Computer-Use Agents as Judges for Generative User Interface

Anonymous ACL submission

Abstract

Computer-Use Agents (CUA) are becoming increasingly capable of autonomously operating digital environments through Graphical User Interfaces (GUI). Yet, most GUI remain designed primarily for humans (such as aesthetics), and there is limited discussion of whether such human-oriented interfaces are optimal for agents. At the same time, rapid advances in coding-oriented language models (Coder) have transformed automatic GUI design. This raises a fundamental question: *Can CUAs serve as judges to assist Coders in automatic UI design*? To investigate, we introduce **AUI-Gym**, a benchmark for Automatic UI development spanning 52 applications across diverse domains. Using language models, we synthesize 1560 tasks that simulate real-world scenarios. To ensure task reliability, we further develop a verifier that programmatically checks whether each task is executable within its environment. Building on this, we propose a **Coder–CUA in Collaboration** framework: the Coder acts as Designer, generating and revising websites, while the CUA serves as Judge, evaluating functionality and refining designs. Success is measured not by visual appearance, but by task solvability and CUA navigation success rate. To turn CUA feedback into usable guidance, we design a **CUA Dashboard** that compresses multi-step navigation histories into concise visual summaries, offering interpretable guidance for iterative redesign. By positioning agents as both designers and judges, our framework shifts interface design toward agent-native efficiency and reliability. Our work takes a step toward shifting agents from passive use toward active participation in digital environments.

1 Introduction

Recent advances in language agents have shown that **Computer-Use Agents** (openai, 2025; Anthropic, 2025) can autonomously operate within GUIs—performing tasks such as online shopping

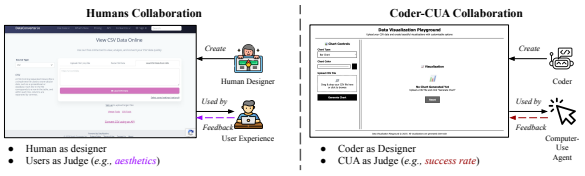


Figure 1: **Illustration of Human Collaboration *vs.* our Coder-CUA Collaboration in terms of UI designs. Left:** Most GUIs are designed by humans and optimized for user experience (*e.g.,* aesthetics), forcing trained agents to adapt to human-oriented behaviors. **Right:** Our Coder-CUA Collaboration framework leverages Coder as Designer and CUA as Judge together, enabling more reliable task execution and improved usability for agents.

by sequentially clicking through multiple buttons (Zhou et al., 2024). However, today's environments remain fundamentally human-centric, optimized for aesthetics and usability through features like dynamic animations or colorful layouts. To adapt to these settings, researchers typically train CUA on large-scale human demonstration trajectories, click logs, or static screenshots (Xu et al., 2024; Lin et al., 2025; Seed, 2025), effectively forcing agents to imitate human behavior. This approach binds automation to human-oriented design choices, where stylistic details crucial for humans are redundant for agents whose primary objective is efficient task completion. In parallel, coding-oriented language models—**Coders**—have already demonstrated strong capabilities, capable of generating functional HTML pages or even entire websites from a single instruction (Si et al., 2024). Yet these outputs remain confined to human-facing loops: even when generated by agents, interfaces are still judged by human rather than agent interaction. Despite terminal are recognized as agent-native interfaces for coder agents. However, it remains unclear *whether visually represented GUIs can be optimized for CUAs* (see Appendix F for more discussion).

Observing that LLMs and Agents serve as effective judges (Gu et al., 2024; Chen et al., 2024;

Zhuge et al., 2024; Gou et al., 2025), we ask: can CUAs assist Coders in redesigning UIs automatically, where environments are created for, and evaluated by, agents themselves, with CUA acting as judges? This offers practical value such as automatic UI testing (Liu et al., 2025). In this work, we reconceptualize the UI as a tunable environment. The core idea is to employ the ***Coder as Designer***, responsible for initializing and revising UIs, while the ***CUA acts as Judges***, navigating through tasks and collecting trajectories as feedback.

There is no testbed that aligns with our goal, we introduce **AUI-Gym** to pioneer evaluation in this setting. AUI-Gym automatically develops websites across 52 applications spanning six domains. Unlike most coders that focus on single-page generation, AUI-Gym requires agents to produce fully automated, executable, application-level designs with an emphasis on functional completeness. Enabling sufficient, scalable, and human-minimal evaluation is non-trivial. To ensure that each website can be reliably tested, we use LLMs to generate a customized rule-based functional checker for each individual task, determining whether the task is feasible within the given interface. This infrastructure establishes a human-minimal, reliable foundation for subsequent CUA exploration and feedback-driven UI refinement.

To this end, we develop a **Coder–CUA collaboration** framework to study whether we can design visual interface to optimize the CUA task success rate. The *Coder acts as Designer*, responsible for UI initialization and refinement, while the *CUA serves as Judge*, supplying feedback. The central challenge is how to transform raw CUA interactions into effective revision signals from an agent perspective. We address this through two complementary dimensions of feedback: **(a) CUA Judge**, where the agent executes tasks through atomic actions such as clicks or typing and judges success or failure; and **(b) Task Solvability**, where unsolvable tasks are accumulated as functionality failures and returned to the Coder as precise indicators of missing features. CUA produces long, multi-step trajectories interleaved with screenshots, making direct feedback difficult to interpret. To overcome this, we introduce the **CUA Dashboard**, which condenses each task, its outcome, actions, and intermediate states into a single $2K$ resolution image. Rather than storing every screenshot, the dashboard highlights only key interactive regions, with region sizes adaptively scaled by the number of steps. This dynamic design reduces redundancy by average 76.2% while preserving essential cues, offering a clear step-by-step view of how the CUA perceives and acts on the interface. As a result, success and failure cases become visible, and the dashboard provides interpretable feedback that enables the Coder to guide UI refinement.

Our empirical results show that while state-of-the-art Coders can generate complete GUIs that appear suitable to humans, they still encounter notable limitations: **(i) Task solvability as a foundation.** Initial UIs often fail to capture many practical scenarios, resulting in low usability. However, by collecting failure cases, the Coder can readily boost performance through language-based functional summarization. **(ii) CUA navigation success rate as a key bottleneck.** Even when UIs achieve high functional completeness, CUAs initially exhibit low success rates due to the complexity of multi-step navigation. Through our Coder–CUA collaboration, we substantially improve navigation success rates, particularly showing that CUA feedback-driven redesigns—such as *de-stylization*, increased contrast, and simplified layouts—significantly enhance CUA execution. Together, these findings highlight the promising potential of agents for automatic UI design and testing, improving both task success and robustness. To summarize, our contributions are threefold:

1. **AUI-Gym: a scalable testbed for automatic GUI development and testing**, covering 52 applications across six domains with 1560 human-validated tasks and per-task rule-based checkers. This enables human-free development of automatic UI building and testing.

2. **Method: Coder–CUA collaboration with CUA Dashboard.** The Coder initializes and refines UIs while the CUA judges via two signals: navigation outcomes and task solvability. A single-image $1K$ CUA Dashboard compresses task goal, actions, intermediate states, and outcome by highlighting key interactive regions with adaptive scaling, reducing visual tokens by 76.2% on average while preserving essential cues for redesign.

3. **Insights: simplified UIs are preferred by CUAs.** Our evaluations show that task solvability is foundational yet readily improved via failure-driven functional summarization, whereas CUA navigation success rate is the main bottleneck. Feedback-driven redesigns

(*e.g.,* de-stylization, higher contrast, simplified layouts) substantially raise execution success and overall robustness.

2 Related Work

2.1 Computer-Use Agents

Recent studies reveal the potential of LLMs beyond language modeling, with advancements in demonstrating their ability to autonomously complete complex tasks using tool integration (Schick et al., 2023) like humans. This has prompted the development of GUI automation agents that learn to operate digital user interfaces by imitating human trajectories. This learning is primarily achieved in two ways: (i) by steering general multimodal foundation models (Achiam et al., 2023) with in-context human trajectory examples, and the general models perceive the UI through intermediate representations like HTML, accessibility trees (Drouin et al., 2024; Gao et al., 2023; Zheng et al., 2024), Optical Character Recognition (Lu et al., 2024), or Set of Masks (Yang et al., 2023). (ii) by pre-training specialized GUI foundation models through extensive supervised fine-tuning or reinforcement learning on large-scale vision-text UI data (*e.g.,* screenshots and instructions) (Xu et al., 2024; Lin et al., 2025; Gou et al., 2024; Lin et al., 2024; Nayak et al., 2025; Lu et al., 2025a; Seed, 2025). While foundational, these data-driven approaches suffer from heavy requirements for high-quality human trajectories to achieve agent performance improvements (Zhuge et al., 2024; Nguyen et al., 2025; Zhang et al., 2024; Liu et al., 2025). Despite their methodological differences, these approaches share a common, agent-centric paradigm, focusing on improving the agent's capabilities to navigate a static and often complex environment. Notably, we investigate a complementary approach. Instead of adapting the agent, we explore how to dynamically tune the environment to enhance the performance of a frozen agent.

2.2 Automatic Software Design

Besides CUAs, there has been extensive research on software automation, automatic interface design (Lu et al., 2023; Kong et al., 2008) and generation (Si et al., 2024; Beltramelli, 2018; Laurençon et al., 2024; Chen et al., 2025; Peng et al., 2024; Cai et al., 2024; Wan et al., 2025; Lu et al., 2025b). Programmatic and semantic UI components—such as accessibility layers, ARIA tags, and declarative interface frameworks (*e.g.,* React Native, Flutter)—illustrate how environments can be

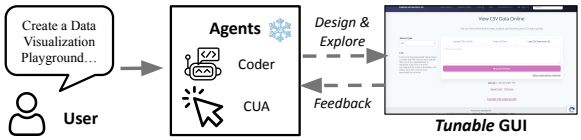


Figure 2: **AUI-Gym task definition.** A user issues a request (*e.g.,* "Create a Data Visualization Playground"), and agents (*e.g.,* Coder or CUA) interact with the GUI through design, exploration, and feedback. In this setup, the GUI serves as a *tunable* environment.

annotated or abstracted for automated processes, such as automatic UI testing (Liu et al., 2025). Similarly, benchmarks in automated software interaction, such as WebArena (Zhou et al., 2024) and GAIA (Mialon et al., 2023), assume agent operates within fixed, human-oriented systems for task automation. More recently, embodied AI environments (*e.g.,* ALFRED (Shridhar et al., 2020), Habitat (Puig et al., 2023), MineDojo (Fan et al., 2022)) show how environments can be crafted to accelerate agent training, though primarily in physical or simulated domains. These efforts highlight the growing recognition that environments can be reimagined for machine interaction, yet a systematic framework for designing agent-centric digital environments remains absent.

3 AUI-Gym Benchmark

3.1 Task Definitions

We develop AUI-Gym for automatic GUI development and testing. Given a language user query $\mathcal{Q}$ as input and several agents (*e.g.,* Coder or CUA), the output is a complete website that serves as a *tunable* environment $\mathcal{E}$. We detail the input and output respectively below.

Input Query $\mathcal{Q}$. Since the outcome is a website, the user query $\mathcal{Q}$ should be both descriptive and concrete. To this end, we standardize queries into the structured format illustrated above. This supplements the query with a name, goal, functional features, and UI theme.

Input formulation

Create a single-page app in a single HTML file with the following requirements:
- Name: {Camping Gear Checklist}
- Goal: {Track gear for camping trips}.
- Features: {Checklist items, weight calculator, save lists.}
- Theme: {The UI should be outdoor-themed.}

Output website $\mathcal{E}$. The website is an application-level deliverable that must be fully functional, going beyond a static page to support navigation, transitions, button interactions, and completion of functional goals, with the objective of maximizing the agent's success rate. Constructing an effective evaluation framework in this setting

is non-trivial and introduces several challenges. We next present our scalable, automatic solutions.

3.2 Task Creation

The full curation pipeline is illustrated in Fig. 3—an LLM-assisted, human-filtered strategy. To construct the benchmark, we collect 52 task prompts from OpenAI's playground [1], covering multiple domains.

Synthesize candidate rubric tasks Ω. Applications are typically designed to support a variety of relevant tasks, and a key evaluation is whether they can smoothly handle such tasks. We leverage GPT-5 (OpenAI, 2025) to synthesize diverse user requirements: given an instruction $\mathcal{I}$, it generates a set of candidate tasks Ω that simulate practical usage and serve as a *rubric* for evaluation, requiring the website to satisfy multiple requirements. As illustrated in Fig. 3, for the application '*Micro Habit Tracker*', an example task is: "Create a habit named 'Meditate 5 min,' then view today's column and the habit chart." These tasks serve as fine-grained probes that capture the potential demands of the environment $\mathcal{E}$.

Manual quality control. As the tasks are automatically generated by GPT-5, human oversight is required to ensure their quality. Different applications demand different characteristics: for example, tasks for game UIs should emphasize interactivity and control, while tasks for utility tools should capture information accessibility and workflow patterns. To this end, humans define domain-specific principles and filter out low-quality tasks (*e.g.,* trivial clicks) or nonsensical ones (*e.g.,* beyond the application scope), ambiguous queries (cross-application), ensuring that the proposed tasks remain concrete, meaningful and aligned with each domain's design philosophy. This human filtering takes approximately 10-15 minutes per application, totaling around 10 hours for all 52 applications. Furthermore, we validated a random sample of 100 generated rules against human judgments, achieving 91% agreement, confirming the reliability of GPT-5.

Data Statistics. Based on the above strategy, we obtain 30 tasks for each application. The benchmark spans 52 web applications across six domains, yielding a total of 1,560 tasks and enabling comprehensive evaluation across diverse applications. As illustrated in Table 6, the domains include: (i) App, general-purpose applications; (ii) Landing, commercial and promotional interfaces; (iii) Game, puzzle and arcade-style challenges; (iv) Interactive, dynamic user engagement with real-time feedback; (v) Tool, specialized utilities; and (vi) Utility, everyday organizational support. This diverse coverage captures distinct GUI challenges, ensuring robust evaluation across varied interaction paradigms and functional complexities (full statistics in Appendix Tab. 6). Beyond task diversity, the generated websites exhibit substantial implementation complexity (Please see Appendix C for statistics).

3.3 Evaluation with Verifiers

Even with the proposed tasks, it remains challenging to determine whether a given GUI can truly satisfy them, as websites are interactive and highly diverse environments. More importantly, since the *GUIs are generated at test time*, it is difficult to design fixed standards that generalize across all cases, given the variety of possible implementation approaches. A naive solution is to adopt a VLM-as-Judge approach, but this inevitably introduces bias and uncertainty (Appendix Tab. 3). Ideally, the most reliable solution would be concrete functional checks with manual validation, yet this approach is prohibitively expensive.

To address this, we define a **Verifier** $\mathcal{V}(\cdot)$ powered by GPT-5 at test time, which takes as input a candidate GUI together with a specific task. It analyzes the available elements and states, reasoning over the presence of required UI components, their properties, and potential interaction paths. If the task is deemed solvable, the Verifier produces a task-specific verification **function checker** $\widetilde{\mathcal{V}}(\cdot)$ (in `JavaScript`) that encodes the success condition by element status; otherwise, the task is discarded as invalid, preventing noisy or unachievable goals from disrupting evaluation. As shown in Fig.3, for task "Create a habit named 'Meditate 5 min,' then view today's column and the habit chart.", based on the candidate website (right), the verifier generates the rule `gridContainer contains 'Meditate 5min'` In this way, the Verifier is customized for each website and each task at test time, ensuring reliable validation. Importantly, while GPT-5 generates the checker code, the actual evaluation is executed as a deterministic, rule-based JavaScript function, not an LLM judgment, eliminating potential bias from using the same model for both generation and evaluation.

Metrics. With the support of function checkers as reliable verification, we can ensure that a web-

[1] https://github.com/openai/gpt-5-coding-examples

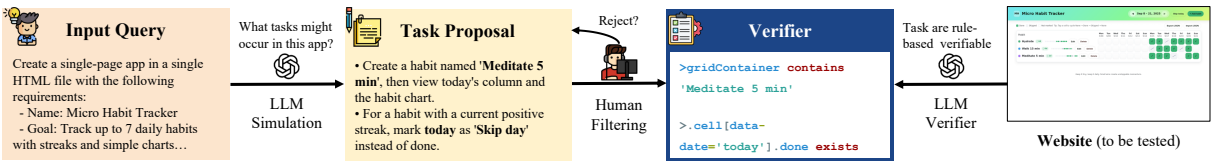
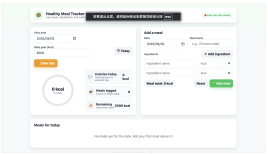


Figure 3: **AUI-Gym construction pipeline. (i)** An input query specifies the app requirements. **(ii)** GPT-5 proposes candidate tasks with explicit goals. **(iii)** Humans filter and refine tasks using domain-specific principles. **(iv)** A test-time Verifier reads the website HTML and generates task-specific, rule-based checkers to validate success on the to-be-tested website.

Table 1: **Examples of App domains in AUI-Gym.** For each domain, we show a website created by GPT-5, paired with 30 tasks (blue) simulating real-world usage. Each task is further linked to a rule-based verifier (green). See full distribution and examples in Tab.6.

Domain	#Apps	Pct.	Example Instruction	UI created by GPT-5
App	11	21%	Create a single-page app in a single HTML file with the following requirements: - Name: Healthy Meal Tracker - Goal: Log meals and nutrition info. - Features: Ingredient list, calories per meal, daily summary. - The UI should be clean with food icons. **Task: Add five meals for today's date (any names/ingredients) so today's meal count reaches at least 5.** **Rule: `#dailyMealCount >= 5`**	

site is both actionable and workable for the CUA. This further allows us to evaluate whether tasks are completed after CUA navigation, thereby measuring task success rate within the UI environment. In this way, we devise the following measure:

(i) CUA Success Rate (SR). This measures the average success rate over all tasks executed by CUA. If CUA successfully completes a task, it is counted as a success; otherwise, it is counted as a failure. Notably, if the Coder fails to yield a functional checker, the task is counted as a failure.

$$\text{SR} = \frac{1}{|\Omega|} \sum_{t \in \Omega} \mathbf{1}\left(\text{task } t \text{ is completed}\right), \quad (1)$$

where Ω denotes the set of all tasks and $\mathbf{1}(\cdot)$ is the indicator function.

(ii) Function Completeness (FC). While CUA performance reflects the ultimate goal, it may be sparse if most CUAs fail to complete tasks. Therefore, we devise a second metric to evaluate only whether the Coder-created website functionally supports the task (valid), independent of CUA navigation. This metric reflects task validity and serves as a more basic measure.

$$\text{FC} = \frac{1}{|\Omega|} \sum_{t \in \Omega} \mathbf{1}\left(\text{task } t \text{ has a checker}\right). \quad (2)$$

4 CUA–Coder in Collaboration

Overview. We present our framework for enabling collaboration between the CUA and the Coder, consisting of two main components: the Coder as Designer while the CUA as Judge. Given a user instruction $\mathcal{Q}$, AUI generates an initial UI environment $\mathcal{E}_0$, which is iteratively revised through interaction and feedback. The framework involves two central roles: a Coder policy π_{Coder} that proposes and revises UI designs, and a CUA policy π_{CUA} that explores the UI and evaluates its functionality. We formalize this iterative loop as a Markov Decision Process where the "decision" is a design revision. The state is the current UI $\mathcal{E}_t$, the action is a design update proposed by π_{Coder}, and the transition deterministically $\mathcal{E}_{t+1} \leftarrow \pi_{\text{Coder}}(\mathcal{E}_t, \mathcal{R}_t)$. The feedback $\mathcal{R}_t$ is related to the metrics (*i.e.,* Eq.1 and Eq.2) results achieved by the CUA when interacting with $\mathcal{E}_t$, *i.e.,* $\mathcal{R}_t \leftarrow S(\mathcal{E}_t, \pi_{\text{CUA}})$. The Coder is optimized to maximize the total reward $\mathbb{E}\left[\sum_t \gamma^t \mathcal{R}_t\right]$. In this formulation, the CUA acts as a user that provides feedback by testing the environment, while the Coder serves as a designer who integrates this feedback into code revisions to iteratively improve the UI. Unlike conventional CUA setups, where the agent adapts to a fixed environment $\pi_{\text{CUA}} \leftarrow \mathcal{E}$, our framework adapts the environment itself based on CUA feedback $\mathcal{E} \leftarrow \pi_{\text{CUA}}$, thereby optimizing UIs for agent-native success. We illustrate the full workflow in Fig. 4 and detail each role in the following subsections.

4.1 Coder as Designers

Recent advances in Coder (OpenAI, 2025; Qwen, 2025; anthropic, 2025) demonstrate strong capabilities in generating UI applications. In our framework, we position Coders as *designers*, responsible not only for creating new environments but also for refining them based on feedback from CUAs. Accordingly, Coders operate in two complementary modes: one dedicated to the initial creation of UIs, and the other focused on their iterative improvement through CUA-guided feedback.

i. Initialization. Given a user query defined in formulation 3.1 and enriched with multiple details, the Coder progressively generates long-context

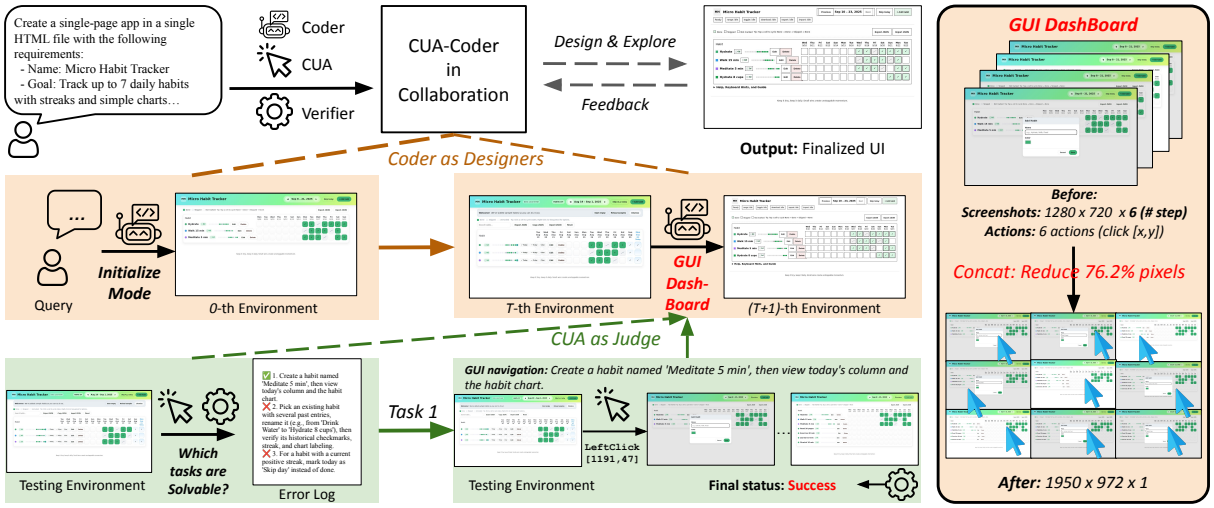


Figure 4: **Overview of the (left) Coder–CUA collaboration framework and (right) GUI dashboard illustration.** The process starts with the Coder acting as a *Designer*, which initializes the UI and iteratively revises it based on feedback. In parallel, the CUA serves as a *Judge*, executing task-driven navigation in the evaluation environment and generating trajectories and error logs to assess task solvability. A verifier checks functional correctness, while feedback from CUA navigation guides subsequent UI revisions. The GUI dashboard (Table 7) visualizes CUA trajectories for clearer and efficient analysis.

code to construct a HTML-rendered UI $\mathcal{E}_0$ from scratch, which serves as the base environment for subsequent interactions.

ii. Revision from Feedback. After constructing the initial environment $\mathcal{E}_0$, the Coder enters an iterative refinement loop to update the UI: $\mathcal{E}_{t+1} \leftarrow (\mathcal{E}_t, \mathcal{R}_t)$, where $\mathcal{R}_t$ denotes the feedback signal expressed as a language caption, described in the next section.

4.2 CUA as Judges

We employ Computer-Use Agents (CUAs) as *Judges* to trial and diagnose the UIs $\mathcal{E}_t$ generated by the Coder, providing actionable feedback for iterative redesign. Specifically, we define two complementary forms of reward signals:

***(i)* Task Solvability Feedback $\mathcal{R}_{\text{task}}$.** Before navigation begins, we verify whether a task τ is implementable on the current UI. Let $\mathcal{V}$ denote the verifier in Sec. 3.3. A task is deemed solvable if and only if $\mathcal{V}(\mathcal{E}_t, \tau) = 1$; otherwise it is labeled a *functionality failure*. This gate prevents wasted rollouts on impossible tasks and sharpens the feedback signal. We collect all failed tasks into $\mathcal{T}_{\text{fail}} = \{\tau : \mathcal{V}(\mathcal{E}_t, \tau) = 0\}$ and return them to the Coder as precise indicators of missing features. The Coder then aggregates and summarizes these failures into a language feedback signal $\mathcal{R}_{\text{task}}$.

***(ii)* CUA Judge Feedback $\mathcal{R}_{\text{cua}}$.** For solvable tasks $\mathcal{T}_{\text{succ}} = \{\tau : \mathcal{V}(\mathcal{E}_t, \tau) = 1\}$, evaluation proceeds as a UI navigation problem. At step k, the CUA receives an observation o_k (a screenshot of the current state), emits an action $a_k \in \{\text{CLICK}, \text{TYPE}, \text{SCROLL}, \ldots\}$ with an optional reasoning trace, and the environment transitions to the next state, yielding o_{k+1}. The trajectory $\mathcal{H} = (o_0, a_0, \ldots, o_K)$ terminates when either (a) the function checker signals success $\widetilde{\mathcal{V}}(\mathcal{E}_t, \tau) = 1$, or (b) a step limit is reached, which we record as a failure. We log full trajectories—observations, actions, and intermediate rationales—and use them to construct targeted feedback for UI refinement.

CUA Dashboard for Compact Feedback. Raw trajectories $\mathcal{H}$ are long and interleaved, making them ill-suited for direct ingestion by the Coder. We therefore distill each rollout into a CUA Dashboard: a single, fixed-resolution (1920×1080) canvas that compresses key evidence from the trial. This transformation is shown in Fig.4 (right). Rather than storing full frames, we crop and tile only *interactive regions* touched by the CUA, allocating dynamic region sizes based on step order to preserve temporal structure. The detailed illustration is shown in Appendix Fig. 7. This yields a substantial reduction in redundancy (*e.g.,* a 76.2% drop in visual content for 6-step trajectories) while retaining the cues needed to localize failure modes (missed affordances, hidden state, ambiguous labels) and success paths at a glance. The dashboard provides a step-by-step visual trace aligned with actions, making error locations immediately visible. Finally, we convert the dashboard into a language summary $\mathcal{R}_{\text{cua}}$ by passing it to the Coder and as the feedback used in the revision rule.

5 Experiments

5.1 Settings

Baselines. For the Coder, we select representative models including GPT-5 (OpenAI, 2025), GPT-

Table 2: **Main Result of Benchmark per Coder.** Top: Function Completeness Rate (%). Bottom: CUA Success Rate (%). UI-TARS-1.5-7B is used as the CUA for evaluation. Green = improvement over baseline, Red = decline from baseline. Color intensity indicates magnitude of change.

Coder	Feedback Type	landing (%)	game (%)	app (%)	utility (%)	interactive (%)	tool (%)	Overall (%)
				Metric: Function Completeness				
GPT-5	Baseline	53.0	77.8	70.6	63.3	73.0	70.0	67.9
	+Task Solvability	19.7	100.0	69.4	65.6	55.6	56.2	60.5
	+CUA Judge	53.3	87.8	74.2	70.0	70.4	69.5	70.8
	+Task & CUA	75.3	92.2	85.2	73.3	82.6	76.7	**81.5**
Qwen3-Coder-30B	Baseline	16.3	50.4	41.2	43.9	52.2	54.8	42.1
	+Task Solvability	55.0	79.6	58.5	67.8	56.3	74.3	**64.3**
	+CUA Judge	23.3	50.4	38.8	49.4	39.3	55.2	41.3
	+Task & CUA	47.7	72.2	59.7	56.7	57.0	69.5	60.1
GPT-4o	Baseline	9.7	55.2	36.1	38.9	44.8	37.6	36.3
	+Task Solvability	23.7	55.9	52.1	55.0	58.9	65.2	**50.6**
	+CUA Judge	8.3	55.2	28.2	34.4	26.3	35.7	30.4
	+Task & CUA	16.3	68.5	36.4	51.7	51.1	41.4	43.1
Gemini-3-Pro	Baseline	68.3	97.4	52.4	55.6	80.7	76.3	71.7
	+Task Solvability	52.4	70.4	95.4	73.8	79.4	53.9	70.1
	+CUA Judge	56.4	64.0	91.2	77.0	81.0	65.6	71.4
	+Task & CUA	66.7	91.5	62.7	56.1	77.8	78.7	**72.5**
				Metric: CUA Success Rate				
GPT-5	Baseline	34.7	24.8	27.3	14.4	18.1	21.9	24.5
	+Task Solvability	16.3	39.3	26.7	16.1	20.7	11.9	22.6
	+CUA Judge	17.7	43.3	30.0	21.1	21.1	17.6	25.7
	+Task & CUA	40.7	27.4	31.5	22.2	14.1	12.9	**26.0**
Qwen3-Coder-30B	Baseline	5.3	9.3	9.1	11.7	7.0	1.4	7.3
	+Task Solvability	14.7	42.2	19.1	14.4	11.1	4.3	18.3
	+CUA Judge	6.7	20.7	9.1	11.1	12.2	11.4	11.7
	+Task & CUA	23.7	30.7	22.4	7.8	9.3	13.8	**19.0**
GPT-4o	Baseline	4.7	12.6	12.4	6.7	9.3	5.7	8.8
	+Task Solvability	8.7	18.5	19.1	5.6	8.5	22.9	14.1
	+CUA Judge	5.7	31.5	10.0	8.3	10.4	6.7	12.3
	+Task & CUA	10.3	27.4	13.9	13.3	15.2	16.7	**16.1**
Gemini-3-Pro	Baseline	32.2	40.3	15.0	21.0	39.4	59.6	35.8
	+Task Solvability	38.2	44.2	39.7	52.5	46.9	18.6	41.5
	+CUA Judge	55.9	59.9	47.9	58.7	43.5	31.4	**51.0**
	+Task & CUA	62.5	41.3	46.9	26.7	37.1	62.0	47.0

4o (Hurst et al., 2024), Gemini-3-Pro, and the open-source Qwen3-Coder (Qwen, 2025). For the CUA, we adopt UI-TARS-1.5-7B (Seed, 2025) and Operator (openai, 2025). UI-TARS-1.5-7B is a lightweight yet strong performer among open models with high efficiency, and Operator is among the state-of-the-art close-source API-based CUAs. Note that the Commenter role (which analyzes visual feedback including screenshots and dashboards) uses a multimodal model: GPT-5 for GPT-5 Coder experiments and Qwen2.5-VL-72B-Instruct for Qwen Coder experiments. The experiment results highlight that the gain brought by our proposed method is universal for both lightweight and powerful CUAs.

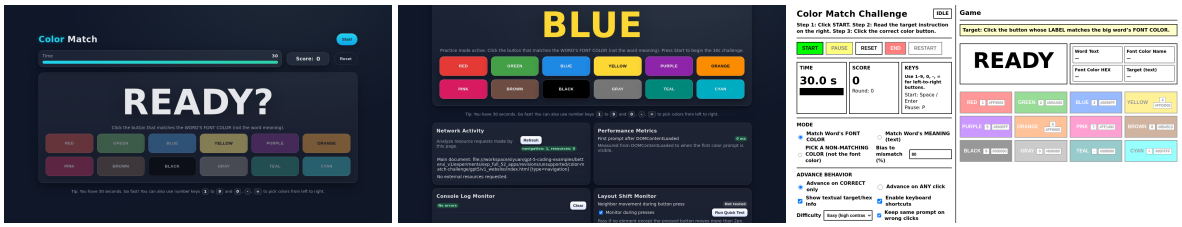
5.2 Main Results

Table 2 reports results across six domains for four coders. Several key findings emerge: **(i) Function completeness.** Revision based on 'Task Solvability' feedback leads to substantial gains, consistently boosting the overall functionality completeness for all coders. After applying full revision for GPT-5, the function completeness is increased to 81.5% from 67.9%, reaching the highest. Notably, the landing, game and app domains have dramatic improvements, with the maximum improvement of 38.7%. Interestingly, Revision based on 'Task Solvability' feedback or 'CUA Judge' feedback alone does not guarantee function completeness improvements, but the full revision combining these two components bring stable improvements on all domains for all coders, highlighting the strength of our design. Moreover, fixing unresolved functionalities alone also benefits CUA task solving, yielding a 4.8% average improvement on CUA evaluation, highlighting the mutual reinforcement between task solvability and CUA navigation. **(ii) CUA success rate.** Open-source CUAs (UI-TARS-1.5-7B) initially perform poorly, with an average overall CUA success rate of only 19.1%. However, our framework can consistently improve the CUA success rate, with an average 7.9% improvement. Interestingly, our framework brings large improvements to weak coders such as Qwen3-Coder-30B and GPT-4o, with a maximum overall improvement of 11.7%, showcasing that our framework can greatly empower weak models. Overall, these results demonstrate the effectiveness of our framework: *'Task Solvability' feedback guides to robust UI design, while leveraging 'CUA Judge' feedback optimizes interfaces toward agent-centric success.* We further show that revision generalizes across CUA policies (Appendix Tab. 4) and reduces grounding errors (Appendix Tab. 5).

5.3 Key Ablations

Effects by different CUA as Judge. Fig. 6(a–b) shows that UI-TARS and Operator provide complementary CUA-judge signals with comparable function-completeness gains; detailed analysis is provided in Appendix Sec. B.

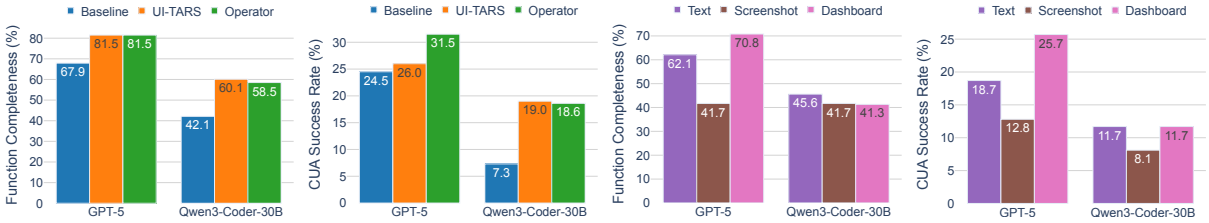
Effects by GUI Dashboard. Fig.6 (c-d) demon-



Initial UI w. Task Solvability Feedback w. CUA Judge Feedback

Task of `color-match-challenge`: Create a single-page app in a single HTML file for a fast-paced "color match" game. - Show a word (e.g., "RED") in a random font color — player must click the correct color button (not the word meaning). - Keep score based on correct answers within 30 seconds. - Use large typography, color-coded buttons, and smooth button press animations.

Figure 5: **Qualitative comparison of initialized UI *v.s.* refined UI.** An initial UI (left), revision based on function (middle), and revision based on CUA-failed tasks (right).



(a) **FC:** UI-TARS vs. Operator (b) **SR:** UI-TARS vs. Operator (c) **FC:** Commenter mechanism (d) **SR:** Commenter mechanism

Figure 6: **Ablation studies of (a-b) different CUA policies** (UI-TARS, Operator) and **(c-d) different Commenter mechanisms** (Text-only, Screenshot-only, GUI Dashboard).

strate the results when using different types of commenters for revision based on CUA navigation feedback. Besides dashboard commenter, there are two additional variants which either use text only or screenshot only for CUA policy trajectory information. **Text-only** commenter will only percept the same text information used in the dashboard, *i.e.,* task description, task expected outcome, CUA thought and action, etc. **Screenshot-only** commenter will only percept the same screenshots used in the dashboard. Besides the CUA trajectory information, all the three commenters will see the full UI screenshot for UI analysis. It can be observed that the dashboard can bring significant improvements for GPT-5 commenter and coder, but its performance is slightly lower than text-only commenter for Qwen commenter and coder. This indicates that *dashboard offers comprehensive visual and textual cues, which requires commenter has strong visual perception to fully capture the information*. We further show that Dashboard can **reduce average 70.4% visual context** compared with the screenshot-only baseline.

6 Qualitative Analysis

In Fig.5, we present a representative case: *color-match-challenge*. As shown, the initial UI has a launch period. During this period, the buttons have lower brightness compared to other components in the page. This reduces the button visibility, which may confuse the agents about where to click. With 'Task Solvability' feedback, the middle one is more straightforward without the launch time period, with more elements on the page compared to the initial UI. Further with 'CUA Judge' feedback, it introduced clear, bounded displays and buttons, exposed primary controls upfront (start, pause, reset, etc.) to avoid confusion, and added explicit visual hints for action order. Thus, improve agent accessibility. Together, these changes improved functional completeness (via robust parsing) and CUA success rate (via clearer affordances and shorter action paths), illustrating how function fixes and agent-oriented refinements complement each other in practice. Please refer to Appendix Sec. D for more examples.

7 Conclusion

We study whether Computer-Use Agents can act as Judges to assist Coders in automatically redesigning GUIs toward agent-native efficiency. We introduce AUI-Gym, a 52-application testbed with rule-based verifiers, and a Coder–CUA collaboration loop that iteratively refines websites via CUA execution feedback, complemented by a one-page CUA dashboard for interpretable trajectory analysis. Experiments show that failure-driven functional fixes readily improve task solvability, while simplifying interfaces (*e.g.,* de-stylization, higher contrast, cleaner layouts) consistently boosts CUA navigation success. Our work takes a step toward agent-native GUI design and automated UI testing.

Limitations

Our study has several limitations. First, AUI-Gym currently focuses on applications whose interfaces can be improved primarily through layout, styling, and interaction simplification. Extending the framework to highly complex interfaces, such as deep DOM trees, long scrollable pages, or multi-page workflows, remains challenging. Second, our conclusions depend on the CUA policies evaluated in this work. Open-source CUAs such as UI-TARS may make grounding and navigation mistakes, while stronger closed-source CUAs such as Operator may exhibit different preferences and failure modes. Third, our verifier pipeline relies on LLM-generated tasks and rule-based checks. Although we validate verifier quality against human judgments, these checks may still miss edge cases and do not fully capture human usability or accessibility.

Ethics Statement

Our work automates GUI evaluation and redesign via Computer-Use Agents, raising several ethical considerations. First, agent-centric UI revisions may reduce visual appeal or accessibility for human users by prioritizing machine navigability over human usability; we encourage future work to balance both objectives. Second, while our framework augments rather than replaces human designers, continued progress in automated UI generation could affect employment in UI/UX design. Third, our benchmark relies on LLM-generated tasks and rule-based verifiers (91% agreement with human judgments), which may contain biases inherited from the underlying models. Fourth, agent-driven UI simplifications (*e.g.,* removing animations, increasing contrast) may inadvertently affect accessibility for users with special needs (*e.g.,* screen reader compatibility); future work should evaluate redesigned interfaces against established accessibility standards such as WCAG. We do not anticipate immediate negative societal consequences beyond those typical of advances in machine learning and automation.

Reproducibility Statement

We will release the full benchmark, including all 52 applications, 1,560 tasks, and the verifier generation pipeline. The codebase for the Coder–CUA collaboration framework, CUA dashboard construction, and evaluation scripts will also be made publicly available. All experimental settings, including model configurations, CUA step limits, and commenter variants, are detailed in the main text and appendix.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anthropic. 2025. Claude 3.7 sonnet system card.

anthropic. 2025. Introducing claude 4.

Tony Beltramelli. 2018. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems*, pages 1–6.

Yuzhe Cai, Shaoguang Mao, Wenshan Wu, Zehua Wang, Yaobo Liang, Tao Ge, Chenfei Wu, WangYou WangYou, Ting Song, Yan Xia, and 1 others. 2024. Low-code llm: Graphical user interface over large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 3: System Demonstrations)*, pages 12–25.

Dongping Chen, Ruoxi Chen, Shilin Zhang, Yaochen Wang, Yinuo Liu, Huichi Zhou, Qihui Zhang, Yao Wan, Pan Zhou, and Lichao Sun. 2024. Mllm-as-a-judge: Assessing multimodal llm-as-a-judge with vision-language benchmark. In *Forty-first International Conference on Machine Learning*.

Jiaqi Chen, Yanzhe Zhang, Yutong Zhang, Yijia Shao, and Diyi Yang. 2025. Generative interfaces for language models. *arXiv preprint arXiv:2508.19227*.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, and 1 others. 2024. Workarena: How capable are web agents at solving common knowledge work tasks? *arXiv preprint arXiv:2403.07718*.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. 2022. Minedojo: Building open-ended embodied agents with internet-scale knowledge. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Difei Gao, Lei Ji, Zechen Bai, Mingyu Ouyang, Peiran Li, Dongxing Mao, Qinchen Wu, Weichen Zhang, Peiyi Wang, Xiangwu Guo, and 1 others. 2023. Assistgui: Task-oriented desktop graphical user interface automation. *arXiv preprint arXiv:2312.13108*.

Boyu Gou, Zanming Huang, Yuting Ning, Yu Gu, Michael Lin, Weijian Qi, Andrei Kopanev, Botao Yu, Bernal Jiménez Gutiérrez, Yiheng Shu, and 1 others. 2025. Mind2web 2: Evaluating agentic search with agent-as-a-judge. *arXiv preprint arXiv:2506.21506*.

Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. 2024. Navigating the digital world as humans do: Universal visual grounding for GUI agents. *arXiv preprint arXiv:2410.05243*.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, and 1 others. 2024. A survey on llm-as-a-judge. *The Innovation*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. GPT-4o system card. *arXiv preprint arXiv:2410.21276*.

Jun Kong, Keven L Ates, Kang Zhang, and Yan Gu. 2008. Adaptive mobile interfaces through grammar induction. In *2008 20th IEEE International Conference on Tools with Artificial Intelligence*, volume 1, pages 133–140. IEEE.

Hugo Laurençon, Léo Tronchon, and Victor Sanh. 2024. Unlocking the conversion of web screenshots into html code with the websight dataset. *Preprint*, arXiv:2403.09029.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Qinchen WU, Mingyi Yan, Zhengyuan Yang, Lijuan Wang, and Mike Zheng Shou. 2024. Videogui: A benchmark for gui automation from instructional videos. *Advances in Neural Information Processing Systems*, 37:69329–69360.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Stan Weixian Lei, Lijuan Wang, and Mike Zheng Shou. 2025. ShowUI: One vision-language-action model for GUI visual agent. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 19498–19508.

Chenxu Liu, Junheng Wang, Wei Yang, Ying Zhang, and Tao Xie. 2025. Judge: Effective state abstraction for guiding automated web GUI testing. *ACM Transactions on Software Engineering and Methodology*.

Fanbin Lu, Zhisheng Zhong, Shu Liu, Chi-Wing Fu, and Jiaya Jia. 2025a. Arpo: End-to-end policy optimization for gui agents with experience replay. *arXiv preprint arXiv:2505.16282*.

Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. 2024. Omniparser for pure vision based GUI agent. *arXiv preprint arXiv:2408.00203*.

Yuwen Lu, Ziang Tong, Qinyi Zhao, Chengzhi Zhang, and Toby Jia-Jun Li. 2023. Ui layout generation with llms guided by ui grammar. *arXiv preprint arXiv:2310.15455*.

Zimu Lu, Houxing Ren, Yunqiao Yang, Ke Wang, Zhuofan Zong, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2025b. Webgen-agent: Enhancing interactive website generation with multi-level feedback and step-level reinforcement learning. *arXiv preprint arXiv:2509.22644*.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*.

Shravan Nayak, Xiangru Jian, Kevin Qinghong Lin, Juan A Rodriguez, Montek Kalsi, Nicolas Chapados, M Tamer Özsu, Aishwarya Agrawal, David Vazquez, Christopher Pal, and 1 others. 2025. Ui-vision: A desktop-centric gui benchmark for visual perception and interaction. In *Forty-second International Conference on Machine Learning*.

Dang Nguyen, Jian Chen, Yu Wang, Gang Wu, Namyong Park, Zhengmian Hu, Hanjia Lyu, Junda Wu, Ryan Aponte, Yu Xia, and 1 others. 2025. GUI agents: A survey. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 22522–22538.

OpenAI. 2025. Introducing gpt-5. OpenAI website. Available at https://openai.com/index/introducing-gpt-5/, accessed August 10, 2025.

openai. 2025. Operator.

Yi-Hao Peng, Faria Huq, Yue Jiang, Jason Wu, Xin Yue Li, Jeffrey P Bigham, and Amy Pavel. 2024. Dreamstruct: Understanding slides and user interfaces via synthetic data generation. In *European Conference on Computer Vision*, pages 466–485. Springer.

Xavi Puig, Eric Undersander, Andrew Szot, Mikael Dallaire Cote, Ruslan Partsey, Jimmy Yang, Ruta Desai, Alexander William Clegg, Michal Hlavac, Tiffany Min, Theo Gervet, Vladimir Vondrus, Vincent-Pierre Berges, John Turner, Oleksandr Maksymets, Zsolt Kira, Mrinal Kalakrishnan, Jitendra Malik, Devendra Singh Chaplot, and 4 others. 2023. Habitat 3.0: A co-habitat for humans, avatars and robots.

Qwen. 2025. Qwen-3-coder. https://qwenlm.github.io/blog/qwen3-coder.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

ByteDance Seed. 2025. Ui-tars-1.5. https://seed-tars.com/1.5.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke

Zettlemoyer, and Dieter Fox. 2020. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10740–10749.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2024. Design2code: Benchmarking multimodal code generation for automated front-end engineering. *arXiv preprint arXiv:2403.03163*.

Yuxuan Wan, Tingshuo Liang, Jiakai Xu, Jingyu Xiao, Yintong Huo, and Michael R Lyu. 2025. Automatically generating web applications from requirements via multi-agent test-driven development. *arXiv preprint arXiv:2509.25297*.

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

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. 2023. Set-of-mark prompting unleashes extraordinary visual grounding in GPT-4V. *arXiv preprint arXiv:2310.11441*.

Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Guyue Liu, Qingwei Lin, and 1 others. 2024. Large language model-brained GUI agents: A survey. *arXiv preprint arXiv:2411.18279*.

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

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. Webarena: A realistic web environment for building autonomous agents. *Preprint*, arXiv:2307.13854.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, and 1 others. 2024. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*.

A Why GUI as Testbeds? 12

B Additional Experiments 12

C Application Complexity 13

D Qualitative Analysis 14

E Full Statistics and Examples 14

F GUI DashBoard illustration 14

G Agent's Prompts 17

A Why GUI as Testbeds?

While agents can operate on structured representations (*e.g.,* DOMs, code-centric CLI or terminals), we focus on visual GUI interaction for two reasons.

1. First, real-world deployment requires handling diverse interfaces without *privileged* access, necessitating human-like visual perception.

2. Second, whereas code-centric agents (*e.g.,* those using bash) have achieved robust performance, general-purpose CUAs remain limited by visual capabilities—the core challenge our benchmark targets.

3. Third, our framework provides a perspective on how to optimize an existing environment for a particular agent, which is meaningful when aiming to deliver *personalized* user experiences, such as enlarge text or simplifying colors for people with myopia.

We believe it is meaningful to brige GUI and Code such as how to convert GUIs into CLIs or tool-based interactions, which we leave for future works.

B Additional Experiments

Below, we show the experiment results additional to the results showcased in the main paper.

Effects by different CUA as Judge. Following up on Fig. 6(a–b), we compare UI-TARS and Operator as CUA policies within the full revision loop. We evaluate with two coders, GPT-5 and Qwen3-Coder-30B, to cover both stronger closed-source and lighter open-source coders. Both CUA policies yield comparable gains in functional completeness, with UI-TARS slightly outperforming on Qwen3-Coder-30B. Although the task-solvability signal is identical across CUAs, UI-TARS tends to fail more tasks, thereby surfacing richer failure cases and driving greater function-oriented revisions. For CUA success rate, Operator delivers larger gains with the stronger coder GPT-5, while improvements are similar across CUAs for the weaker coder. This suggests Operator's navigation strengths are best realized on more complex UIs, whereas weaker coders often produce simpler interfaces.

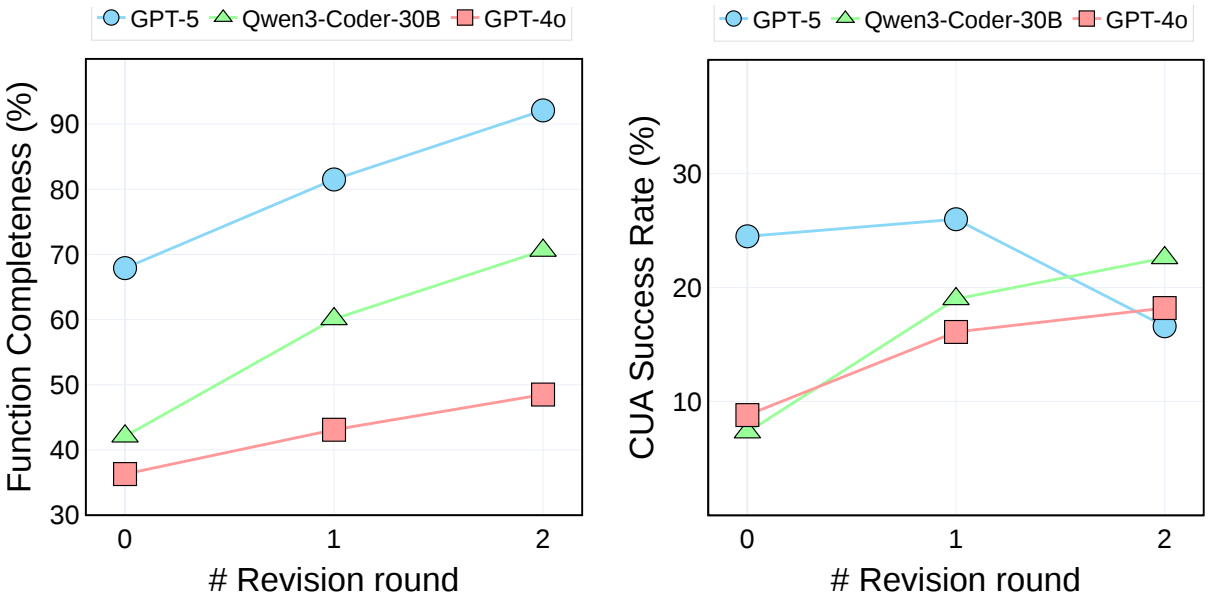
Effects by refinement round. Fig. 7 shows that iterative revision consistently improves function completeness for all coders. CUA success may drop after repeated revisions for GPT-5, while Qwen3-Coder-30B and GPT-4o continue to gain, suggesting that strong coders can saturate earlier whereas weaker coders benefit from more feedback rounds.

Table 3: **VLM evaluation on final screenshot only is unreliable.** Given the final screenshot of CUA trajectory and the expected outcome, the accuracy of VLM evaluation is only slightly above the naive all-fail baseline; Balanced Accuracy is near 0.55; Recall/F1 and Cohen's κ are low.

Metric	Overall	GPT-5	Qwen2.5-VL-72B	GPT-4o
Naive all-fail baseline accuracy	0.720	0.720	0.720	0.720
Accuracy vs. oracle	0.735	0.736	0.738	0.732
Balanced accuracy	0.556	0.549	0.568	0.552
Precision (Pass)	0.616	0.660	0.612	0.589
Recall (Pass)	0.147	0.121	0.178	0.142
F1 (Pass)	0.237	0.205	0.276	0.229
Cohen's κ	0.145	0.128	0.175	0.132

Can we use VLM as Judge as Verifiers? As noted in Sec. 3.3, VLM-as-Judge introduces bias. Table 3 further demonstrates that VLM evaluation on CUA task trajectory is unreliable. It can be observed that the the compared to rule-based oracle evaluation, the VLM evalation tends to judge the CUA policy outcome as failure, thus having very low balanced accuracy, recall and F1. Moreover, the low Cohen's κ indicates very weak agreement of VLM evaluation compared to rule-based oracle evaluation. This indicates that VLM evaluation on the final screenshot only is unreliable, and may requires more screenshots along the CUA policy task trajectory for more reliable evaluation, leading to much higher computational cost.

Cross-CUA Generalization. Following up on the main results, we investigate whether revision based on one CUA policy can generalize to improve performance for a different CUA policy. Specifically, we use Operator to generate navigation feed-



(a) **FC** along revisions number. (b) **SR** along revisions number.

Figure 7: **Ablation studies of different revision rounds** under three different Coders.

back for revision, then evaluate the revised websites using UI-TARS. This setup tests whether the UI improvements driven by Operator's failure patterns transfer to benefit a different navigation agent.

Table 4: **Cross-CUA Generalization.** Revision based on Operator feedback, evaluated with UI-TARS. Results show CUA Success Rate.

Model	Version *(Operator as Judge)*	Overall *(UI-TARS)*
GPT-5	Baseline	26.7%
	+ **Task & CUA**	**29.3%**
Qwen3-Coder-30B	Baseline	11.8%
	+ **Task & CUA**	**30.9%**
GPT-4o	Baseline	19.6%
	+ **Task & CUA**	**35.7%**

As shown in Table 4, revision based on Operator feedback consistently improves UI-TARS success rate across all three coders, with overall success rate for three coders increasing from 19.4% to 32.0%. This demonstrates that the UI improvements generalize across different CUA policies, suggesting that the revision process captures fundamental usability issues rather than agent-specific quirks.

CUA Grounding Error Analysis. Beyond the CUA success rate improvements reported in the main paper, we analyze visual grounding capability by measuring *grounding errors*, *i.e.,* clicks whose coordinates fall outside interactive UI elements (buttons, inputs, links, canvas, etc.).

As shown in Table 5, the grounding error rate decreases across all three coders after applying **Task & CUA** revision. GPT-5 shows a reduction from 34.42% to 19.89%, GPT-4o improves from 54.22% to 36.08%, and Qwen3-Coder-30B improves from 32.15% to 29.64%. This indicates that our CUA-driven revision not only improves task success rate but also enhances the visual clarity and affordance of interactive elements, making them easier for CUA policies to accurately ground.

Table 5: **Grounding Error Analysis.** Clicking coordinates outside interactive element bounding boxes are counted as grounding errors, evaluated with UI-TARS. Lower is better.

Model	Version	Error Rate ↓
GPT-5	Baseline	34.42%
	+ **Task & CUA**	**19.89%**
Qwen3-Coder-30B	Baseline	32.15%
	+ **Task & CUA**	**29.64%**
GPT-4o	Baseline	54.22%
	+ **Task & CUA**	**36.08%**

C Application Complexity

As discussed in Data Statistics, beyond task diversity, the generated websites exhibit substantial implementation complexity. On average, each application contains 447 lines of JavaScript, 15.1 event listeners, and 14.2 interactive elements (buttons, inputs, selects). 96% of applications exceed 100 JS lines, 92% include CSS transitions, and 31% employ canvas for graphics rendering—demonstrating that these are fully functional interactive applica-

tions rather than static pages.

D Qualitative Analysis

We extend the qualitative analysis from the main paper with additional revision cases.

In this section, we present four representative revision cases—*artisan-csa*, *color-match-challenge* and *festival-lights-show*. Each row displays the initial UI alongside its revised versions, evaluated under two criteria: Task solvability test and CUA judge. Across the four cases, the revisions demonstrate distinct patterns of improvement. Revisions based on the task solvability, which addresses unsupported tasks, tend to focus on adding underlying functionality, sometimes with subtle visual changes. For example, the *festival-lights-show* revision added a crucial "Running" state indicator, and the *csv-to-charts* revision added a button to select a delimiter.

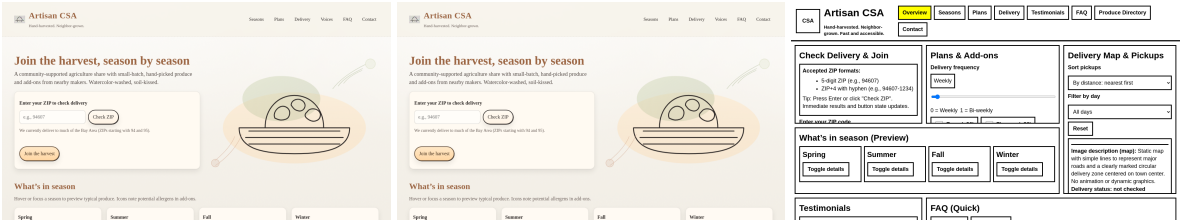
In contrast, revisions based on the CUA judge consistently yield more significant visual modifications geared towards agent accessibility. For most websites, this meant adding buttons with clear boundaries and visual hints. In both *color-match-challenge* and *csv-to-charts*, both revision types improved accessibility by presenting more information and controls upfront, reducing the need for scrolling. A key CUA-friendly adaptation is seen in *festival-lights-show*, where "increase" and "reduce" buttons were added as a complement to sliders, providing a more direct and reliable interaction method for agents.

E Full Statistics and Examples

We provide the complete per-domain breakdown referenced in Sec. 3.2. Figure. 6 displays the full statistics and corresponding examples.

F GUI DashBoard illustration

In Figure. 7, we display how our GUI Dashboard work to reduce the observation token number. It integrate multiple observations (including screenshots and actions) then result one figure as dashboard, which highlight the important information (such as coordinates) by visual prompts (red point) as well as organize the action information under individual screenshot. Thus, offer a friendly reading experience.

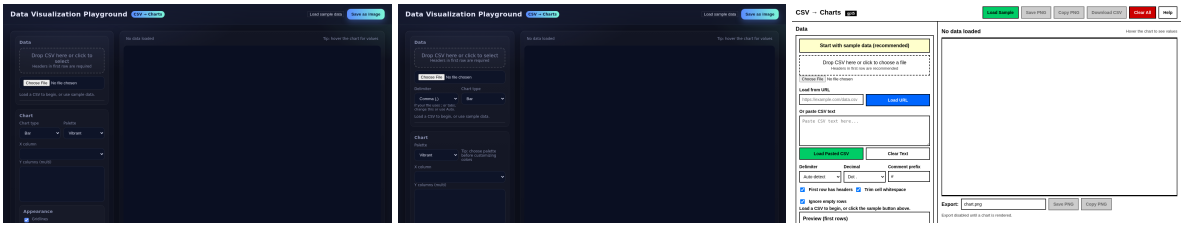


Initial UI w. Task Solvability Feedback w. CUA Judge Feedback

(a) `artisan-csa`: Create a single-page app, in a single HTML file, for a community-supported agriculture program with a hand-drawn, watercolor aesthetic.

Initial UI w. Task Solvability Feedback w. CUA Judge Feedback

(b) `color-match-challenge`: Create a single-page app in a single HTML file for a fast-paced "color match" game. - Show a word (e.g., "RED") in a random font color — player must click the correct color button (not the word meaning). - Keep score based on correct answers within 30 seconds. - Use large typography, color-coded buttons, and smooth button press animations.



Initial UI w. Task Solvability Feedback w. CUA Judge Feedback

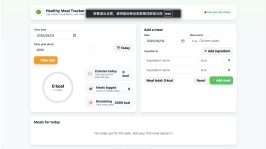
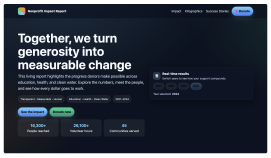
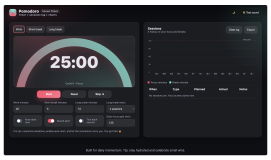
(c) `csv-to-charts`: Create a single-page app in a single HTML file with the following requirements: - Name: Data Visualization Playground - Goal: Upload CSV and generate charts. - Features: Chart type selector, color customization, save as image. - The UI should be modern with a focus on charts.

Initial UI w. Task Solvability Feedback w. CUA Judge Feedback

(d) `festival-lights-show`: Create a single-page app in a single HTML file with the following requirements: - Name: Festival Lights Show - Goal: Control a virtual light show. - Features: Color changes, patterns, music sync. - The UI should be vibrant and dynamic.

Figure 8: Qualitative comparison of initialized UI *v.s.* refined UI. Each row shows an initial UI (left), its revision based on function (middle), and its revision based on CUA-failed tasks (right).

Table 6: **Distribution and examples of six domains in AUI-Gym.** For each domain, we show a website created by GPT-5, paired with 30 tasks (blue) simulating real-world usage. Each task is further linked to a rule-based verifier (green).

Domain	#Apps	Pct.	Example Instruction	GUI by GPT-5
App	11	21%	Create a single-page app in a single HTML file with the following requirements: - Name: Healthy Meal Tracker - Goal: Log meals and nutrition info. - Features: Ingredient list, calories per meal, daily summary. - The UI should be clean with food icons. **Task: Add five meals for today's date (any names/ingredients) so today's meal count reaches at least 5.** **Rule: `#dailyMealCount >= 5`**	
Landing	10	19%	Create a single-page app in a single HTML file with the following requirements: - Name: Nonprofit Impact Report - Goal: Show measurable results of programs. - Features: Infographics, success stories, donation link. - The UI should be inspiring and visually engaging. **Task: Navigate to Success Stories and expand the first story card to reveal the full narrative.** **Rule: `#slides .slide:first-child button[aria-expanded] == 'true' OR #slides .slide:first-child.expanded exists`**	
Game	9	17%	Create a single-page app in a single HTML file with the following requirements: - Name: Typing Rain - Goal: Type falling words before they reach the bottom. - Features: Increasing difficulty, accuracy tracker, score. - The UI should be the city background with animated raindrop words. **Task: In a single run, achieve a score of at least 500 points.** **Rule: `#scoreValue >= 500`**	
Interactive	9	17%	Create a single-page app in a single HTML file with the following requirements: - Name: Festival Lights Show - Goal: Control a virtual light show. - Features: Color changes, patterns, music sync. - The UI should be vibrant and dynamic. **Task: Enable Music Sync, start playback, then pause the built-in track; confirm audio status is Paused while Music Sync remains enabled.** **Rule: `#audioStatus == 'Paused' AND #syncBadge != 'Sync: Off'`**	
Tool	7	13%	Create a single-page app in a single HTML file with the following requirements: - Name: Customer Journey Flow - Goal: Sketch customer journey stages and connections. - Features: Add/edit stages, connect nodes, view JSON of the flow. - The UI should be simple and full-screen. **Task: Create 'Social Ad' and 'Search Ad' leading to 'Landing Page', then to 'Consideration' and 'Purchase' (two branches merging into one path).** **Rule: `#io-json contains 'Social Ad' AND #io-json contains 'Search Ad' AND #io-json contains 'Landing Page' AND #io-json contains 'Consideration' AND #io-json contains 'Purchase'`**	
Utility	6	12%	Create a single-page app in a single HTML file with the following requirements: - Name: Pomodoro - Goal: Time focus and break sessions. - Features: Focus/break modes, timers, basic controls. - The UI should be minimal and distraction-free. **Task: Start a short break and verify the mode label and starting time show a 5-minute break.** **Rule: `#lblSession == 'Short Break' AND #lblTime contains '05:00'`**	

G Agent's Prompts

Task Proposer Prompt. Below, we show the prompt used by the Task Proposer.

Task Proposer Prompt

Propose a comprehensive set of 30 diverse, realistic user tasks for the following {tag_type} application:

Application: {app_title}
Description: {app_description}

Task Requirements
Each task should be:
1. Clear and specific in its description
2. Represent realistic user scenarios
3. Cover different complexity levels and use cases
4. Grounded in an observable outcome: The task's completion must be marked by a clear and unambiguous change in the application's state or interface. The expected outcome description must precisely define this terminal state.
5. Avoid single element grounding (focus on complete workflows)
6. Test the application's core functionality effectively

{tag_specific_content}

Tag Philosophy Template:

Game:
Focus on GAME-SPECIFIC user tasks:
1. Playing complete game rounds or levels
2. Achieving high scores and personal bests
3. Completing specific game objectives or challenges
4. Using game controls and input methods
5. Navigating game menus and settings
6. Restarting games and trying different strategies
7. Progressing through difficulty levels

Additional task requirements:
1. Focus on actual gameplay actions and goals
2. Include winning and losing scenarios
3. Cover different skill levels and strategies
4. Test game restart and replay functionality
5. Emphasize user enjoyment and engagement

Tool:
Focus on TOOL-SPECIFIC user tasks:
1. Creating or generating content using the tool
2. Inputting data in various formats and types (typed/-pasted text or on-page controls)
3. Transforming and processing information
4. Previewing results in-page (no file uploads/downloads)
5. Using tool-specific features and options
6. Working with both simple and complex inputs
7. Completing end-to-end workflows within the page

Additional task requirements:
1. Focus on practical use cases and workflows
2. Include both basic and advanced tool usage
3. Cover different input types and scenarios without external files
4. Verify visible in-page outputs or status changes in the DOM
5. Emphasize real-world problem solving

Utility:
Focus on UTILITY-SPECIFIC user tasks:
1. Setting up and configuring the utility for personal use
2. Adding, organizing, and managing data or items
3. Tracking progress and monitoring status over time
4. Using timers, reminders, and scheduling features
5. Customizing settings and preferences
6. Completing daily or routine activities
7. Accessing and updating information quickly

Additional task requirements:
1. Focus on everyday productivity scenarios
2. Include setup and personalization tasks
3. Cover routine and habitual usage patterns
4. Test organization and tracking features
5. Emphasize practical daily life applications

Interactive:
Focus on INTERACTIVE-SPECIFIC user tasks:
1. Exploring and experimenting with interactive elements
2. Creating and manipulating visual or audio content
3. Adjusting parameters and settings in real-time
4. Playing with creative tools and features
5. Experiencing immersive visual or audio effects
6. Using touch, click, and gesture interactions
7. Customizing appearance and behavior

Additional task requirements:
1. Focus on creative and exploratory activities
2. Include experimentation and play scenarios
3. Cover different interaction methods
4. Test customization and personalization
5. Emphasize sensory and aesthetic experiences

Landing:
Focus on LANDING-SPECIFIC user tasks:
1. Browsing and exploring page content and sections
2. Reading and understanding key information
3. Clicking on call-to-action buttons and links
4. Navigating through different page sections
5. Finding contact information and ways to engage
6. Viewing team, product, or service details
7. Accessing additional resources and links

Additional task requirements:
1. Focus on visitor browsing and exploration
2. Include information-seeking behaviors
3. Cover engagement and conversion actions
4. Test navigation and content discovery
5. Emphasize typical visitor journey scenarios

App (Default):
Focus on APP-SPECIFIC user tasks:
1. Creating, editing, and managing content or data
2. Using multiple features in combination
3. Setting up and personalizing the application

4. Completing complex multi-step workflows
5. Organizing and categorizing information
6. Accessing and updating saved information

Additional task requirements:

1. Focus on practical in-app usage
2. Include multi-feature workflows and combinations
3. Cover content creation and management
4. Test personalization and customization
5. Verify completion via visible state changes in the DOM (no external integrations)

Task Categorization Framework:
Each task must be categorized into one of the following three archetypes, which provides a structured approach to evaluating different facets of the application's functionality:

1. "core_function": Tests a single, primary feature in isolation.
2. "user_workflow": Tests a sequence of features that represent a complete user goal.
3. "edge_case": Tests non-standard inputs, boundary conditions, or less common interaction patterns.

Please respond in JSON format:

```
{
  "app_name": "<app_name>",
  "tags": ["<tag1>", "..."],
  "tasks": [
    {
      "id": 1,
      "description": "Clear, specific
          task description",
      "category": "core_function|
          user_workflow|edge_case",
      "expected_outcome": "What
          should happen when task
          completes"
    }
  ]
}
```

Coder Prompt. Below, we show the prompt used by the Coder.

Coder Prompt

Initial Website Generation
Create a single-page web application based on the following specification:
{instruction}
Requirements:

1. Create a complete HTML file with embedded CSS and JavaScript
2. The app should be fully functional and interactive
3. Use modern HTML5, CSS3, and vanilla JavaScript (no external libraries)
4. Include proper semantic HTML structure
5. Make the UI clean, responsive, and user-friendly
6. Add unique IDs to interactive elements for easier automation testing
7. Ensure the app works in a 1280x720 viewport

Please generate the complete HTML file:

Revision from CUA Failures
You are tasked with improving a web application based on detailed failure analysis from automated testing.
Context
Application: {app_name}
Model: {model_name}
Total Failed Tasks: {len(failed_tasks)}
Failure Categories: {list(failure_categories.keys())}
Original HTML Length: {len(initial_html.strip())}
Output Format
Generate a single, complete, and self-contained HTML file. The file must be fully functional, including all necessary CSS and JavaScript, from `!DOCTYPE html` to `/html`. Do not use placeholders or truncate the code.
Original Initial Website (Full)

```
{initial_html}
```

Commenter UI Analysis
{(failure_analysis or "No visual UI analysis available").strip()}
{(non_regression_contract_prompt or ").strip()}
Improvement Requirements
Core Issues to Address
Based on the failure analysis, you must:

1. Identify missing DOM elements that tasks expect to exist
2. Add missing JavaScript functionality for user interactions
3. Fix timing issues that prevent task completion
4. Ensure proper event handling and state management
5. Add missing visual feedback and UI updates

Specific Fixes Needed
For each failed task category:

1. **basic_usage**: Ensure fundamental interactions work (clicking, displaying, updating)
2. **workflow**: Support complete user workflows and multi-step processes
3. **advanced_feature**: Implement sophisticated UI behaviors and animations
4. **edge_case**: Handle unusual inputs and boundary conditions properly

Technical Implementation Guidelines

1. Preserve ALL existing working functionality from initial version
2. Add missing HTML elements with unique IDs for automation
3. Implement complete JavaScript event handlers and state updates
4. Ensure synchronous UI updates for immediate feedback
5. Do NOT introduce new input constraints that would block task inputs implied by the tasks (e.g., accept plain text or non-HTTP payloads if tasks need them). Validation must be permissive and never reduce what the initial version allowed.
6. Do NOT auto-trigger flows on page load that would change initial states relied upon by tasks (e.g., auto-generation, auto-download, auto-navigation). Initial state should be neutral and idle.

7. Keep critical controls visible within a 1280x720 viewport without scrolling. Avoid multi-panel "hub" layouts; prefer single-view, compact layouts that fit important controls on screen.
8. Avoid adding non-essential animations/transitions; prioritize high visibility and clarity over decoration.
9. Make sure timers, counters, and dynamic content work correctly

DOM Structure Requirements

1. Every interactive element MUST have a unique ID
2. Form controls must have proper event listeners
3. Dynamic content areas must update immediately on state changes
4. Visual feedback must be implemented for all user actions

JavaScript Functionality Requirements

1. All user interactions mentioned in failed tasks must be fully implemented
2. State changes must be reflected in the DOM immediately
3. Event handlers must properly update all related UI elements
4. Any game logic, scoring, timing must be complete and functional

Surgical Revision Policy

1. Preserve existing IDs; do not rename or remove working elements from initial version.
2. Avoid large rewrites. Patch only the functions, event handlers, and minimal markup necessary to satisfy the failed/unsupported tasks.
3. Preserve working logic from initial version; do not regress features that already work.
4. Reuse existing elements/IDs for state wherever possible; only add new IDs if strictly necessary to expose the state of new logic.
5. Preserve initial version immediacy semantics. Do NOT introduce extra confirmation steps as prerequisites where initial version achieved completion via immediate interactions. Implement functional logic first, then expose proxies from the same code path; never update proxies without the underlying state change.

Commenter JSON (if provided)

1. If the COMMENTER UI ANALYSIS is a JSON object, prioritize applying entries in `actionable_changes` precisely.
2. Keep changes surgical and bounded by those actionable suggestions; do not broaden scope beyond them.

Output Requirements

Generate a COMPLETE, FULLY FUNCTIONAL HTML file that:

1. Addresses ALL failure points identified in the analysis
2. Maintains existing successful functionality from initial version
3. Implements missing features causing task failures
4. Provides proper DOM elements for automation testing
5. Ensures immediate UI feedback for all user actions

Agent-Centric Design Principles

While improving functionality, apply the following design principles to optimize the UI for automated agents. The goal is functionality and testability, not human aesthetics.

Visual Clarity and Simplicity

1. Use a simple color scheme (e.g., black text on a white background).
2. Avoid decorative elements that do not serve a functional purpose, such as animations, gradients, or shadows.
3. Establish a clear visual hierarchy using typography and spacing. Logically group related controls.

Robust Agent Interaction

1. All interactive controls must be clearly labeled and sized appropriately to be easily and unambiguously targeted by automation tools.
2. Support keyboard-based interaction for all core functionality. Navigable elements should have clear focus indicators.
3. Prioritize immediate state updates upon interaction. Avoid complex, multi-step confirmation dialogs for actions where direct manipulation is sufficient.
4. All critical functionality should be accessible within a standard 1280x720 viewport without requiring scrolling.

Transparent State Management

1. The DOM must serve as a reliable, single source of truth for the application's state.
2. Ensure that any significant state change (e.g., a result is generated, a calculation is complete) is clearly and synchronously reflected in the DOM. This can be achieved by updating element attributes, text content, or values.
3. Interactive elements and state indicators must have unique and stable IDs to facilitate reliable testing and interaction.

Versatile Input Handling

1. For continuous inputs (like sliders), provide alternative discrete control mechanisms (e.g., step buttons, direct text input). No interaction should rely solely on pointer-dragging.
2. Input validation should be permissive and should not block inputs that an automated task might reasonably provide.
3. Distinguish between actions that cause immediate, reversible state changes (e.g., selecting an option) and those that trigger irreversible, multi-step processes (e.g., submitting a form).

Behavior Preservation

1. Simplifying the visual design must not alter the core interaction logic.
2. Any user action that was immediate in initial version must remain immediate in the revised version.

Please generate the complete improved HTML file:

Revision from Unsupported Tasks

You are tasked with improving a web application to support additional tasks that are currently unsupported.

Context

Application: {app_name}
Model: {model_name}
Total Unsupported Tasks: {len(unsupported_tasks)}
Original HTML Length: {len(initial_html.strip())}

Output Format

Generate a single, complete, and self-contained HTML file. The file must be fully functional, including all necessary CSS and JavaScript, from `!DOCTYPE html` to `/html`. Do not use placeholders or truncate the code.

Original Initial Website (Full)

```
{initial_html}
```

Unsupported Tasks Analysis
{unsupported_summary}
Code Preservation Contract (Non-Regression)
{" if ablate_no_contract else (non_regression_contract_prompt or ").strip()}
Improvement Requirements
Task Support Issues to Address
Based on the unsupported task analysis, you must ADD missing functionality:
1. Add missing DOM elements that tasks expect to exist
2. Implement missing JavaScript functionality for user interactions
3. Add missing form controls and input handling
4. Implement missing display areas and visual feedback
5. Add missing navigation and UI components

Implementation Guidelines
1. PRESERVE all existing working functionality from initial version
2. ADD new HTML elements with unique IDs for automation
3. IMPLEMENT complete JavaScript event handlers for new features
4. ENSURE new UI elements are properly styled and visible
5. DO NOT introduce new input constraints that would block task inputs implied by tasks; validation must be permissive and must not reduce what the initial version allowed.
6. DO NOT auto-trigger flows on load that change initial states (no auto-generation, auto-download, auto-navigation). Start in a neutral, idle state.
7. FIT critical controls within a 1280x720 viewport without scrolling. Avoid multi-panel hub layouts and unnecessary panels that push controls below the fold.
8. IMPLEMENT missing workflows and user interaction patterns

DOM Structure Requirements
1. Every new interactive element MUST have a unique ID
2. New form controls must have proper event listeners
3. New content areas must update appropriately on state changes
4. New visual feedback must be implemented for added interactions

JavaScript Functionality Requirements
1. All new user interactions mentioned in unsupported tasks must be fully implemented
2. New state changes must be reflected in the DOM immediately
3. New event handlers must properly update all related UI elements
4. Any new game logic, scoring, timing must be complete and functional

Output Requirements
Generate a complete and fully functional HTML file that:
1. Maintains all existing functionality from initial version.
2. Adds the missing functionality required to support the new tasks.
3. Implements all necessary DOM elements and JavaScript for task support.
4. Ensures all new features are robust and testable.

Commenter JSON (if provided)
1. If upstream provides a commenter JSON analysis with `actionable_changes`, follow those changes first, precisely and surgically.

Surgical Revision Policy
1. Preserve existing IDs; do not rename or remove working elements from initial version.
2. Avoid large rewrites. Patch only the functions, event handlers, and minimal markup necessary to satisfy the failed/unsupported tasks.
3. Preserve working logic from initial version; do not regress features that already work.
4. Reuse existing elements/IDs for state wherever possible; only add new IDs if strictly necessary to expose the state of new logic.
5. Preserve initial version immediacy semantics. Do NOT introduce extra confirmation steps as prerequisites where initial version achieved completion via immediate interactions. Implement functional logic first, then expose proxies from the same code path; never update proxies without the underlying state change.

Please generate the complete improved HTML file:

CUA Policy Prompt. Below, we show the prompt used by the CUA Policy.

CUA Policy Prompt

You are a GUI agent. You are given a task and your action history, with screenshots. You need to perform the next action to complete the task.
Output Format

```
Thought: ...
Action: ...
```

Action Space
1. `click(point='x1 y1')`
2. `left_double(point='x1 y1')`
3. `right_single(point='x1 y1')`
4. `drag(start_point='x1 y1', end_point='x2 y2')`
5. `hotkey(key='ctrl c')` — Split keys with a space and use lowercase. Also, do not use more than 3 keys in one hotkey action.
6. `type(content='xxx')` — Use escape characters \', \", and \n in content part to ensure we can parse the content in normal python string format. If you want to submit your input, use \n at the end of content.
7. `scroll(point='x1 y1', direction='down or up or right or left')` — Show more information on the `direction` side.
8. `wait()` — Sleep for 5 s and take a screenshot to check for any changes.
9. `finished(content='xxx')` — Use escape characters \', \", and \n in content part to ensure we can parse the content in normal python string format.

Note
1. Use {language} in `Thought` part.
2. Write a small plan and finally summarize your next action (with its target element) in one sentence in `Thought` part.

User Instruction
{instruction}

Dashboard Commenter Prompt. Below, we show the prompt used by the Dashboard Commenter.

Dashboard Commenter Prompt

You are diagnosing a UI design issue that caused a task failure for a Computer-Use Agent (CUA). Your goal is to conduct a root cause analysis based on a core set of design principles and output a structured diagnostic report in JSON format. This report will guide the next iteration of UI code generation.

You will be provided with two images:
1. The current website state (Resolution: {width}x{height})
2. A storyboard summarizing the failed task attempt, arranged as a grid of step screenshots (variable count) fitted into a 1920x1080 canvas

Your analysis must be guided by the following Agent-Centric UI Design Principles:
1. **State Visibility:** Any significant state change resulting from an agent's action must be clearly and synchronously reflected in the DOM. This can be achieved by updating element attributes, text content, or values. Ambiguous or out-of-band feedback (like temporary toast notifications) is considered a violation.
2. **Interaction Robustness:** All UI components critical for task completion must be visible and actionable within a standard 1280x720 viewport without requiring scrolling. Elements should have clear, stable identifiers.
3. **Input Permissiveness:** Input fields and controls should accept the most general data format required for the task, avoiding overly restrictive client-side validation that may block agent inputs.
4. **Predictable Behavior:** The UI should remain in a stable, neutral state upon loading.

Based on these principles, analyze the provided materials and output a compact JSON object.

Output Format
Output strictly as JSON with these keys only:
1. `issues`: An array of up to 3 short strings identifying the primary UI problem categories, derived from the violated principles (e.g., "visibility", "interaction", "feedback").
2. `actionable_changes`: An array of 3–6 diagnostic statements. Each statement must identify a specific UI element (referencing selectors/IDs) and explain which design principle it violated, providing a root cause for the failure. Example: "The element '#submit-btn' violates the Interaction Robustness principle, as it is not visible in the default viewport."
3. `fit_within_screen`: A diagnostic boolean flag. Set to `true` only if the primary reason for failure was a violation of the Interaction Robustness principle concerning viewport visibility.
4. `avoid_regressions`: A confirmation flag, set to `true`, signifying that the diagnosis adheres to a "minimal intervention" philosophy. This confirms the analysis focuses solely on fixing the observed failure without disturbing unrelated, functional parts of the UI.

Respond with JSON only, no extra text.

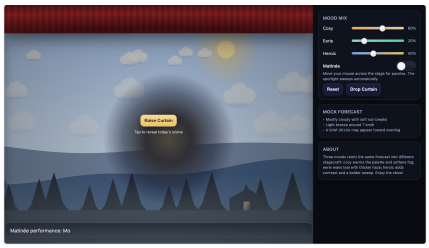
Task	Load the app for the first time and wait for the curtain reveal to complete.
Website	 1280×720
Dashboard	**Task:** Load the app for the first time and wait for the curtain reveal to complete. **Result:** Failure **Expected:** An element representing the curtain becomes hidden/removed and the canvas stage is visible; no user interaction needed. **1. Action:** Click(458,346) **Thought:** I noticed a prominent yellow button in the center of the screen that says "Raise Curtain." This button is clearly the key to starting the curtain-raising sequence. According to the task requirements, I need to **2. Action:** Click(456,348) **Thought:** I noticed a prominent yellow button in the center of the screen that says "Raise Curtain." According to the task requirements, I need to click this button to reveal the scene. This button seems to be the key to **3. Action:** Click(455,346) **Thought:** I noticed that the curtain on the stage is still closed. To see the scene that will unfold today, I need to click the "Raise Curtain" button in the center of the screen to lift the curtain. This button is quite prominent, **4. Action:** Click(456,346) **Thought:** I noticed a button in the center of the screen that says "Raise Curtain," which is exactly what I need to click on. The interface is quite simple, and the button is clearly visible. By clicking it, I can reveal today's **5. Action:** Click(453,342) **Thought:** I noticed there's a "Raise Curtain" button in the center of the screen, which seems to be the key to revealing the scene. The interface is quite minimalist, and this yellow button stands out prominently. I'm about to click **6. Action:** Click(1080,268) **Thought:** I noticed a button labeled "Drop Curtain" in the upper right corner of the interface. This seems to be the key to controlling the curtain. Since the task requires us to experience the reveal of the curtain, I should go Before: $6 \times 1280 \times 720 \rightarrow$ After: $1 \times 1950 \times 975$, 76.2% tokens reduction
Result	Failure
Comments	The weather-theatre app requires button clicks to trigger curtain reveal, but the task expects automatic curtain opening on first load without user interaction, creating a fundamental mismatch between expected auto-start behavior and actual manual activation requirement.

Table 7: **Illustration of CUA Dashboard.** The dashboard generates one informative image that clearly demonstrates how the CUA performs each step along with the corresponding observations, while reducing visual tokens by cropping to the key interactive regions.