UI-Mate: Advancing Open-Weight Foundation GUI Agents with In-Context Demonstrations

Tencent Hy Team

Foundation GUI agents can automate complex digital tasks, but deployment is hindered by scarce and biased training data, ambiguous prompts, and unreliable execution. Routine workflows rely on user-specific tools and tacit conventions, so unstated instructions can produce arbitrary variations across runs. We present UI-Mate, a foundation GUI agent that integrates an environment-grounded training stack with in-context demonstration learning. UI-Mate makes three contributions: A Scalable Environment-Grounded Training Stack: A closed-loop data engine automates task generation, environment construction, rollout, filtering, capability balancing, SFT, and online RL across massively parallel environments via unified task-verifier bundles. In-Context Demonstration Learning: A mechanism that transforms multimodal demonstrations into flexible subtask-level workflows, follows relevant demonstrated steps, and re-plans from the live interface. OSWorkerBench Benchmark and Insights: A benchmark of 100 long-horizon office tasks across 41 applications that supports instruction-only and demonstration-guided evaluation. Its demonstration resources separate a 33-task self-demo setting, built from successful strong-agent rollouts of the same targets, from a 45-task variant-demo setting, built from human recordings of related but non-identical tasks. Experiments show that UI-Mate-27B sets a new open-weight state of the art on general computer-use benchmarks, scoring 77.0% on OSWorld-Verified and 66.2% on WindowsAgentArena. On OSWorkerBench, it reaches 41.0% strict success and 76.9% progress, outperforming its Qwen3.6-27B base by 17.7 and 24.5 points. On the 33-task self-demo subset, one demonstration raises strict success from 17.2% to 35.4% and progress from 67.9% to 81.1%, substantially improving long-horizon reliability. Project page: https://ui-mate.github.io

Date: August 17, 2026

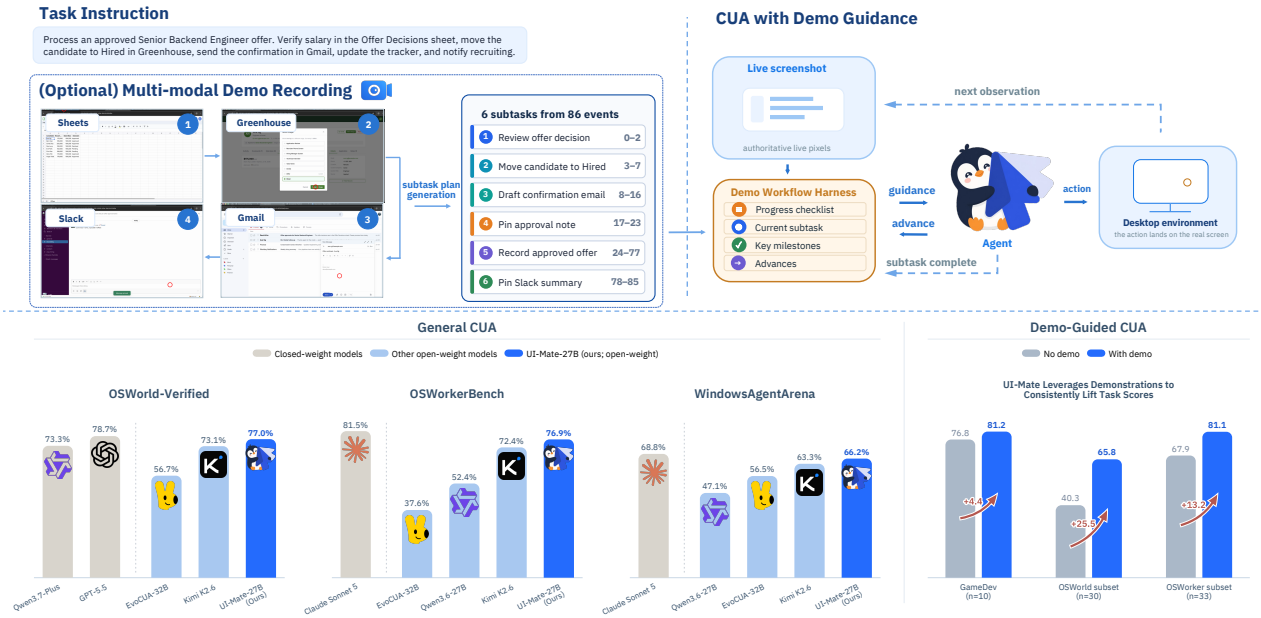


Figure 1 UI-Mate combines strong general computer-use capabilities with demonstration-guided execution. *Top:* For an underspecified cross-application task, an optional multimodal demonstration is distilled into a subtask-level workflow. During execution, the harness provides the current subtask and key milestones; the agent grounds them in the live interface, acts, and advances the workflow upon completion. *Bottom:* In the general CUA evaluation setting, UI-Mate-27B is competitive with leading open- and closed-weight systems across OSWorld-Verified, OSWorkerBench, and WindowsAgentArena. In the reported self-demo evaluation, one same-task demonstration raises task scores from 76.8 to 81.2 on GameDev, 40.3 to 65.8 on the OSWorld subset, and 67.9 to 81.1 on the 33-task OSWorkerBench subset.

Contents

1 Introduction

Foundation GUI agents promise to turn natural-language intent into multi-step action across software ecosystems. Recent CUA systems [1, 2], including UI-TARS [3, 4] and Qwen-UI-Agent [5], show that vision-language models can perceive interfaces, ground actions, and execute extended workflows directly on screen. Yet benchmark progress alone does not yield reliable real-world deployment, which remains constrained by two complementary bottlenecks. The first is a *training bottleneck*: scalable GUI learning requires not only trajectories, but also executable environments, reliable verifiers, and broad capability coverage. The second is an *interaction bottleneck*: an instruction usually specifies the desired outcome more readily than the user-specific procedure by which it should be achieved. The former limits what an agent can learn, whereas the latter limits how consistently it applies what it already knows.

GUI interaction data are inseparable from the environments in which they are generated. Unlike static text or image corpora, a trajectory is useful for learning only when its initial state can be instantiated, its actions can be executed, and its outcome can be verified. Recent work has responded with scalable synthetic experience, cross-platform trajectory collection, and automated construction of verifiable environments [2, 6–8]. Scale alone, however, does not guarantee coverage. Data production naturally favors what is inexpensive to instantiate: short, single-application tasks accumulate, whereas long-horizon workflows, cross-application information transfer, and recovery from execution errors remain sparse. Training on this skewed distribution encourages narrow execution patterns. The central data problem is therefore not merely how many trajectories to collect, but how to diagnose and correct what the corpus fails to cover.

The interaction bottleneck persists even for a capable agent because the missing information is absent from the instruction itself. Routine work is shaped by a user's tools, file organization, templates, naming conventions, and required output formats rather than by a single universal procedure. Encoding every such choice in a prompt can require as much effort as performing the task manually, so users provide concise instructions and leave procedural details implicit. Those details may then be resolved differently across runs, causing an agent that succeeds once to fail on an ostensibly identical request. Mean success rates obscure this distinction: occasional correct resolutions of ambiguity and consistently correct resolutions can produce the same average, although only the latter supports dependable delegation. For end users, reliability is not secondary to capability but the means through which capability is experienced.

We address both bottlenecks with **UI-Mate**, an open-weight foundation GUI agent that combines environment-grounded training with in-context demonstration learning. Figure 1 summarizes the resulting system and its two evaluation regimes, while §2 formalizes the task and presents the complete architecture. On the training side, a closed-loop data pipeline (§3) constructs tasks and executable environments, collects and filters rollouts, and uses a hierarchical capability tree to identify and rebalance gaps in coverage. The resulting verified trajectories and task–verifier bundles support supervised fine-tuning followed by online agentic reinforcement learning in the general computer-use training stack (§4).

On the interaction side, UI-Mate introduces DemoCUA (§5) to communicate procedural intent through a multimodal demonstration recorded from either a human or an agent. Prior human-taught GUI agents have shown that screen recordings can expose reusable procedural knowledge [9]. UI-Mate converts such a recording into a subtask-level workflow rather than treating it as an action sequence to replay. At execution time, the live screenshot remains authoritative: the agent follows demonstrated steps when they remain relevant, supplies omitted low-level actions, and re-plans when the target task diverges from the recording. UI-Mate App (§8) realizes this paradigm on the user's own desktop by recording a demonstration and subsequently running the agent against the same live environment.

Measuring the value of this guidance requires more than the instruction-only protocols used by established desktop benchmarks such as OSWorld [10] and WindowsAgentArena [11]. We therefore introduce OSWorker-Bench (§6), an office-centric benchmark of 100 long-horizon tasks spanning 41 normalized applications and 10 job families. Its 67 Long-Memory tasks require delayed reuse of dynamic information, while 49 Multi-App tasks require substantive information transfer across at least three applications. It supports instruction-only and demonstration-guided evaluation, with two distinct demonstration settings. The 33-task *self-demo* set pairs each target with a successful strong-agent rollout of that same task and supports the quantitative DemoCUA evaluation in this report. The 45-task *variant-demo* set instead pairs each target with a multimodal

human recording of a semantically related but non-identical source task and is intended to measure procedural transfer. In either paired comparison, the target instruction, initialized environment, interaction budget, and executable verifier remain fixed; only demonstration availability changes.

Our evaluation (§7) shows that UI-Mate-27B reaches 77.0% on OSWorld-Verified and 66.2% on Windows Agent Arena, establishing the strongest open-weight results among the systems in our comparison. On OSWorkerBench, it achieves 41.0% strict success and 76.9% progress, improving over its Qwen3.6-27B base model by 17.7 and 24.5 percentage points, respectively. On a 30-task subset of OSWorld-Verified, one demonstration raises the average task score from 40.3% to 65.8%. We observe similarly substantial gains on the 33-task OSWorkerBench self-demo subset, introduced in this work, where progress increases from 67.9% to 81.1% and strict success from 17.2% to 35.4%. These results support a central empirical finding: demonstrations improve not only average task completion, but also the consistency with which an agent realizes underspecified user intent.

In summary, this report makes three primary contributions:

- **Environment-grounded data and training.** We develop a closed-loop pipeline that jointly constructs tasks and environments, filters trajectories, diagnoses capability coverage, and produces supervision for both SFT and online RL.
- **Demonstration-guided computer use.** We introduce an in-context learning formulation that converts multimodal human or agent demonstrations into adaptive subtask-level workflows, preserving procedural intent without reducing execution to rigid replay.
- **Benchmark and empirical evidence.** We introduce OSWorkerBench and its controlled paired protocol, and show that UI-Mate advances open-weight general computer use while demonstrations substantially improve long-horizon execution reliability.

2 Overview

2.1 Task Formulation

General computer use. We define a computer-use task as a pair

$$\mathcal{T} = (x, \mathcal{E}), \tag{1}$$

where x is a natural-language instruction and $\mathcal{E}$ is a computer-use environment: a machine with an operating system, installed applications, and user files, which the agent operates directly.

The agent interacts with $\mathcal{E}$ in discrete decision steps. At step t it receives an observation o_t, the screenshot of the screen after the previous action has been executed. It then produces one response

$$y_t = (r_t, a_t) \sim \pi_\theta(\cdot \mid x,\, h_t,\, o_t)\,, \tag{2}$$

where r_t is intermediate reasoning, a_t is the action to execute, and h_t is the interaction history. We keep h_t to a bounded window of recent responses, so the input does not grow with the length of the task. We write $c_t = (x, h_t, o_t)$ for this step context.

A single response may carry more than one action,

$$a_t = \left(a_t^{(1)}, \ldots, a_t^{(K_t)}\right), \qquad a_t^{(k)} \in \mathcal{A}, \tag{3}$$

where $\mathcal{A}$ is the action space. The K_t actions run consecutively, and the agent receives no new observation until they finish. One response is therefore one *decision turn*, and we measure interaction length in decision turns rather than in individual operations.

Execution ends when the agent emits a terminal action or exhausts its step budget. The resulting trajectory is

$$\tau = (x,\, o_1,\, y_1,\, \ldots,\, o_T,\, y_T). \tag{4}$$

Whether the task succeeded cannot be read off τ alone. It is decided by an executable verifier that inspects the final state of $\mathcal{E}$ and returns a binary outcome $R(\tau) \in \{0, 1\}$. The agent optimizes $\mathbb{E}\left[R(\tau)\right]$.

Demonstration-guided computer use. A demonstration-guided task additionally provides a demonstration d of how the task, or a closely related one, has been carried out before. We do not consume d as a flat action sequence; it is segmented into an ordered set of subtasks

$$d = (s_1, \ldots, s_N), \qquad s_n = \big(\ell_n,\, v_n,\, u_n\big), \tag{5}$$

where ℓ_n is a natural-language subtask goal, v_n is a verifiable completion criterion for it, and $u_n = \big(u_n^{(1)}, \ldots, u_n^{(M_n)}\big)$ is the ordered sequence of action descriptions performed inside that subtask. Every $u_n^{(m)}$ is a textual description of an operation together with the visual cues that identify its target; it carries no pixel coordinates and is therefore advisory rather than executable. At each step the agent is shown only the part of d that is currently relevant. A pointer $n_t \in \{1, \ldots, N\}$ tracks the active subtask, and the agent conditions on the *workflow view*

$$g_t = \Phi(d, n_t) = \big(\underbrace{\ell_1, \ldots, \ell_N}_{\text{progress checklist}},\, \ell_{n_t},\, v_{n_t},\, u_{n_t}\big), \tag{6}$$

i.e. the goals of all subtasks annotated with completed / current / upcoming markers, but the detailed action steps of the active subtask only. The policy becomes

$$y_t = (r_t, a_t) \sim \pi_\theta(\,\cdot \mid x,\, h_t,\, o_t,\, g_t)\,, \tag{7}$$

and the action space is extended with one control action, $\mathcal{A}^+ = \mathcal{A} \cup \{\textsc{subtask\_complete}\}$. The agent emits $\textsc{subtask\_complete}$ once o_t satisfies v_{n_t}, which advances the pointer,

$$n_{t+1} = \begin{cases} \min(n_t + 1,\, N), & a_t = \textsc{subtask\_complete}, \\ n_t, & \text{otherwise}, \end{cases} \tag{8}$$

so progress through the demonstration is driven by the observed screen state rather than by a fixed step counter. Two properties of this formulation matter for what follows. First, g_t is a *prior*, not a target: u_{n_t} may omit the low-level operations needed on the live screen, may reference elements that are absent, or may be misaligned with the current state, so the correct a_t is the one the observation o_t demands and the observation retains veto power over the demonstration. Second, the objective is unchanged—the verifier still inspects the final state of $\mathcal{E}$ and the agent still maximizes $\mathbb{E}[R(\tau)]$—because the demonstration alters only the conditioning of the policy. General computer use is thus the special case $g_t = \varnothing$, which occurs whenever no useful demonstration is available, and a demonstration-guided agent must remain competent in that regime.

2.2 UI-Mate System Overview

UI-Mate delivers five connected components that together support the full lifecycle of a computer-use agent: an environment-grounded data pipeline, a training stack for general computer use, DemoCUA for learning from in-context demonstrations, OSWorkerBench for controlled evaluation, and a unified harness for deployment and evaluation. Together, they connect executable data construction, policy training, procedural adaptation, benchmark evaluation, and real-world execution.

Environment-grounded data pipeline. UI-Mate uses an environment-grounded data pipeline to produce diverse and executable training data. Starting from task instructions, the pipeline constructs the required applications, files, and initial states, executes agent rollouts, and removes invalid or incomplete trajectories. A hierarchical capability tree tracks data coverage and identifies applications, operations, and workflows that remain underrepresented. The pipeline produces verified trajectories for supervised fine-tuning and executable task–verifier bundles for online reinforcement learning. Rollout outcomes and capability diagnostics are fed back into task and environment construction, allowing later collection to focus on weak or missing capabilities. In §3 we describe the pipeline in detail.

Training for general computer use. Built on the outputs of the data pipeline, the general computer-use training stack combines supervised fine-tuning with agentic reinforcement learning. Supervised fine-tuning teaches the

interaction protocol, visual grounding, application workflows, and multi-step execution, while reinforcement learning allows the policy to act in executable environments and learn from verifier scores assigned to complete trajectories. Together, these stages produce a policy that can plan over long horizons, track cross-application state, recover from errors, and complete tasks from natural-language instructions. In §4 we present the training method.

DemoCUA. DemoCUA enables the general policy to learn user-specific procedures from in-context multimodal demonstrations. A recorded human execution or successful agent rollout is converted offline into a structured workflow containing subtasks, completion criteria, and visually grounded action descriptions. During execution, the model receives the current workflow together with the live screenshot and interaction history. Because the screenshot remains the primary source for action selection, the model can follow useful demonstrated steps, skip unnecessary ones, and revise the plan when the target task differs from the recording. Training examples cover cases in which the demonstration agrees with, diverges from, or is unrelated to the current interface state, teaching the model to use the workflow as guidance rather than as a fixed script.

OSWorkerBench. OSWorkerBench evaluates both general computer-use ability and the ability to use a procedure supplied by a demonstration. It contains 100 long-horizon office tasks across 41 applications, all available for instruction-only evaluation. Its demonstration-guided mode has two distinct settings: 33 targets have self-demos obtained from successful strong-agent rollouts on those same tasks, while 45 targets have human-recorded variant-demos from related but non-identical tasks. Within either paired protocol, the target is evaluated with and without its demonstration under the same instruction, environment, interaction budget, and executable verifier. The performance difference therefore isolates the value of demonstration availability from the model's instruction-only ability, while only the variant-demo setting measures transfer across task variants. The benchmark reports both strict task success and partial progress on long workflows. In §6 we describe its design and construction.

Unified harness for deployment and evaluation. A shared harness connects the trained policy to executable computer environments during both deployment and evaluation. It manages the observation–reasoning–action loop, interaction history, model invocation, action execution, and trajectory recording, while platform-specific adapters support local desktops, virtual machines, and sandbox environments. UI-Mate App uses this harness for general and demonstration-guided execution, demonstration recording, and user oversight of the execution trace. The evaluation system uses the same interaction format in controlled environments and attaches executable verifiers to completed trajectories. In §8 we describe UI-Mate App and its deployment.

The remainder of this report presents these components in turn, covering the environment-grounded data pipeline, general computer-use training, DemoCUA, OSWorkerBench, and the shared harness for evaluation and deployment.

3 Data Pipeline

3.1 Task Instructions

Task instructions for rollout and training general computer-use agent must satisfy two requirements that pull in different directions: they should reflect what people actually do on a computer, and they should systematically cover the functional surface an agent is expected to operate. We meet both with four complementary sources. Open-source computer-use datasets, including AgentNet [1] and ScaleCUA [12], provide a broad foundation of everyday tasks derived from real user activity. Atomic subtasks decomposed from failed or stalled rollouts concentrate the distribution on operations that agents demonstrably find difficult. Instructions generated from real documents, spreadsheets, presentations, and static websites [7] reference concrete entities and relationships rather than synthetic placeholders, and support long-horizon workflows that span multiple applications. Finally, capability trees built from application specifications surface fine-grained operations that common workflows would otherwise leave untouched. We deliberately bias the distribution toward everyday office use by incorporating both tasks performed by real users and tasks derived from authentic working materials. This combination promotes diversity and realism while ensuring systematic coverage of the capabilities required for GUI agents. The source of the task instructions is detailed in Appendix A.

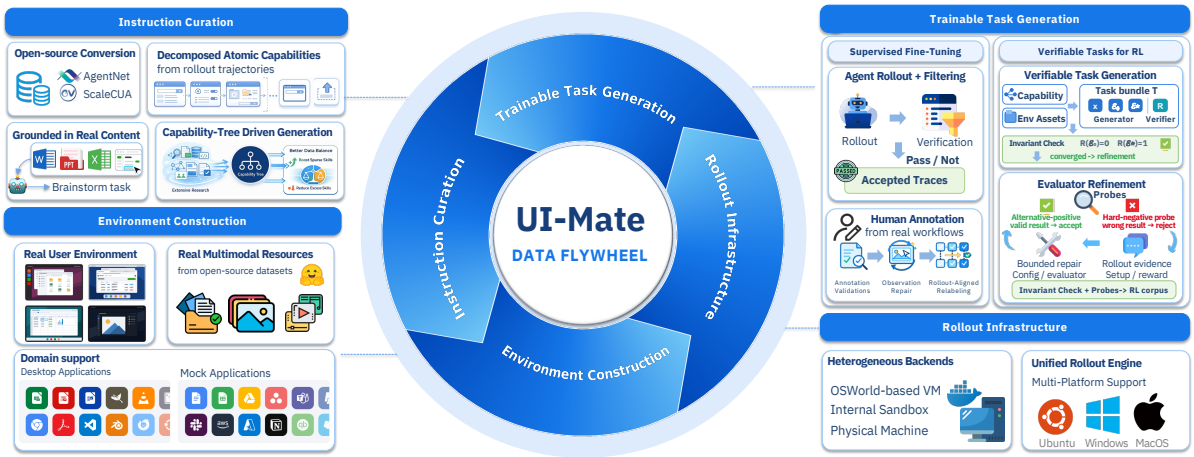


Figure 2 Overview of the UI-Mate data flywheel. The pipeline jointly scales instruction curation, environment construction, trainable task generation, and rollout infrastructure. SFT data combines filtered agent rollouts with validated and repaired human trajectories; RL data is built from verifiable task bundles whose evaluators are refined using complementary probes and rollout feedback. Coverage and outcome diagnostics are fed back to rebalance capabilities and continuously improve the diversity, difficulty, and reliability of the training distribution.

3.2 Environment Construction

From Instruction to Runnable Environment We design an automated pipeline that converts each instruction into an executable environment. An LLM identifies required files, such as documents, spreadsheets, presentations, or images, and generates executable code to create them when needed. The pipeline uploads the resources and configures the operating system, applications, and task-specific state. This produces the initial setup and environment resources needed for execution. Real user environments vary substantially even when they support the same task. Randomized setup code therefore varies wallpapers, desktop layouts, application settings, and sidebar positions while preserving task feasibility. This diversifies the data and reduces dependence on incidental visual or interface configurations.

Grounding Environments with Real Resources LLM-generated resources are often shorter and more homogeneous than real files and may contain placeholders. These artifacts create shortcuts that distort realism and difficulty. We therefore index open-source documents, presentations, spreadsheets, images, videos, and audio. The indexed files better reflect the richness and irregularity of real working materials. During construction, the LLM retrieves and copies a real file through setup code, using synthesis when none exists. Tasks already grounded in real files or static websites use those resources directly. This approach simplifies access to realistic materials while reducing synthetic artifacts.

3.3 Rollout and Filtering

3.3.1 Rollout Infrastructure

Our rollout infrastructure is designed to execute large numbers of heterogeneous computer-use tasks efficiently and reliably. It supports Ubuntu, Windows, and macOS environments through a unified rollout interface, while accommodating the different virtualization and deployment constraints of each operating system. To support parallel rollouts and complex reinforcement-learning environments, we develop an internal cloud virtual machine backend optimized for lightweight environment provisioning and high-throughput parallel execution. It exposes programmable interfaces for environment creation, interaction, observation, and teardown, allowing workers to be allocated dynamically while maintaining isolation between concurrent rollouts. It also manages the complete rollout lifecycle, including resource upload, environment setup, agent interaction, observation collection, and reward computation. To sustain high-throughput data collection and reinforcement learning, rollouts are scheduled independently onto available workers, allowing large numbers of long-horizon trajectories to proceed concurrently across heterogeneous machines and backend types.

3.3.2 Trajectory Filtering

The filtering of the rollout trajectories undergoes two stages, including task and environment validity checks and step-level outcome filtering. For the first stage, a multimodal judge will check the setup configuration as well as the initial environment state, and retain a trajectory only when its task and environment are valid, and rollout evidence supports every expected deliverable. Before assessing agent behavior, the judge rejects ambiguous or infeasible tasks, tasks inconsistent with the setup or already satisfied in the initial state, and trajectories with malformed actions, missing visual observations, or environment failures. This prevents task and infrastructure defects from entering supervision. For the second step-level outcome verification stage, the judge extracts independently verifiable deliverables from each instruction and tracks supporting evidence across GUI observations and actions. It retains a trajectory only when every deliverable is supported. Tracking evidence throughout the rollout prevents plausible final states or unsupported agent narration from masking partial failure and captures intermediate evidence absent from the final frame.

3.4 Capability Tree and Data Diagnosis

Large-scale data production favors inexpensive tasks, while application-level statistics obscure underrepresented behaviors. We therefore map each task to a shared capability taxonomy and associate its rollout outcomes with the corresponding capabilities, making coverage gaps actionable for subsequent data generation.

3.4.1 Extracting Capabilities

For each application, we maintain an inventory of atomic user-facing capabilities. We initialize it from application documentation and expand it with behaviors found in training instructions. During annotation, we match each instruction to an existing entry and create a new one only when no suitable match exists. We maintain a separate cross-application domain for behaviors that connect applications, such as transferring and transforming information between them.

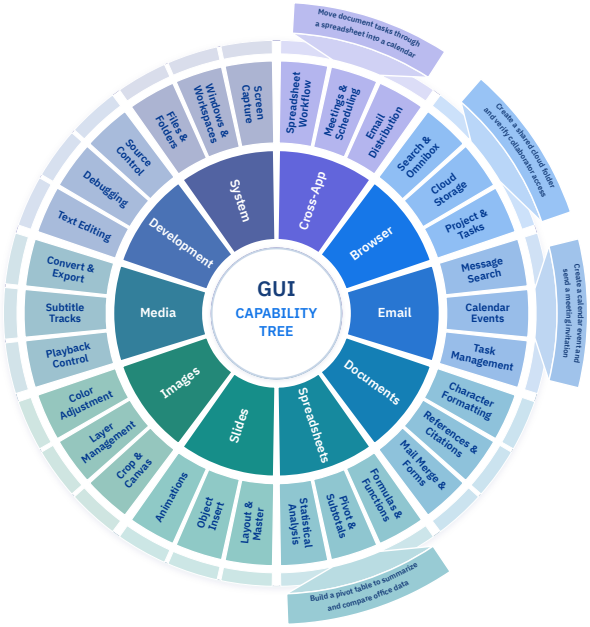


Figure 3 Representative subset of the collected GUI capability tree. The hierarchy proceeds from application domains to fine-grained capabilities and representative workflows; sector sizes are schematic.

3.4.2 Constructing the Capability Tree

As a flat inventory grows, matching becomes ambiguous and data budgets become difficult to allocate consistently. We therefore organize capabilities into three levels: application, coarse capability, and fine-grained operation. Each task is first routed to a coarse capability and then matched within the corresponding subtree, reducing confusion among near-duplicates. We keep the coarse level stable for budgeting, add fine-grained nodes as needed, and periodically consolidate redundant or inactive nodes. Equivalent entries are aligned across applications to expose shared interaction patterns.

3.4.3 Data Rebalancing

We rebalance the corpus using target coverage, observed data density, rollout success, and filtering rejection rates. Low density relative to the target triggers additional generation, whereas oversupplied capabilities are down-weighted. Sufficient volume combined with low rollout success or high rejection instead signals task or environment defects. We treat task length as a separate sampling dimension to prevent abundant short tasks from masking long-horizon gaps. Newly generated tasks pass through the same annotation and evaluation pipeline, and their outcomes update the next allocation cycle. This closes the loop between capability diagnosis and data generation.

3.5 Human Annotation

Automated rollouts provide scale, but remain biased toward tasks that current agents can execute and environments that are inexpensive to instantiate. We therefore complement them with human-annotated trajectories from real workflows across operating systems and applications, capturing natural interaction patterns and long-tail behavior underrepresented in synthetic data. Because raw annotations may contain inconsistent labels, recording artifacts, and supervision that differs from the rollout format, we subject them to validation, observation repair, and aligned relabeling.

Annotation Validation and Observation Repair Deterministic checks verify structural integrity and action–parameter agreement, while multimodal review assesses task completion and whether each action matches the visible state transition. We pay particular attention to observation timing: a pre-action screenshot may leak the target through cursor or hover state, or capture an unstable interface transition. Affected steps are repaired by selecting an earlier buffered frame for target leakage or a later one for incomplete rendering, with time-bounded recovery from the source recording when needed. Repaired frames are revalidated; only trajectories with irreparable corruption or annotation-tool contamination are removed. This repair-first policy preserves costly supervision without admitting visual shortcuts or inconsistent actions.

Rollout-Aligned Relabeling Human recordings provide action types and arguments but not the reasoning and natural-language action descriptions expected by the policy. A teacher model adds them using the same system prompt, interaction history, observation folding, and action space as model rollout. The outputs are generated jointly and sequentially, while all human-annotated action parameters remain fixed. Consistency checks reject action mismatches and privileged annotation leakage, yielding the same decision-step representation without altering the executable human supervision.

3.6 Verifiable Tasks for Reinforcement Learning

Reinforcement learning from verifiable rewards (RLVR) requires more than instructions and runnable environments: every rollout must terminate in a verifiable signal. Authoring such tasks manually is expensive, because the instruction, the configuration, and the reward must agree on the same initial conditions and completion criteria. We therefore automate construction in two stages: *generation* turns capability and environment assets into diverse, self-contained task bundles, and *refinement* independently stress-tests and repairs the evaluators. Both stages operate on a common data contract. Beyond the instruction x and environment $\mathcal{E}$ of Equation 1, a task usable for RL fixes the initial state $\mathcal{E}_0$ from which rollouts begin, a reference completion state $\mathcal{E}^\star$ the generator can reach, and a per-task executable verifier R evaluated on environment state. Every such task must satisfy the execution invariant

$$R(\mathcal{E}_0) = 0, \qquad R(\mathcal{E}^\star) = 1. \tag{9}$$

Generation uses this invariant for internal consistency; refinement asks whether the evaluator faithfully captures the instruction rather than merely agreeing with the reference.

3.6.1 Verifiable Task Generation

Scaling instructions and executable configurations. We reuse the instruction sources and environment-construction machinery described in §3.1 and §3.2, but couple them more tightly for RL. Capability-guided sampling controls which behaviors are exercised and how they are composed, while configuration synthesis instantiates the resources, preconditions, and application state required to execute each instruction. The same capability can therefore be realized under different files, user contexts, application states, and system configurations, increasing interaction diversity rather than merely producing paraphrases. Conversely, complex instructions retain the environmental dependencies needed to make them genuinely executable instead of being simplified to what is easiest to initialize or evaluate. This joint expansion of instruction and configuration space is the primary mechanism by which we scale the coverage and difficulty of verifiable tasks.

Decoupled construction of environments and rewards. Inspired by the generator–discriminator formulation of CUA-Gym [8], we separate the roles of realizing a task and judging its outcome. A generator constructs the executable configuration together with the initial and reference completion states, whereas a verifier agent

derives the reward from the task specification. This information boundary keeps evaluation faithful—seeing only the specification and the resulting state, the verifier judges whether the user-visible objective is met, not how the reference completion arose. We further extend this separation with application-aware configuration synthesis: the state that determines task success may live in different places depending on the application — user artifacts, application profiles, structured stores (e.g., databases), or operating-system state — and the generator resolves this per task while exposing all such state to the verifier through a single, common evaluator interface. Co-designing configuration observability and reward semantics makes verifiability a construction-time constraint, rather than an annotation added after rollout collection.

Generation-time convergence checks. A bundle is retained only once its components converge on the same task semantics. Static and model-based checks reject unsupported preconditions, inconsistent artifacts, weak proxy rewards, and configurations that do not expose the state required for evaluation; execution then tests the invariant directly. This removes malformed or trivially satisfied tasks cheaply, but establishes only internal consistency, since the reference completion and the reward can share a semantic blind spot—making convergence a prerequisite for, not a substitute for, independent refinement.

3.6.2 Evaluator Refinement

Independent probes beyond self-consistency. Refinement takes converged bundles as read-only inputs and re-examines instruction–reward alignment along a reasoning path independent of generation. We first make evaluator decisions observable by decomposing a scalar pass/fail result into per-requirement diagnostics, then apply two complementary probes. A *hard-negative* probe perturbs a successful state into a plausible but incorrect completion and should be rejected; acceptance indicates an under-specified reward. An *alternative-positive* probe constructs a different legitimate completion and should be accepted; rejection indicates an over-strict reward. With the original initial and reference states, the probes provide evidence about both false acceptance and false rejection instead of validating a single solution.

Rollout feedback and bounded repair. Real rollouts expose discrepancies that generation cannot anticipate: the configuration may instantiate an unintended state, a precondition may not survive initialization, a valid outcome may be represented differently by the application, or the reward may respond to a shortcut. Flagged tasks enter a bounded repair loop rather than being discarded; repairs may target the configuration, reference state, evaluator, or—when the objective itself is ambiguous—the instruction. Every modification must pass the invariant and the probes again, and changes that improve one signal while degrading evaluator observability or task compliance are reverted. Only tasks free of hard setup defects and supported by both positive and negative evidence are promoted to the RL corpus.

Scaling as a coupled design objective. These components make scaling a coupled objective rather than three independent throughput problems: added generation capacity contributes new capabilities, interaction contexts, and difficulty instead of paraphrases, easy-to-verify tasks, or noisy rewards, broadening the frontier of the training distribution while preserving the executability and reliability RLVR requires.

4 Training UI-Mate for General Computer Use

4.1 Training Recipe

UI-Mate is trained in two stages: supervised fine-tuning (SFT), followed by agentic reinforcement learning (RL). SFT learns from filtered offline trajectories (§3) and equips the model with capabilities for GUI interaction: producing well-formed and executable responses under the interaction protocol, grounding semantic targets to screen coordinates, and selecting actions based on screenshots and interaction history.

Agentic RL improves the model through online interaction in the verifiable environments (§3.6). It uses task-level rewards to optimize for successful task completion rather than action imitation. This stage improves long-horizon planning, state tracking, error recovery, and reliable task completion.

4.2 Supervised Fine-Tuning

Training data. Our SFT corpus is constructed using the data pipeline described in §3. For each task instruction, we instantiate a runnable environment (§3.2) and execute an agent rollout using the infrastructure (§3.3.1). We retain only trajectories for which step-level verification confirms all required outcomes (§3.3.2). The retained trajectories are sampled according to the capability tree in §3.4, with balanced coverage across applications, capability levels, and task lengths. The resulting training mixture covers atomic operations, single-application workflows, and cross-application tasks, and includes trajectories both with and without explicit intermediate reasoning [1].

Objective. Each training instance is a single decision turn of a retained trajectory. Given the instruction x, the interaction history h_t carried into turn t, and the current screenshot o_t, the model reproduces the target response y_t, which contains its reasoning followed by the action to execute. Writing $c_t = (x, h_t, o_t)$ for the turn context, SFT minimizes the next-token prediction loss over the response tokens,

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\,\mathbb{E}_{(c_t, y_t)\sim\mathcal{D}_{\mathrm{SFT}}}\left[\frac{1}{|y_t|}\sum_{k=1}^{|y_t|}\log \pi_\theta(y_{t,k} \mid y_{t,<k},\, c_t)\right], \tag{10}$$

where $y_{t,k}$ is the k-th token of the response and $|y_t|$ its length. The turn context carries no loss and acts purely as conditioning.

4.3 Agentic Reinforcement Learning

Starting from an SFT policy, we optimize UI-Mate online in the verifiable GUI environments described in §3.6, using the decision-turn interaction defined in §2.1. Each completed trajectory is scored by an environment verifier. Figure 4 summarizes the overall RL pipeline, which combines adaptive task sampling and grouped online rollouts with verifier-based credit assignment and asynchronous GRPO updates. Decision-turn centering and token-level normalization form the outcome-only credit-assignment path, while an optional *Process Credit Model* (PCM) further localizes verifier-derived credit and provides process-level diagnostics.

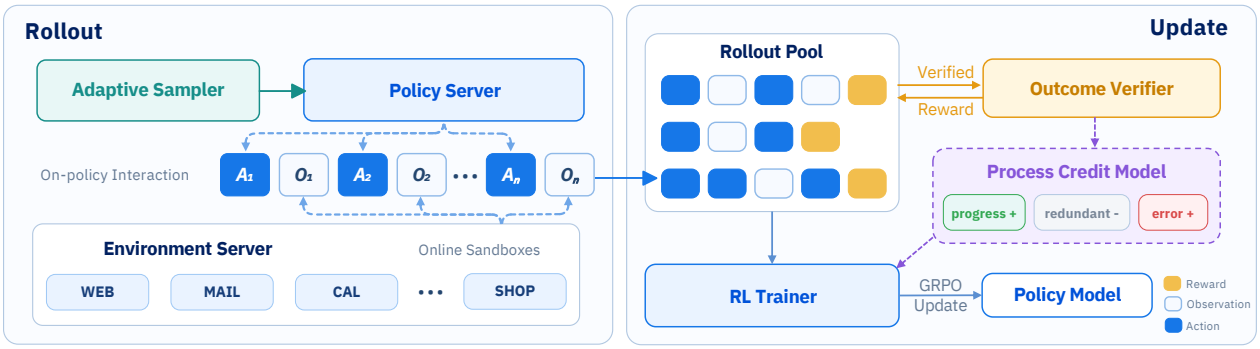


Figure 4 Agentic RL system of UI-Mate. Our RL Pipeline can be divided into rollout stage (left) and update stage (right). Adaptive sampler drives online sandbox rollouts under a fixed policy snapshot; completed trajectories are pooled per task group for outcome verification, optional process credit assignment, and asynchronous GRPO updates.

4.3.1 Online GUI RL Preliminaries

Grouped online interaction. Following §2.1, let $\mathcal{T} = (x, \mathcal{E})$ denote a selected verifiable task. At decision turn t of trajectory i, the rollout-policy snapshot $\pi_{\bar{\theta}}$ samples

$$y_{i,t} = (r_{i,t}, a_{i,t}) \sim \pi_{\bar{\theta}}(\cdot \mid c_{i,t}), \qquad c_{i,t} = (x, h_{i,t}, o_{i,t}). \tag{11}$$

Each response $y_{i,t}$ is one decision turn even when $a_{i,t}$ contains multiple consecutive operations. A trajectory terminates when the model emits a terminal action or reaches the interaction budget. To construct a rollout group, we use the same fixed policy snapshot to run the selected task in multiple independent environment

instances. Each run produces one complete trajectory. We retain both successful and failed task executions, discarding only rollouts corrupted by environment or infrastructure failures. The remaining N trajectories form the rollout group $\mathcal{G} = \{\tau_i\}_{i=1}^{N}$. Because they share the same task and policy snapshot, their outcomes can be compared directly.

Group-relative outcome supervision. The executable verifier for the selected task inspects the final environment state and returns a binary outcome $R_i = R(\tau_i) \in \{0, 1\}$, where 1 indicates task success and 0 indicates failure. We adopt Group Relative Policy Optimization (GRPO) [13, 14], which estimates relative advantages by comparing outcomes within the rollout group $\mathcal{G}$, without requiring a learned value model. Groups in which all trajectories receive the same outcome are excluded because they provide no relative learning signal [15].

4.3.2 Trajectory-to-Token Credit Assignment

A verifier supplies one outcome per trajectory, while the actor is optimized over tokens produced across many decision turns. Applying the same trajectory-level GRPO advantage to every decision turn raises two issues. First, under outcome-only supervision, trajectories within the same group can contain different numbers of decision turns. Failed trajectories often contain more turns because they involve more trial and error. Broadcasting the same advantage to every turn therefore gives longer trajectories more weight, resulting in a decision-turn bias. We address this bias with decision-turn centering. Second, even after centering, an outcome alone does not reveal which steps are critical to success or failure; PCM uses process evidence to identify these steps and reweight their learning signals accordingly.

Decision-turn centering. To correct the decision-turn bias, we compute the group baseline under the decision-turn measure:

$$\mu^{\text{turn}} = \frac{\sum_{i=1}^{N} T_i R_i}{\sum_{i=1}^{N} T_i}, \qquad A_{i,t}^{\text{base}} = R_i - \mu^{\text{turn}}, \tag{12}$$

where T_i is the number of decision turns in τ_i and $t \in \{1, \ldots, T_i\}$. The resulting advantage is constant within each trajectory but is indexed by decision turn. This approximately centers the advantage under the decision-turn measure to make sure $\mathbb{E}[A] \approx 0$. Within each task group, we center only the mean and do not divide by the group standard deviation. This preserves the reward scale and avoids unstable amplification when group outcomes are nearly uniform.

Trajectory-merge process credit. The outcome-only objective can optimize the policy directly from the final reward R_i, but the outcome does not reveal which decision steps were critical to success or failure. As a result, every step in the same trajectory receives the same advantage. Building on process supervision [16–18], we optionally apply PCM during RL training. When enabled, PCM uses teacher annotations to identify critical steps and reweight their learning signals accordingly. PCM only redistributes the learning signal across decision steps and does not change the final task-level reward R_i. Without PCM, training follows the outcome-only objective without step-level reweighting.

The teacher first extracts observable milestones from verified successful trajectories and merges equivalent subgoals and alternative valid branches into a shared task structure. This structure is then used to annotate every trajectory. For each failed trajectory, the teacher selects the closest valid branch and labels each milestone as `completed`, `attempted_failed`, `not_attempted`, or `skipped_by_branch`. The last two labels prevent blocked or irrelevant subgoals from being treated as policy failures.

At the trajectory level, failures are classified as policy-related, external, mixed, or unclear. At the decision-turn level, annotations identify progress, causal errors, recovery, and redundancy. Related errors are grouped to distinguish their root causes from downstream symptoms. If a task group contains no verified successful trajectory, PCM is not applied and training follows the outcome-only path.

Let $e_{i,t}$ denote the resulting process evidence for decision step t. PCM converts this evidence into a nonnegative importance weight:

$$w_{i,t} := \text{clip}\big(b + \phi\big(e_{i,t}, \text{sgn}\big(A_{i,t}^{\text{base}}\big)\big), 0, w_{\max}\big), \qquad \widetilde{w}_{i,t} := \frac{w_{i,t}}{\frac{1}{T_i}\sum_{u=1}^{T_i} w_{i,u}}. \tag{13}$$

Here b and $w_{\max}$, with $0 < b \leq w_{\max}$, set the base and maximum weight scales. The mapping ϕ rewards steps that make progress or recover from mistakes in successful trajectories. In failed trajectories, it penalizes the steps that introduce an error and the later steps that fail to correct it. Redundant steps and steps blocked by the environment are also penalized. We normalize the resulting weights within each trajectory so that their average is one. If all weights become zero after clipping, we assign equal weights to all steps.

A sign-aware selector $m_{i,t} \in \{0, 1\}$ retains decision turns with positive evidence or causal responsibility for failure. PCM produces

$$A_{i,t}^{\text{proc}} := m_{i,t}\, A_{i,t}^{\text{base}}\, \widetilde{w}_{i,t}\, s_i, \tag{14}$$

where $s_i \in [0, 1]$ discounts failures only partially attributable to the policy and equals one otherwise. With valid annotations, $A_{i,t}^{\text{credit}} = A_{i,t}^{\text{proc}}$; otherwise, $A_{i,t}^{\text{credit}} = A_{i,t}^{\text{base}}$. Nonselected responses remain as context but carry no actor loss. PCM thereby concentrates learning on informative decisions and provides process-level data insights.

Token-level normalization. Beyond decision-turn-level credit assignment, varying response lengths introduce another source of bias because the actor objective is computed over generated tokens. Failed trial-and-error turns often require a larger thinking budget and therefore contain more response tokens. Their advantages are consequently repeated over more tokens and can disproportionately affect the token-level objective. We address this response-length bias by normalizing the advantages over all response tokens in the optimization batch [19]. This aligns the advantage statistics with token-level aggregation, controls the scale of policy updates, and further improves training stability.

4.3.3 Asynchronous Group-Relative Optimization

Synchronous training waits for all trajectories in a rollout group to finish. This is inefficient because GUI rollouts vary widely in duration across environments and task horizons [20]. Instead, rollout workers use the latest available policy snapshot, and the learner starts optimization as soon as enough complete trajectories from the same task group are buffered. We never truncate individual trajectories, and all trajectories compared within a group share the same task configuration and policy snapshot.

Asynchronous training introduces policy staleness because the rollout policy may lag behind the current learner policy. For compactness, let $y = (y_1, \ldots, y_{|y|})$ denote a generated response, $y_{<k}$ its prefix before token k, c its turn context, and $\bar{\theta}$ the rollout-policy snapshot that generated it. We measure the train–rollout mismatch at token k by

$$\rho_k(\theta) = \frac{\pi_\theta(y_k \mid y_{<k}, c)}{\pi_{\bar{\theta}}(y_k \mid y_{<k}, c)}. \tag{15}$$

where ratios $\rho_k(\theta)$ far from one indicate substantial train–rollout mismatch. We apply IcePop [21] and SeqClip [22] as complementary mismatch filters. IcePop rejects isolated tokens with abnormal likelihood ratios, whereas SeqClip evaluates the geometric mean of $\rho_k(\theta)$ over the response to detect coherent policy drift. Let $M_k \in \{0, 1\}$ indicate that token k passes both filters. The resulting PPO-style clipped actor objective [23] is

$$\mathcal{L}_{\text{RL}}(\theta) = -\,\mathbb{E}_y \left[\frac{1}{|y|} \sum_{k=1}^{|y|} M_k \min(\rho_k(\theta) A_k, \text{clip}(\rho_k(\theta), 1-\epsilon, 1+\epsilon)\, A_k) \right]. \tag{16}$$

Here $\epsilon > 0$ is the PPO clipping radius. Tokens rejected by either mismatch filter have $M_k = 0$ and therefore contribute no actor loss. The mismatch filters determine whether a stale token is eligible for learning, while PPO clipping bounds the contribution of each retained token; the two mechanisms therefore address different aspects of train–rollout mismatch.

4.3.4 Adaptive Curriculum Sampling

As the policy improves, uniform sampling increasingly spends rollouts on solved tasks. We therefore use a domain-level *adaptive curriculum sampler* that combines broad coverage with emphasis on weak application domains [24–28]. It only reallocates rollout computation over the fixed RL corpus; task construction and corpus rebalancing remain in §3.

Base and adaptive allocation. At each training iteration, we select a fixed number of tasks, with each task producing one rollout group. We divide this task budget between base and adaptive sampling. Base sampling follows the domain distribution of the RL corpus to maintain broad coverage. Adaptive sampling allocates its portion to weak domains identified from recent on-policy results. The two components select disjoint tasks, so no task appears twice in the same batch.

Estimating weak domains. We estimate domain performance over a recent rollout window. Only domains with enough valid trajectories and a low environment-error rate are included, preventing infrastructure failures from being mistaken for policy weakness. Let $\mathcal{D}_{\text{elig}}$ denote these eligible domains, and let V_d and S_d be the numbers of valid and successful trajectories in domain d. We compute the domain success rate $\widehat{p}_d$ and the overall success rate $\bar{p}$ as

$$\widehat{p}_d = \frac{S_d}{V_d}, \qquad \bar{p} = \frac{\sum_{d \in \mathcal{D}_{\text{elig}}} S_d}{\sum_{d \in \mathcal{D}_{\text{elig}}} V_d}. \tag{17}$$

A domain is considered weak when $\bar{p} - \widehat{p}_d \geq \delta$, where $\delta > 0$ is the minimum required success-rate gap.

We activate adaptive sampling only when at least two domains have reliable estimates and at least one weak domain is identified. Once these weak domains are identified, the adaptive component samples tasks from them without replacement, subject to a per-domain cap. When candidate tasks have similar priority, we prefer tasks whose recent rollout groups contain both successful and failed trajectories, since these partially learned tasks provide informative group-relative signals. If the statistics are insufficient, no weak domain is identified, or a weak domain runs out of eligible tasks, the unused adaptive budget is reassigned to base sampling.

5 DemoCUA: Learning from In-Context Demonstrations

5.1 Demonstration Representation

How to Get the Demo. A demonstration is a recorded successful desktop execution, with every mouse and keyboard action and a screenshot before and after each one. Its source depends on the evaluation setting: a *self-demo* may be a successful rollout from a stronger GUI agent on the same task, whereas a *variant-demo* is recorded by a human on a related but non-identical task. The raw trace is normalized into a uniform action-and-frame format, then annotated offline by a vision-language model along four axes: screen state, the intent, the action taken, and how its target was located visually. The annotated trace is finally segmented into a few coherent subtasks, each with a short goal and an explicitly checkable completion criterion. Figure 5 summarizes this offline pipeline and how the resulting demonstration is consumed online at every step.

How to Use the Demo. A demonstration serves as a guide rather than a script to replay. Before each model call, the harness provides a short summary of completed, current, and upcoming subtasks, along with key goals and milestones. It omits pixel coordinates and low-level actions, so the agent must rely on the live screenshot and can adapt when the interface changes. The agent marks each subtask as complete before moving to the next. Figure 6 shows a complete example and the context provided to the agent.

5.2 Training with Demonstration

Data Generation. We build a three-stage pipeline to generate both self-demo (i.e. identical tasks) and variant-demo (i.e. similar tasks) data, consisting of Rollout, Score, and Filter & Repair. (1) The Rollout stage collects observation-to-action trajectories on perturbed AgentNet data [1]. Demo-guided prompting and a subtask-report protocol are used to improve rollout success. (2) The Score stage evaluates each rollout using a hybrid mechanism: a VLM judge assesses semantic completion, while rule-based metrics verify format validity. Each rollout is scored at three levels—trajectory, subtask, and demo-following—as detailed in Table 8. (3) The Filter & Repair stage retains data that meet task-specific quality thresholds. Instead of discarding trajectories with fixable errors, it recovers them through rule-based and VLM-based repair specified in Table 9, improving data efficiency.

Demo-Augmented Training Data Format. Each training sample represents one decision step in a trajectory.

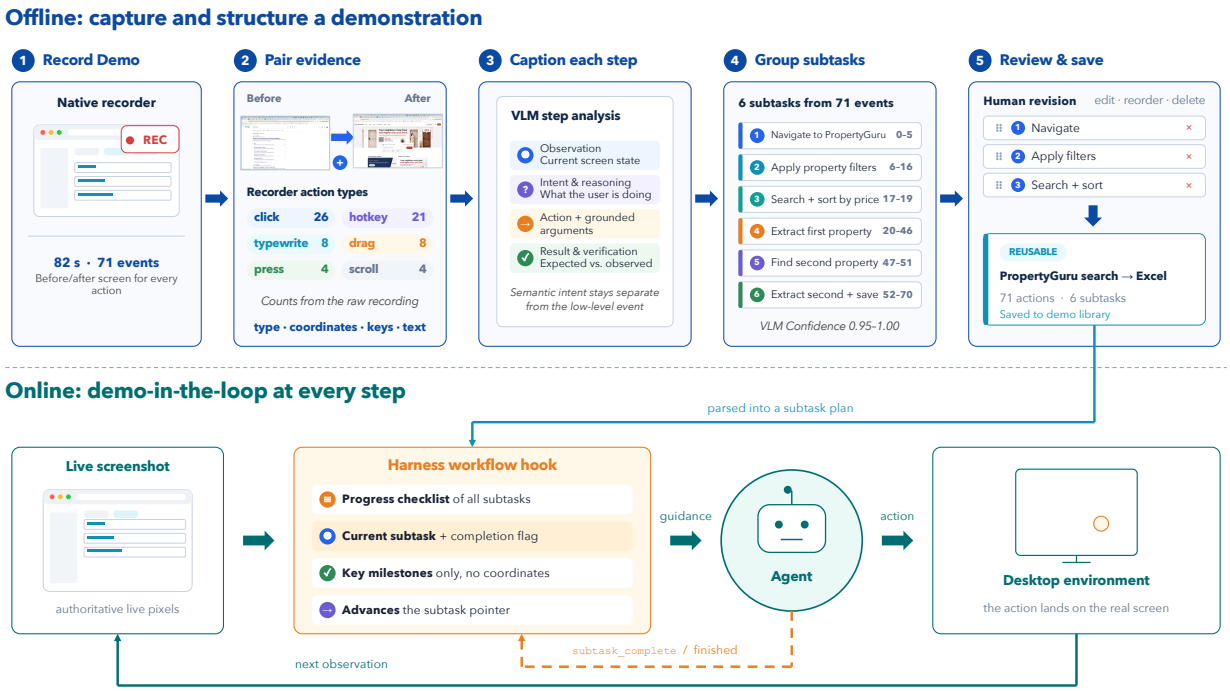


Figure 5 Demonstration representation. Offline (top), a human recording is annotated and segmented into subtasks with goals and verifiable completion criteria. Online (bottom), the agent follows the current subtask using live screenshots and advances upon completion.

Figure 7 shows two changes to our standard CUA format. First, the system prompt adds a demonstration workflow contract and a new `computer_use` action, `subtask_complete`. Second, the first user turn includes a snapshot of the demonstration workflow at the supervised step. This snapshot contains the subtask checklist and its progress, the current subtask and its completion criterion, and its ordered action steps (see Figure 6 for an example). The remaining format is unchanged. Assistant turns retain the `<think>`/`<action>`/`<tool_call>` structure, while subsequent user turns contain only screenshots. Thus, the multi-turn format remains identical to the demonstration-free baseline. When a subtask is completed, the agent emits `subtask_complete` after observing the post-action screenshot. Each episode ends with an explicit `finished` turn.

Demo-Augmented Training Data Category Ratios. Because the demonstration workflow and target query are closely aligned, the model may simply copy the next workflow step without checking the screenshot. This shortcut fails when the workflow and live interface diverge. To prevent this shortcut, we introduce three types of demonstration-workflow–screen relationships: (1) full-alignment, where the model follows the workflow; (2) partial-misalignment, where the model corrects mismatched steps using the screenshot; (3) irrelevance, where it ignores the workflow and acts from the screenshot alone. Note that full-alignment remains the majority case so that the workflow still provides useful guidance.

Training from Incomplete Demonstration Workflows to Prevent Shortcut Learning. We keep the full trajectory as the supervision target, but show only key actions in the demonstration workflow. Intermediate actions, such as focus clicks, scrolling, and popup dismissal, are omitted. Thus, even in full-alignment cases, the model cannot simply copy the workflow; it must infer the missing actions from the screenshot. The workflow provides milestones, while the model learns how to reach them.

5.3 Inference with Demonstration

At inference, we provide the subtask's complete action sequence without LLM-based key-action extraction. The action list is more detailed than during training. This mismatch is intentional: omitting actions during training prevents blind copying, while including them at inference provides fuller guidance once the model

A concrete workflow, and where it sits in the context window

Godot "timer and bullet firing" task. The demonstration contains 323 actions and is segmented into 7 subtasks. The episode below took 289 steps in total; the snapshot shown is runtime step 105, at which subtask 4 of 7 is the current one (the agent declared it complete at step 106).

☐ derived from the demonstration ☐ given by the benchmark task ☐ produced at run time by the harness, the agent, or the environment

(a) The context window at runtime step 105

Messages in the order the model sees them. Exactly one turn carries the demonstration; every other turn is ordinary agent history.

system — Base agent system prompt and tool schema, plus a short "# Workflow" section explaining how to read the injected blocks and how to signal that a subtask is done. Identical on every step. — harness, static

assistant — "# Progress so far (earlier steps folded into this summary)": steps 1 to 64 of this episode have been compressed by context folding, so their raw text no longer occupies the window. — folded at run time

user (step 65 (the first turn of the window)) — a screenshot placeholder (this old frame has already been collapsed)

<workflow_progress> ... </workflow_progress>
<current_subtask> ... </current_subtask>
<current_subtask_action_list> ... </current_subtask_action_list>

one guidance line, then `Instruction: the verbatim benchmark task`

the whole demonstration contribution sits in these three blocks

expanded in panel (b) below; rewritten in place at every step to describe the current subtask

assistant / user (steps 65 to 104) — Each past step contributes the model's own previous output followed by a bare tool response holding that step's screenshot. Only the five most recent screenshots remain as images; older frames become a short placeholder. No demonstration text appears on these turns. — agent and environment

user (step 105) — The live screenshot from which the next action must be decided. No coordinates from the demonstration are ever given.

(b) The three blocks of that single turn, in full

Every line below comes from the demonstration, except the done / current / todo markers, which the harness maintains at run time.

<workflow_progress> the subtask checklist — subtasks from demo — markers from run time

```
[done]    subtask 0 : Create the Bullet.tscn scene with an Area2D root, a Sprite2D using Bullet.png and a fitted CollisionShape2D ...
[done]    subtask 1 : Create and attach Scripts/bullet.gd, declare an exported bullet_speed float defaulting to 100.0 ...
[done]    subtask 2 : Set the Bullet Inspector override for bullet_speed to about 300, then open Player.tscn for editing ...
[done]    subtask 3 : Add a Timer child to the Player, autostart, ~1 s wait time, connect timeout to _on_fire ...
[current] subtask 4 : Export a PackedScene bullet_scene in player.gd, implement _on_fire to instantiate the bullet ...
[todo]    subtask 5 : Remove any pre placed Bullet instance from Game.tscn, add a muzzle offset and guard clauses ...
[todo]    subtask 6 : Await a ~3 s SceneTree timer then queue_free in bullet.gd's _ready, save and playtest ...
```

<current_subtask> the goal to work on now, and how completion is judged — from demo

```
index: 4
sub_instruction:    Export a PackedScene bullet_scene variable in player.gd, implement _on_fire to instantiate the
                    bullet at the player's position and add it to the current scene, then bind bullet_scene to Bullet.tscn.
subtask_complete_flag:    player.gd contains the bullet_scene export and _on_fire instantiation logic,
                          and the Inspector shows Bullet.tscn assigned to the Bullet Scene property.
```

<current_subtask_action_list> milestones the human passed through inside this subtask, in order, with no coordinates — from demo

72 milestones in total for this subtask, abbreviated here

```
Key Step  0: Click inside the _on_fire function body to begin implementing the firing logic.
Key Step  2: Scroll up to the export variable section at the top of player.gd.
Key Step  7: Start declaring the bullet scene variable: var bullet_scene: PackedScene.
Key Step 20: Annotate it with @export so it becomes visible in the Inspector.
       ...
Key Step 57: Complete get_tree().current_scene.add_child(bullet_node) so the bullet joins the running scene.
Key Step 63: Select the Player root node in the Scene tree to reach its Inspector.
Key Step 69: Open the resource browser for the Bullet Scene property.
Key Step 71: Select Scenes/Bullet.tscn and confirm the assignment.
```

Takeaways — The demonstration occupies a single user turn, the first one in the window, and is never repeated on later turns. The benchmark task itself is carried verbatim on that same turn. That turn is rewritten in place at every step, so the blocks always describe the current subtask.
The agent takes one action per step from the live screenshot; once the screenshot satisfies the completion criterion it emits a subtask completion signal instead of an action, and the blocks are re-rendered for the next subtask.

Figure 6 A complete demo-workflow for a Godot task in which the agent adds timer-based bullet firing to a simple game. Panel (a) shows the context available to the agent at one point during execution. It includes the original task instruction, one user turn containing the demonstration, and the agent's interaction history. Panel (b) breaks down the demonstration into a subtask checklist, the current subtask and its completion criterion, and key milestones.

```
System :  tools: computer_use(click|type|scroll|... , subtask_complete, finished)
          # Demonstration Workflow  -- contract; the live screenshot is authoritative
User 1 :  <image>
          <workflow_progress>            all subtasks; done / current / upcoming
          <current_subtask>              sub_instruction + subtask_complete_flag
          <current_subtask_action_list>  ordered action steps of THIS subtask only
          Instruction: {task}
```

Figure 7 DemoCUA SFT Format: the demonstration is pinned to the first user turn as a snapshot of the workflow state.

relies on the screenshot. It also removes a model call and extraction errors.

Long-Horizon Context Management. Long episodes can produce interaction histories that exceed the model's context window. We control context growth through proactive folding and reactive truncation. Before each inference call, the harness estimates the request size using approximately one token per three text characters and a fixed token cost for each screenshot. When the estimated usage approaches a configurable threshold, an additional LLM call summarizes the oldest interaction steps into a compact progress note. Recent steps remain verbatim, while the most recent screenshots are retained separately rather than summarized.

Limitations. A current limitation is that the workflow appears at the beginning of the context. Each subtask update therefore invalidates the shared prefix and prevents efficient KV-cache reuse. Future work will move the workflow to the end, allowing the context to grow append-only and reuse the cache throughout the episode.

6 OSWorkerBench: Realistic Cross-Application Office Workflows with Multimodal Demonstrations

Most existing GUI benchmarks evaluate agents under an instruction-only protocol, requiring them to infer and execute a workflow from a natural-language request [10, 11]. While this setting is essential for measuring general computer-use competence, it does not directly test whether an agent can use an example of how a procedure should be carried out. This capability is particularly important for personalized workflows, proprietary or organization-specific applications, and complex processes whose conventions may be absent from public training data and cumbersome to specify completely in text. In such settings, a demonstration can expose both the intended actions and the resulting visual state transitions. To evaluate demonstration-guided execution alongside standard instruction following, we introduce **OSWorkerBench**, *an office-centric benchmark of long-horizon, cross-application workflows in enterprise productivity environments.*

OSWorkerBench comprises 100 realistic office tasks spanning 41 normalized applications and 10 consolidated job families (Figure 9a), all of which support standard instruction-only evaluation. To capture the complexity of real-world office work, we further identify two independently annotated, potentially overlapping capability subsets: 67 **Long-Memory** tasks require delayed reuse of dynamic information or sustained tracking of constraints and workflow state, while 49 **Multi-App** tasks require faithful transfer of dynamic, multi-field information across at least three logical applications. OSWorkerBench supports two evaluation modes: **instruction-only**, in which the target instruction is the only procedural input, and **demonstration-guided**, in which one multimodal demonstration is additionally supplied. The demonstration-guided mode contains two settings with different purposes. The **self-demo** setting pairs 33 targets with successful strong-agent rollouts of those same tasks and is used for the quantitative DemoCUA evaluation in Section 7.3. The more challenging **variant-demo** setting pairs 45 targets with human recordings of semantically related but non-identical source tasks. The numbers 33 and 45 therefore refer to different demonstration collections, not to a partition of the 100 benchmark tasks. The 45 variant-demo targets are deliberately challenging and long-horizon: under instruction-only evaluation, Kimi-2.6 requires more than 100 observation-to-action decision turns per task on average before termination (see Section 7.2.4 for the decision-turn definition and full trajectory analysis). These cases are not demonstration-only tasks: the same targets and evaluators can be used with and without guidance. To the best of our knowledge, **OSWorkerBench is the first CUA benchmark to provide multimodal**

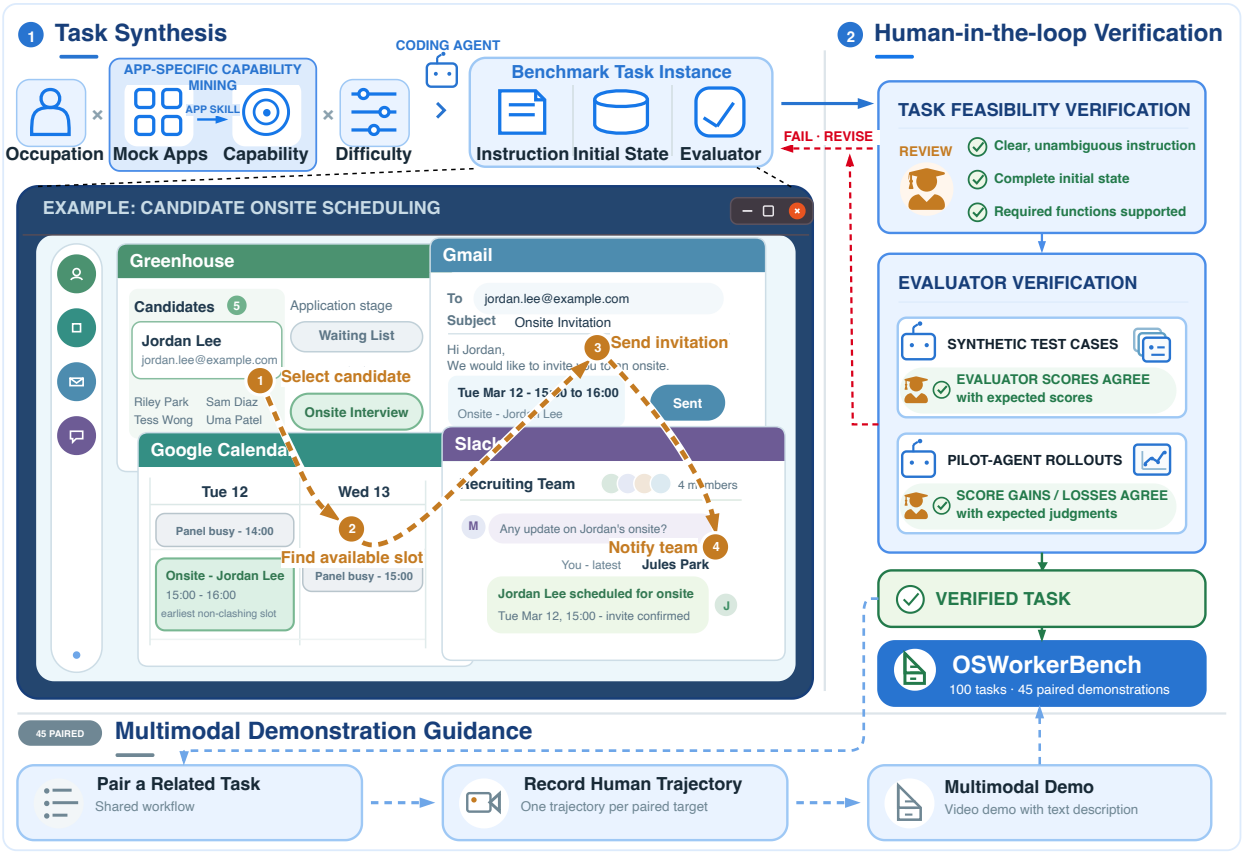


Figure 8 OSWorkerBench benchmark construction pipeline. Candidate task instances are synthesized and validated through human review, evaluator tests, and pilot-agent rollouts. Forty-five targets are additionally paired with human-recorded demonstrations of related but non-identical tasks for the variant-demo setting. The 33 same-task strong-agent rollouts used in the self-demo evaluation are constructed separately and are not depicted.

demonstrations as explicit one-shot guidance during evaluation. We report systematic results for the 33-task self-demo setting in Section 7; systematic evaluation of the 45-task variant-demo setting is left to future work.

6.1 Benchmark Design and Construction Pipeline

Figure 8 provides an overview of the OSWorkerBench construction pipeline, which proceeds from capability-grounded task synthesis and dense evaluator generation through human-in-the-loop verification and, for 45 selected targets, human-recorded variant-demo pairing. The 33 self-demos are produced separately by retaining successful strong-agent rollouts on the finalized target tasks.

Capability-grounded workflow synthesis. OSWorkerBench synthesizes mock-app tasks from an occupation, a set of applications, verified application capabilities, and a target difficulty. For each application, we inspect its implemented UI and state schema to identify functions that are both executable through the interface and programmatically observable, and use these verified functions as workflow building blocks. The synthesizer requires at least one explicit cross-application dependency rather than merely co-locating unrelated actions. In the onsite-scheduling example in Figure 8, the candidate selected in Greenhouse determines what must be scheduled, Calendar availability constrains the Gmail invitation, and the confirmed schedule supplies the content of the Slack notification.

Difficulty is assigned during synthesis rather than inferred retrospectively from model performance. We control three complementary axes: *breadth*, the number of applications and transitions among them; *depth*, the number and diversity of nontrivial UI access points that require navigation or discovery; and *reasoning*, the amount of task state that must be derived, filtered, or deliberately left unchanged rather than directly

copied. A task's construction-time difficulty profile jointly varies these axes through its applications and information transfers, access points, records and distractors, decision branches, dependencies, and estimated human-reference horizon. Harder tasks increase multiple dimensions together rather than scaling any single factor in isolation. The horizon is estimated from the GUI actions in a complete human reference solution and is not an observed model trajectory length.

Once the workflow and difficulty profile are fixed, a coding-agent pipeline realizes the specification as a natural-language instruction, a deterministic initial state, and an executable evaluator. The setup and evaluator are authored separately under a shared task and answer-key contract, while CUA-Gym's state-injection design provides an isolated, resettable session for reproducible execution [8].

Dense evaluator generation and human-in-the-loop verification. To evaluate task outcomes without requiring imitation of a reference trajectory, each evaluator measures functional outcomes rather than similarity to a reference trajectory by decomposing task completion into independently verifiable checkpoints. Record-level checks detect missing, incorrect, or extra outputs; gates enforce prerequisites among dependent outcomes; and weights prioritize primary business outcomes over cross-application outputs and auxiliary confirmations. Most checkpoints compare the final and initial application states, including session-scoped mock-application data exposed through the unified API. When backend state is insufficient, task-specific evaluators inspect spreadsheet cells, generated files, images, compound documents, or conditional outputs. Evaluators contain 1–13 checkpoints (mean 4.86; median 5), enabling dense progress measurement while reserving strict success for tasks satisfying all required final-state conditions.

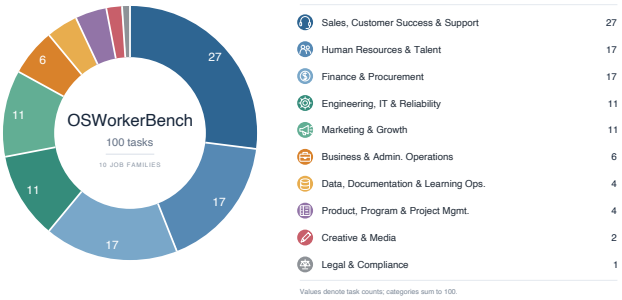
Before deployment, each evaluator is tested on controlled terminal states. The untouched initial state must score 0 and a golden completion 1. Checkpoint-specific and valid partial states test intermediate scores, while negative cases introduce missing prerequisites, wrong values or branches, distractors, and over-action errors. Each case applies a controlled mutation to the initialized state, with its expected score fixed from the rubric before execution. The production evaluator is then run unchanged across all cases, and a per-task manifest records every mutation and expected score for reproducibility.

Human reviewers additionally verify instruction clarity, setup completeness, task feasibility, checkpoint coverage, weights, and expected partial scores. As an independent execution check, representative agents, including Kimi-2.6 and other CUAs, run each task from its initialized environment. Reviewers compare their trajectories and terminal states against checkpoint-level and aggregate scores. Any mismatch triggers revision of the instruction, setup, or evaluator, followed by rerunning all affected validation checks.

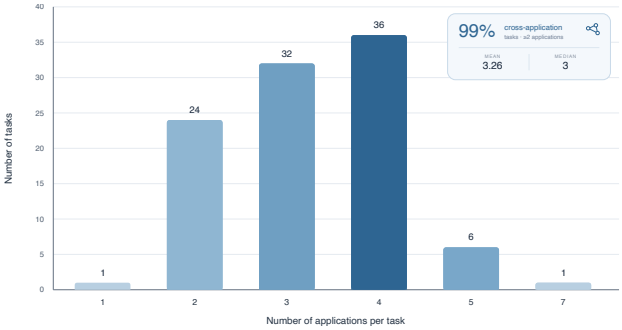
6.2 Multimodal Demonstration Guidance

Self-demo setting. The self-demo collection contains 33 multi-application targets used in the reported DemoCUA evaluation. For each target, a stronger GUI agent first completes that same task successfully. Its rollout is converted into a multimodal demonstration that preserves the screenshots and action types while omitting concrete pixel coordinates from the workflow shown to the evaluated agent. The target is then reset and evaluated with and without this same-task demonstration. Self-demo measures how effectively a model can use a demonstrated execution path; because the source and target tasks are identical, it does not by itself measure procedural transfer across task variants.

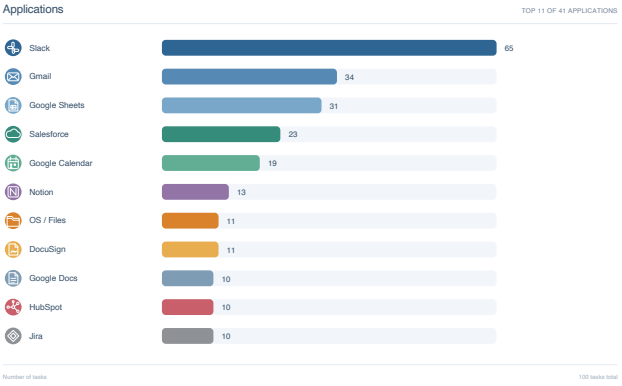
Variant-demo setting. OSWorkerBench additionally selects 45 procedurally demanding, multi-stage targets for demonstration-guided procedural transfer. These workflows require information obtained in one application to guide subsequent decisions or artifacts in another, testing intermediate-state retention, stage ordering, and reliable cross-application handoffs. Each target is paired with one successful human demonstration from a semantically related but non-identical source task. Source and target share a transferable workflow structure but differ in entities, values, initial states, and execution details, preventing direct replay. Each raw demonstration contains its source instruction and, for every tool call, the pre-action screenshot, action type and arguments, and post-action screenshot. Natural-language reasoning is excluded. Before inference, the trace is converted into the coordinate-free subtask workflow described in Section 5. These 45 human-recorded pairs define the variant-demo benchmark resource; they are distinct from the 33 same-task demonstrations used for the reported self-demo results.



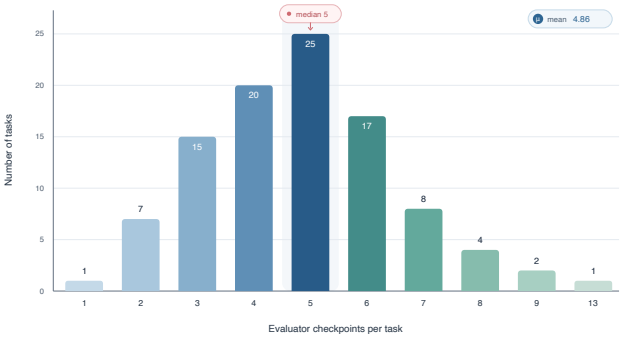
(a) Primary job-family distribution. Each task is assigned to exactly one family according to its main business objective.



(b) Number of distinct required applications per task across the full benchmark.



(c) Most frequent normalized applications across the benchmark.



(d) Distribution of evaluator checkpoints per task.

Figure 9 OSWorkerBench dataset overview. All statistics are computed over the canonical 100-task set. Application aliases and mock variants are normalized before counting.

Controlled evaluation protocols. All 100 tasks can be evaluated under the **instruction-only** protocol. Under either demonstration-guided setting, the associated targets are rerun after adding exactly one demonstration to the context. Guided and unguided runs use identical target instructions, initialized environments, interaction budgets, and evaluators; only demonstration availability differs. On the 33-task self-demo set, the paired difference measures the value of same-task execution guidance. On the 45-task variant-demo set, it measures transfer of an observed procedure to a related but non-identical task. The remaining 55 tasks lack a variant-demo pairing but remain part of the full instruction-only benchmark.

6.3 Task Coverage and Characteristics

We assign each task to one mutually exclusive job family according to its primary business outcome, rather than the applications it happens to use. As shown in Figure 9a, sales, customer success, and support form the largest family with 27 tasks. Human resources and talent and finance and procurement each contribute 17 tasks, followed by engineering, IT, and reliability and marketing and growth with 11 tasks each. The remaining 17 tasks cover business and administrative operations, data, documentation and learning operations, product/program/project management, creative and media work, and legal and compliance. This distribution reflects a deliberate focus on enterprise work while retaining a meaningful long tail of professional scenarios.

Capability-oriented task slices. Job-family and application statistics describe where a task is situated, but they do not reveal the information dependencies that make the workflow difficult. We therefore annotate every task along two complementary requirement-level dimensions: whether successful completion demands long-term retention of task state, and whether it demands substantive information transfer across applications. These tags characterize properties of the task specification rather than the behavior of a particular agent: they

are assigned independently of model scores and observed failure modes. Each case is semantically reviewed against its instruction and available setup, evaluator, and trajectory evidence, while automated scripts are used only to align evidence, count applications, and validate label consistency.

Long-memory workflows (67 cases). Real office processes often separate the moment when information is discovered from the moment when it must be used. We label a task as Long-Memory when it requires either (i) reading or deriving a dynamic value, retaining it across substantive intermediate operations or application switches, and reproducing it consistently in a later evaluated outcome; or (ii) maintaining multiple constraints, decisions, completed stages, and pending items across a multi-stage workflow. Both patterns create an explicit dependency between earlier observations and later actions. Merely producing a long trajectory, taking detours, repeating clicks, or eventually failing does not qualify a task for this subset.

Cross-application information flow (49 cases). Likewise, opening several applications does not by itself constitute meaningful cross-application reasoning: the actions may be independent and require no information to flow between tools. We reserve the Multi-App tag for tasks that involve at least three distinct logical applications and require the agent to obtain at least two dynamic facts, a multi-field record collection, or a substantive passage of text from one application and faithfully reproduce, organize, or transform it in another. A single transferred identifier, constants already supplied in the instruction, and unrelated fixed actions across several applications are excluded. Application counts include only tools that must be read or modified to complete the evaluated workflow; browser and desktop shells, incidental exploration, and background-only applications are excluded, while multiple tabs or windows of the same product count once. The Long-Memory and Multi-App subsets are assigned independently and may overlap: the former captures temporal dependencies within a workflow, whereas the latter captures information dependencies across tools.

Cross-application breadth. Because demonstrations alter the evaluation context rather than the required workflow, Figure 9b aggregates all 100 tasks. Cross-application execution is a defining property of OSWorkerBench: 99 tasks require at least two applications, 68 require three or four, and the mean is 3.26 applications per task (median 3; maximum 7). This count is broader than the 49-task Multi-App subset, which additionally requires dynamic information transfer across applications.

Application frequencies exhibit a hub-and-spoke structure (Figure 9c). Slack appears in 65 tasks, followed by Gmail (34), Google Sheets (31), Salesforce (23), and Google Calendar (19), where the most common pairs are Google Sheets–Slack and Gmail–Slack (23 tasks each). These hubs support communication and state synchronization across a long tail of 41 applications. Together, application breadth and evaluator granularity characterize intrinsic task structure, with evaluators containing 1–13 checkpoints (Figure 9d).

Long-horizon interaction difficulty. OSWorkerBench is deliberately designed around realistic office workflows that unfold over many dependent stages rather than isolated GUI operations. To reach the final business outcome, an agent must carry forward dynamically discovered information, preserve constraints across application switches, and track both completed and pending steps throughout the workflow. This structure produces genuinely long interaction horizons: in the Kimi-2.6 rollouts, the median trajectory contains 68 observation–decision turns (mean 88.3), and 38 of the 100 trajectories extend to at least 100 turns. OSWorkerBench therefore directly stresses long-memory, cross-application state tracking, and sustained end-to-end workflow control. Section 7.2.4 provides the decision-turn definition and full trajectory analysis.

6.4 Executable Evaluation

Each task is paired with an evaluator that measures functional outcomes rather than similarity to a reference trajectory. Eighty-eight tasks use state-based evaluators over initialized enterprise application backends. The remaining 12 use task-specific evaluators for spreadsheets, images, compound document deliverables, or conditional workflows. This mixture preserves deterministic final-state verification while accommodating outputs whose correctness cannot be represented as a single application-state predicate.

We report *strict task success*, which requires all final-state conditions, and *partial progress*, the checkpoint-weighted score $S_{\text{partial}} = (\sum_i w_i c_i)/(\sum_i w_i)$, where $c_i \in [0, 1]$ is the score and w_i the weight of checkpoint i. For the Long-Memory and Multi-App subsets, we report only strict success because task-level labels may correspond to only a subset of checkpoints, making partial scores capability-ambiguous. Guided and unguided

Table 1 Overall average scores (%) on OSWorld-Verified. We report the overall average score of each model under the OSWorld evaluation protocol. Size denotes the publicly disclosed model scale when available. Higher is better. The best baseline result is shown in bold, and our models are shaded. * denotes reproduced following the official OSWorld repository.

Model	Size	Average Score
General-purpose multimodal models		
Kimi-K2.6	1T-A32B	73.1
Qwen3.7-Plus	Closed-source	73.3
Qwen3.6-27B	27B	52.5*
GPT-5.5	Closed-source	78.7
Claude Sonnet 5	Closed-source	81.2
Claude Opus 4.8	Closed-source	83.4
Specialized computer-use agents		
EvoCUA	32B	56.7
UI-TARS-1.5	7B	25.4
ScaleCUA-Qwen3.5	9B	68.7
Our models		
UI-Mate-9B (Ours)	9B	66.2
UI-Mate-27B (Ours)	27B	77.0

runs use identical evaluators, ensuring that measured gains reflect task outcomes rather than action imitation. We plan to release the task specifications, the 33 self-demo and 45 variant-demo pairings, taxonomy metadata, and evaluators.

7 Evaluations

7.1 Evaluation Setup

7.1.1 Benchmarks

Public Benchmarks. We evaluate UI-Mate on OSWorld-Verified [10] and WindowsAgentArena (WAA) [11] as public references for general computer-use performance.

OSWorkerBench. We additionally evaluate on OSWorkerBench (Section 6). Under the instruction-only protocol, all models receive only the target instruction, and we report strict binary success and checkpoint-based progress over all 100 tasks. We further report binary success on two overlapping requirement-based subsets: 67 Long-Memory tasks and 49 Multi-App tasks. Demonstration-guided evaluation uses separate pairings: the reported quantitative experiment uses 33 same-task strong-agent rollouts in the self-demo setting, while the benchmark also provides 45 human-recorded demonstrations of related but non-identical tasks for the variant-demo setting. A systematic aggregate result on the 45-task variant-demo set is left for future work exploration.

7.1.2 Baselines

We compare UI-Mate with nine representative baselines, divided into two groups. The general-purpose group comprises Kimi-K2.6 [29], Qwen3.6-27B [30], Qwen3.5-9B [31], GPT-5.5 [32], Claude Sonnet 5 [33], and Claude Opus 4.8 [34]. These models span different scales, model families, and deployment regimes, providing strong reference points for computer-use capabilities that emerge from broadly trained multimodal models. The specialized group includes EvoCUA-32B [6], UI-TARS-1.5-7B [3], and ScaleCUA-Qwen3.5-9B [35], each explicitly trained or adapted for graphical user-interface interaction. Together, these baselines position UI-Mate relative to both general-purpose multimodal models and specialized computer-use agents. Because the systems differ in architecture, training data, and agent scaffolding, the comparisons reflect end-to-end

Table 2 Main results on OSWorkerBench under the instruction-only protocol. We report the average checkpoint-based progress score across all 100 tasks, followed by strict binary success rates on the overlapping requirement-based Multi-App (49 tasks) and Long-Memory (67 tasks) subsets and on the full benchmark. Models are grouped by weight availability, with model sizes omitted when they are not publicly disclosed. All values are percentages, and higher is better. The best baseline result in each column is shown in bold.

Model	Model Size	Progress Score (%)	Binary Success Rate (%)		
		Overall (100)	Multi-App (49)	Long-Memory (67)	Overall (100)
Closed-weight models					
GPT-5.6-Sol		**87.67**	**65.31**	**67.16**	**71.00**
Claude Opus 4.8		81.54	53.06	55.22	62.00
Claude Sonnet 5		81.46	42.86	50.75	55.00
Open-weight models					
UI-TARS-1.5 [3]	7B	9.22	0.00	0.00	4.33
Qwen3.5 [31]	9B	18.11	2.04	1.49	5.05
EvoCUA [6]	32B	37.62	2.04	4.48	16.00
ScaleCUA-Qwen3.5 [35]	9B	38.27	3.40	7.96	16.33
Qwen3.6 [30]	27B	52.35	7.48	12.94	23.33
Kimi-K2.6 [29]	1T	72.42	18.37	25.37	40.67
UI-Mate (Ours)	9B	66.55	16.33	25.37	34.00
UI-Mate (Ours)	27B	**76.86**	**28.57**	**32.84**	**41.00**

system performance rather than the isolated effect of model scale or computer-use specialization. Unless otherwise stated, all systems are evaluated in the same initialized target environments with identical task instructions, interaction budgets, and executable evaluators. Where a demonstration-guided comparison is reported, each system receives the same demonstration through its model-specific input format; the quantitative OSWorkerBench comparison in this report uses the 33-task self-demo collection.

7.1.3 Evaluation Protocols

We use benchmark-specific evaluation configurations as described below.

OSWorld-Verified. We evaluate our model on OSWorld-Verified [10], a benchmark designed to assess multimodal agents on open-ended tasks in real computer environments. OSWorld-Verified performance reflects the complete interactive process: interpreting visual observations, grounding actions in the graphical interface, executing multi-step operations, and maintaining progress over long-horizon workflows. We follow the official OSWorld codebase and evaluation protocol. Evaluation is conducted using the official Docker/AWS provider configurations. The Docker provider supplies isolated, reproducible desktop environments with KVM acceleration where available, while the AWS provider uses OSWorld's official host-client architecture to support large-scale parallel evaluation. Parallel execution affects only evaluation throughput and does not change individual task configurations or scoring criteria. We report the end-to-end task success rate calculated by the official evaluators.

WindowsAgentArena. To further assess cross-platform generalization, we evaluate our model on WindowsAgentArena (WAA) [11], which tests computer-use agents on Windows applications and system-specific configurations. We follow OS-SYMPHONY's evaluation protocol for WAA and use the corresponding OSWorld configurations as references for model-specific settings. Our implementation supports both WAA's original Docker-based environment and our internal sandbox. We report the end-to-end task success rate computed by WAA's task-specific evaluators.

OSWorkerBench. To support long-horizon state tracking, UI-Mate-27B retains its generated thinking throughout the interaction history, keeping intermediate constraints, decisions, and pending actions available during later stages of a workflow. For the baseline agents with instruction-only setup, we follow their corresponding

OSWorld evaluation configurations and adapt them to the OSWorkerBench environment without changing their model-specific interaction mechanisms. All agents are evaluated with a maximum budget of 200 interaction steps per task.

7.2 Main Results

7.2.1 OSWorld-Verified

UI-Mate is competitive with general-purpose models and advances specialized computer-use agents. As shown in Table 1, UI-Mate-27B achieves an average score of 77.0% on OSWorld-Verified, outperforming the general-purpose Kimi-K2.6 (73.1%) and Qwen3.7-Plus (73.3%) while approaching GPT-5.5 (78.7%) with a gap of 1.7%. Among specialized agents, it exceeds ScaleCUA-Qwen3.5 (68.7%), EvoCUA-32B (56.7%), and UI-TARS-1.5 (25.4%). UI-Mate-9B reaches 66.2%, slightly below ScaleCUA-Qwen3.5-9B but above the larger EvoCUA-32B by 9.5%. These comparisons indicate that environment-grounded computer-use training can deliver performance competitive with substantially larger general-purpose systems and that model scale alone does not determine agent capability.

Scaling primarily improves application-level workflow execution. UI-Mate-9B and UI-Mate-27B obtain the same OS score of 91.7%, suggesting that the smaller model already acquires strong basic operating-system interaction capabilities. The 27B model's overall gain of 10.8% instead comes primarily from Office (+11.9%), Daily (+12.7%), Professional (+6.1%), and Workflow (+13.0%) tasks. Thus, increasing model capacity contributes less to atomic OS control than to coordinating longer, application-specific execution paths. UI-Mate-9B nevertheless reaches 66.2% and surpasses EvoCUA-32B (56.7%), showing that model size alone does not determine computer-use performance; the coverage and quality of agent training data are also critical.

UI-Mate's relative strengths lie in system interaction and workflow coordination. Compared with Kimi-K2.6, UI-Mate-27B achieves higher scores on OS (91.7% vs. 79.2%), Office (85.4% vs. 80.0%), and Workflow tasks (63.7% vs. 55.0%), while the two models perform similarly on Daily tasks (76.8% vs. 77.1%). In contrast, UI-Mate-27B remains weaker on Professional tasks (75.5% vs. 81.6%). This performance profile indicates that UI-Mate's training transfers particularly well to operating-system control, office applications, and multi-stage workflow execution, while specialized professional software remains an important direction for improving coverage and generalization.

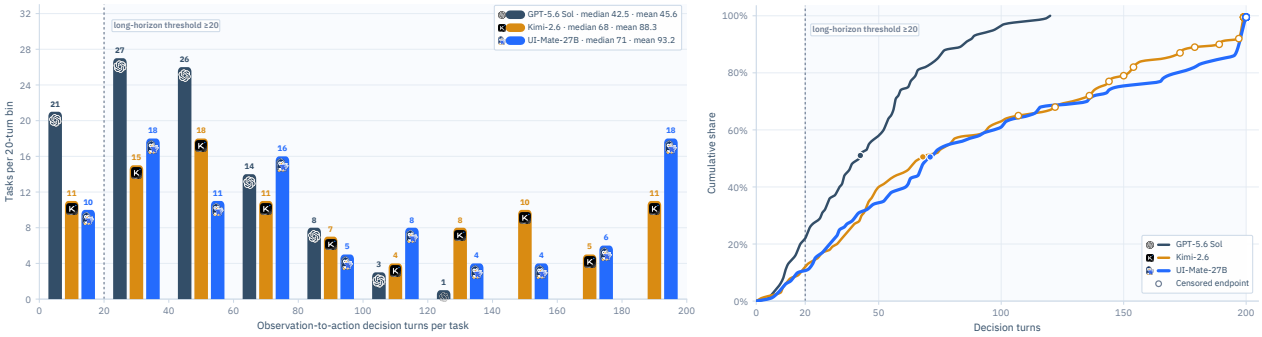
7.2.2 WindowsAgentArena

UI-Mate establishes the strongest open-weight performance on WindowsAgentArena. In Table 3, UI-Mate-27B achieves a task success rate of 66.2%, outperforming all open-weight baselines from 7B to 1T parameters. It surpasses the substantially larger Kimi-K2.6 (63.3%) by 2.9 percentage points and exceeds specialized computer-use agents including EvoCUA-32B (56.5%), UI-TARS-1.5-7B (42.1%), and ScaleCUA-Qwen3.5-9B (38.1%). UI-Mate-27B also approaches frontier closed-source models, trailing Claude Sonnet 5 (68.8%), Claude Opus 4.8 (69.3%), and GPT-5.5 (70.4%) by only 2.6, 3.1, and 4.2 percentage points, respectively. These comparisons demonstrate that environment-grounded computer-use training can produce an open-weight agent competitive with substantially larger general-purpose models and leading proprietary systems.

UI-Mate delivers substantial gains across model scales. UI-Mate-9B reaches a success rate of 61.7%, outperforming its Qwen3.5-9B base model (37.5%) by 24.2 percentage points and ScaleCUA-Qwen3.5-9B (38.1%), a specialized agent at the same scale, by 23.6 points. Despite using only 9B parameters, it also surpasses the larger EvoCUA-32B (56.5%) by 5.2 points and comes within 1.6 points of the 1T-parameter Kimi-K2.6. Similarly, UI-Mate-27B improves upon Qwen3.6-27B (47.1%) by 19.1 percentage points while holding model scale fixed. Together, these results show that UI-Mate's gains arise not merely from model capacity, but from the coverage, quality, and executable feedback provided by our training stack.

7.2.3 OSWorkerBench

UI-Mate delivers competitive performance at the 27B scale. In Table 2, UI-Mate-27B attains 41.00% overall binary success and a 76.86% progress score. Relative to its Qwen3.6-27B base model, our computer-use training



(a) Trajectory-length histogram (20-turn bins) **(b)** Cumulative distribution

Figure 10 Decision-turn distributions for GPT-5.6 Sol, Kimi-K2.6, and UI-Mate-27B on the same 100 OSWorkerBench tasks. A decision turn is one observation-to-action model response. A response containing multiple UI actions still counts as one turn because no new observation occurs within the batch; terminal responses and duplicate log records are excluded. Trajectory-length histograms using 20-turn bins show the prevalence of extended interactions, while empirical cumulative distributions summarize the full trajectory-length profiles; open triangles mark trajectories censored by the turn limit or technical failures. Each system has one rollout per task, and decision-turn counts characterize end-to-end interaction patterns rather than atomic UI-operation counts.

improves these metrics by 17.67 and 24.51 percentage points, respectively, while holding architecture and parameter count fixed. This brings UI-Mate-27B slightly ahead of Kimi-K2.6 in both end-to-end completion (41.00% vs. 40.67%) and overall progress (76.86% vs. 72.42%). UI-Mate-27B also exceeds the best prior specialized agent in overall success, although a clear gap to frontier models remains.

UI-Mate achieves substantial improvements at the 9B scale. As shown in Table 2, UI-Mate-9B improves overall binary success from 5.05% to 34.00% and progress from 18.11% to 66.55% over its Qwen3.5-9B base model, corresponding to gains of 28.95 and 48.44 percentage points. It also substantially outperforms ScaleCUA-Qwen3.5-9B, a specialized agent built on the same base model, in both overall success (34.00% vs. 16.33%) and progress (66.55% vs. 38.27%).

UI-Mate is stronger on long-horizon and cross-application tasks. UI-Mate-27B achieves 28.57% success on Multi-App tasks and 32.84% on Long-Memory tasks, substantially improving over its Qwen3.6-27B base model (7.48% and 12.94%) and outperforming Kimi-K2.6 (18.37% and 25.37%). Given the near tie between UI-Mate-27B and Kimi-K2.6 on overall success, this relative advantage suggests that our training particularly benefits workflows requiring information transfer across applications and sustained state tracking.

Trajectory-level evidence. Representative trajectories illustrate this difference. In a five-application pipeline-management workflow, UI-Mate preserves record-specific attributes through the final updates, whereas Kimi-K2.6 completes most stages but fails to persist required dates. In a multi-student report-card workflow, UI-Mate maintains the correspondence among records, reports, recipients, and status updates, while Kimi-K2.6 omits required attributes from the final email. These cases point to late-stage information preservation, rather than basic interface reachability, as an important source of the performance difference.

Remaining failure modes. UI-Mate-27B's progress score exceeds its binary success rate by 35.86 percentage points, indicating substantial partial progress on tasks that are not completed end to end. The trajectory examples point to late-stage omissions, such as an unwritten field or missing final notification, as an important remaining failure mode. Closing the gap between partial progress and strict completion is therefore a primary opportunity for improvement. Long-Memory and Multi-App are overlapping task populations rather than matched variants, so their success rates characterize relative strengths but do not isolate the causal effect of any single training component.

7.2.4 Decision-Turn Analysis on OSWorkerBench

Protocol and metric. Task-level scores summarize outcome quality but not the interaction horizon required to achieve it. We therefore analyze trajectories from GPT-5.6 Sol, Kimi-K2.6, and UI-Mate-27B on the same 100

Table 3 Agent success rates on WindowsAgentArena. Results are grouped by task category, with models organized by model family and input modality type. All values are percentages, and higher is better. The best baseline result in each column will be shown in bold, and our models are shaded.

Model	Access / Size	Total Score (%)
General-purpose multimodal models		
Kimi-K2.6	Open, 1T-A32B	63.3
Qwen3.6-27B	Open, 27B	47.1
Qwen3.5-9B	Open, 9B	37.5
GPT-5.5	Closed-source	70.4
Claude Sonnet 5	Closed-source	68.8
Claude Opus 4.8	Closed-source	69.3
Specialized computer-use agents		
EvoCUA-8B	Open, 8B	37.4
EvoCUA-32B	Open, 32B	56.5
UI-TARS-1.5-7B	Open, 7B	42.1
UI-TARS-2	Closed-source, 230B-A23B	50.6
ScaleCUA-Qwen3.5-9B	Open, 9B	38.1
Our models		
UI-Mate-9B (Ours)	9B	**61.7**
UI-Mate-27B (Ours)	27B	**66.2**

OSWorkerBench tasks. A *decision turn* consists of one interface observation followed by one model response. A response containing multiple UI actions still counts as a single turn because the model does not receive another observation or make another decision within the batch. Terminal responses and duplicate records are excluded. This metric therefore measures cycles of observation and decision making rather than atomic mouse or keyboard operations. All systems use screenshot observations, PyAutoGUI control, 1920×1080 environments, and a nominal 200-turn budget, with one rollout per task.

OSWorkerBench requires long-horizon interaction. Figure 10a reports trajectory-length histograms, while Figure 10b shows the corresponding cumulative distributions. The median trajectory contains 68 decision turns for Kimi-K2.6 and 71 for UI-Mate-27B, and 40 UI-Mate trajectories extend to at least 100 turns. Even GPT-5.6 Sol, whose trajectory distribution is shorter, requires sustained interaction on most tasks. Several Kimi and UI-Mate trajectories reach the turn limit or terminate because of technical failures, so their observed lengths are lower bounds. These distributions establish that OSWorkerBench evaluates extended workflows requiring sustained planning, state tracking, and execution rather than isolated GUI operations.

Shorter GPT trajectories largely reflect action batching. GPT's lower decision-turn count does not imply that the underlying workflows are shorter or require fewer atomic operations. A separate audit of its action logs shows that GPT produces an average of 3.83 action records per turn, with nearly half of its turns containing multiple non-wait UI actions. A substantial part of the distributional difference in Figure 10 therefore reflects action batching and system-level interaction policies rather than a reduction in the underlying task horizon.

UI-Mate maintains execution over extended workflows. UI-Mate-27B operates across a median of 71 decision turns, with a substantial portion of its trajectories extending beyond 100 turns. Together with the outcome results in Table 2, this distribution provides evidence that UI-Mate can preserve task state and accumulate verified progress across long, multi-stage workflows. The central conclusion is therefore not that shorter trajectories are inherently better, but that OSWorkerBench exposes long-term execution demands and that UI-Mate can operate effectively over these extended horizons.

7.3 DemoCUA Results

All quantitative No-Demo versus Demo results in this subsection use the **self-demo** setting, in which each target is paired with a demonstration of that same task. For the 33-task OSWorker-Subset, each self-demo is a successful rollout produced by a stronger GUI agent on the corresponding target task. This setting is separate from OSWorkerBench's **variant-demo** resource, which contains 45 human-recorded demonstrations of related but non-identical source tasks (Section 6). We conduct preliminary variant-demo exploration on OSWorkerBench targets, where Section 10 summarizes an observation from a 10-task pilot out of the 45 recorded demos. We do not include variant-demo results in the aggregate tables, and leave systematic evaluation of all 45 pairs to future work.

7.3.1 Benchmark

We evaluate the effectiveness of DemoCUA in the self-demo setting on three benchmarks comprising 73 tasks in total: a 30-task subset of OSWorld, a newly constructed 33-task subset of OSWorkerBench, and our 10-task GameDev benchmark. Together, these benchmarks cover diverse applications, long-horizon interactions, and workflows involving repeated or branching subtasks, enabling a comprehensive evaluation of demonstration-guided computer use.

GameDev. We introduce a manually curated test set of 10 exceptionally long-horizon GUI tasks, each requiring more than 200 human actions on average. The benchmark centers on eight Godot tasks that collectively cover end-to-end 2D game development from scratch. In the demonstration-supported setting, we provide human-recorded demonstrations together with relevant online tutorials. Further details are provided in Appendix C.

OSWorkerBench-Subset. OSWorkerBench-Subset comprises 33 multi-application tasks selected from OSWorkerBench, each requiring coordinated interaction across three to five applications. These tasks feature repeated subtask patterns and branching execution paths, making them particularly challenging for long-horizon agents. Without prior knowledge of application-specific operations and workflow structure, an agent may fail to discover necessary interactions, lose track of repeated objectives, or terminate after completing only part of the task. For each target, we pair the evaluated model with a successful stronger-agent rollout of that same task. This 33-task self-demo subset therefore provides a focused testbed for evaluating whether same-task workflow guidance improves task completeness.

OSWorld-Subset. OSWorld-Subset contains 30 feasible tasks that UI-Mate-27B fails without demonstration guidance but that a stronger reference agent can solve. This selection isolates tasks that are challenging yet demonstrably executable, enabling us to evaluate whether demonstrations can provide the procedural knowledge needed to close the capability gap. Tasks determined to be infeasible are excluded.

7.3.2 Demonstration Construction.

We construct a demonstration for each task using a model-assisted pipeline. First, we deploy a capable GUI agent to interact with the target applications and record its interaction trajectories as screen-capture videos. We then convert the raw trajectories into structured action sequences paired with annotated screenshots. Human annotators subsequently refine each demonstration by (1) completing any unfinished portions of the task that the agent was unable to solve, (2) removing redundant interactions such as error-recovery loops, and (3) retaining the key actions that convey the essential workflow logic. The resulting demonstrations are concise and correct, cover the full scope of each task, and do not reveal task-specific answers such as underlying data values or classification labels.

7.3.3 Evaluation Setup

We evaluate Kimi K2.6 and UI-Mate-27B under both No-Demo and Demo conditions. Unless explicitly stated otherwise, "Demo" in this subsection means self-demo. The two conditions use identical task instructions, initialized environments, interaction budgets, and evaluators; demonstration guidance is provided only in the Demo condition. Both agents use the long-horizon context-management mechanism described in Section 5.3 and are allowed up to 1,000 interaction steps per episode.

For proactive folding, each screenshot is assigned a fixed cost of 1,800 tokens when estimating context usage. Folded summaries are generated at a temperature of 0.2 and limited to 2,048 tokens. All other inference settings are held fixed between the No-Demo and Demo conditions for each agent.

Kimi K2.6 uses a 96K-token context window, with 32K tokens reserved for generation. It retains the eight most recent interaction steps verbatim and triggers context folding when estimated usage reaches 85% of the available input budget. UI-Mate-27B uses a 128K-token context window, with 64K tokens reserved for generation. It retains up to 40 recent textual steps and the five most recent screenshots. Because UI-Mate-27B produces substantially longer reasoning traces, folding is triggered earlier, at 60% of its input budget.

7.3.4 Results

Table 4 GameDev performance with and without demonstrations on UI-Mate-27B. Results are evaluated over five runs. The two Avg. columns report the corresponding mean success rates, while Δ denotes the Demo - No-Demo difference in percentage points. Higher is better.

Task	No-Demo					Avg.	Demo					Avg.	Δ
	1	2	3	4	5		1	2	3	4	5		
godot-01	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
godot-02	71.43	64.29	78.57	78.57	57.14	70.00	71.43	71.43	64.29	71.43	64.29	68.57	-1.43
godot-03	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
godot-04	22.22	38.89	44.44	38.89	38.89	36.67	55.56	50.00	55.56	66.67	50.00	55.56	18.89
godot-05	76.47	82.35	82.35	82.35	76.47	80.00	64.71	76.47	70.59	76.47	88.24	75.29	-4.71
godot-06	86.67	86.67	80.00	80.00	80.00	82.67	93.33	93.33	73.33	86.67	86.67	86.67	4.00
godot-07	63.16	36.84	78.95	84.21	68.42	66.32	89.47	94.74	73.68	89.47	73.68	84.21	17.89
godot-08	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
obsidian-01	66.67	80.00	33.33	53.33	70.00	60.67	66.67	60.00	36.67	76.67	53.33	58.67	-2.00
qgis-01	68.75	81.25	68.75	68.75	68.75	71.25	75.00	100.00	87.50	75.00	75.00	82.50	11.25
Average	75.54	77.03	76.64	78.61	75.97	**76.76**	81.62	84.60	76.16	84.24	79.12	**81.15**	**4.39**

GameDev. As shown in Table 4, demonstration guidance improves the average score of UI-Mate-27B from 76.76% to 81.15%, a gain of 4.39 percentage points. The largest improvements occur on godot-04 (+18.89 points), godot-07 (+17.89), and qgis-01 (+11.25), all of which require long, structured sequences of fine-grained operations. Performance remains unchanged on the three tasks already solved perfectly without demonstrations. We observe a similar improvement with the stronger Kimi K2.6, whose results and trajectory lengths are reported in Appendix Tables 12 and 13.

Table 5 Paired self-demo performance on the OSWorld-Subset30 and OSWorkerBench-Subset33 with UI-Mate-27B. Each target is evaluated under instruction-only and self-demo-guided conditions with identical initial states, budgets, and evaluators. The self-demo is a successful stronger-agent rollout of the same task. OSWorld-Subset results are averaged over five runs per target, while OSWorkerBench-Subset results are averaged over three runs per target. Paired gains are computed as self-demo guided minus instruction only and reported in percentage points (pp). Higher is better.

Evaluation Set	Condition	Performance (%)		Paired Gain (pp)	
		Progress Score	Binary Success	Δ Progress	Δ Success
OSWorld (Subset-30)	Instruction only	40.27	–	–	–
	Demo guided	**65.75**	–	+25.48	–
OSWorkerBench (Subset-33)	Instruction only	67.85	17.17	–	–
	Demo guided	**81.14**	**35.35**	+13.29	+18.18

OSWorkerBench-Subset. Table 5 presents the self-demo results on the more demanding OSWorkerBench subset. Demonstrations improve the average normalized task score from 67.85% to 81.14%, a gain of 13.29

percentage points, with improvements on 28 of the 33 tasks. The number of tasks receiving a perfect score also increases from one to five. At the same time, the average trajectory length increases from 173.3 to 216.0 steps (Table 16). This increase does not simply indicate lower execution efficiency: OSWorkerBench tasks contain repetitive and branching subtasks, and agents without demonstrations frequently terminate after completing only part of the requested workflow. Demonstrations help the agent identify and execute the remaining branches or repeated operations, producing longer but more complete trajectories.

OSWorld-Subset. On the OSWorld subset, Table 5 and Table 14 in Appendix shows a substantially larger improvement: demonstrations raise the average score from 40.27% to 65.75%, a gain of 25.48 percentage points. Performance improves on 18 of the 30 tasks and remains unchanged on eight. Four tasks that receive zero score without demonstrations: chrome-02, chrome-03, multi-02, and os-01, are solved perfectly in all demonstration-conditioned runs. These results show that demonstrations can provide critical application-specific procedures that the agent is unlikely to discover reliably from the instruction alone. Nevertheless, performance decreases on four tasks, suggesting that demonstration guidance can be harmful when the demonstrated procedure is misapplied or conflicts with the current interface state.

7.3.5 Case Study

We present two representative cases illustrating demo-in-the-loop execution: one from our GameDev benchmark and one adapted from OSWorld2 [36]. For the latter visa-application case, we remove the requirement to invoke `ask_user`.

GameDev-Godot As illustrated in Fig. 11, the task requires configuring which bullet a game character should fire. The no-demonstration agent correctly identified the bullet file, but attached it to a character instance in the current game level rather than to the reusable character template (Fig. 11(c)). Because both operations look nearly identical in the interface, the agent incorrectly considered the task complete. The demonstration showed not only which file to select, but also that the reusable character template should be opened, modified, and checked for a visible confirmation icon (Fig. 11(a)). Consequently, the demo-conditioned agent followed the demonstrated procedure and modified the correct underlying object (Fig. 11(b)), whereas the no-demonstration agent performed an apparently correct operation in the wrong context. This difference is confirmed by the final artifact evaluation: the demo-conditioned run passed the `player-bullet-reference` check, while the no-demonstration run failed it.

OSWorld2-Visa Application As illustrated in Fig. 12, the task [36] requires reading the supporting PDF documents and completing a DS-2019 application with consistent form values and financial evidence. The no-demonstration agent attempted to enlarge the document view, but did not adequately reposition the viewport to expose the relevant fields (Fig. 12(c)). This resulted in incorrect visual/OCR readings and, consequently, incorrect form values and document selection. The demonstration showed not only which documents to inspect, but also that the PDF viewer should be maximized, the viewport should be adjusted to locate the critical fields, and the extracted values should be cross-checked before submission (Fig. 12(a)). Consequently, the demo-conditioned agent correctly identified the $18,000 funding requirement, recognized that the initial $12,000 certificate was insufficient, located the alternative $18,000 certificate, and submitted consistent evidence (Fig. 12(b)). This difference is confirmed by the final artifact evaluation: the demo-conditioned run passed all JSON checks and uploaded the correct financial certificate, achieving a score of 99.5%, whereas the no-demonstration run submitted incorrect field values and the insufficient certificate, achieving only 24.5%.

7.4 Findings

7.4.1 Data

Task difficulty lives in the environment, not the instruction. The same instruction can be trivial over a synthesized spreadsheet of a dozen uniform rows, but demands locating, disambiguating, and cross-checking content when the workbook is a real one with multiple sheets, merged headers, and columns irrelevant to the task. Concretely, for office-related tasks, retrieved real resources are roughly 6× larger than their synthesized counterparts, and trajectories over them average 58.4 steps against 38.5, a 51.7% increase. Environment

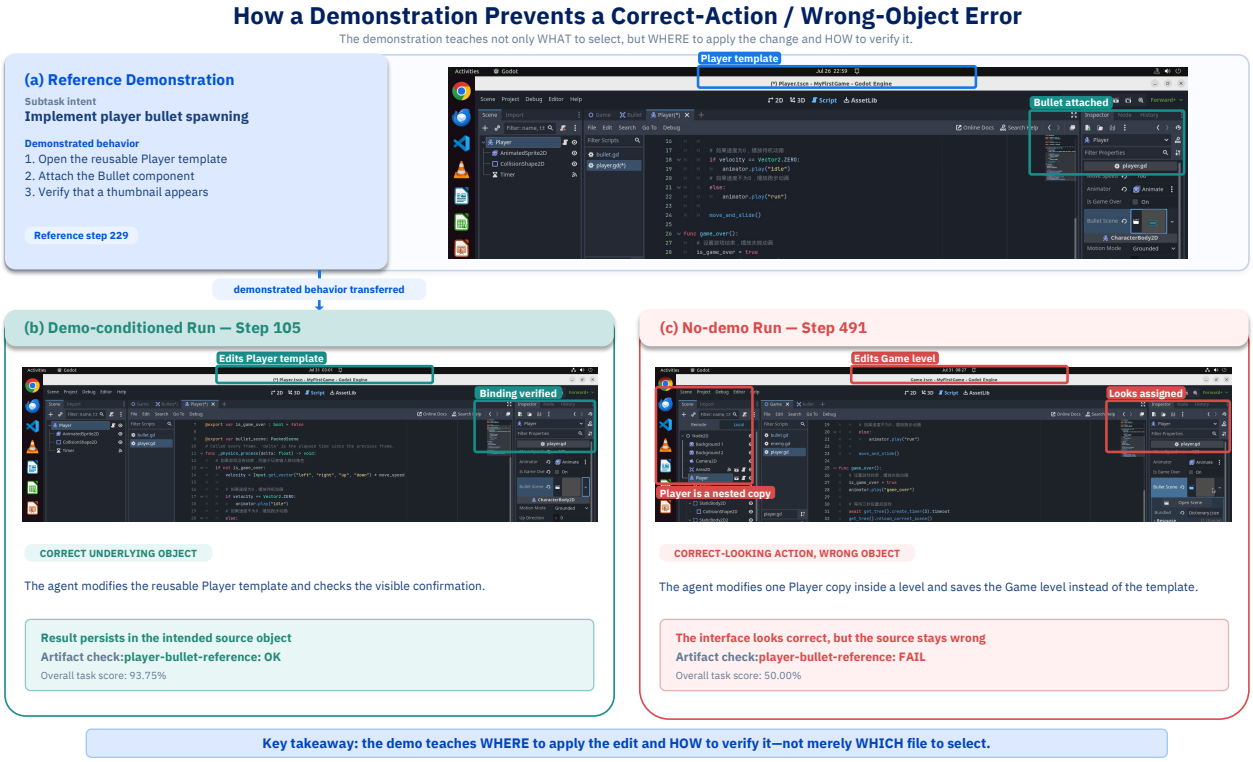


Figure 11 Demonstrations help the agent modify the correct underlying object. The reference demonstration and demo-conditioned run modify the reusable Player template, whereas the no-demo run modifies only one Player copy inside a game level. The two operations look similar in the interface, but only the former persists in the intended source object.

realism, rather than instruction complexity alone, is therefore the primary target when we construct training data.

Capability-aware allocation matters more than raw data volume. As data production scales, uniform sampling increasingly revisits well-covered capabilities while leaving long-tail gaps obscured by application-level statistics. Thus, we route each instruction through the capability tree of Section 3.4 to a fine-grained operation and allocate data according to the coverage of the capability. This tree-guided rebalancing expands capability coverage, with the added coverage concentrated primarily in Multi-App workflows, and improves Multi-App performance by 15.5 pp over training without capability-aware sampling. Gains are also larger on tasks whose capabilities were previously underrepresented. Although this evidence is correlational rather than a controlled isolation of capability tagging, it suggests that the tree provides an effective interface for diagnosing long-tail gaps and grouping superficially different tasks by shared operational structure. We therefore budget data by capability coverage rather than by raw trajectory count or application.

The hard part of a verifiable task is making its reward mean the instruction. The execution invariant of Equation 9 removes inconsistent tasks but cannot certify that a reward means what its instruction says: it only checks the reward against a single reference completion, which says nothing about the other solutions it should accept or the wrong ones it should reject. An LLM audit of evaluators that had passed this filter found roughly 18% still misaligned, with over-strict matching (40%), semantically vacuous assertions (23%), and checks applied to the wrong target (19%) as the leading modes. Auditing therefore belongs in the pipeline, where flagged tasks enter repair rather than discard, as this preserves the hard, environment-rich tasks most likely to be misjudged, while what is not recovered is not promoted. We treat reward validity, not artifact checkability, as the binding constraint on synthesizing verifiable tasks at scale.

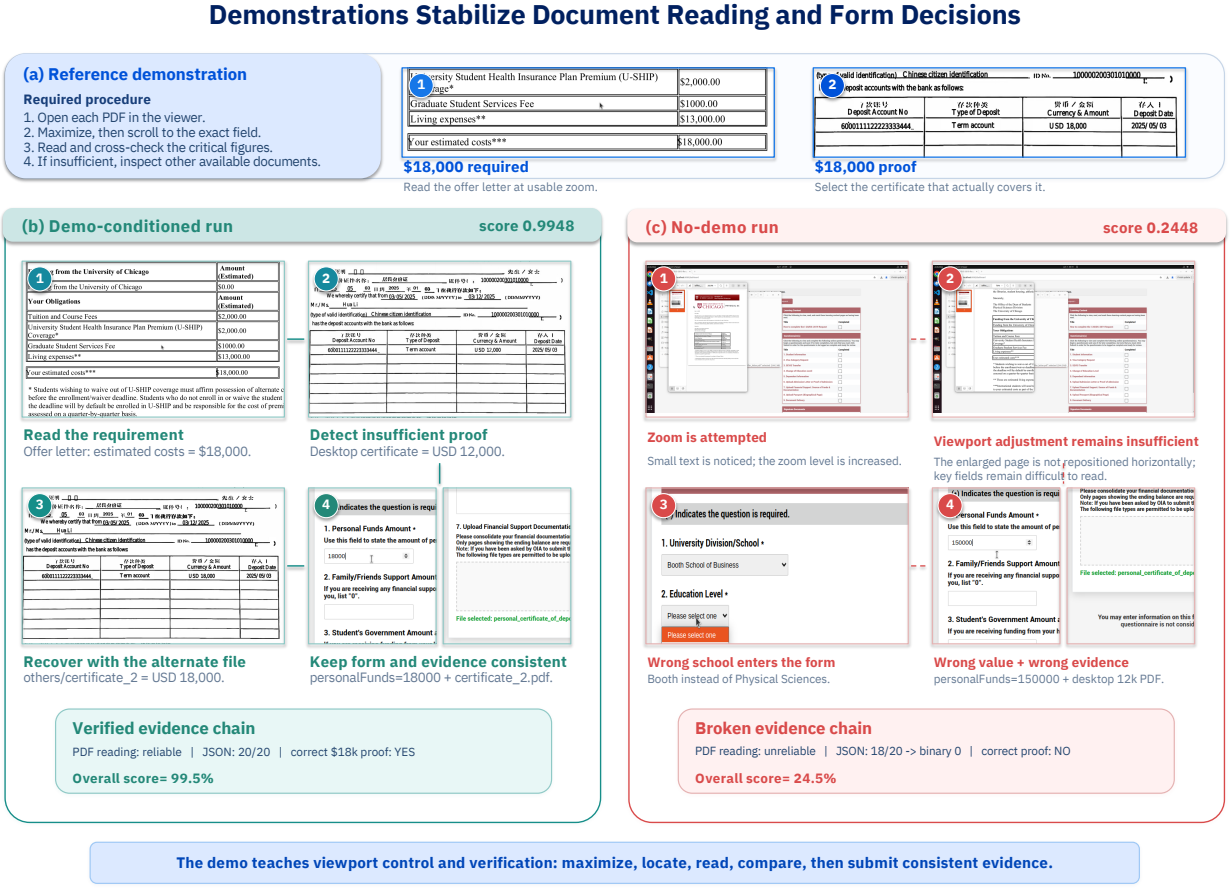


Figure 12 Demonstrations teach reliable viewport control for document-grounded reasoning. The reference demonstration and demo-conditioned run maximize the PDF viewer, locate and cross-check the critical values, and submit consistent form entries and supporting evidence. The no-demo run enlarges the document view without adequately repositioning the viewport, leading to incorrect visual/OCR readings and erroneous downstream decisions.

7.4.2 Model Training

Historical reasoning improves inference but undermines exploration during RL rollouts. Historical reasoning—preserving the reasoning traces from preceding interaction steps—consistently improves inference-time performance. Activating historical reasoning exclusively at evaluation yields gains of +3.43 pp for the SFT model and +2.27 pp for the RL model trained without historical reasoning. When historical reasoning is additionally incorporated during SFT, it yields a further improvement of approximately +2.85 pp on long-horizon, cross-application tasks, but with negligible gains on shorter and simpler tasks. Qualitative analysis attributes these improvements to more robust cross-application state tracking, consistent preservation of entity chains across Greenhouse, Gmail, Calendar, Slack, HubSpot, and QuickBooks, and stronger constraint adherence and source grounding in table aggregation and cross-application referencing. Historical reasoning also facilitates recovery from intermediate execution errors, reducing premature termination with `DONE`.

In contrast, incorporating historical reasoning into RL training affects the policy's exploration dynamics. Conditioning each decision on accumulated historical reasoning traces increases predictive confidence and accelerates the reduction of policy entropy. Empirically, this configuration exhibits signatures of entropy collapse, thereby restricting exploration during RL rollouts and resulting in lower evaluation performance. These results highlight the need for a more effective strategy to integrate historical reasoning into RL, one that preserves policy diversity without compromising its advantages for long-horizon state tracking.

Adaptive curriculum sampling and process credit improve training efficiency rather than final performance. Adaptive

Table 6 Trajectory lengths and task scores on GameDev on UI-Mate-27B. Human Demo reports the number of steps in the human demonstration, while No-Demo and Demo report the average model steps and normalized task scores over five runs. Tasks requiring longer trajectories are generally associated with lower scores.

Task	Human Demo	No-Demo		Demo	
	Steps	Avg. Steps	Score (%)	Avg. Steps	Score (%)
godot-01	35	29.8	100.00	30.0	100.00
godot-02	223	178.8	70.00	186.6	68.57
godot-03	202	173.8	100.00	131.0	100.00
godot-04	247	410.4	36.67	357.8	55.56
godot-05	380	580.0	80.00	362.6	75.29
godot-06	323	638.4	82.67	379.2	86.67
godot-07	376	442.8	66.32	448.0	84.21
godot-08	46	15.0	100.00	36.2	100.00
obsidian-01	255	427.4	60.67	457.6	58.67
qgis-01	305	139.6	71.25	142.2	82.50
Average	**239.2**	**303.6**	**76.76**	**253.1**	**81.15**

curriculum sampling and PCM do not consistently improve final task success over standard outcome-only training. Their main benefit is faster convergence: models using these mechanisms often reach comparable performance in fewer than half as many optimization updates. Adaptive curriculum sampling prioritizes weak domains, while PCM concentrates verifier-derived credit on decisions most relevant to success or failure. These mechanisms therefore improve how efficiently a fixed training corpus is learned, whereas further gains remain primarily constrained by data coverage and quality.

Smaller models benefit from explicit reasoning and staged, verifier-grounded training. Model scale changes the training recipe that works best. During SFT, the 9B model benefits from training on trajectories with explicit reasoning, whereas the 27B model remains effective when trained on a mixture of trajectories with and without reasoning. During RL, the 9B model is more sensitive to evaluator correctness: incomplete success criteria or permissive reward proxies more readily reinforce partial or unintended behaviors. Finally, the 9B model benefits from revisiting the same tasks across multiple training stages. These observations suggest that smaller GUI agents require more explicit reasoning supervision, more carefully validated evaluators, and repeated on-policy exposure to the same task distribution.

7.4.3 DemoCUA

Subtask-level demonstrations keep the model on track. Providing the full task demonstration at once can confuse the model about its current progress, causing it to follow steps from later stages. We instead show only the active subtask in detail and summarize the others in a progress checklist. This keeps the model focused while preserving the overall workflow (see Figure 6).

Less guidance helps training; More guidance helps inference. During training, extracting key actions from the current subtask forces the model to infer omitted intermediate steps from the screenshot rather than copy the demonstration workflow. Once trained, the model treats the demonstration as a fallible reference and resolves conflicts using the live screenshot. Key-action extraction is therefore unnecessary at inference; providing the full sequence is simpler, more informative, and empirically better.

Longer trajectories are associated with greater task difficulty. Table 6 shows that longer tasks tend to score lower on GameDev: godot-01 and godot-08 are finished in 29.8 and 15.0 steps at 100%, whereas every task needing more than 400 steps stays below 83%. The length of the human demonstration is a cleaner predictor of the steps the model needs (rank correlation +0.83), suggesting that trajectory length mainly reflects how much work a task inherently requires.

Demonstrations improve execution efficiency. Demonstrations shorten the average trajectory from 303.6 to

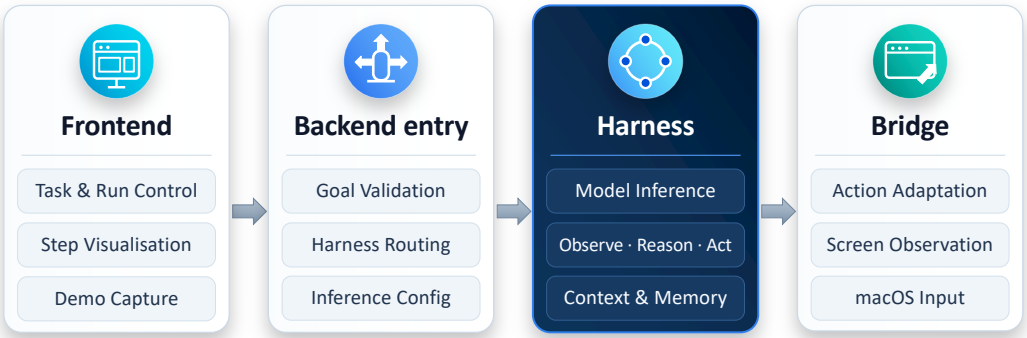


Figure 13 UI-Mate App's four-layer architecture: the frontend controls runs and demonstrations, the backend configures requests, the harness drives the agent loop, and the bridge provides macOS observation and input.

253.1 steps on GameDev, a reduction of 50.5 steps or 16.6%, while raising the average score from 76.76% to 81.15% (Tables 4 and 6), so shorter runs do not sacrifice completion. The saving is concentrated on the five tasks that exceed 400 steps without demonstrations, where the average falls from 499.8 to 401.0 steps, indicating that demonstrations mainly remove exploratory detours on long-horizon tasks.

8 UI-Mate App

8.1 Overview

UI-Mate App turns a configured VLM into a computer-use agent (CUA) on the user's machine. Given a task, it observes the screen and operates applications through the mouse and keyboard. It also records demonstrations for in-context learning in the same environment. Because it drives the desktop directly, installed applications require no plugin, API or scripting interface.

Design. Two decisions define the architecture. First, *the application contains no model*: it sends OpenAI-format requests to a configured endpoint, allowing one build to support every arrangement in §8.4. Second, *the agent logic is separate*: an API-connected harness handles prompting, action selection and completion judgement. Models and agent logic can thus change independently of desktop control, demonstration capture and execution visibility.

Architecture. Figure 13 shows four layers: the *frontend* controls and visualizes runs and records demonstrations; the *backend entry* validates and configures requests; the *harness* selects actions; and the platform-specific *bridge* executes them. The same agent can therefore run in offline evaluation or a virtual machine. A stream of JSON-line events connects the frontend and backend processes, enabling live updates and mid-run commands. On macOS the bridge is a native Swift helper; Linux uses `pyautogui`, and Windows support is in progress.

8.2 General CUA

A general run executes a natural-language goal without a demonstration. The trace in Figure 14(a) summarizes each step and expands to its observation, reasoning, action and latency. Users can pause, resume or stop the run and send a message for the next step. The application marks the controlled display and highlights actions while excluding its own windows from captures. Each run is saved incrementally as a JSONL trajectory of screenshots, model inputs and actions, exportable as training data or a benchmark task.

Step cycle. In each step, the application calls the model on the current screen, extracts and executes its action, waits for the interface to settle, then captures the observation for the next step. Engines such as UI-Mate and Kimi [29] differ in prompts and history policy but share the action space and execution path.

Action adaptation. The bridge grounds platform-neutral, `pyautogui`-style actions in the current desktop. It maps pointer locations from a 1,000-unit image grid through capture downscaling, Retina scaling and display origin into global coordinates. The macOS Accessibility (AX) API then identifies the element: actionable

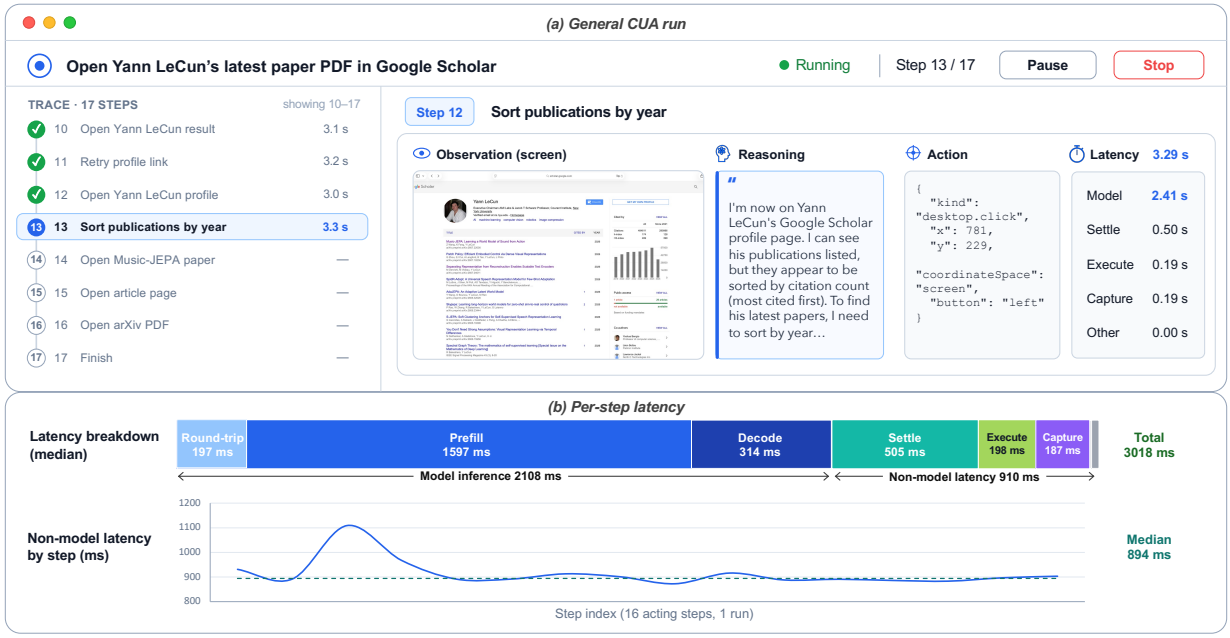


Figure 14 UI-Mate interface and runtime latency. (a) Execution trace with an expanded step. (b) Per-step latency and breakdown in a real CUA run.

elements are invoked directly, tolerating small coordinate errors; otherwise the bridge simulates a click, supporting canvases and interfaces absent from AX. It also maps **ctrl** to **cmd** outside terminals.

Per-step latency. Figure 14(b) profiles a run on the latency-tuned 27B endpoint of §8.4. The median step takes 3030 ms: 2108 ms for the model call and 910 ms elsewhere. Prefill and decoding account for roughly 76% and 15% of the call; action settling, execution and screen capture dominate the remainder. Halving all non-model costs would reduce the median by only about 15%. UI-Mate therefore exposes image history, resolution, prompt-prefix reuse and reasoning-output controls that target prefill or decoding.

8.3 Demo CUA

Many desktop workflows involve internal tools, personal conventions or required action orders that a model does not know. A user can demonstrate such a workflow once: an offline stage converts the recording into a saved workflow, which then guides autonomous runs online.

Recording and Demo2Workflow. A native macOS recorder saves video and raw input, groups related events into actions, and extracts frames at and after each action to show its target and effect. Demo2Workflow then uses a VLM to describe each action from its frame pair and recorded facts, groups the steps into named subtasks, and lets the user revise the result. Manual edits remain separate from model output so neither overwrites the other.

Guided runs. Processed workflows form a personal library from which the user explicitly attaches one to a task. A guided run uses the general loop but also provides each step with workflow progress and the recorded steps of the current subtask. The agent reports subtask completion, advancing both the workflow and its displayed progress.

8.4 Deployment

Model serving is separate from client installation. For each agent engine, the user configures the base URL and model name of an OpenAI-compatible endpoint, allowing one installation to switch among the arrangements in Table 7 without rebuilding.

Table 7 Serving arrangements and approximate step times on our devices.

Where the model runs	Served as	Per step
Hosted gateway	Vendor's own	varies
Self-hosted, one 8-GPU machine	27B, bf16	3–5 s
	27B, ours	2–3 s
On device, one Apple Silicon Mac	9B, 6-bit	10–20 s

Accelerated self-hosted serving. The timings in §8.2 use QuaRot-style W8A8 quantization [37] and speculative decoding with a DFlash-trained draft model [38]. Together they reduce a step from 3–5 s to 2–3 s on the same machine without application changes.

On-device serving. On a single Mac, visual encoding and prefill dominate. We use a six-bit 9B model of roughly 7 GB, reduce captures from 1080p to 720p, and patch `mlx-vlm` to reuse prompt prefixes and cached image features. These changes reduce typical step time from about 45 s to 10–20 s.

Client distribution. The signed and notarized disk image includes the native helper and Python runtime. After installation, UI-Mate requests the macOS Screen Recording and Accessibility permissions needed to observe and control the desktop.

9 Related Work

Computer Use Agents. Computer-use agents have shifted from agentic scaffolds that compose planning, grounding, and reflection modules around general-purpose vision–language models [39–41] toward native models that internalize perception, grounding, and action end-to-end [42–47]. Native foundation agents such as UI-TARS [4], OpenCUA [1], ScaleCUA [2], Mobile-Agent-v3 [48], UI-Venus-1.5 [49], and MAI-UI/Qwen-UI-Agent [5, 50] train on large trajectory corpora and increasingly close the loop with environment interaction: online and curriculum reinforcement learning [25, 27, 51], multi-turn RL at scale [4], EvoCUA's self-evolution from scalable synthetic experience [6], scalable environment construction [8], and verifiable task synthesis with efficient online RL [35]. Yet the resulting corpora drift toward what is cheap to instantiate, and most standard training and evaluation protocols condition primarily on task instructions. UI-Mate advances both fronts: its closed-loop data engine budgets generation by a hierarchical capability tree and promotes only audited task–verifier bundles to RL; and it consumes in-context multimodal demonstrations, distilled from a single human recording into a subtask-level workflow that is followed where the user's steps carry procedural intent and overridden by the live screen where the target task diverges, targeting forms of prompt ambiguity and execution unreliability that scaling instruction-only training alone does not directly address.

Computer Use Benchmarks. GUI-agent evaluation spans static grounding suites [52], offline web-navigation benchmarks [53], and interactive environments with executable, state-based verification. These environments cover the web [54, 55], mobile devices [56, 57], and desktop operating systems, anchored by OSWorld [10] and WindowsAgentArena [11]. Recent benchmarks further raise realism and interaction horizon: OSWorld 2.0 [58] evaluates 108 hour-scale workflows with dynamic mid-task events, WeaveBench [59] forces interleaved GUI–CLI execution under trajectory-level auditing, ScienceBoard [60] targets scientific workflows, and office-oriented suites assess professional deliverables across business applications [61, 62]. Most standard protocols nevertheless specify target tasks through instructions and environment-provided artifacts, without explicitly controlling demonstration availability for the same target. Even tutorial-following tasks [58] test adherence to materials supplied by the environment rather than acquisition of a user's own procedure. OSWorkerBench addresses this gap with 100 long-horizon office tasks across 41 applications, requirement-level Long-Memory and Multi-App subsets, and validated checkpoint evaluators. It separates two demonstration-guided settings: self-demos from successful strong-agent rollouts of the same tasks, and variant-demos from multimodal human demonstrations of related but non-identical tasks. Both support controlled comparison with instruction-only execution under the same target instruction, environment, budget, and verifier. This controlled protocol measures demonstration-guided procedural generalization and task completion.

Demonstration-Guided Computer Use. Traditional robotic process automation (RPA) records or scripts fixed sequences of interface operations, providing efficient and deterministic execution for repetitive workflows but remaining brittle to changes in task logic or UI state [63]. Recent work instead exposes procedural knowledge as reusable agent skills. CUA-Skill represents human-authored desktop procedures with parameterized execution and composition graphs [64], while MMSkills augments textual procedures with state cards and visual keyframes so that agents can recognize when a skill applies and verify its progress [65]. Most closely related, ShowUI-Aloha converts a human screen recording into a semantic action trace that guides planning and execution [9]; its demonstrated transfer primarily reuses one taught procedure across task instances sharing the same workflow logic. In contrast, UI-Mate distills either a human recording or an agent rollout into subtask-level procedural intent, selectively follows, skips, or adapts demonstrated steps, and grounds every decision in the live interface, thereby enabling demonstration-guided procedural generalization rather than workflow replay.

10 Discussion and Future Work

Verifier-Grounded Progress Credit. PCM's efficiency gains motivate verifier-grounded progress estimation from its milestone structure. A task-conditioned model could reward milestone completion and recovery, penalize regression, and retain signal in homogeneous-outcome groups. The environment would remain authoritative: a terminal residual would keep total process reward equal to the executable outcome. Deterministic checks would score verifiable milestones, with learned judges used only for uncertain states. Neutral spans could remain as context but be masked from actor loss, shortening the optimization horizon without losing causal evidence.

Beyond Self-Demo: the Variant-Demo Setting. All quantitative DemoCUA results in this report use the self-demo setting: the demonstration and evaluation target are the same task. On OSWorker-Subset, this means 33 targets are paired with successful same-task rollouts from a stronger GUI agent. These results establish the value of execution guidance, but should not be interpreted as procedural transfer across task variants.

The open problem is the variant-demo setting. OSWorkerBench provides 45 human-recorded demonstrations whose source tasks are related to, but non-identical to, their paired targets. We do not report a systematic aggregate result on this 45-task collection. In a preliminary pilot on ten of these targets, replicating the demonstrated segment to match the true number of target entities makes the variant setting net positive, but performance is not yet sufficiently stable for a main benchmark claim. Three directions follow: (1) model capability, training agents to extract the transferable structure of a partially matching demonstration and to recognize what it does not cover, rather than replay it step by step; (2) offline acquisition, harvesting demonstrations at scale from books, curated documentation and instructional video instead of recording one per task; (3) retrieval, querying a demonstration corpus against the live state so that coverage grows with the corpus rather than with annotation effort.

11 Author Contributions

We would like to express our sincere gratitude to all contributors, including those not listed in the paper, for their invaluable support and efforts. Authors within each role are listed alphabetically by their last name.

Core Contributors

Zihan Ding
Longxu Dou*
Qi Gao
Xiangwu Guo
Shengchao Hu
Zilong Huang†
Zihang Jiang*
Lei Ke*
Mengcheng Lan
Weixian Lei*
Hanxuan Li
Honglin Li
Xiyun Li
Zaitang Li
Leowei Liang‡
Xin Luo
Haozhe Ma
Jiayi Mao

Zhoujie Pan
Can Qin
Tianyuan Qu
Weiqi Wang
Wenkai Wang
Yonglin Wang
Yuxin Wang
Chenxu Wu
Yingchen Yu*
Chenyu Zhang
Yuhao Zheng

Contributors

Tianqing Fang
Zhenpeng Huang
Zhongwei Wan
Jiahao Xu
Ruihan Yang
Yidi Zhang

*Project Co-Lead. †Project Lead. ‡Project Supervisor.

References

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

[2] Zhaoyang Liu, JingJing Xie, Zichen Ding, Zehao Li, Bowen Yang, Zhenyu Wu, Xuehui Wang, Qiushi Sun, Shi Liu, Weiyun Wang, et al. Scalecua: Scaling open-source computer use agents with cross-platform data. In *International Conference on Learning Representations*, volume 2026, pages 62707–62766, 2026.

[3] Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. UI-TARS: Pioneering automated GUI interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

[4] ByteDance Seed. UI-TARS-2 technical report: Advancing GUI agent with multi-turn reinforcement learning. *CoRR*, abs/2509.02544, 2025.

[5] 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.

[6] Taofeng Xue, Chong Peng, Mianqiu Huang, Linsen Guo, Tiancheng Han, Haozhe Wang, Jianing Wang, Xiaocheng Zhang, Xin Yang, Dengchang Zhao, et al. EvoCUA: Evolving computer use agents via learning from scalable synthetic experience. *arXiv preprint arXiv:2601.15876*, 2026.

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

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

[9] Yichun Zhang, Xiangwu Guo, Yauhong Goh, Jessica Hu, Zhiheng Chen, Xin Wang, Difei Gao, and Mike Zheng Shou. ShowUI-Aloha: Human-taught gui agent. *arXiv preprint arXiv:2601.07181*, 2026.

[10] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

[11] Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Keunho Jang, and Zheng Hui. Windows agent arena: Evaluating multi-modal OS agents at scale. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pages 4874–4910. PMLR, 2025.

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

[13] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[14] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

[15] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: an open-source LLM reinforcement learning system at scale. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025.

[16] Danyang Zhang, Situo Zhang, Ziyue Yang, Zichen Zhu, Zihan Zhao, Ruisheng Cao, Lu Chen, and Kai Yu. Progrm: Build better GUI agents with progress rewards. *CoRR*, abs/2505.18121, 2025.

[17] Congmin Zheng, Xiaoyun Mo, Xinbei Ma, Qiqiang Lin, Yin Zhao, Jiachen Zhu, Xingyu Lou, Jun Wang, Zhaoxiang Wang, Weiwen Liu, Zhuosheng Zhang, Yong Yu, and Weinan Zhang. Adaptive milestone reward for GUI agents. *CoRR*, abs/2602.11524, 2026.

[18] Zhiheng Xi, Chenyang Liao, Guanyu Li, Zhihao Zhang, Wenxiang Chen, Binghai Wang, Senjie Jin, Yuhao Zhou, Jian Guan, Wei Wu, Tao Ji, Tao Gui, Qi Zhang, and Xuanjing Huang. Agentprm: Process reward models for LLM agents via step-wise promise and progress. In Hakim Hacid, Yoelle Maarek, Francesco Bonchi, Ido Guy, and Emine Yilmaz, editors, *Proceedings of the ACM Web Conference 2026, WWW 2026, Dubai, United Arab Emirates, originally scheduled for April 13-17, 2026, rescheduled for June 29 - July 3, 2026*, pages 4184–4195. ACM, 2026.

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

[20] Zhongwen Xu and Zihan Ding. Single-stream policy optimization. In *The Fourteenth International Conference on Learning Representations*, 2026.

[21] Ling Team, Anqi Shen, Baihui Li, Bin Hu, Bin Jing, Cai Chen, Chao Huang, Chao Zhang, Chaokun Yang, Cheng Lin, Chengyao Wen, et al. Every step evolves: Scaling reinforcement learning for trillion-scale thinking model, 2025.

[22] Guanhua Huang, Tingqiang Xu, Jinbo Wang, Guangming Sheng, Siheng Li, Evander Yang, Kejiao Li, Yunxiang Li, Zenan Xu, Qi Yi, Kyrierl Deng, Ziyuan Nan, Yuhao Jiang, Chenchen Zhang, Taiqiang Wu, Feiyuan Zhang, Junhao Wang, Bo Zhou, Alex Chen, Di Wang, and Shunyu Yao. Stabilizing RLVR via token-level gradient diagnosis and layerwise clipping, 2026.

[23] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[24] Weiqi Wang, Xin Liu, Binxuan Huang, Hejie Cui, Rongzhi Zhang, Changlong Yu, Shuowei Jin, Jingfeng Yang, Qingyu Yin, Zhengyang Wang, Zheng Li, Yifan Gao, Priyanka Nigam, Bing Yin, Lihong Li, and Yangqiu Song. Heapa: Difficulty-aware heap sampling and on-policy query augmentation for LLM reinforcement learning. In *Third Conference on Language Modeling*, 2026.

[25] Fanbin Lu, Zhisheng Zhong, Shu Liu, Chi-Wing Fu, and Jiaya Jia. Arpo:end-to-end policy optimization for GUI agents with experience replay. *CoRR*, abs/2505.16282, 2025.

[26] Songqin Nong, Jingxuan Xu, Sheng Zhou, Jianfeng Chen, Xiaoxuan Tang, Tao Jiang, and Wenhao Xu. CRAFT-GUI: curriculum-reinforced agent for GUI tasks. *CoRR*, abs/2508.11360, 2025.

[27] Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, Jie Tang, and Yuxiao Dong. Webrl: Training LLM web agents via self-evolving online curriculum reinforcement learning. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[28] Zhaoyang Liu, JingJing Xie, Zichen Ding, Zehao Li, Bowen Yang, Zhenyu Wu, Xuehui Wang, Qiushi Sun, Shi Liu, Weiyun Wang, Shenglong Ye, Qingyun Li, Xuan Dong, Yue Yu, Chenyu Lu, YunXiang Mo, Yao Yan, Zeyue Tian, Xiao Zhang, Yuan Huang, Yiqian Liu, Weijie Su, Gen Luo, Xiangyu Yue, Biqing Qi, Kai Chen, Bowen Zhou, Yu Qiao, Qifeng Chen, and Wenhai Wang. Scalecua: Scaling open-source computer use agents with cross-platform data. *CoRR*, abs/2509.15221, 2025.

[29] Moonshot AI. Kimi K2.6: Advancing open-source coding, 2026.

[30] Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026.

[31] Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026.

[32] OpenAI. GPT-5.5 system card, April 2026.

[33] Anthropic. Introducing Claude Sonnet 5, June 2026.

[34] Anthropic. Introducing Claude Opus 4.8, May 2026.

[35] 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, 2026.

[36] Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, Dunjie Lu, Haikong Lu, Qi Zhen, Xinyuan Wang, Jiaqi Deng, Yuhao Yang, Cheng Chen, Boyuan Zheng, Alex Su, Xiao Yu, Hao Zou, Saaket Agashe, Xing Han Lu, Manpreet Kaur, Zhengyang Qi, Vincent Sunn Chen, Frederic Sala, Dayiheng Liu, Junyang Lin, Zhou Yu, Yu Su, Siva Reddy, Xin Eric Wang, Peng Qi, Tianbao Xie, and Tao Yu. Osworld 2.0: Benchmarking computer use agents on long-horizon real-world tasks, 2026.

[37] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

[38] Jian Chen, Yesheng Liang, and Zhijian Liu. Dflash: Block diffusion for flash speculative decoding. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026.

[39] Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent s2: A compositional generalist-specialist framework for computer use agents. *arXiv preprint arXiv:2504.00906*, 2025.

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

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

[42] Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for gui agents. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14281–14290. IEEE, 2024.

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

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

[45] Boyu Gou, Demi 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. In *International Conference on Learning Representations*, volume 2025, pages 30851–30883, 2025.

[46] Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous gui interaction. *arXiv preprint arXiv:2412.04454*, 2024.

[47] Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Stan Weixian Lei, Lijuan Wang, and Mike Zheng Shou. Showui: One vision-language-action model for gui visual agent. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 19498–19508. IEEE, 2025.

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

[49] Venus Team, Changlong Gao, Zhangxuan Gu, Yulin Liu, Xinyu Qiu, Shuheng Shen, Yue Wen, Tianyu Xia, Zhenyu Xu, Zhengwen Zeng, et al. Ui-venus-1.5 technical report. *arXiv preprint arXiv:2602.09082*, 2026.

[50] Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, et al. Mai-ui technical report: Real-world centric foundation gui agents. *arXiv preprint arXiv:2512.22047*, 2025.

[51] Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. *Advances in Neural Information Processing Systems*, 37:12461–12495, 2024.

[52] Kaixin Li, Ziyang Meng, Hongzhan Lin, Ziyang Luo, Yuchen Tian, Jing Ma, Zhiyong Huang, and Tat-Seng Chua. Screenspot-pro: Gui grounding for professional high-resolution computer use. In *Proceedings of the 33rd ACM International Conference on Multimedia*, pages 8778–8786, 2025.

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

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

[55] Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 881–905, 2024.

[56] Chris Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Androidworld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, volume 2025, pages 406–441, 2025.

[57] Quyu Kong, Xu Zhang, Zhenyu Yang, Nolan Gao, Chen Liu, Panrong Tong, Chenglin Cai, Hanzhang Zhou, Jianan Zhang, Liangyu Chen, et al. Mobileworld: Benchmarking autonomous mobile agents in agent-user interactive and mcp-augmented environments. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6142–6167, 2026.

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

[59] 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.

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

[61] Frank Fangzheng Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, et al. Theagentcompany: benchmarking llm agents on consequential real world tasks. *Advances in Neural Information Processing Systems*, 38, 2026.

[62] Zilong Wang, Yuedong Cui, Li Zhong, Zimin Zhang, Da Yin, Bill Yuchen Lin, and Jingbo Shang. Officebench: Benchmarking language agents across multiple applications for office automation. *arXiv preprint arXiv:2407.19056*, 2024.

[63] Wil M. P. van der Aalst, Martin Bichler, and Armin Heinzl. Robotic process automation. *Business & Information Systems Engineering*, 60(4):269–272, 2018.

[64] Tianyi Chen, Yinheng Li, Michael Solodko, Sen Wang, Nan Jiang, Tingyuan Cui, Junheng Hao, Jongwoo Ko, Sara Abdali, Qing Xiao, Leon Xu, Suzhen Zheng, Hao Fan, Pashmina Cameron, Justin Wagle, and Kazuhito Koishida. CUA-Skill: Develop skills for computer using agent. *arXiv preprint arXiv:2601.21123*, 2026.

[65] Kangning Zhang, Shuai Shao, Qingyao Li, Jianghao Lin, Lingyue Fu, Shijian Wang, Wenxiang Jiao, Yuan Lu, Weiwen Liu, Weinan Zhang, and Yong Yu. MMSkills: Towards multimodal skills for general visual agents, 2026.

Appendix

A Source of Task Instructions

Converted open-source instructions. We adapt instructions from computer-use datasets, including AgentNet [1] and ScaleCUA [12], to our platforms. The selected instructions undergo both cleaning and platform conversion. We remove tasks requiring unavailable applications or containing ambiguous targets or unverifiable outcomes. We preserve operational intent while adapting application and platform references; for example, an Ubuntu application may be replaced by a Windows equivalent. This retains existing capability coverage while making instructions executable and unambiguous.

Atomic capabilities decomposed from existing rollouts. Following [6], we analyze failed or stalled rollouts, identify the operation or decision responsible, and isolate it as an independent subtask. This turns difficult portions of long workflows into targeted instructions for evaluating and improving atomic capabilities while directing data collection toward interactions that agents demonstrably find challenging. Decomposition also expands coverage of fundamental operating-system interactions and makes failure modes easier to evaluate in isolation.

Instructions grounded in real content. Although open-source and rollout-derived tasks cover atomic operations well but provide limited support for authentic cross-application workflows. We address this gap by generating instructions from real documents, spreadsheets, presentations, and static websites from InfiniteWeb [7]. Content previews let instructions reference concrete entities and relationships rather than artificial placeholders. Operational patterns from application manuals guide realistic long-horizon sequences, such as transferring website information into a spreadsheet and using the analysis to update a document or presentation. Grounding instructions in real content makes the tasks more realistic and their outcomes easier to evaluate.

Capability-tree-driven task generation. Finally, we construct capability trees from application specifications, organizing functionality into operational modes and fine-grained capabilities. This top-down process complements collection from datasets, rollouts, and real files by surfacing underrepresented operations that common workflows might crowd out. The resulting targeted instructions systematically fill remaining capability gaps.

B Details of DemoCUA Data Generation

To supplement the three-stage data generation pipeline outlined in Section 5.2, we provide fine-grained specifications for the evaluation (Score) and post-processing (Filter & Repair) phases.

During Score, each rollout is evaluated across three dimensions via a hybrid mechanism (Table 8). First, for trajectory-level Whole-Task Completion (S1), a VLM judge with a Skeptical-Auditor prompt assesses task success from a visual chain. Second, for subtask-level Per-Subtask Completion (S2), the VLM evaluates subtasks from boundary frames and self-reports. Third, for Demo-following adherence (S3–S4), rule-based metrics calculate Subtask Completion Rate and measure Execution-Order Alignment against the reference workflow via Longest Common Subsequence (LCS).

During Filte & Repair, rather than discarding trajectories with fixable errors, we recover valid training data by applying deterministic and LLM-based repairs (Table 9). Deterministic program rules (R1, R2, R4) handle format normalization by collapsing duplicate reports, closing unclosed trajectories, re-deriving subtask IDs, and stripping leading WAIT or empty actions. For missing boundary reports where subtask transitions occur without explicit calls (R3), an LLM synthesizes the missing report from surrounding context.

Table 8 Scoring rules used to evaluate each rollout. Every rollout is assessed along three orthogonal dimensions. K is a small constant specifying how many trailing frames are appended to the visual chain used by the VLM in rule S1. Let $\mathcal{S}$ be the set of subtasks defined by the workflow and $|\mathcal{S}|$ its cardinality (i.e., the total number of subtasks).

ID	Dimension	Item	Type	Judging Method
S1	Trajectory-level	Whole-Task Completion	VLM	A Skeptical-Auditor prompt is fed to the VLM together with a multi-frame visual chain (initial + uniformly sampled intermediate + last-K frames); the VLM returns a binary verdict on whether the full task is completed.
S2	Subtask-level	Per-Subtask Completion	VLM	For every subtask, its boundary frames and its self-report sentences are passed to the VLM, which independently returns $\text{passed}(s) \in \{\text{True}, \text{False}\}$ for each subtask $s \in \mathcal{S}$.
S3	Demo-following	Subtask Completion Rate	Rule	Fraction of subtasks judged as passed by S2, computed as $\#\{\, s \in \mathcal{S} \mid \text{passed}(s) = \text{True} \,\} \,/\, |\mathcal{S}|$.
S4	Demo-following	Execution-Order Alignment	Rule	Longest-common-subsequence similarity between the agent's actual visit order $\mathbf{o}_{\text{agent}}$ (extracted from the trajectory) and the reference visit order $\mathbf{o}_{\text{demo}}$ defined by the demo workflow, normalized as $\text{LCS}(\mathbf{o}_{\text{agent}}, \mathbf{o}_{\text{demo}}) \,/\, |\mathcal{S}|$.

Table 9 Repair rules applied to rollouts. Each rule targets one action/report issue, executed by a deterministic program (**Rule**) or language model (**LLM**). Subtask reports refer to `report(subtask_id, status)` calls at boundaries.

ID	Item	Type	Repair Method
R1	Duplicate Reports	Rule	Detect adjacent `report` calls that share the same `subtask_id` and `status`, and collapse them into one.
R2	Unclosed Trajectory	Rule	If the trajectory terminates without a final `report(subtask_id, done)`, append one to close the last subtask.
R3	Missing Boundary	LLM	When a subtask transition is present in the action stream but the corresponding boundary `report` is absent, an LLM synthesizes the missing report from surrounding context.
R4	Step-Level Cleanup	Rule	Re-derive subtask boundaries from the report stream and rewrite each step's `subtask_id` to align with the correct segment; in the same pass, strip any leading `WAIT` or empty action so that the first step is action-effective.

C Details of GameDev Tasks

Table 10 Ten curated task case studies (1/2). Instructions are verbatim; rubric entries enumerate equal-weight artifact checks.

Original instruction (verbatim)	Detailed artifact-based rubric
obsidian-01 **Case 1: Appearance and workspace**	30 checks
Starting from the default Ubuntu desktop, install Obsidian and launch its AppImage from a terminal with the `-password-store=basic` flag (for example, `./Obsidian-1.12.7.AppImage -password-store=basic`), then create or open an Obsidian vault. In Obsidian, enable community plugins, install and enable the Style Settings plugin, then install and enable the AnuPpuccin theme. Make sure the Fira Code and Maple Mono NF CN fonts are actually installed on the system. In Obsidian's Appearance settings, set the interface font to Maple Mono NF CN, set the monospace font to Fira Code, select AnuPpuccin as the theme, and use the dark appearance mode. In Style Settings, configure the AnuPpuccin options as follows: set the dark theme flavor to Frappe, set the active line highlight to the border style, and set callouts to the block style. Enable file icons, the floating header, collapsed folder styling, and custom vault title styling. Set rainbow folders to the simple rainbow style, and enable rainbow coloring for folder titles, collapse icons, indentation lines, folder icons, and subfolders. Also enable the floating status bar, mini tabs, and card layout. Finally, arrange the Obsidian right sidebar into two vertical sections, with the graph view in the upper section and the outline view in the lower section. Make sure all plugin, theme, font, Style Settings, and sidebar layout changes are saved.	**Installation (6):** Obsidian executable plus registered vault; Style Settings enabled; real plugin installation; real AnuPpuccin installation; Fira Code installed; Maple Mono NF CN installed. **Appearance (4):** AnuPpuccin selected; dark mode; Fira Code monospace font; Maple Mono NF CN interface font. **Style Settings (16):** Frappe; border active line; block callouts; file icons; floating header; collapsed folders; custom vault title; simple-rainbow style; rainbow titles; collapse icons; indentation lines; folder icons; subfolders; floating status bar; mini tabs; card layout. **Workspace (4):** right sidebar exists; top/bottom split; Graph in upper branch; Outline in lower branch.
qgis-01 **Case 2: CSV to labeled map**	16 checks
Starting from the default Ubuntu desktop, download and install QGIS, then download the cities dataset CSV file (`Cities_Asia.csv`, containing Name, Latitude, and Longitude columns for 14 Asian cities) from this Google Drive folder into the Downloads folder: https://drive.google.com/drive/folders/1sp-LIgRxUvkHaGEG16nToIW4NjLC77U2?usp=sharing. In QGIS, install and enable the QuickMapServices plugin from the official plugin repository, then add the OSM Standard basemap to the map and set its transparency to 50%. Import the CSV as a delimited-text point layer using the Longitude and Latitude fields with the WGS 84 (EPSG:4326) coordinate reference system. Style the city points using a simple marker with a point size of 4 millimeters, and enable single labels on the layer using the Name field. Export the point layer to the Desktop as an ESRI Shapefile named `cities_asia` (producing the `.shp/.shx/.dbf/.prj` files) in EPSG:4326, keeping the Name, Latitude, and Longitude attributes. Finally, save the QGIS project to the Desktop as `cities_asia.qgz`. Make sure QGIS, the downloaded CSV, the installed plugin, the exported shapefile, and the saved project are all persisted to disk.	**Installation (1):** real QGIS executable plus profile or parseable-project evidence. **Shapefile (6):** all four files/openable layer; Point geometry; exactly 14 features; EPSG:4326; all coordinates match without longitude/latitude swap; exactly the Name/Latitude/Longitude fields and matching values. **Project (7):** Desktop QGZ exists; project CRS is EPSG:4326; vector layer points to `cities_asia`; 4 mm SimpleMarker; single labels from Name; OSM Standard tile source; 50% opacity/transparency. **Plugin (1):** QuickMapServices both installed and enabled. **Source data (1):** exact CSV exists in Downloads, or all core exported data is correct.
godot-01 **Case 3: Install and initialize**	7 checks
Install the supplied offline Godot 4.3 Linux build from `/home/user/Desktop/Godot_v4.3-stable_linux.x86_64.zip`. Extract the archive into `/home/user/Desktop/Godot` so the executable is exactly `/home/user/Desktop/Godot/Godot_v4.3-stable_linux.x86_64`, make that file executable, and launch it successfully. In the Godot Project Manager, create a new project named MyFirstGame at `/home/user/Desktop/myfirstgame`. Use the Godot 4.3 Forward Plus renderer and create the project in its own folder with Git version-control metadata and the normal default project files, including `project.godot`, `icon.svg`, `.gitignore`, and `.gitattributes`. Open the new project in the editor and leave all project files saved on disk. Use only the supplied offline build; do not install a different Godot version.	**Installation (3):** dedicated Desktop directory; pinned executable at the exact path; executable reports Godot 4.3. **Project (4):** project at the required path; name is MyFirstGame; Forward Plus configuration; default icon reference plus `.gitignore` and `.gitattributes`.
godot-02 **Case 4: Assemble the game scene**	14 checks
Using Godot 4.3, create the Forward Plus project MyFirstGame at `/home/user/Desktop/myfirstgame` and copy the supplied `/home/user/Desktop/AssetBundle` folder into the project root so its Sprites, Audio, and `Uranus_Pixel_11Px.ttf` resources are available under `res://AssetBundle`. Keep canvas texture filtering on nearest/no filtering so the pixel art remains crisp. Create a Scenes folder and save a main 2D scene as `Scenes/Game.tscn`. Use a Node2D root. Add Sprite2D children named Background 1 and Background 2 that both use `AssetBundle/Sprites/ForestBackground.png`. Place them side by side with no gap and center the combined continuous forest/road map around the world origin. Add a Camera2D at the origin with uniform zoom (2.415, 2.415) so the complete map is framed. Set `Game.tscn` as `run/main_scene`. Create and save `Scenes/Player.tscn` with a CharacterBody2D root named Player and an AnimatedSprite2D child. Create a SpriteFrames animation named idle from `AssetBundle/Sprites/Foxy.png`: slice `Foxy.png` along its natural, evenly spaced character-frame grid without crossing frame boundaries, use the first four cells of the first row in order, loop the animation at 12 fps, and make idle autoplay. Instance `Player.tscn` into `Game.tscn`, save both scenes and the project, keep the Player instance at the Game origin, and verify the main scene runs with the idle fox visible on the two-part background.	**Project (4):** MyFirstGame/Forward Plus identity; `Game.tscn` is main; nearest texture filter; complete sprite/audio/font asset bundle. **Scene roots (2):** Game has Node2D root; Player has CharacterBody2D root. **Idle animation (3):** four frames at 12 fps; autoplay; 33×32 first-row atlas cells. **World (3):** two ForestBackground sprites; one left of center; one right of center. **Camera/player (2):** Camera2D at approximately 2.4× zoom; Game instances `Player.tscn`.
godot-03 **Case 5: Implement WASD movement**	10 checks
In the supplied MyFirstGame project, configure four Input Map actions with the default 0.5 deadzone: left bound to the physical A key, right to D, up to W, and down to S. Preserve the existing Game and Player scenes and idle animation. Create `Scripts/player.gd`, make it extend CharacterBody2D, and attach it to the Player root in `Scenes/Player.tscn`. Declare `@export var move_speed: float = 50.0` as the script default, then set the Player scene's Inspector override for `move_speed` to 100 so the saved scene value differs from the script default. In `_process(delta)`, read `Input.get_vector("left", "right", "up", "down")`, multiply the result by `move_speed`, assign it to `velocity`, and call `move_and_slide()`. Save `player.gd`, `Player.tscn`, and `project.godot`, then ensure `Scripts/player.gd` contains no debug print statements or hard-coded velocity assignment in `_ready()`, run the game, and verify WASD moves the player in all four directions.	**Input Map (4):** left=A; right=D; up=W; down=S using physical key codes. **Script/configuration (2):** `player.gd` attached to CharacterBody2D; exported speed default 50 and scene override about 100. **Movement (3):** all four actions feed an input vector and velocity; movement runs in a frame/physics callback; `move_and_slide()` is called. **Cleanup (1):** no temporary debug print.

Table 11 Ten curated task case studies (2/2). Cases are independently evaluated; all listed rubric checks have equal weight within each case.

Original instruction (verbatim)	Detailed artifact-based rubric
godot-04 **Case 6: Animate running and add borders**	18 checks
Download and install Godot 4, then open the MyFirstGame project located at `/home/user/Desktop/myfirstgame` (a top-down 2D game with a fox player on a forest/road map). Make the following two changes and save everything to disk. 1) Player run animation. On the player's AnimatedSprite2D, add a SpriteFrames animation named "run" using the `Foxy.png` sprite sheet from the asset bundle on the Desktop (the AssetBundle folder at `/home/user/Desktop/AssetBundle`): figure out the sprite sheet's grid, slice it into cells, and take the second row (the running frames) as its 6 frames, and set the playback speed to 12 fps. Keep the existing "idle" animation, and keep autoplay on idle (run must not auto-play). In `player.gd`, run the movement in `_physics_process`: drive velocity from the existing left/right/up/down input actions, and switch the AnimatedSprite2D between "idle" when stopped and "run" when moving, then call `move_and_slide()`. Expose the AnimatedSprite2D to the script (e.g. an `@export`/`@onready` reference) and make sure it is actually assigned to that node, so the animation calls work. Also add a CollisionShape2D to the player with a CircleShape2D whose center is moved down toward the fox's feet. 2) Map borders. In the Game scene add four WorldBoundaryShape2D static walls confining the player - near the bottom, left, and right map edges, and a top wall along the middle of the map (the forest/road divide, not the very top) - each oriented so its solid side faces inward. Group the four wall bodies under one parent Node2D and lock that node.	**Animation (6):** "run" exists; "idle" remains; run has 6 frames; run speed passes the requested minimum; run is not autoplay; frames are sliced atlas regions. **Controller (5):** script derives velocity from input; switches run/idle; calls `move_and_slide()`; uses `_physics_process`; animator reference is bound. **Collider (2):** player collision shape exists; center is shifted toward the feet. **Borders (5):** bottom, left, right, and middle-top boundaries are correctly placed; all four are grouped under a locked parent.
godot-05 **Case 7: Trigger collision and game over**	17 checks
Create and save `Scenes/Slime.tscn` as a reusable Area2D enemy scene. Add an AnimatedSprite2D using `AssetBundle/Sprites/Slimer.png`, split the strip into its complete sequence of evenly sized slime poses, and create a looping animation using all frames at 12 fps with autoplay. Add a CollisionShape2D with a CircleShape2D sized to the slime's solid body and shift its center downward toward the slime's base. Create `Scripts/enemy.gd`, attach it to the Area2D, and declare `@export var slime_speed: float = -100.0` as the script default. Use `slime_speed` to move the slime left at a frame-rate-independent pace during physics updates, then set the Slime scene's Inspector override for `slime_speed` to about -50 so the saved scene value differs from the script default. Connect the Slime Area2D `body_entered` signal to `enemy.gd`. In the handler, detect a CharacterBody2D player and call its `game_over()` function. Instance one fully visible Slime in `Game.tscn` on the road to the right of the player for testing, while preserving Y-sorted gameplay rendering. Extend `Player.tscn` with a looping `game_over` animation built from the complete failure-action row in `Foxy.png`, ordered from left to right and played at about 6 fps. In `player.gd`, add an `is_game_over` boolean initially false. Only process normal input, idle/run animation, and `move_and_slide` while the game is not over. Implement `game_over()` to set the flag, play the `game_over` animation, await a SceneTree timer of about three seconds, and reload the current scene. Save and verify touching the slime stops player movement, shows the failure animation, and restarts.	**Slime scene (6):** Area2D root with script; eight ordered 41×38 first-row frames at 12 fps; autoplay; circle collider; downward collider offset; speed default −100 and override about −50. **Signal/movement (4):** `body_entered` connection; physics movement; frame-rate-independent use of exported speed; handler calls Player `game_over()`. **Placement (2):** Slime instance exists in Game; instance lies in the lower-right road region. **Animation (2):** six-frame looping `game_over`; speed about 6 fps. **Player logic (3):** game-over state/function; normal movement guard; delayed scene reload. Preserved main-scene Y-sort is an unweighted regression gate.
godot-06 **Case 8: Fire bullets on a timer**	15 checks
Create `Scenes/Bullet.tscn` with an Area2D root named Bullet. Add a Sprite2D using `AssetBundle/Sprites/Bullet.png` and a CollisionShape2D using a RectangleShape2D fitted closely to the visible bullet sprite. Create and attach `Scripts/bullet.gd`, and declare `@export var bullet_speed: float = 100.0` as the script default. Set the Bullet scene's Inspector override for `bullet_speed` to about 300 so the saved scene value differs from the script default, then move the bullet right in `_physics_process` with `position += Vector2(bullet_speed, 0) * delta`. In `_ready`, await a SceneTree timer of about three seconds and queue_free the bullet so missed shots cannot accumulate indefinitely. In `Scripts/player.gd`, declare `@export var bullet_scene: PackedScene`. In `Player.tscn`, bind that exported property to `Bullet.tscn`. Add an automatically starting, continuously repeating Timer child with an approximately one-second trigger interval and connect timeout to `player.gd::_on_fire`. In `_on_fire`, return without shooting whenever velocity is not `Vector2.ZERO` or `is_game_over` is true. Otherwise instantiate `bullet_scene`, place the new bullet at the player's right-side gun muzzle so it visibly emerges from the character's weapon, and add it to `get_tree().current_scene`. Ensure `Game.tscn` contains no pre-placed Bullet instance, save all files, and verify the stationary player fires repeatedly while moving or game-over states suppress firing and old bullets self-destruct.	**Bullet scene (7):** Area2D root with `bullet.gd`; `Bullet.png`; closely fitted rectangle collider; fast scene speed override; speed-driven physics movement; timer plus `queue_free`; lifetime about 3 seconds. **Reference (1):** Player exports `bullet_scene` and binds it to `Bullet.tscn`. **Timer (4):** autostart; about 1 second; repeating; timeout connected to `_on_fire`. **Firing (3):** dynamic instantiation with no pre-placed Bullet; blocked while moving/game-over; player-relative right-muzzle offset.
godot-07 **Case 9: Add death and dynamic spawning**	19 checks
Add the Bullet Area2D root to a Godot group named bullet. In `Slime.tscn`, rename the existing looping animation to idle and keep it autoplaying. Add a non-looping death animation using every pose in `AssetBundle/Sprites/SlimerDeath.png` at the demonstrated 12 fps pace. Connect the Slime Area2D `area_entered` signal to `enemy.gd`. In `enemy.gd`, add an `is_dead` boolean initially false and only move the slime while it is not dead. At the start of the area-entered handler, return immediately when `is_dead` is already true. Otherwise check `area.is_in_group("bullet")`; on the first hit play death, set `is_dead` true, queue_free the bullet area, await about 0.6 seconds for the death animation, and then queue_free the slime itself. Preserve the separate `body_entered` player-collision game-over behavior. Create `Scripts/GameManager.gd` extending Node2D and attach it to the Game scene root. Export `slime_scene` as PackedScene and `spawn_timer` as Timer, and bind them to `Slime.tscn` and a Game Timer child. Configure that Timer with `wait_time` 3, autostart enabled, and timeout connected to `_spawn_slime`. Remove the old pre-placed Slime instance. In `_spawn_slime`, instantiate `slime_scene`, place it just beyond the map's right edge at a random height that keeps the complete sprite within the playable road band, and add it to the current scene. In `_process`, gradually subtract about `0.2 * delta` from `spawn_timer.wait_time` and clamp the interval between 1 and 3 seconds. Save and verify slimes spawn with increasing frequency and bullets trigger one clean death sequence.	**Bullet group (1):** Bullet Area2D belongs to group `bullet`. **Spawner (9):** GameManager attached; scene/Timer bindings; no pre-placed Slime; 3-second autostart Timer; timeout signal; instantiate/add logic; beyond-right randomized road-safe position; interval adjusted and clamped; decay about 0.2/s within 1–3 seconds. **Animations (3):** eight-frame `idle` at 12 fps; idle autoplay; seven-frame non-looping `death` at 12 fps. **Death logic (6):** `area_entered` connection; bullet-group test; repeated-hit guard and death state; bullet and slime cleanup; about 0.6-second delay; dead slime stops moving. The inherited `body_entered` game-over path is an unweighted regression gate.
godot-08 **Case 10: Clean up off-screen enemies**	2 checks
Add off-screen slime cleanup in `Scripts/enemy.gd`. The map's left collision wall is approximately x = -235; use x = -267 as the off-screen cleanup threshold. In `_physics_process`, after updating slime movement, call `queue_free()` when `position.x < -267`. Save `enemy.gd`.	**Boundary (1):** `enemy.gd` checks that the slime has moved beyond the left boundary. **Cleanup order (1):** the physics callback updates movement before freeing a slime that crossed the boundary.

D GameDev Performance of Kimi-K2.6

Table 12 Kimi K2.6 performance on GameDev without and with demonstrations. Results are evaluated over five runs. The two Avg. columns report the corresponding mean success rates, while Δ denotes the Demo - No-Demo difference in percentage points. Higher is better.

Task	No-Demo					Avg.	Demo					Avg.	Δ
	1	2	3	4	5		1	2	3	4	5		
godot-01	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
godot-02	92.86	71.43	92.86	78.57	92.86	85.71	92.86	71.43	50.00	85.71	85.71	77.14	-8.57
godot-03	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
godot-04	78.95	52.63	84.21	63.16	68.42	69.47	94.74	84.21	68.42	84.21	84.21	83.16	13.68
godot-05	82.35	41.18	64.71	82.35	29.41	60.00	70.59	58.82	70.59	76.47	70.59	69.41	9.41
godot-06	86.67	93.33	86.67	80.00	93.33	88.00	93.33	86.67	86.67	100.00	86.67	90.67	2.67
godot-07	84.21	89.47	36.84	78.95	84.21	74.74	78.95	78.95	84.21	89.47	52.63	76.84	2.11
godot-08	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
obsidian-01	80.00	90.00	90.00	83.33	90.00	86.67	76.67	90.00	76.67	96.67	96.67	87.33	0.67
qgis-01	75.00	68.75	68.75	68.75	68.75	70.00	100.00	100.00	100.00	100.00	100.00	100.00	30.00
Average	88.00	80.68	82.40	83.51	82.70	**83.46**	90.71	87.01	83.66	93.25	87.65	**88.46**	**5.00**

Table 13 Trajectory lengths and task scores for Kimi K2.6 on GameDev. Human Demo reports the number of steps in the human demonstration, while No-Demo and Demo report the average model steps and normalized task scores over five runs. Tasks requiring longer trajectories are generally associated with lower scores.

Task	Human Demo	No-Demo		Demo	
	Steps	Avg. Steps	Score (%)	Avg. Steps	Score (%)
godot-01	35	22.2	100.00	31.8	100.00
godot-02	223	253.4	85.71	190.6	77.14
godot-03	202	103.0	100.00	145.8	100.00
godot-04	247	387.4	69.47	418.2	83.16
godot-05	380	450.6	60.00	445.0	69.41
godot-06	323	213.8	88.00	233.6	90.67
godot-07	376	433.2	74.74	388.6	76.84
godot-08	46	104.8	100.00	121.0	100.00
obsidian-01	255	193.0	86.67	463.2	87.33
qgis-01	305	100.4	70.00	136.8	100.00
Average	**239.2**	**226.2**	**83.46**	**257.5**	**88.46**

Table 12 shows that Kimi K2.6 already has strong long-horizon task-completion ability without demonstrations, achieving a mean score of 83.46 and fully solving three tasks. Demonstrations raise the mean from 83.46 to 88.46, a gain of 5.00 points, and improve six of the seven tasks not already at ceiling. The largest gains are 30.00 points on QGIS, 13.68 on `godot-04`, and 9.41 on `godot-05`, indicating better coverage of fine-grained artifact requirements. Table 13 provides a trajectory-level view: without demonstrations, we observe that Kimi often uses CLI commands to bypass lengthy GUI sequences, allowing it to complete tasks in fewer steps on average than the pure-GUI human reference. With demonstrations that primarily follow GUI workflows, Kimi takes 257.5 steps on average but achieves higher scores, suggesting that these workflows help it better satisfy fine-grained task requirements.

E Details of the OSWorld-Subset under DemoCUA setting

Table 14 reports per-task results on the 30-task OSWorld subset. Self-demonstrations increase the average score from 40.27% to 65.75%, a gain of 25.48 percentage points. Performance improves on 18 tasks, remains unchanged on eight, and decreases on four, demonstrating a substantial overall benefit despite several cases of negative transfer.

Table 15 further compares trajectory lengths with the human demonstrations. Self-demonstrations reduce the average model trajectory from 33.3 to 30.0 steps while improving task completion.

Table 14 OSWorld Per-task performance with and without demonstrations on UI-Mate-27B. Results are evaluated over five runs. The two Avg. columns report the corresponding mean success rates, while Δ denotes the Demo - No-Demo difference in percentage points. Higher is better.

Task	No-Demo					Avg.	Demo					Avg.	Δ
	1	2	3	4	5		1	2	3	4	5		
chrome-01	100.00	0.00	0.00	0.00	100.00	40.00	100.00	100.00	100.00	100.00	100.00	100.00	60.00
chrome-02	0.00	0.00	0.00	0.00	0.00	0.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
chrome-03	0.00	0.00	0.00	0.00	0.00	0.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
chrome-04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
gimp-01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	100.00	40.00	40.00
calc-01	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
calc-02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	20.00	20.00
calc-03	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
calc-04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00	20.00	20.00
calc-05	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
impress-01	0.00	100.00	0.00	0.00	0.00	20.00	100.00	100.00	100.00	100.00	100.00	100.00	80.00
impress-02	100.00	0.00	0.00	100.00	100.00	60.00	100.00	100.00	100.00	100.00	100.00	100.00	40.00
writer-01	0.00	100.00	0.00	100.00	0.00	40.00	100.00	100.00	100.00	100.00	0.00	80.00	40.00
writer-02	0.00	100.00	0.00	0.00	100.00	40.00	0.00	0.00	0.00	0.00	0.00	0.00	−40.00
writer-03	100.00	0.00	0.00	0.00	100.00	40.00	0.00	0.00	0.00	100.00	0.00	20.00	−20.00
multi-01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
multi-02	0.00	0.00	0.00	0.00	0.00	0.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
multi-03	100.00	100.00	100.00	100.00	100.00	100.00	0.00	0.00	0.00	0.00	0.00	0.00	−100.00
multi-04	0.00	0.00	0.00	100.00	0.00	20.00	0.00	100.00	100.00	100.00	0.00	60.00	40.00
multi-05	83.99	58.60	62.49	84.11	52.29	68.30	93.85	93.85	93.85	93.85	93.85	93.85	25.55
multi-06	100.00	0.00	0.00	0.00	100.00	40.00	0.00	100.00	100.00	100.00	0.00	60.00	20.00
multi-07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
multi-08	100.00	100.00	100.00	100.00	100.00	100.00	0.00	100.00	100.00	100.00	100.00	80.00	−20.00
multi-09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
os-01	0.00	0.00	0.00	0.00	0.00	0.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
vlc-01	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00
vlc-02	100.00	100.00	0.00	0.00	100.00	60.00	100.00	100.00	100.00	100.00	100.00	100.00	40.00
vlc-03	98.70	0.00	0.00	0.00	0.00	19.74	98.63	98.63	98.63	98.70	98.63	98.64	78.90
vlc-04	100.00	100.00	0.00	100.00	100.00	80.00	100.00	100.00	100.00	100.00	100.00	100.00	20.00
vlc-05	100.00	100.00	100.00	100.00	0.00	80.00	100.00	100.00	100.00	100.00	100.00	100.00	20.00
Average	49.42	41.95	25.42	39.47	45.08	**40.27**	56.42	73.08	66.42	73.08	59.75	**65.75**	**25.48**

Table 15 Trajectory lengths and task scores with and without demonstrations on OSWorld-Subset 30 problems with UI-Mate-27B. Human Demo reports the number of steps in the human demonstration, while No-Demo and Demo report the average model steps and normalized task scores over five runs. Demonstrations make the model trajectory length track the human demonstration much more closely (rank correlation with the human step count rises from 0.66 to 0.86).

Task	Human Demo	No-Demo		Demo	
	Steps	Avg. Steps	Score (%)	Avg. Steps	Score (%)
chrome-01	5	9.0	40.00	8.8	100.00
chrome-02	13	87.4	0.00	21.6	100.00
chrome-03	24	10.8	0.00	18.0	100.00
chrome-04	17	34.4	0.00	64.0	0.00
gimp-01	4	4.8	0.00	6.6	40.00
calc-01	17	22.8	100.00	21.8	100.00
calc-02	14	25.2	0.00	35.6	20.00
calc-03	7	9.0	100.00	10.2	100.00
calc-04	45	26.2	0.00	50.6	20.00
calc-05	10	22.6	100.00	15.6	100.00
impress-01	16	55.0	20.00	26.4	100.00
impress-02	45	44.8	60.00	54.8	100.00
writer-01	4	12.8	40.00	7.0	80.00
writer-02	16	85.2	40.00	31.0	0.00
writer-03	22	40.0	40.00	25.6	20.00
multi-01	32	60.8	0.00	48.4	0.00
multi-02	21	36.0	0.00	44.0	100.00
multi-03	26	45.0	100.00	50.6	0.00
multi-04	39	84.2	20.00	49.6	60.00
multi-05	20	37.4	68.30	25.6	93.85
multi-06	21	19.8	40.00	27.0	60.00
multi-07	11	11.4	0.00	15.2	0.00
multi-08	39	69.8	100.00	73.6	80.00
multi-09	13	28.4	0.00	22.4	0.00
os-01	42	35.8	0.00	54.0	100.00
vlc-01	8	12.8	100.00	15.2	100.00
vlc-02	3	5.6	60.00	10.0	100.00
vlc-03	8	18.4	19.74	13.0	98.64
vlc-04	12	30.2	80.00	26.8	100.00
vlc-05	14	13.8	80.00	25.8	100.00
Average	**18.9**	**33.3**	**40.27**	**30.0**	**65.75**

F Details of the OSWorker-Subset under DemoCUA setting

Table 16 Trajectory lengths and task scores for UI-Mate-27B on the 33-task OSWorkerBench self-demo subset. For each target, Demo provides a successful stronger-agent rollout of that same task (self-demo setting), represented by screenshots and action types without concrete pixel coordinates. No-Demo and Demo report the average model steps and normalized task scores over three runs under otherwise identical conditions. Demos improve average task score by 13.29 percentage points but also increase trajectory length, because these tasks involve repetitive or branching subtasks that the model may fail to infer from instructions alone and thus declares completion prematurely. The longer trajectories reflect higher task completeness rather than lower efficiency.

Task	No-Demo		Demo	
	Avg. Steps	Score (%)	Avg. Steps	Score (%)
task-01	250.3	95.67	265.3	98.33
task-02	145.0	78.89	203.3	68.33
task-03	224.0	53.33	313.3	18.33
task-04	174.3	43.75	264.0	41.67
task-05	270.3	79.58	278.0	81.25
task-06	139.7	93.33	123.3	93.33
task-07	384.3	0.00	454.3	15.67
task-08	220.7	93.33	393.3	97.67
task-09	80.3	86.56	120.3	100.00
task-10	230.7	43.33	191.7	49.17
task-11	196.7	72.33	213.3	76.00
task-12	319.0	61.69	205.7	93.35
task-13	205.0	76.58	253.0	78.48
task-14	164.3	32.67	300.7	58.00
task-15	168.7	98.33	244.3	100.00
task-16	125.3	54.17	301.3	94.78
task-17	133.7	57.22	99.7	93.33
task-18	106.7	52.97	165.7	97.92
task-19	165.0	72.00	171.3	97.33
task-20	273.7	63.29	329.0	85.62
task-21	200.7	62.00	293.0	84.00
task-22	157.3	48.83	219.7	90.00
task-23	82.3	78.36	83.7	84.17
task-24	156.7	68.55	140.7	100.00
task-25	183.0	58.67	214.7	84.33
task-26	109.0	93.33	122.7	100.00
task-27	160.0	53.96	223.7	62.50
task-28	101.0	98.33	112.0	100.00
task-29	140.3	46.19	198.0	66.81
task-30	122.0	100.00	89.0	82.22
task-31	140.3	56.67	283.7	96.19
task-32	59.3	82.19	114.3	91.26
task-33	130.0	82.92	142.3	97.50
Average	**173.3**	**67.85**	**216.0**	**81.14**

Example. Task 18 requires the model to classify six inbound emails by tier and then, for each qualifying lead, update Salesforce, create calendar events, and post messages to Slack. Without a demonstration, the model correctly classifies all six emails but processes only one of the two "Hot" leads, losing track of the second in the long interaction history before incorrectly declaring the task complete (with an average of 106.7 steps and a score of 52.97%). Although the demonstration does not explicitly specify which leads should be processed, it provides a reference workflow that illustrates how the same sequence of actions should be repeated for every qualifying lead. This guides the model to process both "Hot" leads more completely, including updating Salesforce, creating follow-up tasks, and scheduling calendar events, resulting in an average of 165.7 steps and a score of 97.92%.