize-then-unify curriculum first builds solid, standalone capabilities in visual grounding, GUI interaction, and coding and terminal execution before teaching the model to route and combine both channels within a single episode.

Context construction and training units. Context folding converts trajectories into turn-level examples, with supervision applied to the next assistant response. Each context retains the three most recent screenshots and four complete responses, while compressing earlier steps into imperative **Action:** summaries immediately after the system prompt. Each assistant response pairs a grounded `<think>` rationale, a concise **Action:** summary, and the executed `<tool_call>`; together with the folded **Action:** history once older screenshots slide out of the window, these structured reasoning, summary, and tool-call traces act as a compact working memory, enabling the policy to track prior intent and completed operations across long-horizon, interdependent multi-step tasks (Appendix B.3).

Midtraining and SFT use two epochs over each dataset, packed into sequences with a capacity of 32,768 tokens. Table 1 summarizes the per-stage compute shares, trajectory counts, and token volumes; Appendix B gives the data and compute-estimation details.

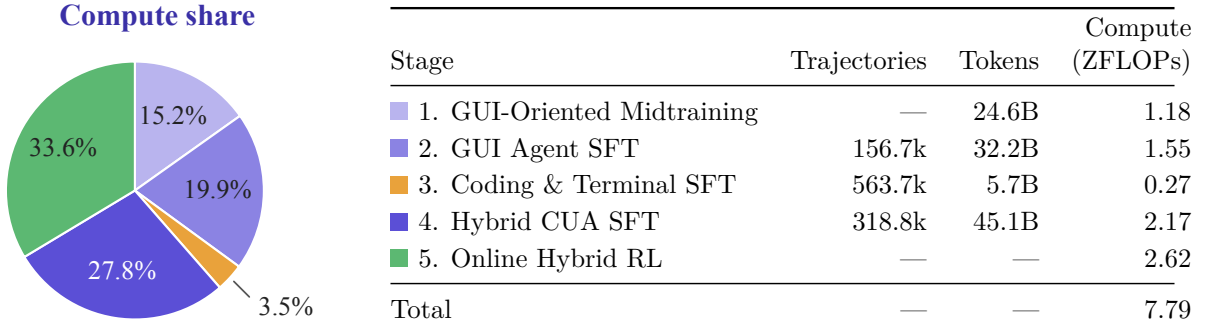


Table 1 Compute shares, training data, and nominal compute across the five stages. The pie chart (left) shows stage-wise compute shares, with colors corresponding to the stage rows. Tokens are reported in billions per epoch ($B = 10^9$ / epoch) of packed capacity, including padding and image tokens; midtraining and SFT use two epochs (ZFLOPs = 10^{21} FLOPs). Stage 1 packs multi-source examples and Stage 5 uses online RL rollouts, so their fixed trajectory or token entries are shown as dashes. Appendix B details the compute-estimation scope.

4.1 Stage 1: GUI-Oriented Midtraining

We initialize training from an intermediate 4B Mage-VL checkpoint. Evaluating this starting checkpoint on the core atomic capabilities required for agentic computer use—visual grounding, chain-of-thought reasoning, and structured tool calling—reveals that all three remain underdeveloped: the model exhibits an *answer-machine* tendency [22], defaulting to terse direct replies rather than reasoning before acting or emitting schema-compliant tool calls. Stage 1 therefore introduces GUI-oriented midtraining to establish these three atomic capabilities before multi-step trajectory post-training.

Targeting desktop and browser computer use, we curate an OS and web grounding corpus from open-source GUI grounding datasets including UI-E2I-Synth and others [12, 21, 23–26], formatting all spatial annotations as `computer_use click` tool calls and packing same-screenshot queries into shared sequences with target-level deduplication [27]. To jointly acquire screen localization, structured function calling, and chain-of-thought reasoning without mutual interference, we combine this grounding corpus with tool-calling and GUI trajectory data during Stage 1.

To monitor whether midtraining checkpoints have acquired these atomic capabilities without waiting for downstream post-training and multi-step rollouts, we construct Single Step Cowork Bench directly from real downstream desktop trajectories, using the target agent action space and system prompt to provide fast diagnostic feedback. Appendix B.1 details our midtraining data design and the two complementary recipes we explored for reconciling non-reasoning grounding

supervision with reasoning-annotated tool calls, and Appendix B.2 describes Single Step Cowork Bench.

4.2 Stage 2: GUI Agent SFT

The second stage turns the visual and tool-format foundation into a GUI execution policy. We collect teacher trajectories with the GUI-only harness. In this setting, task actions are performed through the GUI channel, and the training examples pair the available screenshot and interaction history with the teacher’s next action. We use these trajectories to fine-tune the midtrained checkpoint on complete computer-use workflows.

GUI trajectories connect single-step grounding to multi-step interaction, teaching the model which control to operate next, when to scroll, how to enter text, and how to continue after the interface changes. Building strong standalone GUI execution first ensures that when the model later learns hybrid channel routing, it chooses between two capable channels rather than routing around a weak visual policy.

4.3 Stage 3: Coding & Terminal SFT

The third stage develops coding and terminal execution capabilities, including terminal operation, code generation, repository navigation, and debugging with Terminal-Bench-style [28] and SWE-bench-style training data [29].

We combine general software-engineering data with desktop terminal harness trajectories from productivity tasks we synthesized in Section 3. While repository-level tasks cover source files, tests, and build tools, desktop coding trajectories teach the model to read and modify workbooks, documents, presentations, and application state using specialized Python libraries and command-line utilities.

Specifically, we run a terminal-only teacher harness on the same batch of desktop tasks used for GUI trajectory collection in Stage 2. Retaining the successful coding and terminal trajectories alongside general terminal and software-engineering data gives the student direct examples of how to solve target desktop tasks through code before Stage 4 combines both channels.

4.4 Stage 4: Hybrid CUA SFT

Stage 4 brings the visual and coding foundations together by training on the 318,837 trajectories collected under the evolved hybrid harness (Section 3.1). Within a shared task context, a single policy learns turn-level channel routing—deciding when to act through the GUI, when to use coding and terminal execution, and when to switch between them, such as using code to edit a structured file and then returning to the application window to inspect the rendered result.

Because a 4B model is sensitive to noisy or ambiguous demonstrations, optimizing the teacher harness for both task success and token efficiency and filtering trajectories before training, which is critical. Rather than merely exposing the student to a larger tool inventory, the resulting supervision directly teaches effective channel routing and compact execution.

The final supervised stage draws on the 318,837 retained hybrid trajectories; the final hybrid dataset contains 1,377,805 packed bins. The resulting model combines the visual grounding and GUI execution learned in the first two stages, the general and desktop-specific coding capabilities developed in the third, and the hybrid execution policy demonstrated by the evolved hybrid agent. We refer to the checkpoint obtained after this stage as Mage-VL-Cowork-SFT.

4.5 Stage 5: Online Hybrid RL

The final stage refines Mage-VL-Cowork-SFT with online reinforcement learning in the hybrid harness. We train on the 1,988 tasks released by ScaleCUA [4], each paired with a verifiable reward. Following ScaleCUA, we use frontier sampling, with task weights based on pre-run success rates to allocate rollouts toward the learning frontier. The policy is optimized with GRPO [30]. Rollouts

are generated asynchronously with bounded policy staleness, and we apply vanilla truncated importance sampling (TIS) [31] to correct the mismatch between the rollout and training policies. Each rollout uses the same step budget as our OSWorld-Verified evaluation. The resulting model is Mage-VL-Cowork, which we evaluate in Section 6.

Each optimizer update admits 96 task groups with eight episode slots per group. A four-batch admission window bounds policy lag at three updates, and TIS caps importance weights at 2.0. Mixed-policy episodes are enabled. CPU workers execute desktop environments, while GPU services handle training and inference. The relay carries control, inference, and compact trajectories; weight synchronization uses an intra-GPU-cluster path. Section 5 summarizes the architecture; Appendix C gives the implementation details, runtime profile, and measurement definitions.

5 Reinforcement Learning Infrastructure

Our Rollout-as-a-Service (RaaS) [32] infrastructure decouples desktop execution, model inference, and policy optimization, integrating PoLar [32], Slime, Azure Relay, and Azure Virtual Machine Scale Sets (VMSS). CPU workers run environments and agent loops, an inference service generates responses, and the training client consumes completed trajectories. This separation supports asynchronous learning across long-running desktop tasks; implementation details and runtime measurements are given in Appendix C.

5.1 Workload Characteristics and Design Requirements

Desktop rollouts combine inference with application loading, content entry, and state updates. Individual observed episodes exceed 40 minutes, motivating asynchronous scheduling so that completed tasks can advance training independently of long-running episodes.

The resource requirements are also heterogeneous. Desktop environments need CPU capacity, guest memory, and nested virtualization, whereas model inference and optimization run on GPUs. For a compact model on B200 hardware, environment execution and orchestration can remain important latency sources even when individual model calls are relatively fast. We consequently manage environment capacity separately from model-serving and training capacity, and use asynchronous rollouts rather than requiring lockstep progress across the desktop fleet.

Visual rollouts create repeated screenshot inputs across turn-level examples. Compact episode transfer, prefetching, and node-local image expansion address transport, preparation, and training-input materialization as separate stages of the data path.

5.2 System Architecture and Asynchronous Execution

Figure 5 separates training, inference, and environment execution. Slime and rollout adapters form the training client; SGLang [33] serves model requests; and RaaS workers on Azure VMSS host the resource manager and CUA runners. The resource manager dispatches tasks, schedules sandboxes, and stores completed compact episodes. Each runner coordinates environment interaction, the stateful harness, evaluation, and trajectory construction. Control messages, inference traffic, and bulk trajectory transfer use separate Azure Relay channels; weight synchronization remains inside the GPU cluster.

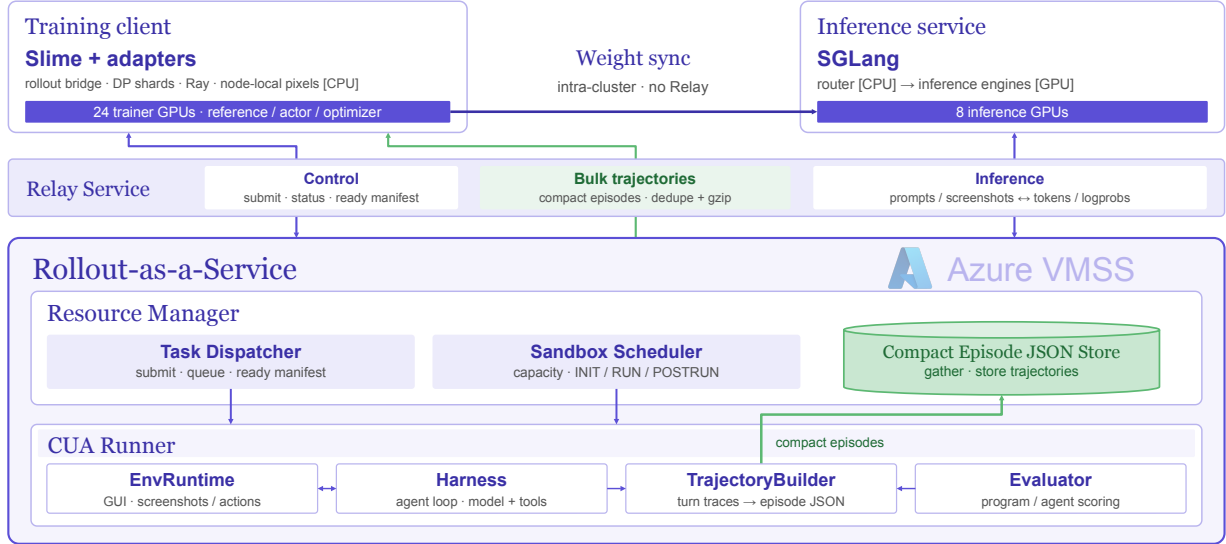


Figure 5 Rollout-as-a-Service architecture for asynchronous desktop RL. The resource manager coordinates task dispatch, sandbox scheduling, and compact episode storage. Runners handle environment interaction, harness execution, trajectory construction, and evaluation. The relay separates control, inference, and trajectory traffic; model weights synchronize within the GPU cluster. The diagram shows a 24-training/8-inference-GPU deployment.

Rollouts and optimization proceed asynchronously under the four-batch admission window and three-update lag bound described in Section 4.5. Compact, deduplicated trajectories are transferred before per-turn sample construction and node-local image expansion. Prefetching overlaps data preparation with training, while lifecycle-specific admission limits manage sandbox initialization, execution, and post-run processing independently.

The reference deployment provisions 1,120 GUI slots across 70 CPU hosts and 32 B200 GPUs, allocating 24 GPUs to training and eight to inference. Appendix C details deployment configurations and runtime measurements.

6 Experiments

Our experiments evaluate four properties of Mage-VL-Cowork: desktop and browser task performance at the 4B model scale, recurring task-generation expenditure, performance–efficiency gains across training stages, and the relationship between teacher-harness quality and student outcomes.

6.1 Benchmarks and Evaluation Setup

We evaluate on held-out task sets.

WebVoyager-Static. Evaluating browser agents on live websites [14] is confounded by both environment instability and unreliable grading: page content varies across time and location, anti-bot checks and access blocks interrupt rollouts, VLM judges lack backend ground truth and often credit unverified claims absent from the final screenshot, and newer models already know much of the target information and tend to guess answers without browsing. To construct a high-fidelity offline benchmark, we build on WebHarbor [20] and substantially expand its self-hosted replicas, reconstructing the missing pages, interactive controls, and local database records required to faithfully support all 643 tasks across the 15 WebVoyager [14] sites. Rather than relying on a final-screenshot VLM judge alone, grading verifies both the execution trace, by confirming that the agent visited the target pages and performed the required UI operations, and the final outcome, by checking the agent’s answer and backend database state against ground truth. This deterministic setup eliminates the score inflation common on live web benchmarks (Table 4) while running at a

fraction of the cost of full desktop VM rollouts, making WebVoyager-Static an effective multi-step diagnostic for validating our web data and iterating on early post-training checkpoints.

OSWorld-Verified. OSWorld-Verified evaluates end-to-end desktop task completion across documents, spreadsheets, presentations, browsers, image editing, and operating-system operations. We evaluate the final RL checkpoint on 361 held-out tasks with a 50-step budget, excluding Google Drive. The reported result averages two evaluation runs. For each run, the overall score is $100N^{-1} \sum_{i=1}^N s_i$, where $N = 361$ and s_i is the benchmark score awarded to task i . Table 2 reports the overall scores and available category breakdowns.

6.2 Main Results on OSWorld-Verified

Mage-VL-Cowork achieves 66.0% on OSWorld-Verified with a 4B policy and a 50-step budget. Table 2 places this result alongside models across a broad range of parameter scales: our 4B policy outperforms open-source agents up to 7.5–18 $\times$ larger, including Muse Glimmer-30B (65.9%), Qwen3.6-27B (63.9%), EvoCUA-32B (56.7%), and OpenCUA-72B (44.9%). At the 4B scale, Mage-VL-Cowork exceeds Fara1.5-4B (13.9%) by 52.1 percentage points overall, while achieving 87.5% on OS and 62.7% on multi-application Workflow tasks, which is the highest OS score in the table and 14.6 percentage points above ScaleCUA-Qwen3.5-9B (48.1%) on Workflow. This capability-to-size efficiency is the central outcome of our evolved-harness supervision and progressive five-stage training curriculum. Baseline step budgets and all reported scores are listed in the table.

6.3 Task-Generation Expenditure

Workflow-first generation reduces recurring per-task expenditure by 82.6%, from \$0.906 to \$0.158, and token consumption by 93.3%, from 1,276K to 85K tokens (Table 3). Both approaches use `gpt-5.6-terra`. Reusing generation workflows and application preparers shifts repeated construction work into reusable artifacts.

Model	Steps	OS	Office	Daily	Professional	Workflow	Overall
Human	—	—	—	—	—	—	72.4
<i>Frontier models</i>							
GPT-5.5	—	—	—	—	—	—	78.7
Claude Opus 4.7	—	—	—	—	—	—	<u>78.0</u>
GPT-5.4	—	—	—	—	—	—	75.0
Kimi-K2.6	—	—	—	—	—	—	73.1
GPT-5.4-mini [†]	50	79.2	68.0	76.6	69.4	63.3	69.6
Claude Sonnet 4.5	100	70.8	72.6	61.4	63.2	49.5	62.9
Seed-1.8	100	66.7	68.8	67.1	71.4	42.4	61.9
Agent S2.5 w/ o3	50	75.0	53.8	54.0	75.5	39.5	54.2
UI-TARS-2	100	41.7	61.1	62.1	61.2	34.1	53.1
Claude Sonnet 4	50	45.8	43.5	55.1	55.1	28.5	43.9
OpenAI CUA	50	70.8	23.0	37.2	51.0	15.8	31.3
<i>Models with reported parameter counts</i>							
ScaleCUA-Qwen3.5-9B	50	79.2	70.6	74.3	89.5	48.1	68.7
Muse Glimmer-30B	—	—	—	—	—	—	65.9
Qwen3.6-27B	—	—	—	—	—	—	63.9
EvoCUA-32B	—	—	—	—	—	—	56.7
Qwen3.5-27B	—	—	—	—	—	—	56.2
ComputerRL-9B	50	75.0	45.8	52.5	71.4	27.9	48.0
ProCUA-SFT	—	—	—	—	—	—	45.0
OpenCUA-72B	50	45.8	42.7	53.8	75.5	22.3	44.9
DART-GUI-7B	30	62.5	39.3	50.8	73.5	16.7	42.1
Qwen3.5-9B	—	—	—	—	—	—	41.8
OpenCUA-32B	50	60.9	29.9	41.0	65.3	14.6	35.1
Fara1.5-4B	50	25.0	7.6	29.5	12.2	6.6	13.9
<i>Mage-VL-Cowork-4B</i>	50	87.5	64.9	65.0	66.3	62.7	<u>66.0</u>

Table 2 OSWorld-Verified results across model scales. Mage-VL-Cowork achieves 66.0% with 4B parameters under a 50-step budget. Baseline scores and step budgets are shown as reported. [†]Evaluated by us in one run on 361 tasks, excluding Google Drive. Blue shading marks the two highest overall scores in each model group.

Env Synth. Method	Tokens per task	Expenditure per task (USD)
Agent-first	1,276K	0.906
Mage-CUA-Forge	85K	0.158

Table 3 Recurring task-generation expenditure per task. Mage-CUA-Forge reuses workflows and application preparers; one-time workflow authoring and debugging are excluded from the per-task comparison.

6.4 Training Progression

During Stage 1 GUI-oriented midtraining, screen grounding accuracy rises steadily as the policy acquires visual localization prior to multi-step trajectory training (Figure 6): from 30.6% to 56.9% on OSWorld-G [26], surpassing Qwen3.5-4B (54.5%), and from 78.8% to 87.2% on ScreenSpot-v2 [25], exceeding Qwen3-VL-4B (85.3%). Design trade-offs for Stage 1 midtraining and single-step diagnostics are detailed in Appendices B.1 and B.2.

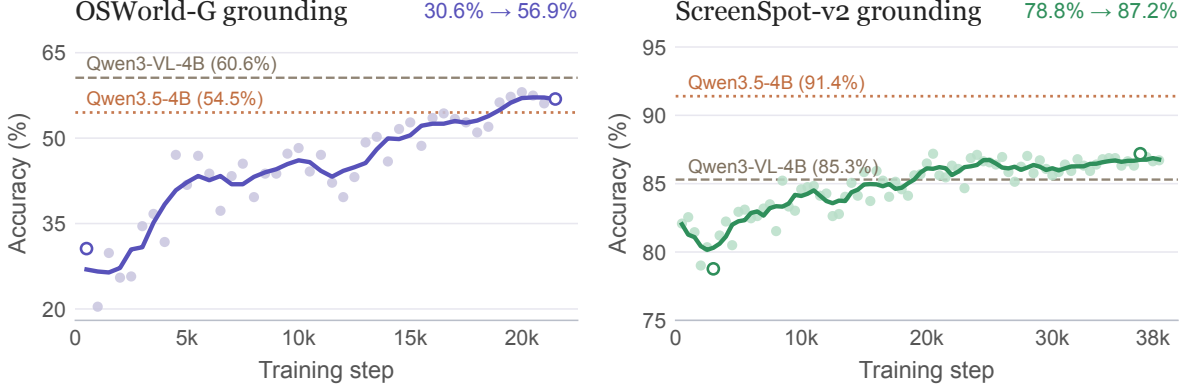


Figure 6 Grounding progression during Stage 1 GUI-oriented midtraining on OSWorld-G (left) and ScreenSpot-v2 (right). Dots show individual checkpoint evaluations, solid curves show five-checkpoint moving averages, and horizontal lines mark the Qwen3-VL-4B and Qwen3.5-4B reference baselines.

Building on this midtrained checkpoint, end-to-end OSWorld-Verified performance rises by 20.44 percentage points across the downstream training stages: from 45.56% after GUI SFT to 55.84% after Hybrid SFT (+10.28 pp), and to 66.0% after online RL (+10.16 pp). Figure 7 shows the progression during RL, and Section 6.5 examines the accompanying changes in execution length and token usage.

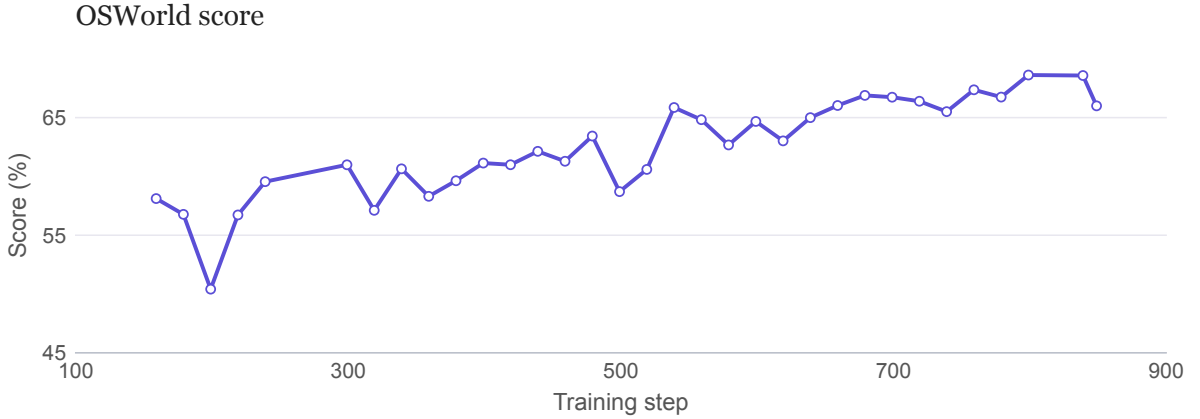


Figure 7 OSWorld-Verified scores during reinforcement learning, using all 33 recorded evaluations. The final model scores 66.0%.

6.5 Execution Efficiency and Token Usage

GUI SFT, Hybrid SFT, and Hybrid RL denote the final checkpoints of the GUI Agent SFT stage, Hybrid CUA SFT stage, and RL stage, respectively. We compare task performance, execution length, and token usage on 361 OSWorld-Verified tasks.

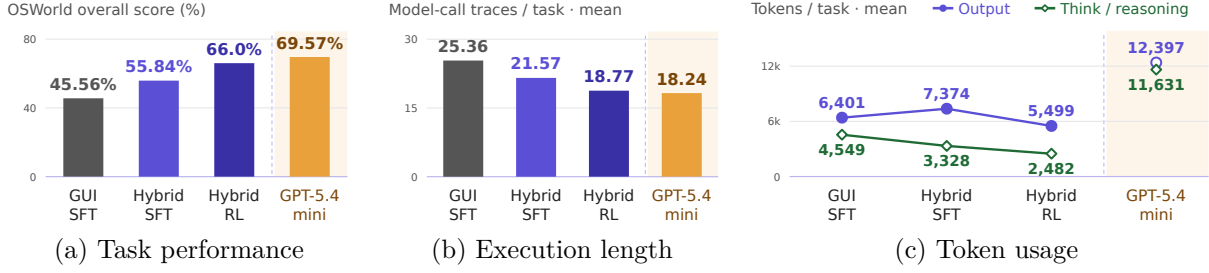


Figure 8 Stage-wise task performance, execution length, and token usage. Execution length counts model-call traces per task. Think/reasoning tokens are a subset of output tokens. Detailed mean, median, and P90 distributions are reported in Appendix B (Table 6).

Figure 8 shows that across the three training stages, task scores rise from 45.56% after GUI SFT to 55.84% after Hybrid SFT and 66.0% after RL, while mean model calls decrease from 25.36 to 21.57 and 18.77 (GPT-5.4-mini averages 18.24). Hybrid SFT pairs a 10.28-percentage-point gain and 14.9% fewer model calls with higher mean output-token usage, averaging 7,374 tokens per task compared with 6,401 for GUI SFT. RL then improves both performance and generation efficiency: it adds 10.16 percentage points over Hybrid SFT, reduces model calls by 13.0%, and reduces both output and think tokens by approximately 25.4%, bringing mean output tokens to 5,499 per task. The final policy therefore combines stronger task performance with shorter interaction sequences and lower output volume.

6.6 Browser Task Performance

On WebVoyager-Static, the 4B Mage-VL-Cowork SFT model achieves 58.6% success (377/643 tasks; Table 4), ranking second overall behind the specialized browser agent Fara1.5-4B (68.0%). Despite being trained as a general desktop-and-browser agent, Mage-VL-Cowork outperforms both larger 8B–9B models—such as Qwen3-VL-8B-Thinking [34] (53.7%) and Qwen3.5-9B (53.3%)—and dedicated open-source browser agents including OpenWebRL-4B [35] (47.1%), MolmoWeb-8B [21] (46.8%), and Fara-7B [36] (40.4%).

Comparing previously published live WebVoyager scores against our offline evaluation reveals a drop of 12.8 to 33.1 percentage points across the five models with self-reported live results [21, 35, 36]. Auditing the rollout traces under both the original live-judge protocol and our offline verifier shows why: because the standard live judge inspects only the final screenshot and explicitly instructs the evaluator to trust unsupported claims in the agent’s text (“choose to believe the content”), it rewards fluent guessing and masks three recurring failure modes in current web agents: (1) *shortcut answering from search snippets or parametric memory*, where agents answer after a few steps from a result card or prior knowledge without opening the detail page that holds the ground-truth evidence; (2) *incomplete multi-constraint filtering*, where agents locate the target site and issue a keyword query but fail to operate interactive filter controls (such as date pickers, price sliders, or sort menus), returning the first plausible candidate that violates secondary constraints; and (3) *uncommitted stateful interactions*, where agents fill out forms or click preliminary buttons and declare success while the backend database records no actual mutation (e.g., an empty cart or unsubmitted booking). These findings suggest that screenshot-only judging can inadvertently incentivize premature, persuasive answers over thorough interaction. By jointly verifying page-visit traces, ground-truth constraints, and backend database state, WebVoyager-Static removes these false positives and provides a more diagnostic reference for evaluating genuine multi-step web interaction.

Model	WebVoyager-Static (%)	Live reported (%)	Δ (pp)
<i>Open-source models</i>			
Fara1.5-4B	68.0	80.8	−12.8
Qwen3-VL-8B-Thinking [34]	53.7	—	—
Qwen3.5-9B	53.3	—	—
Qwen3-VL-4B-Thinking [34]	49.6	—	—
OpenWebRL-4B [35]	47.1	74.1	−27.0
MolmoWeb-8B [21]	46.8	78.2	−31.4
MolmoWeb-4B [21]	42.6	75.2	−32.6
Fara-7B [36]	40.4	73.5	−33.1
Qwen3.5-4B	37.3	—	—
<i>Mage-VL-Cowork-SFT</i>	58.6	—	—

Table 4 WebVoyager-Static results across all 643 tasks under identical offline evaluation settings, alongside self-reported live WebVoyager scores where available [21, 35, 36]. Blue shading marks the best and second-best WebVoyager-Static success rates.

6.7 Effect of the Evolved Harness on Student Performance

We next evaluate how evolving the harness around a fixed frontier teacher model shapes both teacher execution and downstream 4B student learning. Starting from the official GPT-5.4 OSWorld agent, β_1 pairs GUI interaction with a coding channel, whereas β_2 exposes GUI, coding, and accessibility (AX) channels. Both teachers use the same underlying GPT-5.4 model, and each generates the trajectories used to train its corresponding student. All four agents are evaluated on the same 672 held-out multi-application tasks under identical evaluation settings (Table 5).

Metric	Teacher		Student	
	β_1	β_2	β_1	β_2
Success rate (%) $\uparrow$	92.0	89.6	37.6	25.1
Steps $\downarrow$	25.0	35.3	30.91	39.4

Table 5 Teacher–student comparison for two hybrid harness variants. All four agents are evaluated on the same 672 held-out multi-application tasks. Each student is trained on trajectories generated by the corresponding teacher variant. Higher success rates and fewer steps are preferred; bold marks the better student result for each metric.

With the teacher model held fixed, the β_1 harness outperforms the AX-augmented β_2 harness on both success rate (92.0% vs. 89.6%, +2.4 pp) and step count (25.0 vs. 35.3 steps, −29.2%). This gap widens substantially when distilled into the 4B students: the β_1 student reaches 37.6% versus 25.1% for β_2 , while using 21.5% fewer steps (30.91 vs. 39.4). While a frontier teacher can partially absorb an extra channel by taking more steps, a compact student is much more sensitive to ambiguous channel routing. Refining and pruning the teacher harness before rollout therefore yields student gains far larger than the teacher’s raw success-rate difference alone would suggest.

7 Conclusion

Mage-VL-Cowork brings effective hybrid computer use to a 4B policy, achieving 66.0% on OSWorld-Verified under a 50-step budget and outperforming open-source agents several times its size. Rather than post-training a checkpoint already specialized for computer use, our progressive five-stage training curriculum, GUI-oriented midtraining, GUI agent SFT, coding and terminal SFT, hybrid CUA SFT, and online hybrid RL, builds visual grounding, GUI execution, desktop coding and terminal operation, hybrid channel routing, and online RL from a general multimodally aligned

starting point. Upstream of training, evolving the teacher harness, elicits stronger demonstrations than the teacher model’s default setup, while reusable skill-conditioned workflows cut recurring per-task generation expenditure by 82.6%. Together, these results provide a scalable route from reusable environment construction and teacher-harness optimization to compact hybrid computer-use policies, pushing forward the frontier of on-device intelligence.

Contributor List

Contributors: Xiaoyi Zhang^{*†}, Haoqing Wang^{*}, Xinjie Zhang, Xiang Ming, Xing Wang, Senqiao Yang, Kaichen Zhang, Zhaoyang Jia, Andong Tan, Xinjiang Wang, Jinghao Guo, Yifei Shen, Xiao Li, Peng Zhang, Xun Guo, Jiahao Li, Shicheng Zheng, Jinglu Wang, Zongyu Guo, Wenxuan Xie, Bin Li, Yan Lu

^{*}Equal Contribution. [†]Project Lead (xiaoyizhang@microsoft.com).

A Web Replica Environments and Data Counting Definitions

A.1 High-Fidelity Web Replicas

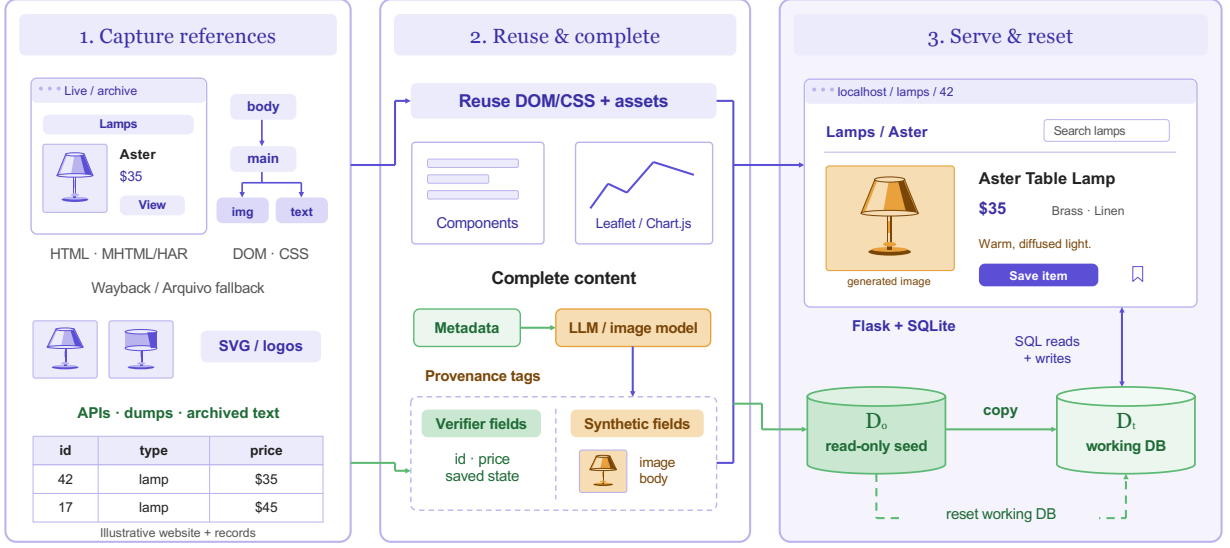


Figure 9 Construction and reset of a web replica. Reference structure, assets, and records are combined with generated content to assemble the local site. Each episode operates on a working database restored from the read-only seed. The website and records are illustrative.

To support the controllable browser workflows and offline evaluations introduced in Sections 3.2 and 6.1, we pair synthesized InfiniteWeb [19] environments with self-hosted website replicas built upon and expanding WebHarbor [20] (Figure 9). Each replica runs as a local **Flask** and **SQLite** service with target domains routed to the local host. Before every episode, the environment restores a fresh working database from a read-only seed snapshot, isolating stateful operations across rollouts while exposing the underlying tables for deterministic verification. Constructing and upgrading these replicas follows two core principles:

Frontend and interactive fidelity. Early mock environments often simplify pages into generic templates or static placeholder blocks, which reduces visual complexity and omits the controls needed for multi-step tasks. To match real-world interfaces, we capture reference snapshots from target websites—spanning DOM hierarchies, computed CSS rules, and static assets (logos, icons, and inline SVGs), supplemented by web archives when live endpoints are restricted—and reconstruct the full page structure. Beyond static layout, we implement the interactive controls required for end-to-end task completion, including multi-attribute filter panels, date pickers, sorting menus, pagination, stateful submission forms, and domain-specific components such as **Leaflet** maps and **Chart.js** views.

Database grounding and multimodal completion. We populate each replica’s seed database primarily from authentic structured sources, including public APIs and data dumps (e.g., ESPN, Stack Exchange, Hacker News, and OpenStreetMap), web crawl archives, and bibliographic catalogs. When raw records lack full-text bodies, detailed item descriptions, or image assets, we condition LLM text generation and image-generation models on the authentic structured attributes to synthesize coherent articles, descriptions, and context-matched thumbnails. All synthesized presentation fields are tagged separately from the canonical attributes queried by task verifiers, ensuring realistic visual and textual density without introducing ambiguity into ground-truth grading.

A.2 Web Task Synthesis and Verifiable Contracts

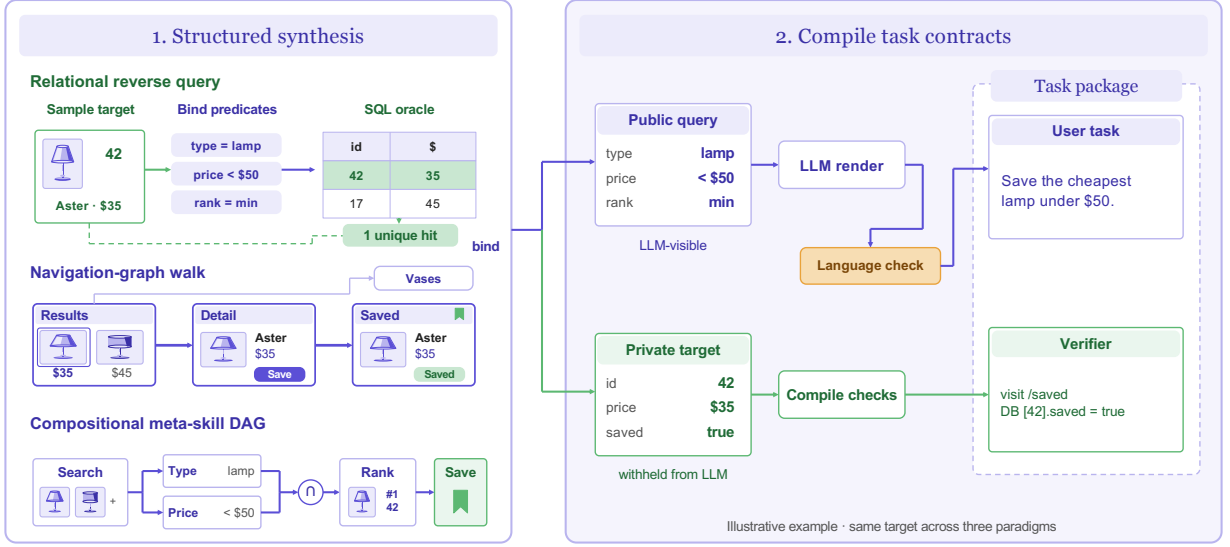


Figure 10 Structured task synthesis and contract compilation. The illustrative example links database predicates, a navigation path, and a compositional skill graph to a shared task specification. The instruction renderer receives only public query constraints; the verifier compiler uses the private target to check page visits and final database state.

Building on the resettable replicas in Appendix A.1, we formulate web task generation as a structured synthesis pipeline that pairs every synthesized instruction with an executable verification contract (Figure 10). Each replica exposes its seed relational database $\mathcal{D}_0$, a site navigation graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ over page templates and UI transitions, and a set of stateful backend operations. From these structured representations, we construct task specifications through three complementary approaches:

1. **Database-grounded query synthesis.** We sample target records and multi-attribute filter predicates directly from the seed database $\mathcal{D}_0$ and execute an SQL query over $\mathcal{D}_0$ to ensure that the resulting constraint set yields a unique, non-trivial answer before turning the predicates into a user task.
2. **Navigation-path synthesis.** We sample executable multi-hop walks over the site’s page-transition graph $\mathcal{G}$ (e.g., *Home* → *Category Listing* → *Filtered Results* → *Item Detail* → *Saved Collection*), converting each valid path into a goal-directed task whose required intermediate pages and terminal state are known by construction.
3. **Compositional skill workflows.** Extending the workflow-first principle from Section 3.2 to browser environments, we abstract recurring web operations into typed *meta-skills*—such as keyword search, multi-attribute filtering, comparative ranking, and stateful form submission—each equipped with explicit preconditions and postconditions. Composing compatible meta-skills into directed acyclic workflows across distinct entities in $\mathcal{D}_0$ produces diverse multi-step tasks that share a reusable verification template.

Contract compilation and targeted curriculum expansion. Each task specification compiles along two separated branches (Figure 10, right). To render natural-language instructions without leaking ground-truth answers into the prompt, we enforce strict information asymmetry: the instruction-rendering LLM receives only the public query constraints and allowed filter attributes, while the private target records and oracle answers remain withheld; candidate prompts that fail uniqueness or naturalness checks are discarded. In parallel, the verifier compiler binds the private

target records to deterministic checks over both the execution trace (confirming required page visits and UI actions) and the final outcome (checking the submitted answer and post-episode SQLite state). Following the iterative feedback loop in Section 3.3, subsequent synthesis rounds increase sampling weight on skill compositions where the student policy underperforms—such as multi-constraint filtering or detail-page evidence verification—supplying rule-verified browser and cross-application trajectories for post-training.

A.3 Dataset Counting Definitions

Tasks are deduplicated across 11 batches by domain and task identifier. A task–file reference is a distinct path or filename extracted from a task’s metadata or text, including declared outputs. References are counted within each task, with corresponding structured and textual declarations merged when resolved.

Application coverage is multi-label, totaling 419,505 application–task incidences across nine application categories. The 17,542 InfiniteWeb [19] tasks are reported separately. File categories are defined by extracted suffixes, including custom suffixes and a no-extension category.

B Midtraining Explorations, Context Design, and Compute Accounting

B.1 GUI Midtraining Design Trade-Offs

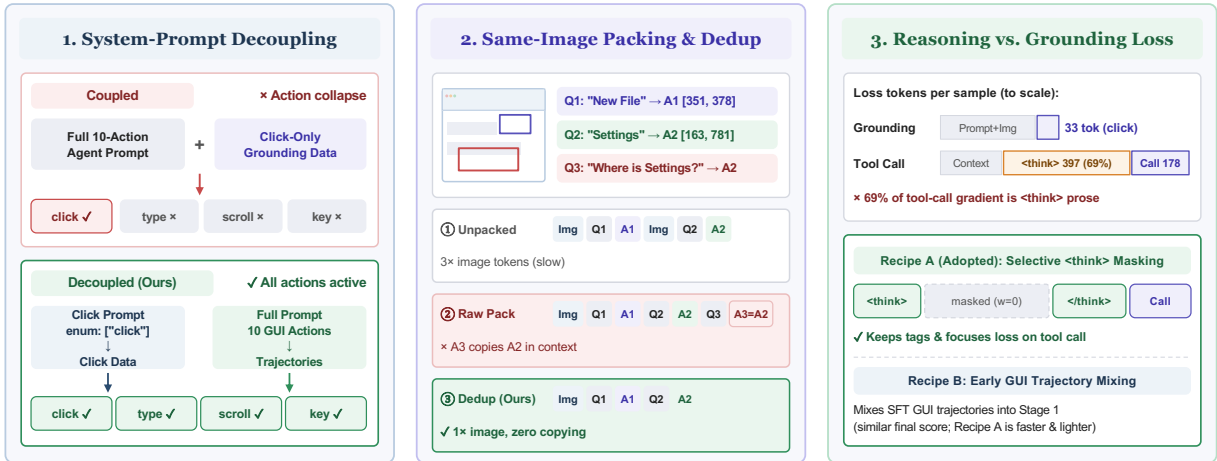


Figure 11 Three design choices in Stage 1 GUI-oriented midtraining: (1) decoupling the click-only grounding prompt from the full agent schema to prevent action-space collapse, (2) deduplicating repeated target elements within packed single-screenshot sequences to eliminate in-context coordinate copying, and (3) selectively masking interior `<think>` tokens to balance reasoning format retention with grounding and tool-syntax learning.

Building on the Stage 1 overview in Section 4.1, Figure 11 illustrates three design choices that proved critical when jointly training screen grounding, structured tool calling, and chain-of-thought reasoning.

Domain-targeted grounding and system-prompt decoupling. We curate our grounding pool from open-source GUI datasets, including UGround [23], UI-E2I-Synth [24], OS-Atlas [25], MolmoWeb [21], JEDI [26], and SeeClick [12], filtering for 720p–1080p desktop and browser screenshots and normalizing all bounding boxes to center coordinates in $[0, 999]$ formatted as `computer_use click` tool calls. We pair these click-only examples with a minimal grounding system prompt declaring only `"enum": ["click"]` (Figure 11, left). Training click-only grounding data under

the full 10-action agent prompt causes the policy to associate the full prompt with unconditional clicking and ignore `type`, `scroll`, or `keypress` downstream; decoupling the two prompts transfers visual localization without collapsing the action space.

Target-level deduplication in same-screenshot packing. Packing multiple grounding queries for the same screenshot into a shared sequence [27] avoids repeated image encoding. However, open-source grounding datasets often contain multiple paraphrased queries targeting the same UI element on a screenshot (Figure 11, center). When packed together, later turns achieve near-zero loss by copying the coordinate pair emitted for an earlier turn ($A_k = A_j$) instead of attending to the image. Retaining at most one query per distinct target location per screenshot removes this shortcut and drives the steady grounding gains in Figure 6.

Reconciling non-reasoning grounding with reasoning tool calls. In packed sequences, a grounding example averages only 33 supervised action tokens, whereas a reasoning-annotated tool-call example averages 575 supervised tokens, 69% of which belong to the `<think>` block (Figure 11, right). Unmasked supervision lets verbose reasoning dominate gradients and dilute localization, whereas stripping `<think>` blocks causes the model to drop reasoning tags at test time. We explored two complementary recipes:

1. **Selective `<think>`-body masking on tool-call blends.** We mask the interior tokens of the `<think>` reasoning while supervising the opening and closing `<think>...</think>` delimiter tags and the `<tool_call>`, preserving the think-then-act format at inference time while focusing gradients on tool syntax and coordinates.
2. **Incorporating post-training GUI trajectories into midtraining.** We also explored mixing synthesized multi-step GUI trajectories directly into Stage 1 alongside grounding data, exposing the model early to the full downstream action space and step-level rationales.

Both recipes reach comparable accuracy after downstream SFT; we adopt selective `<think>`-body masking for the main pipeline because it keeps Stage 1 decoupled from trajectory synthesis, uses fewer tokens, and iterates faster.

B.2 Single Step Cowork Bench

Standard function-calling benchmarks such as BFCL evaluate text-only tool use without visual inputs, standard GUI grounding benchmarks test only single-action coordinate clicking, and end-to-end desktop benchmarks such as OSWorld require downstream SFT and hours of multi-step VM rollouts per checkpoint. To jointly evaluate visual localization, action-type selection, and reasoning format during Stage 1 midtraining, we construct Single Step Cowork Bench from 342 representative interaction steps sampled from real desktop trajectories across office suites (Calc, Writer, Impress), web browsing (Chrome), media and creative tools (GIMP, VLC), development environments (VS Code), and system utilities.

Every example is evaluated under the full downstream agent system prompt and spans eight GUI action primitives (`click`, `double_click`, `triple_click`, `drag`, `scroll`, `type`, `keypress`, and `terminate`). To separate direct localization from intent-driven reasoning, each step is paired with both an *explicit* instruction (naming the target control directly, e.g., “Click the Settings menu item in the popup menu”) and an *intent* instruction (stating the local subgoal, e.g., “Open the VS Code settings page from this menu”). Grading jointly verifies (1) valid `<think>` and `<tool_call>` syntax, (2) exact action-type and argument match, and (3) spatial hit accuracy within the ground-truth bounding box. During Stage 1 development, this lightweight diagnostic surfaced failure modes invisible to click-only grounding benchmarks—including action-space collapse onto `click` under the full agent prompt, hallucinated placeholder function names (`example_function_name`)

copied from the schema, and unterminated or missing `<think>` blocks on dense spreadsheet views—allowing rapid recipe iteration before multi-step post-training.

B.3 Context Window Design and Step-Level Reasoning Synthesis

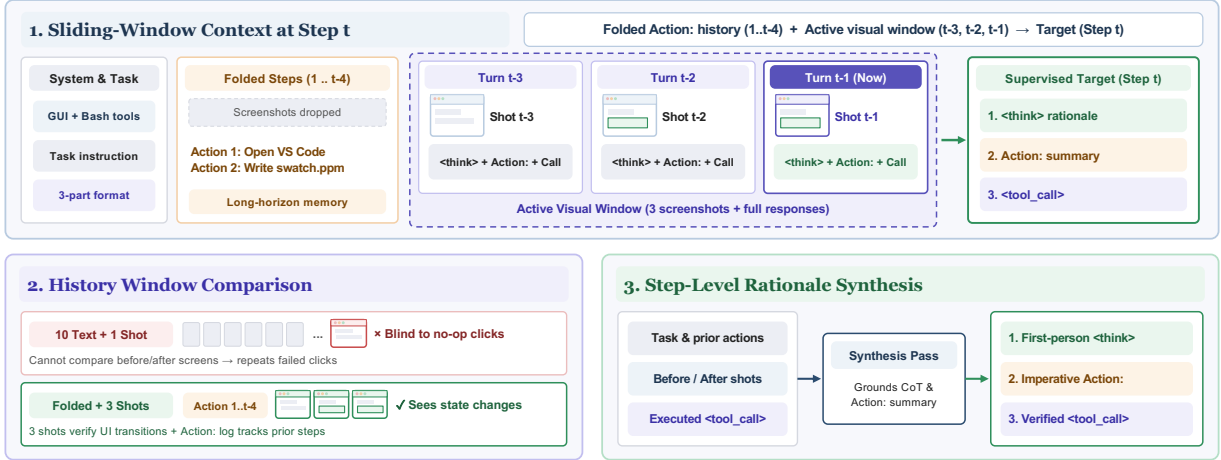


Figure 12 Sliding-window context design and step-level rationale synthesis: (1) each turn conditions on folded imperative **Action:** summaries of steps $1 \dots t-4$ together with the three most recent visual turns ($t-3, t-2, t-1$), (2) retaining three screenshots exposes whether prior actions altered the interface and prevents repetitive no-op click loops, and (3) a reasoning synthesis pass grounds first-person `<think>` traces and **Action:** summaries on before/after screenshots and the executed tool call.

Long-horizon desktop trajectories face a fundamental context trade-off: retaining every historical screenshot quickly exhausts the context budget, whereas keeping only the current screenshot alongside text history leaves the policy blind to whether its previous action actually changed the screen. In early iterations with ten text turns and a single screenshot, a slightly off-target click that left the interface unchanged frequently triggered repetitive click loops (Figure 12, bottom left). To resolve this, each turn-level training example uses an asymmetric sliding window (Figure 12, top): it retains the three most recent visual turns ($t-3, t-2, t-1$) with their screenshots and full assistant responses so the model can directly verify before-and-after UI transitions, while folding earlier steps ($1 \dots t-4$) into single-line imperative **Action:** summaries that preserve a compact long-horizon record of completed operations.

To supply consistent supervision for both in-window reasoning and out-of-window history folding across heterogeneous trajectory sources, we run a step-level reasoning synthesis pass conditioned on the task context, before/after screenshots, and the executed tool call (Figure 12, bottom right). For every step, this pass produces a uniform three-part target: (1) a grounded first-person `<think>` rationale, (2) a concise imperative **Action:** summary that directly serves as the folded memory entry once the turn slides past $t-3$, and (3) the verified `<tool_call>`. Together with the asymmetric window, this uniform output format turns the model’s own generations into a self-sustaining working memory across SFT and RL rollouts: recent turns provide visual and reasoning context for local error recovery, while earlier **Action:** lines accumulate into a compact execution log across long-horizon, interdependent tasks.

B.4 Dataset Volumes and Compute Accounting

For midtraining and SFT, nominal tokens equal the number of packed sequences (751,346 for GUI-oriented midtraining, 983,462 for GUI agent SFT, 174,029 for coding and terminal SFT, and 1,377,805 for hybrid CUA SFT) multiplied by their 32,768-token capacity, including padding. Nominal language-model compute is estimated as $C \approx 6N_{\text{LM}}T$, with $N_{\text{LM}} = 4 \times 10^9$ and T equal to the two-epoch packed-token capacity. The estimates are 1.182×10^{21} FLOPs for midtraining,

3.988×10^{21} for SFT, and 2.618×10^{21} for RL, totaling 7.787×10^{21} FLOPs. Their shares are 15.2%, 51.2%, and 33.6%. The accounting covers the reported language-model training stages, including image tokens in language-model sequences; base pretraining, vision-encoder computation, and separate attention/recomputation overhead terms are outside this scope.

Table 6 reports the mean, median, and P90 of execution length and per-task and per-response token usage across training stages (Section 6.5), using the same counting conventions as Figure 8.

Mean / Median / P90

Token basis: MAGE text counts · GPT API usage

Model / stage	Overall (%)	Traces per task	Output tokens per task	Think / reasoning per task	Output tokens per response	Think / reasoning per response
GUI Agent SFT	45.56	25.36 / 19 / 50	6,401.2 / 4,716 / 13,563.4	4,549.4 / 3,400.5 / 9,657.9	252.5 / 219 / 435	179.6 / 148 / 355
HybridCUA SFT	55.84	21.57 / 15 / 50	7,374.4 / 4,119 / 17,370	3,327.6 / 2,208 / 7,917	343.6 / 257 / 616.3	155.0 / 147 / 218
HybridCUA RL	66.0	18.77 / 14 / 42	5,498.6 / 3,632.5 / 11,551.9	2,482.4 / 1,770.5 / 5,500.7	292.9 / 208 / 528.8	133.2 / 127 / 181
GPT-5.4-mini (xhigh)	69.57	18.24 / 13 / 48	12,397.4 / 9,137 / 27,113	11,630.6 / 8,592 / 24,641	679.5 / 272 / 1,634	637.5 / 230 / 1,587

Table 6 Execution and token statistics by training stage. Triples report mean, median, and P90; token-counting conventions are given in Section 6.5.

C RL Infrastructure Details and Measurements

This appendix describes the RaaS implementation and deployment, followed by workload, transport, update-timing, and policy-lag profiles. Each profile is associated with its measurement run, update window, and aggregation rule.

C.1 Rollout-as-a-Service Architecture

The infrastructure integrates PoLar [32], Slime, Azure Relay, and Azure Virtual Machine Scale Sets (VMSS). The training client uses Slime adapters for desktop rollouts, turn-level samples, and data-parallel shards. SGLang [33] serves model requests, while RaaS workers on Azure VMSS run the resource manager and CUA runners. Routing and image preparation execute on CPUs; inference and optimization execute on GPUs.

The relay exposes three logical communication channels:

- **Control.** The training client submits work and queries task status and ready manifests from the resource manager.
- **Inference.** The harness sends prompts and screenshots to the inference service and receives generated tokens and log probabilities.
- **Bulk trajectories.** Stored compact episodes are delivered to the training adapters using deduplication and gzip compression.

The relay carries control, inference, and compact-trajectory traffic. Trainer-to-inference weight synchronization and training collectives use intra-cluster paths. External judge calls and artifact storage use their respective service interfaces.

C.2 Resource Management and Sandbox Execution

The resource manager contains three components. The *Task Dispatcher* handles submission, queueing, dispatch, status, and ready manifests. The *Sandbox Scheduler* manages fleet capacity and sandbox allocation and release, with separate admission limits for initialization, running, and post-run processing. The *Compact Episode JSON Store* gathers completed trajectory records and supplies the bulk-transfer path. These components separate task coordination and environment capacity from the model-serving process.

Each host has separate concurrency caps for sandbox initialization, execution, and post-run processing: 4 (INIT), 16 (RUN), and 8 (POSTRUN), respectively. Lifecycle-specific admission control manages setup and post-processing alongside the 16 running-episode slots.

Each CUA runner contains four components. *EnvRuntime* provides the desktop environment, captures screenshots, and executes actions. *CUAHarness* maintains the stateful agent loop, issues model requests, and dispatches GUI or terminal tools. *Evaluator* produces the task’s evaluation, reward, and termination outcomes, using rule-based checks or an external judge where required. *TrajectoryBuilder* combines turn traces and evaluator outcomes into compact episode JSON for transfer to the training client.

C.3 Asynchronous Rollouts and Trajectory Preparation

The configured update batch contains 96 task groups with eight episode slots per group, or 768 slots. The admission window spans four batches, with a maximum configured policy lag of three optimizer updates. This organization allows rollout collection and optimization to overlap while bounding the policy-version gap of admitted data. Mixed-policy episodes are enabled.

TIS caps importance weights at 2.0. Admission is configured over a four-batch window, with policy lag bounded at three optimizer updates (Section 4.5).

The data path keeps compact trajectories separate from expanded training tensors. The rollout bridge and adapters construct per-turn samples and data-parallel shards. Ray distributes compressed images, tokens, and metadata; node-local CPU processing decodes and resizes the images and applies the model’s image processor before the tensors are consumed by the trainer GPUs. Prefetching prepares trajectory data ahead of training consumption rather than placing every preparation operation on the immediate update path.

Each turn retains up to three recent images. Compact episodes deduplicate stored screenshots for transfer, and turn-level preparation materializes the image inputs required by each training example.

C.4 Deployment Scale and Runtime Profile

The CPU fleet comprises 70 Standard_D64d_v5 hosts, each with 64 vCPUs and 256 GiB of memory. Sixteen GUI slots per host provide 1,120 configured slots. Each guest receives eight vCPUs and 4 GiB of memory, giving a 2:1 guest-to-host vCPU allocation ratio. The reference GPU deployment uses 32 B200 GPUs, split into 24 training and eight inference GPUs.

A deployment with 28 training GPUs and four inference GPUs reached a peak occupancy of 1,120 running environments. The 24/8 split is the reference configuration used in the architecture diagram.

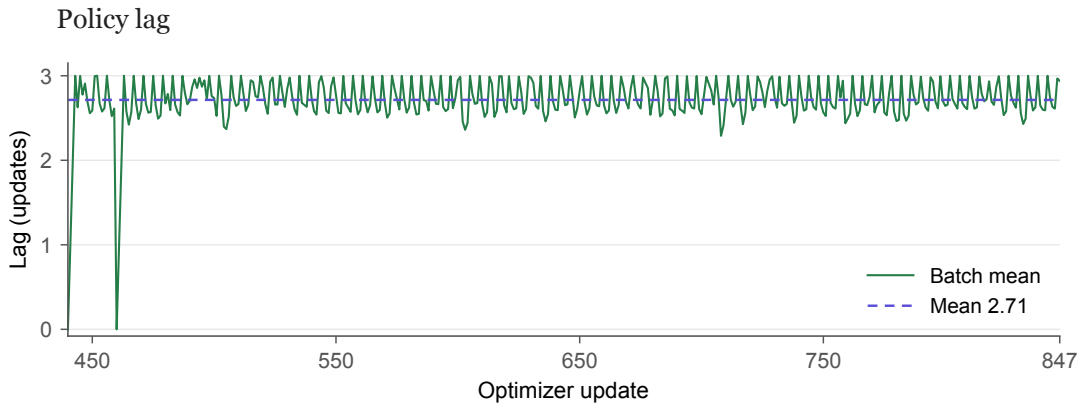


Figure 13 Runtime profile of policy lag.

In the workload-profiling window, each update prepares 18,437.56 turn sequences, 34,973.68 image occurrences, and 136.49 million language-model tokens on average. Compressed episode payloads average 2.44 GB (2.27 GiB) per update. Node-local preprocessing materializes 408.25 GiB of CPU pixel/grid tensors across 24 trainer ranks.

Across 11 post-initialization updates, the mean update time is 570.3 seconds, corresponding to 6.31 updates per hour in that window. The mean of 408 update-level policy-lag measurements is 2.71 updates, with observed batch means ranging from 0.0 to 3.0.

C.5 Workload and Transport Accounting

Turn-sequence counts include all prepared samples, including zero-mask sequences. Image-occurrence counts include screenshot inputs repeated across turn histories. The context retains up to three recent images per turn.

Bulk volume measures successful compressed response payloads for consumed rollout groups, averaging 2.44 GB (2.27 GiB) per update. Protocol headers and retransmissions are outside this payload measure. Prepared CPU pixel/grid tensors average 408.25 GiB per update across 24 trainer ranks.

C.6 Policy-Lag Aggregation

For a consumed turn-level training sample j , the logged lag is

$$\ell_j = \max(0, a_j - v_j), \quad L_u = \frac{1}{|\mathcal{B}_u|} \sum_{j \in \mathcal{B}_u} \ell_j, \quad (2)$$

Here a_j denotes the accepted rollout index, v_j the behavior-policy version index, and $\mathcal{B}_u$ the turn samples consumed at update u . Version indices follow the common numbering convention used to compute the policy-lag statistic.

The reported policy lag of 2.71 updates is the arithmetic mean of the 408 update-level means L_u . These means range from 0.0 to 3.0 updates, within the configured three-update lag bound.

References

- [1] 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, 2025. URL <https://arxiv.org/abs/2510.24563>.
- [2] Siqi Fan, Minghao Li, Xiaoqian Ma, Wenhui Tan, Xiusheng Huang, Juntong Wu, Liujie Zhang, Shuo Shang, and Weihang Chen. Screenshots or Tools? Eliciting Tool Use and Managing Multimodal Context in Hybrid GUI-MCP Computer-Use Agents, 2026. URL <https://arxiv.org/abs/2608.03327>.
- [3] Jaehun Jung, Ximing Lu, Brandon Cui, Muhammad Khalifa, Shaokun Zhang, Hao Zhang, Jin Xu, Amala Sanjay Deshmukh, Karan Sapra, Andrew Tao, Yejin Choi, Jan Kautz, Mingjie Liu, and Yi Dong. ProCUA-SFT Technical Report, 2026. URL <https://arxiv.org/abs/2606.17321>.
- [4] 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.
- [5] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models, 2023. URL <https://arxiv.org/abs/2305.16291>.
- [6] Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. SkillWeaver: Web Agents can Self-Improve by Discovering and Honing Skills, 2025. URL <https://arxiv.org/abs/2504.07079>.
- [7] Tiantian He, Yihang Chen, Keyue Jiang, Ka Yiu Lee, Kaiwen Zhou, Kun Shao, and Shuai Wang. EE-MCP: Self-Evolving MCP-GUI Agents via Automated Environment Generation and Experience Learning, 2026. URL <https://arxiv.org/abs/2604.09815>.
- [8] Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, Ran Xu, and Caiming Xiong. CoAct-1: Computer-using Multi-Agent System with Coding Actions, 2026. URL <https://arxiv.org/abs/2508.03923>.
- [9] Yan Yang, Xiangru Jian, Ziyang Luo, Zirui Zhao, Yutong Dai, Ziji Shi, Hanshu Yan, Jun Hao Liew, Silvio Savarese, and Junnan Li. StateAct: Program State, before Pixels, for Long-Horizon Computer-Use Agents, 2026. URL <https://arxiv.org/abs/2607.22798>.
- [10] Zhaoyang Chu, Jiarui Hu, Xingyu Jiang, Pengyu Zou, Han Li, Chao Peng, Peter O’Hearn, Earl T. Barr, Mark Harman, Federica Sarro, and He Ye. TerminalWorld: Benchmarking Agents on Real-World Terminal Tasks, 2026. URL <https://arxiv.org/abs/2605.22535>.
- [11] Zhiyuan Fan, Tinghao Yu, Yuanjun Cai, Jiangtao Guan, Yun Yang, Dingxin Hu, Jiang Zhou, Xing Wu, Zhuo Han, Feng Zhang, and Lilin Wang. Toward Scalable Terminal Task Synthesis via Skill Graphs, 2026. URL <https://arxiv.org/abs/2604.25727>.
- [12] Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. SeeClick: Harnessing GUI Grounding for Advanced Visual GUI Agents, 2024. URL <https://arxiv.org/abs/2401.10935>.
- [13] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan

- Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments, 2024. URL <https://arxiv.org/abs/2404.07972>.
- [14] Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an End-to-End Web Agent with Large Multimodal Models, 2024. URL <https://arxiv.org/abs/2401.13919>.
- [15] Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, Zhennan Shen, Zhuokai Li, Ryan Li, Xiaochuan Li, Junda Chen, Boyuan Zheng, Peihang Li, Fangyu Lei, Ruisheng Cao, Yeqiao Fu, Dongchan Shin, Martin Shin, Jiarui Hu, Yuyan Wang, Jixuan Chen, Yuxiao Ye, Danyang Zhang, Dikang Du, Hao Hu, Huarong Chen, Zaida Zhou, Haotian Yao, Ziwei Chen, Qizheng Gu, Yipu Wang, Heng Wang, Diyi Yang, Victor Zhong, Flood Sung, Y. Charles, Zhilin Yang, and Tao Yu. OpenCUA: Open Foundations for Computer-Use Agents, 2025. URL <https://arxiv.org/abs/2508.09123>.
- [16] Qiushi Sun, Kanzhi Cheng, Zichen Ding, Chuanyang Jin, Yian Wang, Fangzhi Xu, Zhenyu Wu, Chengyou Jia, Liheng Chen, Zhoumianze Liu, Ben Kao, Guohao Li, Junxian He, Yu Qiao, and Zhiyong Wu. OS-Genesis: Automating GUI Agent Trajectory Construction via Reverse Task Synthesis, 2025. URL <https://arxiv.org/abs/2412.19723>.
- [17] 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, 2026. URL <https://arxiv.org/abs/2605.25624>.
- [18] Shuai Bai, Jiayong Deng, Sicheng Fan, Yikun Fu, Chang Gao, Xuhao Hu, Mianqiu Huang, Yizhen Jiang, Yuheng Jing, Dehui Kong, Keliang Li, Ning Li, Wanli Li, Dayiheng Liu, Dunjie Lu, Changwei Luo, Que Shen, Zheyuan Wang, Zijian Wang, Jie Wu, Gao Wu, Zhihui Xie, Rui Xie, Haiyang Xu, An Yang, Jiakang Yuan, Yanming Zhang, Jiajun Zhang, Xi Zhang, Zhenru Zhang, Zhuo Zhen, Mingkan Zhu, and Bowen Zhou. RecreationWorld: Scalable and Verifiable Environments for Hybrid Computer-Use Agents, 2026. URL <https://arxiv.org/abs/2609.22000>.
- [19] Ziyun Zhang, Zezhou Wang, Xiaoyi Zhang, Zongyu Guo, Jiahao Li, Bin Li, and Yan Lu. InfiniteWeb: Scalable web environment synthesis for GUI agent training. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens, editors, *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 28465–28492, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1313. URL <https://aclanthology.org/2026.acl-long.1313/>.
- [20] WebHarbor Team and Contributors. WebHarbor: Docking Real Websites for Evolving GUI Agent Environments, 2026. URL <https://aiming-lab.github.io/webharbor.github.io>. Project website.
- [21] Tanmay Gupta, Piper Wolters, Zixian Ma, Peter Sushko, Rock Yuren Pang, Diego Llanes, Yue Yang, Taira Anderson, Boyuan Zheng, Zhongzheng Ren, Harsh Trivedi, Taylor Blanton, Caleb Ouellette, Winson Han, Ali Farhadi, and Ranjay Krishna. MolmoWeb: Open Visual Web Agent and Open Data for the Open Web, 2026. URL <https://arxiv.org/abs/2604.08516>.
- [22] Shengbang Tong, Ellis Brown, Penghao Wu, Sanghyun Woo, Manoj Middepogu, Sai C Akula, Jihan Yang, Shusheng Yang, Adithya Iyer, Xichen Pan, et al. Cambrian-1: A fully open,

vision-centric exploration of multimodal llms. *Advances in Neural Information Processing Systems*, 37:87310–87356, 2024.

- [23] Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. Navigating the Digital World as Humans Do: Universal Visual Grounding for GUI Agents, 2024. URL <https://arxiv.org/abs/2410.05243>.
- [24] Xinyi Liu, Xiaoyi Zhang, Ziyun Zhang, and Yan Lu. UI-E2I-Synth: Advancing GUI Grounding with Large-Scale Instruction Synthesis, 2025. URL <https://arxiv.org/abs/2504.11257>.
- [25] Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, and Yu Qiao. OS-ATLAS: A Foundation Action Model for Generalist GUI Agents, 2024. URL <https://arxiv.org/abs/2410.23218>.
- [26] Tianbao Xie, Jiaqi Deng, Xiaochuan Li, Junlin Yang, Haoyuan Wu, Jixuan Chen, Wenjing Hu, Xinyuan Wang, Yuhui Xu, Zekun Wang, Yiheng Xu, Junli Wang, Doyen Sahoo, Tao Yu, and Caiming Xiong. Scaling Computer-Use Grounding via User Interface Decomposition and Synthesis, 2025. URL <https://arxiv.org/abs/2505.13227>.
- [27] Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguis: Unified Pure Vision Agents for Autonomous GUI Interaction, 2024. URL <https://arxiv.org/abs/2412.04454>.
- [28] Renjie Pi, Grace Lam, Mohammad Shoeybi, Pooya Jannaty, Bryan Catanzaro, and Wei Ping. On data engineering for scaling llm terminal capabilities, 2026. URL <https://arxiv.org/abs/2602.21193>.
- [29] OpenThoughts-Agent Team. OpenThoughts-Agent: Data Recipes for Agentic Models. <https://www.openthoughts.ai/blog/agent>, 2026.
- [30] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.
- [31] Feng Yao, Liyuan Liu, Dinghuai Zhang, Chengyu Dong, Jingbo Shang, and Jianfeng Gao. Your efficient RL framework secretly brings you off-policy RL training. <https://fengyao.notion.site/off-policy-rl>, 2025.
- [32] Binfeng Xu, Hao Zhang, Shaokun Zhang, Songyang Han, Mingjie Liu, Jian Hu, Shizhe Diao, Zhenghui Jin, Yunheng Zou, Michael Demoret, Jan Kautz, and Yi Dong. Polar: Agentic RL on Any Harness at Scale, 2026. URL <https://arxiv.org/abs/2605.24220>.
- [33] 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, 2024. URL <https://arxiv.org/abs/2312.07104>.
- [34] Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, Mei Li, Kaixin Li, Zicheng Lin, Junyang Lin, Xuejing Liu, Jiawei Liu, Chenglong Liu, Yang Liu, Dayiheng Liu, Shixuan Liu, Dunjie Lu, Ruilin Luo, Chenxu Lv, Rui Men, Lingchen Meng, Xuancheng Ren, Xingzhang Ren, Sibao Song, Yuchong Sun, Jun Tang, Jianhong Tu, Jianqiang Wan, Peng Wang, Pengfei Wang, Qiuyue Wang, Yuxuan Wang, Tianbao Xie, Yiheng Xu, Haiyang Xu, Jin Xu, Zhibo Yang, Mingkun Yang, Jianxin Yang, An Yang, Bowen Yu, Fei Zhang, Hang

Zhang, Xi Zhang, Bo Zheng, Humen Zhong, Jingren Zhou, Fan Zhou, Jing Zhou, Yuanzhi Zhu, and Ke Zhu. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025.

- [35] Rui Yang, Qianhui Wu, Yuxi Chen, Hao Bai, Wenlin Yao, Hao Cheng, Baolin Peng, Huan Zhang, Tong Zhang, and Jianfeng Gao. OpenWebRL: Demystifying Online Multi-turn Reinforcement Learning for Visual Web Agents, 2026. URL <https://arxiv.org/abs/2606.02031>.
- [36] Ahmed Awadallah, Yash Lara, Raghav Magazine, Hussein Mozannar, Akshay Nambi, Yash Pandya, Aravind Rajeswaran, Corby Rosset, Alexey Taymanov, Vibhav Vineet, Spencer Whitehead, and Andrew Zhao. Fara-7B: An Efficient Agentic Model for Computer Use, 2025. URL <https://arxiv.org/abs/2511.19663>.