CUA-SWE: When Computer-Use Agents Meet Visual Software Engineering

Anonymous authors
Paper under double-blind review

Abstract

Software development requires more than editing code: developers repeatedly run software, interact with its interfaces, visually inspect its behavior, and use these observations to decide what to change next and whether a change works. Existing coding agents and computer-use agents are largely studied in isolation, leaving this integrated development process underexplored. Diagnosing a runtime interaction failure requires agents to connect visual observations with the responsible code, then use the application again to verify the repair. We introduce CUA-SWE, a benchmark, a scalable and verifiable environment, and evaluation pipeline for software engineering with computer use. Beyond studying how GUI feedback supports diagnosis and repair, we ask whether agents can complete software engineering tasks when required specification or operational information is available only through the running application's visual interface. CUA-SWE spans four software engineering domains and requires agents to modify code and configuration, execute commands, interact with running software, and inspect visual feedback within the same task. Each task includes deterministic, task-specific tests that verify whether the resulting software satisfies the requirements and preserves specified behavior. Our evaluation characterizes how frontier agents combine source-level execution with application screenshots and graphical interaction to produce verified software changes. We examine performance across domains and task information requirements, alongside the development behaviors associated with successful repairs. CUA-SWE provides a unified testbed for studying how agents use visual feedback and interaction to guide software engineering, with executable correctness criteria for the resulting software.

1 Introduction

Developing interactive software requires an agent to use the software it writes. Source code and command output do not fully reveal what users see or what happens when they interact with an application (Yang et al., 2024b). An agent needs to run the software, operate its interface, and use visual clues to diagnose problems and guide code changes (Aggarwal & Welleck, 2025). It must then check how those changes affect the running software (Lu et al., 2025). In games, this requires observing behavior over time and checking responses to player input (Chi et al., 2026; Luo et al., 2026). Consider a platform game in which the character falls through a platform after jumping onto it. The game builds and starts, and a static screenshot may look correct. To diagnose the failure, an agent must play the game, observe the jump and landing, and connect that behavior to the movement or collision code. After editing the code, it must replay the jump and check that the character lands correctly while normal movement still works.

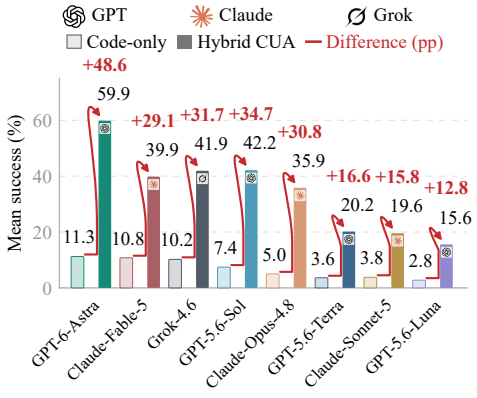


Figure 1: **Code-only and Hybrid CUA agents.** Task success across four equally weighted software engineering domains.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Computer-use agents and coding agents approach software from complementary directions. Computer-use agents operate interfaces through visual observations and mouse and keyboard actions (Agashe et al., 2025; Qin et al., 2025; Wang et al., 2025), while coding agents inspect repositories, edit files, and execute commands and tests (Yang et al., 2024a; Wang et al., 2024). Their benchmarks typically assess either changes to a codebase (Jimenez et al., 2024; Miserendino et al., 2025) or the completion of user tasks in existing applications (Xie et al., 2024; Koh et al., 2024). Visual software benchmarks incorporate runtime feedback, but concentrate on particular development domains, such as user-facing JavaScript software and games (Yang et al., 2024b; Chi et al., 2026). Broader environments emphasize access to development tools through a visual IDE (Aggarwal & Welleck, 2025), or the orchestration of graphical and command-line interfaces across general workflows assessed by an agentic judge (Li et al., 2026). Together, these capabilities motivate two central questions: **(1) How do agents use GUI feedback to diagnose, repair, and verify software? (2) Can agents complete software engineering tasks when required specification or operational information is available only through the running application's visual interface?** Visual observations can thus supply both feedback on an implementation and requirements that the agent must recover and translate into working code. Addressing these questions across software engineering domains calls for an environment that connects software use to source-code modification, together with a benchmark that verifies the resulting behavior. Deterministic tests provide reproducible checks of whether a change satisfies the task and preserves specified functionality.

To address this challenge, we introduce **CUA-SWE**, a benchmark, interactive development environment, and evaluation pipeline for visual software engineering with computer-use agents, spanning web development, game development, mobile app development, and DevOps. Tasks focus on diagnosing, repairing, and verifying software behavior. Each task provides an editable project and running software that the agent can operate and visually inspect throughout development. To reflect how developers work, the environment lets agents choose when to inspect source code, execute commands, interact with the software, examine screenshots, and revise their implementation. Coding and computer use are available within the same development episode: visual observations can guide code changes, and subsequent interaction lets the agent inspect their effects.

CUA-SWE provides deterministic evaluation with verifiable task outcomes. Each task specifies the behavior to implement or repair and the existing functionality that must be preserved. Task-specific tests execute the modified software in a controlled setting, using interaction sequences and assertions on the rendered interface or runtime state to verify the requested behavior and detect regressions. The environment also supports a *code-only* condition in which agents inspect and edit source, execute permitted builds, tests, and custom scripts, and read their outputs. Hybrid CUA adds screenshots and graphical interaction with the running application. The same task requirements and protected tests apply to both conditions, allowing us to measure development with source-level execution and with additional application access through its GUI.

Our contributions are:

- **An environment for software development through coordinated coding and computer use.** Agents work with editable projects, development commands, mouse and keyboard actions, and screenshot observations. Visual runtime evidence connects rendered behavior to code changes, and subsequent interaction lets agents check the effects of their edits.
- **A benchmark with deterministic evaluation and verifiable outcomes.** Across four software engineering domains, tasks connect visually observable application behavior to an implementation or repair requirement and define the existing functionality to preserve. Independent tests verify the resulting software through execution and interaction, providing consistent criteria for task success.
- **An evaluation of frontier models and coding agents.** We evaluate a broad range of frontier models and coding agents on CUA-SWE and characterize task success across domains and application families, including tasks that require recovering specification or operational information from the running application's visual interface and translating it into working software. The code-only reference includes nonvisual execution and command feedback; hybrid CUA additionally provides application screenshots and graphical actions. We examine this comparison by task information requirements and use trajectories to study how agents connect execution feedback and visual observations to implementation and verification.

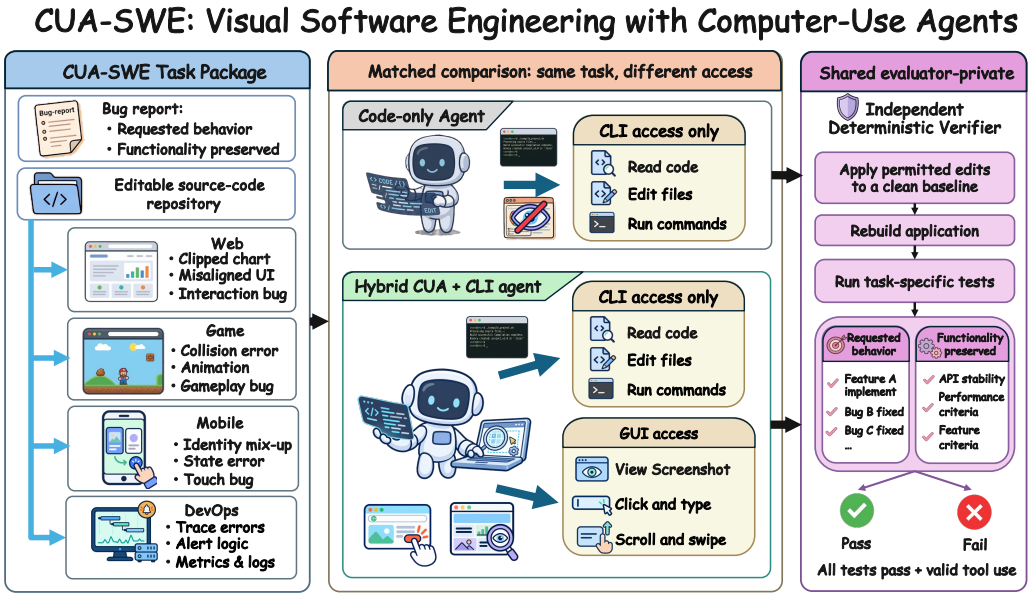


Figure 2: **The CUA-SWE environment and evaluation pipeline.**

2 When CUA Meets Visual Software Engineering

The computer-use channel exposes the rendered application through screenshots and graphical actions, while coding tools provide direct source access. This observation boundary makes interpreting the visible interface and choosing actions part of the measured capability. Structured DOM, HTTP, and application-state queries would supply a different observation channel; the benchmark's visual condition measures development through the interface presented to the agent.

Problem Formulation. For task u, the agent receives an instruction I_u, an editable codebase c_u^0, and access to its running application through screenshots and graphical actions. The task also defines a deterministic verifier V_u that checks whether a submitted codebase satisfies the requested functionality, preserves designated existing behavior, and respects constraints on permitted changes. The agent interacts with the development environment within a finite action budget and produces a final codebase c_T, where T is the number of actions taken before submission or budget exhaustion. We denote its changes relative to c_u^0 by Δc. We model the interaction as a finite-horizon partially observable Markov decision process $\mathcal{M}_u = (\mathcal{S}_u, \mathcal{A}, \mathcal{O}, P_u, \Omega_u, R_u)$, where $\mathcal{S}_u$, $\mathcal{A}$, and $\mathcal{O}$ are the state, action, and observation spaces; P_u and Ω_u are the transition and observation models; and R_u is the terminal repair reward determined by V_u.

State and observation. At step t, the environment state is $s_t = (c_t, x_t, b_t) \in \mathcal{S}_u$, where c_t is the current codebase (with $c_0 = c_u^0$), x_t is the complete runtime state of the development environment and running application, and b_t is the remaining action budget. The agent does not observe s_t directly. Instead, it receives an observation $o_t \in \mathcal{O}$ containing requested file contents, command output, or screenshots of the running application. Screenshots provide visual runtime evidence about the rendered interface and its response to interaction. These observations give a partial view of execution: the visible effect of a code change can depend on prior inputs and whether the application has loaded the edit. After action $a_t \in \mathcal{A}$, the environment evolves according to $s_{t+1} \sim P_u(\cdot \mid s_t, a_t)$ and returns $o_{t+1} \sim \Omega_u(\cdot \mid s_{t+1}, a_t)$.

Action space, verification, and reward. The agent can modify the project, operate the software, or submit its work. We define the agent's action space as follows: $\mathcal{A} = \mathcal{A}_{\text{coding}} \cup \mathcal{A}_{\text{computer}} \cup \{\text{finish}\}$. $\mathcal{A}_{\text{coding}}$ contains commands for inspecting and editing code and running development tools. $\mathcal{A}_{\text{computer}}$ contains screenshot observation, mouse and keyboard input, and application controls such as navigation and resizing. The finish action terminates the episode and submits the final

codebase c_T for verification. The complete interface is given in Appendix B.2. For a candidate codebase c, patch correctness combines three binary checks, $V_u(c) = V_u^{\text{task}}(c) \wedge V_u^{\text{reg}}(c) \wedge V_u^{\text{perm}}(c)$. Here V_u^{task} checks the requested implementation or repair, V_u^{reg} checks designated existing functionality for regressions, and V_u^{perm} checks that the modifications to c_u^0 respect the task's restrictions on permitted changes. Behavioral checks use controlled execution, interaction sequences, and assertions over runtime or rendered application state. For evaluation episode i on task u, we write $P_i = V_u(c_{T_i})$ for patch correctness. A separate trajectory check U_i evaluates the episode's tool access and observation evidence under its assigned condition. The reported task outcome is $S_i = P_i U_i$; Section 4.1 specifies its aggregation. The terminal repair reward is $R_u(s_T) = V_u(c_T) \in \{0, 1\}$, which supplies the correctness signal for post-training (Section 4.4).

Policy and objective. Let $h_t = (o_0, a_0, \ldots, o_{t-1}, a_{t-1}, o_t)$ denote the interaction history. A vision-language policy with parameters θ selects actions conditioned on the instruction and the interaction history, $a_t \sim \pi_\theta(\cdot \mid I_u, h_t)$. The same policy selects coding actions from $\mathcal{A}_{\text{coding}}$ and computer-use actions from $\mathcal{A}_{\text{computer}}$, allowing visual runtime evidence from application interaction to guide subsequent edits. The objective is to maximize the expected terminal reward $\mathbb{E}_{\pi_\theta, P_u, \Omega_u}[R_u(s_T)]$, where the expectation is over the policy's actions and the resulting environment interactions. Because the reward is binary, this objective maximizes the probability that the final codebase passes verification.

3 THE CUA-SWE BENCHMARK

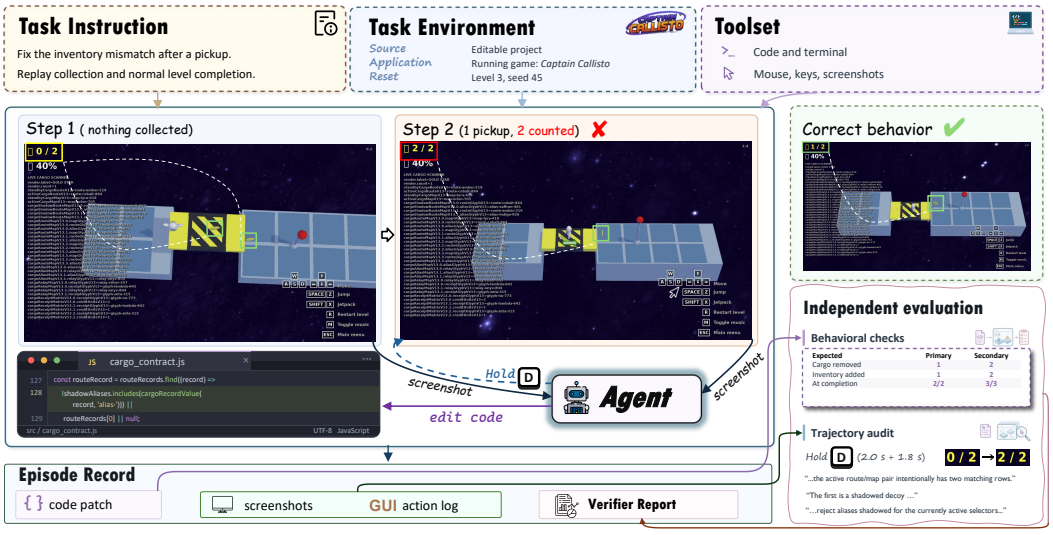


Figure 3: **A concrete benchmark task: repairing inventory updates in Captain Callisto.** Agents combine coding tools with interaction and screenshot observations. In Captain Callisto, one pickup incorrectly changes the inventory from 0/2 to 2/2. The code excerpt shows the agent's final edit, while "Correct behavior" shows a separate reference replay. Independent evaluation checks both the protected behavioral tests and the validity of tool use. The table lists expected outcomes for two replay scenarios.

We construct the benchmark around three principles. First, tasks should expose visual runtime evidence that is relevant to software development through the computer-use interface. Second, repair correctness should be determined independently of the agent's trajectory by executing V_u on the submitted codebase c_T. Third, the construction process should support controlled variations in task demands so that the benchmark can probe where hybrid coding and computer-use agents succeed or fail. We use LLM-assisted task authoring to scale this process, but admit a task only after human review and executable validation of its requirement, runtime behavior, reference solution, and verifier. We then use reviewed construction-time agent trials to identify useful variations in difficulty.

3.1 TASKS AND DOMAIN COVERAGE

CUA-SWE contains 105 tasks spanning four software engineering domains: 36 Web, 29 Game, 20 DevOps, and 20 Mobile tasks. Each task instantiates the same interface described in Section 2: the agent receives I_u and c_u^0, can modify the project through $\mathcal{A}_{\text{coding}}$, and, in the computer-use setting, can observe and operate the running software through $\mathcal{A}_{\text{computer}}$. We select these domains because they expose different relationships between runtime observations and implementation changes. Web tasks emphasize visual layout, geometry, and interaction state; Game tasks involve motion, timing, and state transitions; DevOps tasks connect service-level observations to configuration and processing logic; and Mobile tasks couple displayed interface behavior with application state. Tasks are adapted from open-source applications and documented software behaviors or built as purpose-designed applications when controlled behavior is needed. Appendix A.7 describes the sources and adaptations, and Appendix D lists the tasks.

Specification basis. Tasks also differ in where their requirements are specified. In *source-specified* (S) tasks, the instruction and readable project code define the target behavior. *Application-material-dependent* (M) tasks require interpreting additional application materials, such as imported drawings, service contracts, or graphical reference cards. These labels record specification provenance; diagnosis, implementation, and verification are required in both groups. For example, the gear task's drawing specifies component relationships, while the submitted repair must realize those relationships in the moving preview and preserve phase editing, undo, and reload behavior. Appendix A.3 defines the categories, and Appendix D records each task's specification basis.

3.2 TASK CONSTRUCTION AND VALIDATION

From runtime behavior to a task specification. We begin with a concrete software behavior that can be reproduced in the running application and whose implementation can be modified in the codebase. From this seed, an LLM author constructs the instruction I_u, initial codebase c_u^0, reproducible runtime setup, gold reference repair, plausible negative repairs, and protected behavioral tests implementing V_u. The author also specifies an interaction sequence that exposes behavior relevant to the task through the agent's available observations. Human review checks that the instruction describes an externally meaningful development requirement, that the initial software exhibits the intended problem, and that relevant runtime evidence is observable through the benchmark interface. This step connects the task specification to the partial observations o_t available to the agent rather than relying on information that is visible only to the benchmark author. For example, in the Captain Callisto task in Figure 3, collecting one item changes the visible inventory counter from (0/2) directly to (2/2). The task instruction asks the agent to repair this behavior while preserving normal item collection and level completion. The agent can reproduce the fault through game controls and observe its effect in screenshots before modifying the code.

Constructing an independent verifier. For each task, we validate the reference repair $c_u^\star$ and the deterministic verifier V_u defined in Section 2. At minimum, we require $V_u(c_u^0) = 0$ and $V_u(c_u^\star) = 1$ so that the verifier distinguishes the original faulty program from a known correct implementation. The verifier combines checks of the requested behavior V_u^{task}, preserved functionality V_u^{reg}, and permitted modifications V_u^{perm}. Behavioral checks execute the submitted program under controlled interaction sequences and may assert properties of the rendered interface, runtime behavior, or internal application state. Correctness therefore depends on the behavior of c_T, rather than on matching the reference patch. We further test the verifier with plausible incomplete or incorrect repairs. These *negative repairs* target common ways of satisfying only part of the requirement and must remain rejected by V_u. For example, clearing a masked date field should remove the selected date, whereas entering an invalid date should preserve the current selection. A repair that clears the selection in both cases should fail verification. Gold and negative repairs therefore serve different roles: the former establishes that a correct repair exists, while the latter tests whether the verifier discriminates the intended behavior from plausible partial fixes.

Validating the interactive development path. A valid verifier alone does not establish that a task is suitable for computer-use software engineering. We therefore replay the relevant runtime interactions and check that the behavior needed to understand the task can be exposed through

the benchmark interface. We verify that the application can be driven to the intended state, that relevant visual observations are available, that code changes can take effect during development, and that verification remains reproducible from a clean project state. This validation deliberately separates information available during development from information used for evaluation. The agent reasons from its instruction I_u and interaction history h_t, including source, command output, and screenshots, whereas the evaluator may use privileged application state when executing $V_u(c_T)$. For example, a game task may present only the rendered board to the agent while the verifier directly inspects the underlying board state. This allows evaluation to remain deterministic without exposing privileged information to the agent. Appendix B.3 gives further verifier examples.

3.3 Capability-oriented task expansion

Construction-time trials. After a task passes the preceding validation stages, we use construction-time agent trials to calibrate its difficulty and identify related task variants. These trials do not determine correctness (all candidate solutions are still judged by the fixed verifier V_u), but they reveal whether a task is trivial, beyond the tested agent's effective operating range, or informative about a particular failure mode. Appendix A.8 details this construction and expansion loop (see Figure 6). **Controlled variants.** When a task admits controlled variation, we construct related tasks by changing a specific development demand while retaining the underlying application and repair lineage. Examples include increasing the geometric complexity of a selection problem, extending the temporal dependency of a runtime state, or requiring an additional interaction condition to be preserved. Each variant receives its own instruction I_u, initial codebase c_u^0, and verifier V_u, and must independently pass the same human review and executable validation described above. **Task families.** This process produces task families with related semantics but different development demands. Rather than assigning a scalar difficulty label, the families allow us to examine where agent performance changes as a particular requirement becomes more demanding. They also provide a mechanism for extending CUA-SWE as hybrid computer-use agents improve. Construction trials are used only during benchmark development; the released task instructions, initial codebases, runtime conditions, and protected verifiers are fixed before the evaluations reported in Section 4. Additional construction details and task-family definitions appear in Appendices A.8 and H, respectively.

4 Experiments

We study how agents use GUI feedback during software repair and whether they can complete development tasks whose required specification or operational information is available only through the running application's visual interface. Task success measures the resulting implementation; information annotations and trajectory analyses distinguish the demands of acquiring requirements, diagnosing behavior, and implementing a repair.

4.1 Experimental setup

Tasks and agents. We evaluate frontier computer-use agents and harness systems such as Codex and Claude Code on our CUA-SWE benchmark. Full evaluated model list can be found in G. Table 1 reports the single-attempt comparison with code-only and hybrid CUA agents. Appendix F, Table 4, reports a three-attempt hybrid CUA evaluation of GPT-6-Astra, GPT-5.6-Sol, and Claude-Fable-5 on the same task collections.

Coding and Hyrbird CUA. Both conditions allow agents to inspect and edit project source, execute permitted builds, available tests, and custom scripts, and inspect their outputs. The *code-only* condition thus includes nonvisual execution feedback. *Hybrid CUA* additionally provides screenshots and graphical interaction with the running application, including its displayed state and materials. Agents choose when to execute code, observe the interface, interact, and check their changes. Both conditions use the same public task instructions and protected patch tests. Appendix A.2 details the access rules and gives a source-level execution example. Codex and Claude Code are evaluated in the CUA condition.

Metrics. For episode i on task u, patch correctness is $P_i = V_u(c_{T_i})$, combining the functionality, regression, and permitted-change checks defined in Section 2. The trajectory indicator U_i checks

tool access and observation evidence under the assigned condition. The task outcome is $S_i = P_i U_i$. Within each domain, all models follow the same condition-specific scoring rules (Appendix A.1). For evaluation set $\mathcal{E}$, we report task success as $\mathrm{TSR} = 100|\mathcal{E}|^{-1} \sum_{i \in \mathcal{E}} P_i U_i$. Appendix N reports patch correctness, task success, and the proportion of passing patches with $U_i = 0$ for each model and access condition. The four-domain mean gives Web, Game, DevOps, and Mobile equal weights. Table 1 uses one attempt per task. In Table 4, pass@1 uses these same first-attempt outcomes, while pass@3 counts tasks solved in the first or either of two additional attempts. Their difference measures the task coverage added by those attempts. Appendix A.6 defines development-time and workload measures.

4.2 Frontier performance and computer-use access

Table 1: **Single-attempt task success (%) across four domains.** Code denotes code-only access; hybrid CUA adds screenshots and graphical actions. Teal marks column maxima; an em dash indicates no reported result. Full results: Appendix G, Table 8.

	Web		Game		DevOps		Mobile	
Model	Code	Hybrid CUA	Code	Hybrid CUA	Code	Hybrid CUA	Code	Hybrid CUA
GPT-6-Astra	27.8	**66.7**	**17.2**	**37.9**	0.0	**80.0**	0.0	**55.0**
GPT-5.6-Sol	19.4	58.3	10.3	10.3	0.0	55.0	0.0	45.0
Claude-Opus-5	22.2	55.6	13.8	20.7	0.0	60.0	—	—
Claude-Opus-4.8	16.7	27.8	3.4	20.7	0.0	55.0	0.0	40.0
Claude-Sonnet-5	8.3	0.0	6.9	3.4	0.0	40.0	0.0	35.0
Claude-Fable-5	19.4	47.2	13.8	17.2	**5.0**	55.0	**5.0**	40.0
Grok-4.6	**30.6**	55.6	10.3	17.2	0.0	65.0	0.0	30.0

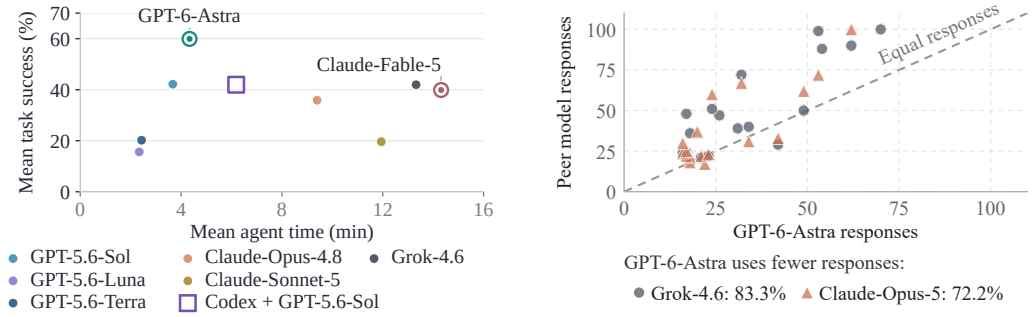
Broad competence across domains. GPT-6-Astra leads the four-domain comparison (Figure 1), with 59.9% mean task success, 17.7 percentage points above GPT-5.6-Sol. It also achieves the highest observed hybrid success within every domain. Its advantage spans graphical editing, game state, service behavior, and mobile application integration. The task-level results locate this lead more precisely: Canvas interactions after geometry changes and the recovery of exact gear and sprite relationships distinguish it from other strong agents. Table 1 summarizes these models; Appendix G provides the complete results, and Appendix H shows their distribution across task families.

Code-only versus hybrid CUA. Hybrid CUA combines source-level execution with application screenshots and GUI interaction (Figure 1). Every frontier model in the four-domain comparison achieves higher aggregate task success with hybrid access; GPT-6-Astra gains 48.6 percentage points. The gains concentrate on tasks whose requirements include application materials; source-specified subsets have near-zero domain-average changes and substantial variation across models (Appendix I). This comparison measures completing visual software engineering tasks through the provided access conditions. In material-dependent tasks, agents must recover requirements from drawings or displayed behavior, implement them, and preserve the surrounding workflow. Source-level execution tests the implementation; application observations supply requirements and feedback that those executions alone may not expose. The following analysis examines how agents turn this evidence into verified repairs under hybrid access.

4.3 Development behavior and reliability

Following the consequences of a code change. The nested-selection task separates fixing a visible defect from preserving the interaction that follows it. GPT-6-Astra first repairs resizing, then drags the object beyond its containing frame. When selection is lost, it extends hit testing to selectable descendants and repeats the interaction. GPT-5.6-Sol and Claude-Opus-5 repair the resize behavior but leave the selection defect in their submitted patches. The paired outcomes therefore identify a concrete advantage: using feedback after an edit to uncover and repair a second, connected behavior. Appendix M traces the observations, changes, and verifier outcomes.

Recovering relations and translating them into behavior. On the Mobile gear task, GPT-6-Astra reconstructs all 23 driving relations and implements their angle, phase, and editing behavior. GPT-5.6-Sol instead chooses the wrong driving rim at two compound assemblies, causing the implemented motion to depart from the drawing. A similar distinction appears in sprite animation: GPT-6-Astra recovers the complete pose sequence, whereas GPT-5.6-Terra's near-complete reconstruction retains incorrect pose assignments. In DevOps, GPT-6-Astra uniquely solves the gauge-calibration task by probing endpoint behavior and updating the inferred mode when the deployment changes. These cases connect its lead to precise recovery of relationships and operating rules, followed by implementation of their consequences across application interactions.



(a) Four-domain success and time

(b) Interaction on shared successes

Figure 4: **Success, time, and interaction effort.** **(a)** Task success rates and mean episode times for nine models and systems, averaged equally across the four domains (circles: frontier models; open square: Codex). Appendix A.6 specifies resource coverage and aggregation. **(b)** Completed responses on the same Web tasks solved by both GPT-6-Astra and each peer (18 tasks per comparison); above the diagonal, GPT-6-Astra uses fewer responses.

Success and development effort. Figure 4(a) places task coverage alongside mean development time. GPT-6-Astra's higher success than GPT-5.6-Sol accompanies a mean episode time of 4.3 rather than 3.7 minutes, combining broader coverage with a modest increase in time per attempt. Its broader coverage is accompanied by economical interaction on tasks that its peers also solve. It uses fewer responses on 83.3% of shared Web successes with Grok-4.6 and 72.2% with Claude-Opus-5 (Figure 4(b)). Each comparison holds the completed task set fixed. Broader task coverage thus coexists with fewer interaction rounds on shared successes. The repair traces show how individual observations inform subsequent edits and checks. Appendix K retains the task-level effort pairs, and Appendix L reports the broader workload profiles.

Capability boundaries and reliability. Reliable repairs require agents to coordinate coupled state and recover relations that remain valid across subsequent interactions. Core Ball exposes event-ordering demands, while Mobile schedule tasks require accurate reconstruction of dense dependencies; local corrections can leave connected behaviors unresolved (Appendix M.1). Repeated attempts distinguish stable competence from additional task coverage. In the three-attempt Game study, GPT-6-Astra solves 34.5% of tasks in all three attempts, compared with 3.4% for GPT-5.6-Sol. On Mobile, later attempts add a sprite repair for GPT-5.6-Sol and a schedule repair for GPT-6-Astra. Thus retries broaden coverage, while consistency across attempts captures how reliably agents complete these interacting requirements. Appendix F gives the full repeated-attempt results and uncertainty estimates.

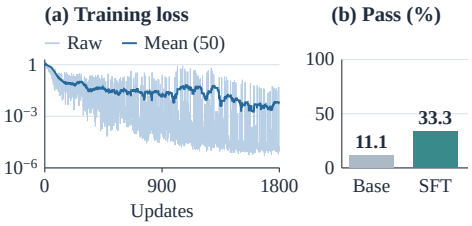


Figure 5: **SFT performance on Web domain.** (a) Batch loss and its 50-update mean (log scale). (b) Native-verifier pass rate, one greedy attempt per task.

4.4 Supervised post-training

Learning from verified development trajectories. CUA-SWE also supplies supervision for hybrid development. We fine-tune Qwen3.8-27B on action prefixes from verified reference replays

with a 27/9 train/test split, and evaluate the base and final checkpoints on the held-out tasks, with one greedy attempt per task and matched interaction settings. The native-verifier pass rate rises from 11.1% to 33.3% after SFT (Figure 5). SFT on gold-patch replays trains the model to reproduce reference repairs, providing early positive indications of a learning signal. This measure is patch correctness P_i; benchmark task success additionally checks trajectory validity U_i. Appendix E.2 preserves the full training configuration, loss summaries, and evaluation details.

Extension to RLVR. We emphasize that our core contribution remains the benchmark and environment, where the above SFT results serve as an initial study for scaling our post-training environment in the future. Our immediate future work is to establish whether CUA-SWE provides a verifiable learning signal and to develop scalable RLVR training in this environment. The executable verifiers can score newly sampled development episodes, supplying rewards for sequences of coding and computer use, while parallel application workspaces support rollout collection. Appendix E.3 specifies the RLVR objective and rollout interface.

5 Related work

Software engineering and coding benchmarks. HumanEval, LiveCodeBench, and BigCodeBench test program synthesis and execution (Chen et al., 2021; Jain et al., 2024; Zhuo et al., 2025); SWE-bench, SWE-bench Pro, and SWE-Lancer assess repository changes and professional development tasks (Jimenez et al., 2024; Deng et al., 2025; Miserendino et al., 2025). Terminal-Bench extends evaluation to complex command-line workflows (Merrill et al., 2026). CUA-SWE connects software changes to runtime observations and graphical interaction, with deterministic tests of the resulting behavior. **Computer-use benchmarks and agents.** Mind2Web, WebArena, and VisualWebArena study web interaction (Deng et al., 2023; Zhou et al., 2023; Koh et al., 2024); OSWorld, Windows Agent Arena, and AndroidWorld cover desktop and mobile workflows (Xie et al., 2024; Bonatti et al., 2024; Rawles et al., 2024). Agent S2, UI-TARS, and OpenCUA develop planning, grounding, and learned policies for computer use (Agashe et al., 2025; Qin et al., 2025; Wang et al., 2025). CUA-SWE studies these capabilities within development, where agents use software to inform and check their own code changes. **Connecting coding and computer use.** Programming with Pixels combines visual IDE interaction, coding tools, and programmatic verification (Aggarwal & Welleck, 2025). WeaveBench evaluates hybrid GUI, CLI, and coding workflows through a trajectory-aware agentic judge (Li et al., 2026). CUA-SWE focuses on software engineering across four domains, providing deterministic correctness criteria for evaluation and verifiable feedback for post-training. Appendix O extends these comparisons to visual software development, agent interfaces, and training data.

6 Limitations

The initial SFT study offers early positive indications of a learning signal from gold-patch replays (Section 4.4). Establishing whether the environment provides a verifiable learning signal and developing scalable RLVR training remain immediate future work. This requires efficient parallel rollout collection, stable optimization over long multimodal trajectories, and effective credit assignment across code edits, observations, and interactions. Appendix E.3 specifies an objective based on executable task feedback. The current benchmark spans Web, Game, DevOps, and Mobile. Expanding to more domains, applications, and tasks will broaden its coverage of visual software engineering. Future extensions can include richer development workflows and longer tasks that require agents to coordinate coding and computer use across multiple tools and applications.

7 Conclusion

CUA-SWE is a benchmark, environment, and evaluation pipeline for visual software engineering across Web, Game, DevOps, and Mobile. It connects code editing and execution with screenshots, GUI actions, and deterministic verification. Comparing code-only and hybrid CUA agents, our evaluation examines how agents use visual runtime evidence to diagnose and repair software and recover requirements supplied through application interfaces. CUA-SWE supports measuring and expanding agents' ability to translate this evidence into verified software changes.

AI USE STATEMENT

AI tools assisted with writing, literature searches, code inspection, conceptual and experimental design, result interpretation, and figure and table preparation. An image-generation model assisted with figure layout studies. Application screenshots and code excerpts in the figures come from the tasks and recorded runs. The authors are responsible for the manuscript's claims, citations, and experimental evidence.

REPRODUCIBILITY STATEMENT

Sections 2 and 3 describe the environment, tasks, and tests. Section 2 specifies the problem formulation. Section 4.4 reports supervised post-training; Appendix E.1 describes SFT data construction, Appendix E.2 gives the training configuration, and Appendices E.3–E.4 describe the RLVR extension. Appendix B documents the implementation and example tests, and Appendix D lists all tasks. Section 4.1 defines the experimental conditions and metrics; Appendix A describes the evaluation protocol, condition comparisons, and workload accounting. Appendix F specifies repeated-attempt metrics and statistical resampling.

ETHICS STATEMENT

The tasks use locally hosted applications and games, including original examples and code adapted from existing projects. Distributing this material requires preserving source and asset attribution. Evaluation tests are kept separate from the agent's editable files to prevent changes to the scoring procedure. The study does not measure human performance. Passing the benchmark tests does not guarantee that a change is safe to deploy.

REFERENCES

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. URL https://arxiv.org/abs/2504.00906.

Pranjal Aggarwal and Sean Welleck. Programming with pixels: Can computer-use agents do software engineering? *arXiv preprint arXiv:2502.18525*, 2025. URL https://arxiv.org/abs/2502.18525v2.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Jang, and Zack Hui. Windows Agent Arena: Evaluating Multi-Modal OS Agents at Scale, 2024. URL https://arxiv.org/abs/2409.08264.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

Wayne Chi, Yixiong Fang, Arnav Yayavaram, Siddharth Yayavaram, Seth Karten, Qiuhong Anna Wei, Runkun Chen, Alexander Wang, Valerie Chen, Ameet Talwalkar, and Chris Donahue. GameDevBench: Evaluating agentic capabilities through game development. *arXiv preprint arXiv:2602.11103*, 2026. URL https://arxiv.org/abs/2602.11103.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a Generalist Agent for the Web, 2023. URL https://arxiv.org/abs/2306.06070.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an End-to-End Web Agent with Large Multimodal Models, 2024. URL https://arxiv.org/abs/2401.13919.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024. URL https://arxiv.org/abs/2403.07974.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan 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). Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.50. URL https://aclanthology.org/2024.acl-long.50/.

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, 2025. URL https://arxiv.org/abs/2504.07981.

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. URL https://arxiv.org/abs/2606.09426v3.

Wei Li, William Bishop, Alice Li, Chris Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. On the Effects of Data Scale on UI Control Agents, 2024. URL https://arxiv.org/abs/2406.03679.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. WebGen-Bench: Evaluating LLMs on generating interactive and functional websites from scratch. *arXiv preprint arXiv:2505.03733*, 2025. URL https://arxiv.org/abs/2505.03733.

Tongxu Luo, Rongsheng Wang, Jiaxi Bi, Chenming Xu, Zhengyang Tang, Jianlong Chen, Juhao Liang, Ke Ji, Shuqi Guo, Yuhao Du, Fan Bu, Wenyu Du, Xiaotong Zhang, Kyle Li, Shaobo Wang, Linfeng Zhang, Yuxuan Liu, Xin Lai, Chenxin Li, Yiduo Guo, Zhexin Zhang, Xinyuan Wang, Tianyi Bai, Ziniu Li, and Benyou Wang. GameCraft-Bench: Can agents build playable games end-to-end in a real game engine? *arXiv preprint arXiv:2606.17861*, 2026. URL https://arxiv.org/abs/2606.17861.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-Lancer: Can frontier LLMs earn $1 million from real-world freelance software engineering? *arXiv preprint arXiv:2502.12115*, 2025. URL https://arxiv.org/abs/2502.12115.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, Xiaojun Xiao, Kai Cai, Chuang Li, Yaowei Zheng, Chaolin Jin, Chen Li, Xiao Zhou, Minchao Wang, Haoli Chen, Zhaojian Li, Haihua Yang, Haifeng Liu, Feng Lin, Tao Peng, Xin Liu, and Guang Shi. UI-TARS: Pioneering automated GUI interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025. URL https://arxiv.org/abs/2501.12326.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. AndroidWorld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*, 2024. URL https://arxiv.org/abs/2405.14573.

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

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2Code: Benchmarking multimodal code generation for automated front-end engineering. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3956–3974. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.naacl-long.199. URL https://aclanthology.org/2025.naacl-long.199/.

SLIME Contributors. SLIME: An LLM post-training framework. Software, commit 4c193f1f37509cca70f0e88807a9305b70f63f4e, 2026. URL https://github.com/THUDM/slime/tree/4c193f1f37509cca70f0e88807a9305b70f63f4e.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024. URL https://arxiv.org/abs/2407.16741.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, Zhennan Shen, Zhuokai Li, Ryan Li, Xiaochuan Li, Junda Chen, Boyuan Zheng, Peihang Li, Fangyu Lei, Ruisheng Cao, Yeqiao Fu, Dongchan Shin, Martin Shin, Jiarui Hu, Yuyan Wang, Jixuan Chen, Yuxiao Ye, Danyang Zhang, Dikang Du, Hao Hu, Huarong Chen, Zaida Zhou, Haotian Yao, Ziwei Chen, Qizheng Gu, Yipu Wang, Heng Wang, Diyi Yang, Victor Zhong, Flood Sung, Y.Charles, Zhilin Yang, and Tao Yu. OpenCUA: Open foundations for computer-use agents. *arXiv preprint arXiv:2508.09123*, 2025. URL https://arxiv.org/abs/2508.09123v3.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *arXiv preprint arXiv:2404.07972*, 2024. URL https://arxiv.org/abs/2404.07972.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024a. URL https://arxiv.org/abs/2405.15793.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. SWE-bench Multimodal: Do AI systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*, 2024b. URL https://arxiv.org/abs/2410.03859.

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. WebArena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023. URL https://arxiv.org/abs/2307.13854.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions, 2025. URL https://arxiv.org/abs/2406.15877.

A Evaluation protocol and measurement

A.1 Patch correctness and trajectory validation

Patch correctness. The verifier applies the submitted changes to a clean task workspace and checks the requested functionality, designated existing behavior, and restrictions on permitted changes. These are the three components of $P_i = V_u(c_{T_i})$ in Section 2. The same patch checks apply to code-only and hybrid CUA executions.

Trajectory validation. The indicator U_i checks that the execution uses the interfaces allowed by its assigned condition and satisfies the domain's observation-evidence requirements. Protected evaluation materials remain outside the agent's access. Web and Mobile require screenshot observations for CUA success. DevOps verifies screenshot use through its execution records. Game validates the assigned CUA capability and records screenshot use as a trajectory measure. Task success is $S_i = P_i U_i$.

Screenshot observations. Screenshot records link the rendered application to the agent's visual observations. The Web API interface associates each screenshot returned by the application launcher with its image-view call. Interfaces with direct image attachments record those observations in the model request; the Mobile audit links captures to the corresponding request and response.

A.2 Agent capabilities and access rules

Both conditions receive the public task instruction and editable project source. Agents can read code and configuration, modify permitted files, run nonvisual builds and available project tests, and execute custom scripts against project modules. Command output, errors, and test results are returned to the agent. For example, the evaluated 2048 code-only trajectory in Appendix C loads game modules in Node, simulates key events and timers, and checks restart behavior. Protected evaluation tests and reference patches remain outside the agent's workspace.

Code-only provides this source-level execution interface. It excludes application screenshots, graphical actions, and access to the served application through a development server, direct HTTP or sockets, or browser automation. Hybrid CUA adds the designated screenshot-and-action interface: the evaluator starts the application, and the agent observes pixels and issues supported mouse, keyboard, and navigation actions. Direct DOM reads, accessibility trees, script evaluation in the browser, and direct HTTP extraction are outside this observation interface.

The same task instruction and protected tests apply to both conditions. In Mobile, both receive the ordinary client source and build inputs; the prepared app also contains imported product materials such as drawings and issuer records. These materials can be inspected through the hybrid condition's application interface. Code-only retains its nonvisual execution and command feedback on the supplied client code.

A.3 What visual runtime evidence contributes

Information and feedback. Visual runtime evidence consists of the rendered application and its visible responses to graphical actions. Screenshots capture layout, spatial relationships, displayed text and values, and changes across interaction states. These observations supply task-specific information and support diagnosis and verification. Information absent from the editable source can include an imported drawing, a service's displayed response, or a target shown in the application. The Mobile gear task, for example, requires reconstructing connections from received assembly drawings before implementing the corresponding motion. Its state report describes the current client model; the received drawing specifies the intended component relations. The permitted code-only tests and scripts exercise the supplied client code, but do not expose the imported drawing. Agreement with that drawing requires reconstructing its relations and implementing their consequences for motion, phase, editing, and persistence. During development, an agent can also inspect a failure, edit the code, and exercise the modified behavior, as in checking a resize gesture or a restart. These activities connect visual requirement recovery to implementation and verification within the same task.

Task information requirements. We annotate each task by the location of its specification. *Source-specified* (S) tasks define the target behavior in the public instruction and readable program text; executing and observing the application supports diagnosis and checking. *Application-material dependent* (M) tasks specify part of the target behavior through application materials, such as an imported drawing, a displayed service contract, or a graphical reference card. Tasks requiring both material interpretation and runtime diagnosis receive M. The labels describe specification provenance; they do not partition tasks into information acquisition versus debugging. Both groups require implementing the requested behavior and verifying the resulting software. M includes runtime-supplied materials and bundled graphical references, as detailed below.

Annotation basis. We inspect task instructions, readable source, reference materials, and the boundary between the coding workspace and the prepared application. Model outcomes are not a classification criterion. A runtime fixture alone does not imply M: 2048 specifies restart invariants in its instruction, Fabric exposes its Help in readable HTML, and Flappy Bird includes target impulses and arcs in its source. These tasks receive S. The received gear drawings and live service contracts receive M because they supply the required interpretation. Five Boxel tasks also receive M: their restart rule appears on a graphical card bundled with the project, which the code-only interface does not display. Thus M covers 58 tasks with runtime-supplied materials and five with bundled graphical references. Appendix D gives every task's label and specification basis.

A.4 Condition comparisons

Code-only and CUA comparisons pair the same tasks within each model and domain.

The four-domain mean weights each domain's task success rate equally. CUA success is computed over the full task inventory for the nine entries evaluated in all four domains. Figure 1 uses matched tasks for eight frontier models, forming 32 model–domain cohorts and 840 paired model–task observations. Differences are computed from the unrounded means.

For the information-requirement analysis, we partition these same tasks by the labels in Appendix A.3. Within each domain, category, and model, code-only and hybrid rates use identical task membership and the original selected attempts. We report the number of tasks, both success rates, and their percentage-point difference. Summary rates average the same eight frontier models with both conditions evaluated in all four domains; full per-model results also include Claude-Opus-5 where evaluated. A category absent from a domain is not applicable. Appendix I reports the results.

Related tasks share applications and interaction patterns: Mobile's 20 tasks cover 12 families, including nine Gantt variants, and Game's 29 tasks cover 12 games. Task-weighted scores measure coverage of the specified requirements; application-family analyses characterize breadth.

A.5 Single- and repeated-attempt evaluation

Evaluation design. Table 8 reports one attempt per task. For GPT-6-Astra, GPT-5.6-Sol, and Claude-Fable-5, we combine these first-attempt outcomes with two further executions per task, each starting from a fresh workspace. Pass@1 uses the first attempt, and pass@3 counts tasks solved in any of the three attempts. Pass@1 matches Table 8. Appendix F reports cumulative coverage, success frequencies, and uncertainty estimates.

Game evaluation. The three rounds cover the same 29 task definitions and protected verifiers. Game success follows the native verifier under the assigned-access policy; browser use and screenshot grounding are recorded as trajectory measures. For each model and task, pass@1 uses the first execution, as in Table 8. Cumulative pass@k counts success in at least one of the first k executions. All repeated-attempt statistics are computed from these task-level triples.

A.6 Execution and workload accounting

Execution settings. Model, provider, interface, and execution settings define each evaluated system. Mobile uses a budget of 60 model responses and 2,700 agent seconds, a maximum of 16,384 output tokens per request, high reasoning, and a 600-second request timeout. Its client makes one

HTTP attempt per request. The Web Anthropic client permits up to ten HTTP attempts with a 300-second timeout per attempt and exponential backoff capped at 30 seconds. HTTP retries occur within a task trial.

Resource measures. Agent wall time measures the development episode; setup and final verification have separate timers where available. DevOps uses the episode timer in its budget attestation. For frontier models, steps count assistant responses, Codex steps count completed items, and Claude Code steps use the larger of assistant records and terminal turns. Native system step counts are reported in their respective units. Screenshot captures and GUI actions measure interaction with the application. Token accounting includes input, cached-input, and output tokens; Web has final token records for 718 of its 720 single-attempt executions. Response length is output tokens divided by completed responses in an episode.

Aggregation. Workload summaries use recorded episode resource measurements. Four-domain summaries average episode measurements within each domain and weight the domains equally. Agent minutes per success is the ratio of this mean episode time to the full-inventory success probability. Domain-level profiles report medians, with workload intervals spanning the 25th–75th percentiles. Shared-success analyses pair tasks completed successfully by both models with resource measurements available for both executions, then summarize their per-task effort ratios.

A.7 Task sources and adaptations

Provenance and reproducibility. Task sources include open-source applications, documented software behaviors, and purpose-built applications with controlled faults. The date-input task adapts a reported clear-on-blur defect in React Datepicker; the DevOps gauge task implements update semantics from StatsD; and Mobile tasks use synthesized applications that import visual materials. Adaptations preserve the relevant behavior while simplifying setup for reproducible execution. Construction records identify the source revision and changes made, together with the initial conditions, triggering interactions, and observable outcomes. Seeds, input materials, and service configurations reproduce the conditions needed to investigate the task.

Domain coverage and task families. Web tasks cover interface geometry, selection, and application contracts. Game tasks cover temporal behavior, physics, and state transitions across 12 games. DevOps tasks connect observations from service interfaces to configuration and data-processing semantics. Mobile tasks cover visually specified relationships and editing behavior across 12 mobile-web application families. Tasks retain their source application, failure mechanism, and shared implementation in their lineage; the nine Mobile wall-chart scheduling tasks, for example, belong to one family. New applications broaden behavioral coverage, while controlled variants extend difficulty within a family. Appendix H reports family-level performance.

A.8 SEEDED TASK CONSTRUCTION AND DIFFICULTY EXPANSION

Construction begins with a seed example and produces a family of executable visual software engineering tasks. Figure 6 shows the process: an LLM author creates the task, human review checks its requirements and visual evidence, and executable validation establishes a correct repair. Reviewed agent trials then guide the generation of neighboring tasks that extend the family's difficulty range.

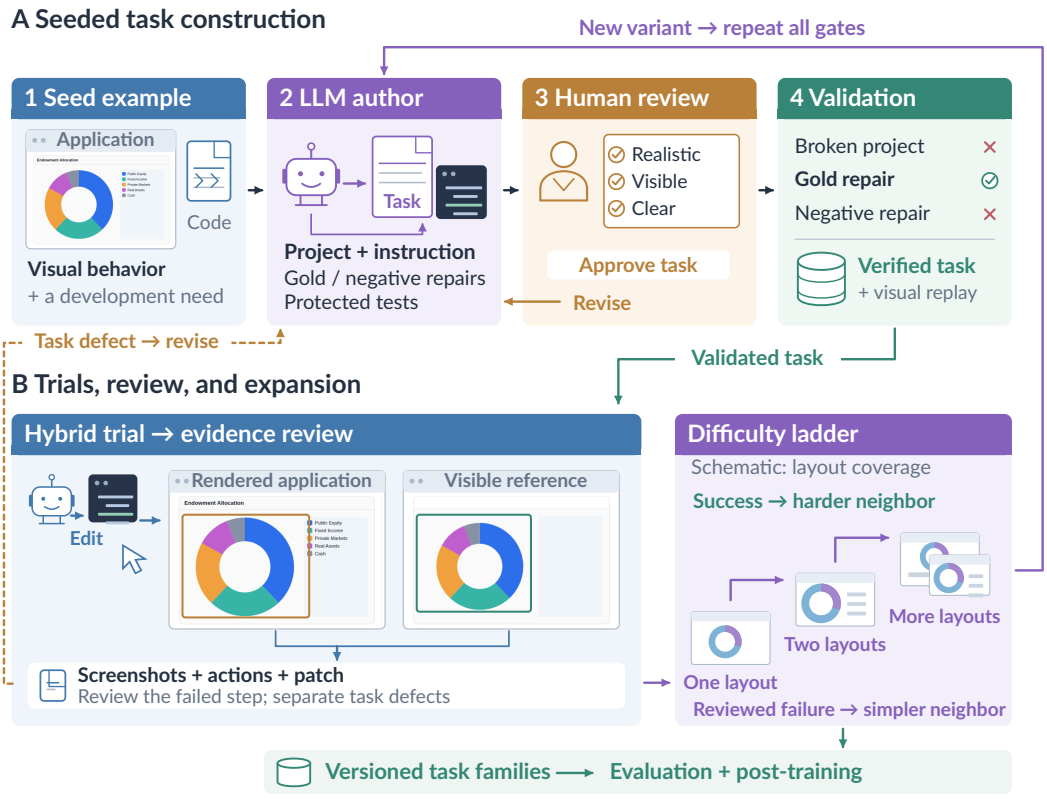


Figure 6: **From a seed example to a verified difficulty ladder.** An LLM author turns a seed into a runnable task. Human review and executable checks establish task validity. Reviewed agent trials drive a separate expansion loop, with every new variant returning through both gates. The chart panels show the rendered application and its visible reference from a recorded observation. The ladder illustrates increasing layout coverage; trials assess the difficulty of proposed variants. Validated task versions support benchmark evaluation and post-training.

Seed examples and task authoring. A seed specifies an application, a concrete development need, and an interaction that exposes the relevant visual behavior. Seeds come from the application sources described in Appendix A.7. The LLM author uses this context to construct a runnable project, a task instruction, a gold reference patch, plausible incomplete repairs, and protected behavioral tests. In the Captain Callisto inventory example (Figure 3), the seed links collecting an item to an incorrect counter update. Authoring turns that example into a requirement to correct inventory updates while preserving cargo removal and level completion, with reproducible scenarios that check these behaviors.

Human review and executable validation. Human reviewers inspect the candidate requirement, the visual evidence available through interaction, and the expected outcome. They check that the development need is meaningful, the instruction is unambiguous, and the screenshots expose the behavior needed to investigate it. Review feedback returns to the LLM author to revise the candidate. Executable checks then require the broken project and incomplete repairs to fail, and the gold patch to pass the protected tests. Interaction replay checks the corresponding visible behavior. The gold patch establishes an executable solution; negative repairs test whether the verifier rejects partial fixes.

A candidate with inconsistent behavior, an inaccessible trigger, or an inadequate verifier returns for correction and revalidation.

Agent trials and reviewed failures. Hybrid agents attempt validated tasks by observing the application, diagnosing its behavior, editing code, and checking the result. A successful trajectory supplies a concrete solution under the tested interface and budget. For a failed attempt, review connects the outcome to the screenshots, actions, and code changes, identifying the step at which the agent lost the relevant evidence or implemented an incorrect repair. Task ambiguity, runtime faults, and verifier defects trigger corrections to the task. Failures attributed to observation, reasoning, or implementation instead provide a target for expanding the task family.

Difficulty ladders and iterative expansion. The LLM author generates neighboring variants by changing one declared demand while retaining the underlying application and behavioral requirement. A reviewed failure motivates a simpler neighbor that isolates the difficult step; a successful repair motivates a harder neighbor that extends it. For example, a selection task can vary the degree of nesting, while a temporal task can vary the history needed to interpret a displayed state. Every variant passes through the same human review and executable checks before another trial. Adjacent successes and reviewed failures locate a capability boundary for the tested agent and budget; trial outcomes determine whether the proposed ordering forms a useful ladder. Repeating this loop extends the benchmark toward more demanding visual SWE problems as agents improve. Task versions retain their seed and family lineage, preserving the distinction between new applications and difficulty variants when reporting benchmark coverage and performance.

B Execution protocol and verifier examples

B.1 Execution and evaluation isolation

Each episode starts from a working copy of the task's source and a running application. Browser actions are validated for required arguments and permitted values before execution. Visual observations contain screenshots without DOM text or accessibility metadata.

The protected evaluation procedure keeps tests separate from editable files and checks the integrity of evaluation files. Patch-only verification reconstructs the application and any task-specific services, applies the candidate repair, and executes the hidden tests without invoking an agent. This separates the behavior of the submitted software from the agent's own claims or development checks.

Task setup specifies dependency installation, application startup and reset, and the viewport or device. Task-specific inputs, such as game seeds, are fixed where applicable. Each attempt receives a fresh project copy. The run protocol limits agent decisions and elapsed time, with an additional generated-token cap where configured. Interaction ends when the agent selects finish or reaches an applicable limit; the candidate change consists of the source and configuration edits accumulated by then.

B.2 Action interface

Each action specifies an operation and its admissible arguments. Shell actions take a command string and a timeout in seconds, capped by the remaining episode time. The computer interface is listed below; operations without arguments use an empty argument tuple.

Action category	Operations and arguments
Startup and observation	start, $\mathrm{observe}$
Mouse input	$\mathrm{click}(p)$, $\mathrm{scroll}(p, \Delta p)$, $\mathrm{drag}(p, p', n)$
Drag continuation	$\mathrm{drag_start}(p)$, $\mathrm{drag_move}(p, n)$, $\mathrm{drag_end}(p, n)$
Keyboard input	$\mathrm{type}(\xi)$, $\mathrm{press}(k)$, $\mathrm{key_hold}(k, d)$
Browser navigation	back, $\mathrm{forward}$
Viewport and timing	$\mathrm{resize}(w, h)$, $\mathrm{wait}(d)$

Here p, p' are viewport pixel coordinates, Δp is a scroll displacement, ξ is text, k is a key or key combination, n is an interpolation-step count, and (w, h) gives viewport dimensions. Computer-

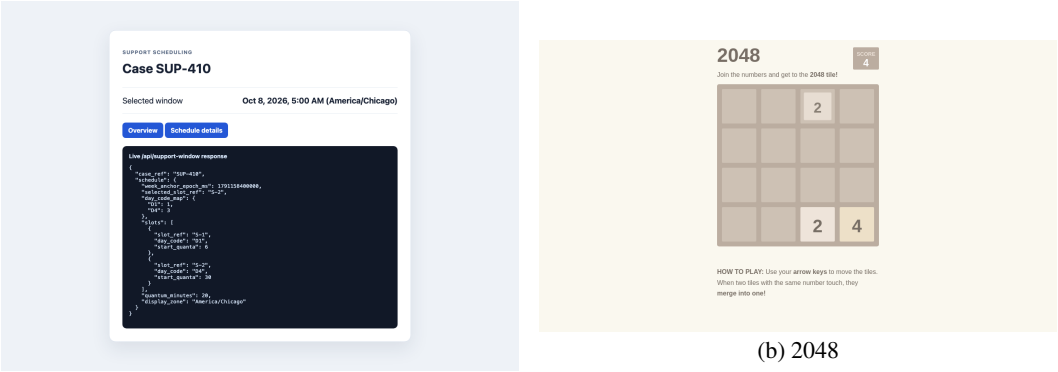
action durations d use milliseconds. Optional arguments use the tool's defaults. The separate finish action submits the current workspace for verification.

B.3 VERIFIER EXAMPLES

The support-appointment and 2048 restart tasks illustrate how tests distinguish complete repairs from changes that fix only part of a requirement. For both tasks, the original buggy version and an intentionally incomplete repair fail the tests, while the reference repair passes.

For the support-window task, the buggy version failed both service-response tests. The incomplete repair displayed the first appointment correctly but reused that value for the second appointment. The reference repair passed both tests while preserving the schedule details.

For 2048, the reference repair passed tests with seeds 42, 1337, and 3735928559. The buggy version and incomplete repair both allowed held input to change the new board after a restart. The tests compare board contents, score, occupied cells, maximum tile, and seed. They record a fresh board and a sequence of moves, hold a direction key through each restart method, and check that the new board and subsequent moves match the originals. A separate check confirms that holding a key still changes the board during normal play.



(a) Support schedule

(b) 2048

Figure 7: The support-schedule and 2048 applications with reference repairs applied, illustrating the reference solutions.

C DEVELOPMENT EXAMPLES

The first example is a code-only trajectory from the Game evaluation. The remaining development episodes and reference replays were collected separately and illustrate how visual observations guide diagnosis, code changes, and behavioral checks. The Web examples combine a view of the bug, a reference patch, and an application replay; the Game examples trace interactions and verifier feedback.

2048: checking restart behavior through code execution. In its successful code-only attempt, Claude-Fable-5 loads the keyboard, grid, tile, game-manager, and game-API modules into a Node test harness. It substitutes event handlers and timers, simulates held keys and restart inputs, and reads assertion output for seeded board restoration, queued movement after restart, and ordinary held-key movement. This execution checks program behavior without launching or observing the browser application. The submitted repair passes the task verifier.

Allocation Ring Studio: sizing a chart within its container. The chart is clipped in the Stacked layout. The reference repair measures the plot area instead of the enclosing card and removes an extra gutter deduction. A replay with the same board and layout shows the ring fitting its card. The verifier checks plot-relative radius and containment across four layout profiles and three boards.

Threadloop: revealing updates without losing reading position. The reference repair chooses between revealing a new paragraph and preserving the reader's position based on who added the text and where that collaborator is reading. The replay shows the paragraph becoming visible. Tests check the update, the reveal-or-preserve behavior, and use of the Quill runtime.

Captain Callisto: persistent corner contact. After a 900-ms `W+Shift` hold and a later 700-ms `D+W` hold, the screen clock advances from 7.4 to 14.0 seconds while the astronaut remains at the corner. The agent's patch resets the collision-contact mask each frame and removes a condition that zeros horizontal velocity. The agent later reports movement after reloading, but does not complete the route and the full task fails.

Flappy Bird: flap strength and contact during motion. A short flap produces an impulse of 120 against a target of 220, while a held flap reaches its target of 250. A contact drill also shows an overlap without a registered collision. The agent changes the impulse calculation, update ordering, and contact handling. After reloading, the short-flap drill still shows 120 against the target of 220. The overall attempt fails.

Core Ball: a completed replay can miss remaining defects. The agent traces a queued pin's incorrect attachment to an angle predicted before launch. It changes the impact-angle calculation and promotes queued shots after the previous attachment updates the core's motion. Its custom replay completes four attachments and a restart, and the build passes. The independent tests still reject the patch: the impact is resolved at the next simulation step rather than its exact arrival time, and a rejected collision still adds a pin to the occupied positions. The verifier checks both exact event timing and state preservation after rejected collisions.

Letter Arc: replay reveals an incomplete animation repair. A completed round initially shows the correct letters and colors. The agent identifies a conflict between the finish animation and the tile's reveal transform, then edits the CSS. Replaying the finish exposes a new problem: the revealed colors disappear in the paused frame. Further edits separate the animation layers and add styles preserving the revealed status during celebration. The recorded interactions do not show a successful replay of the final patch. Independently, the evaluator records a passing build and 24 passing browser checkpoints, covering visual continuity, repeated replay, Continue behavior, restart, daily/practice modes, and reduced-motion completion.

Recipient selection: code checks and a stale interface. Interface guidance indicates that Enter should append the highlighted recipient. The agent updates the selection policy and builds the application. Direct transition checks confirm that Enter appends Iceland to Norway and Sweden, while a keep operation preserves the existing selection. Browser interactions still show cleared recipients, which the agent attributes to a stale application bundle. Thus, the source-level checks pass while the recorded browser replay still shows the failure.

Vector Relay: transferring a scheduled event. The agent launches an orb and observes a lost return: the relay stays armed, the score remains zero, and the failed-return indicator turns red. It traces the failure to a scheduled event whose owner changes when the orb is replaced. The patch cancels the event on manual recall but transfers it on replacement or receiver movement. Source-level checks confirm cancellation, settlement after two transfers, and rejection of repeated contact. A subsequent browser replay shows "Circuit complete" and a score of 500; the independent evaluator also records a pass.

Hextris: continuing after a failed check. The task provides a visible implementation hint. After observing a score of nine, the agent queues blocks that settle again and delays each check until blocks of that color stop moving. The first replay still shows nine. Further inspection, temporary diagnostics, and reloads produce a cascade reaching 27. The agent then removes the diagnostics and revises the queue handling. A local test confirms that the score remains nine while a same-color block moves and reaches 27 after settlement. The final patch preserves the three-block match threshold. The independent evaluator records a pass, including two negative controls; its separate test fixture ends at 41. Figure 8 illustrates several rounds of revision and checking.

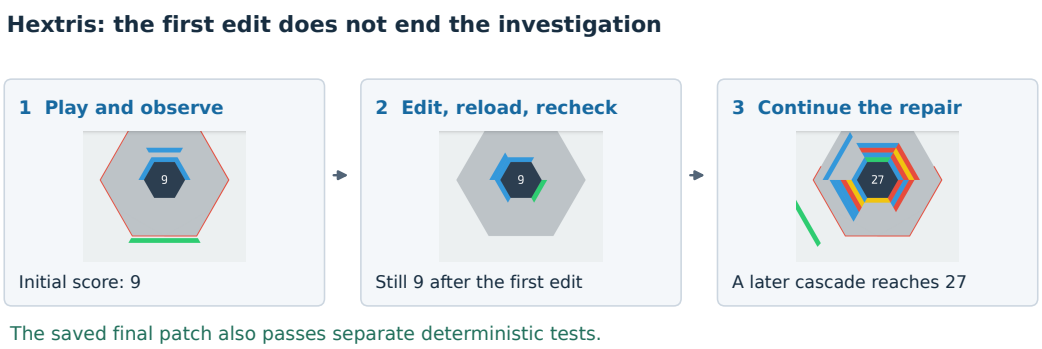


Figure 8: **A replay can show that more work is needed.** Three frames from one Hextris run show the initial score of nine, an unsuccessful recheck after the first edit, and a later cascade reaching 27. The intervening work includes further inspection, temporary diagnostics, and reloads. Screenshots are cropped around the board; the task's implementation hint is above the crop. The final source change is checked separately by the evaluator.

D COMPLETE TASK LIST

Table 2 lists the 36 Web, 29 Game, 20 DevOps, and 20 Mobile tasks, with information-requirement labels and their specification basis. M denotes application-material dependent and S denotes source-specified, as defined in Appendix A.3.

Table 2: Task information requirements. M denotes application-material dependent; S denotes source-specified (Appendix A.3). Each row gives the specification basis. Names recur across distinct scenarios; tasks follow the released manifest order within each domain.

Domain	Task	Type	Specification basis
Web	MapLibre reset touch reacquisition	S	The instruction specifies pointer attachment and preservation of navigation.
Web	Fabric opposite-edge resize	S	Readable Help in index.html specifies the resize and selection contract.
Web	ECharts linked data-zoom toolbar	S	The instruction specifies corresponding chart ranges; source exposes range-mapping strategies.
Web	Fabric viewport rotation controls	S	The instruction specifies frame, handle, and zoom agreement under camera transforms.
Web	JupyterLab Shift-click cell selection	S	The instruction distinguishes cell-range selection from extension of text selection.
Web	Fabric through-edge resize with nested camera	S	Readable Help in index.html specifies the resize and selection contract.
Web	Quill paste and scroll round trip	M	The collaboration bar supplies the presence-dependent meaning of the update.
Web	DatePicker mask, clear and blur	S	The instruction specifies cleared versus invalid-date behavior.
Web	Fabric through-edge resize with nested camera	S	Readable Help in index.html specifies the resize and selection contract.
Web	Recharts responsive pie layout	M	The approved reference image specifies how the ring fits the board.
Web	Slate soft break and scroll	S	The instruction specifies caret visibility while retaining the preceding lines.
Web	MapLibre mouse ridge attachment	S	The instruction specifies pointer attachment and preservation of navigation.
Web	Tiptap keyboard focus position	M	The workspace note and rendered draft distinguish the current bookmark from stale anchors.
Web	MapLibre reset terrain reacquisition	S	The instruction specifies pointer attachment and preservation of navigation.
Web	Fabric Single-to-Nested return	S	Readable Help in index.html specifies the resize and selection contract.

Domain	Task	Type	Specification basis
Web	Account recovery contract	M	The live identity policy specifies the applicable recovery channel.
Web	MapLibre reset integration handoff	S	The instruction specifies pointer attachment and preservation of navigation.
Web	Delegated action contract	M	The live authorization policy specifies the delegated action.
Web	Handsontable menu scroll anchor	S	The instruction specifies menu anchoring and closure on anchor disappearance.
Web	MapLibre terrain pinch anchor	S	The instruction specifies the geographic point to preserve during pinch and pan.
Web	ComboBox selection	M	The workspace keyboard guide specifies Enter, Escape, and Tab behavior.
Web	ComboBox selection	M	The workspace keyboard guide specifies Enter, Escape, and Tab behavior.
Web	Responsive masonry overlap	S	The instruction specifies persistent section visibility and non-overlap on resize.
Web	Saleor export of current search	M	The workspace export guide specifies request fields and filter encoding.
Web	MapLibre post-wheel reacquisition	S	The instruction specifies pointer attachment and preservation of navigation.
Web	MapLibre reset wheel reacquisition	S	The instruction specifies pointer attachment and preservation of navigation.
Web	Subscription renewal contract	M	The live lifecycle contract specifies the renewal interpretation.
Web	Support window contract	M	The service contract specifies the selected support window.
Web	Tax rounding contract	M	The live calculation contract specifies the tax convention.
Web	Fabric through-edge resize	S	Readable Help in index.html specifies the resize and selection contract.
Web	Vue sticky table columns	M	The table preferences supply the active density geometry token and value.
Web	Warehouse cutoff	M	The live dispatch contract specifies cutoff and shipping-day rules.
Web	Warehouse cutoff	M	The live dispatch contract specifies cutoff and shipping-day rules.
Web	Nested Fabric selection	S	Readable Help in index.html specifies the resize and selection contract.
Web	Nested Fabric selection	S	Readable Help in index.html specifies the resize and selection contract.
Web	Nested Fabric selection	S	Readable Help in index.html specifies the resize and selection contract.
Game	Captain Callisto: jetpack timing	S	The instruction and jetpack implementation define burst growth, release, and restart consistency.
Game	Captain Callisto: jetpack timing	S	The instruction and jetpack implementation define burst growth, release, and restart consistency.
Game	Captain Callisto: level-completion transition	S	The instruction requires completion to persist after the approach and release sequence.
Game	Captain Callisto: cargo collection	M	The live cargo scanner supplies the route-to-credit interpretation.
Game	Captain Callisto: platform contact	S	The instruction specifies non-clipping contact and continued passage at the deck corner.
Game	Captain Callisto: lift-transition physics	S	The instruction specifies directional continuity across the moving-support handoff.
Game	Captain Callisto: level-completion transition	S	The instruction requires completion to persist after the approach and release sequence.
Game	2048: seeded restart consistency	S	The instruction specifies deterministic restart without a queued post-restart move.
Game	2048: seeded restart consistency	S	The instruction specifies deterministic restart without a queued post-restart move.
Game	Astray: glass collisions and parallax	S	The instruction specifies the full color sequence and continued movement.

Domain	Task	Type	Specification basis
Game	Core Ball: pin attachment and collision	S	The instruction specifies agreement between pin approach and attachment.
Game	Hextris: stable cascade settlement	S	The instruction specifies cascade completion under the existing match-size rule.
Game	Hextris: cascade settlement	S	The instruction specifies cascade completion under the existing match-size rule.
Game	Pac-Man: warning-state handoff	S	The instruction specifies agreement among warning, contact, and resumed route.
Game	Minesweeper: delayed-loss reset	M	The Episode integrity panel supplies the required delayed-loss timing.
Game	Signal Weave: checkpoint and hazard synchronization	M	The route card supplies the lane handoff orientation.
Game	Vector Relay: deferred-event ownership	S	The instruction specifies one delayed settlement after natural replacement.
Game	Vector Relay: pulse-balance handoff	S	The instruction distinguishes manual recall from natural replacement.
Game	Letter Arc: finish animation and replay	S	The instruction specifies an intact completed row throughout finish and replay.
Game	Boxel Rebound: restart and camera continuity	M	The application image card specifies restart phase and camera reconstruction.
Game	Boxel Rebound: restart and camera continuity	M	The application image card specifies restart phase and camera reconstruction.
Game	Boxel Rebound: restart and camera continuity	M	The application image card specifies restart phase and camera reconstruction.
Game	Boxel Rebound: restart and camera continuity	M	The application image card specifies restart phase and camera reconstruction.
Game	Boxel Rebound: restart and camera continuity	M	The application image card specifies restart phase and camera reconstruction.
Game	Flappy Bird: flight timing and contact	S	The source contains the target impulses and arcs used by the Flight Lab.
Game	Flappy Bird: flight timing and contact	S	The source contains the target impulses and arcs used by the Flight Lab.
Game	Flappy Bird: flight timing and contact	S	The source contains the target impulses and arcs used by the Flight Lab.
Game	Flappy Bird: flight timing and contact	S	The source contains the target impulses and arcs used by the Flight Lab.
Game	Minesweeper: delayed-loss reset	M	The Episode integrity panel supplies the revised delayed-loss timing.
DevOps	Async trace context propagation	M	The compatibility catalog supplies the manager-specific continuation contract.
DevOps	Cross-host trace clock correction	M	Operator notes and calibration guidance specify clock authority and correction scope.
DevOps	Alert inhibition identity	M	The deployed scope registry and operator guidance specify suppression eligibility.
DevOps	Rate-before-aggregation semantics	M	The deployed query definitions and publication lineage identify the correct input binding.
DevOps	SLO availability rollups	M	The reporting agreement specifies the required regional weighting and rollup.
DevOps	Log severity normalization	M	Receiver records and fleet conventions specify the severity interpretation.
DevOps	Tail-sampling decision windows	M	The receiver policy or runbook supplies the contractual decision-time band.
DevOps	Alert pending identity and routing	M	The receiver selector and routing policy specify the stable alert identity.
DevOps	StatsD gauge update modes	M	The endpoint provisioning or relay announce contract specifies update semantics.
DevOps	Rate-before-aggregation semantics	M	The deployed query definitions and publication lineage identify the correct input binding.
DevOps	Tail-sampling decision windows	M	The receiver policy or runbook supplies the contractual decision-time band.
DevOps	StatsD gauge update modes	M	The endpoint provisioning or relay announce contract specifies update semantics.

Domain	Task	Type	Specification basis
DevOps	Rate-before-aggregation semantics	M	The deployed query definitions and publication lineage identify the correct input binding.
DevOps	Async trace context propagation	M	The compatibility catalog supplies the manager-specific continuation contract.
DevOps	SLO availability rollups	M	The reporting agreement specifies the required regional weighting and rollup.
DevOps	Alert inhibition identity	M	The deployed scope registry and operator guidance specify suppression eligibility.
DevOps	Log severity normalization	M	Receiver records and fleet conventions specify the severity interpretation.
DevOps	Cross-host trace clock correction	M	Operator notes and calibration guidance specify clock authority and correction scope.
DevOps	Alert pending identity and routing	M	The receiver selector and routing policy specify the stable alert identity.
DevOps	Scrape response phase deadlines	M	The policy activation board and authority matrix supply the freshness and abort bounds.
Mobile	Brine & Beam: received wall-chart schedules	M	Received drawings specify job durations, dependencies, and report requirements.
Mobile	Brine & Beam: received wall-chart schedules	M	Received drawings specify job durations, dependencies, and report requirements.
Mobile	Brine & Beam: received wall-chart schedules	M	Received drawings specify job durations, dependencies, and report requirements.
Mobile	Brine & Beam: received wall-chart schedules	M	Received drawings specify job durations, dependencies, and report requirements.
Mobile	Brine & Beam: received wall-chart schedules	M	Received drawings specify job durations, dependencies, and report requirements.
Mobile	Brine & Beam: received wall-chart schedules	M	Received drawings specify job durations, dependencies, and report requirements.
Mobile	Brine & Beam: received wall-chart schedules	M	Received drawings specify job durations, dependencies, and report requirements.
Mobile	Brine & Beam: received wall-chart schedules	M	Received drawings specify job durations, dependencies, and report requirements.
Mobile	Repair imported inspection measurements	M	Imported calibration and reports specify annotation and measurement interpretation.
Mobile	Marble Post: imported runs no longer agree with office packets	M	Received track packets specify the simulation and edited-plan receipt contract.
Mobile	Repair Tideglass's received-plan preview	M	Received assembly drawings specify gear connections and motion relationships.
Mobile	Repair Mural Desk's imported projection workflow	M	Imported projection-office packets specify mapping and inverse-edit semantics.
Mobile	Northline tour review: hotspots drift from imported panoramas	M	Imported tour reports specify panorama registration and hotspot interpretation.
Mobile	Repair the received-plate integration	M	Received finder/plate pairs specify field registrations and target offsets.
Mobile	Repair Foldnote proof review	M	Imported artwork and panel reports specify placement and orientation.
Mobile	Repair Lantern's delivered show integration	M	Delivered patchboard and timing sheets specify signal and memory behavior.
Mobile	Repair the Lamp courier pickup review	M	Delivered pose and exposure boards specify the approved performance.
Mobile	Repair the pass wallet	M	Issuer records and receipts specify availability and validation semantics.
Mobile	Repair Campus Pocket's imported-campus routing	M	Imported maps and routing reports specify permitted trips and route interpretation.

E POST-TRAINING IMPLEMENTATION AND RLVR EXTENSION

E.1 SFT DATA CONSTRUCTION ACROSS DOMAINS

Verified reference replays. We use the same SFT data construction pipeline in each domain, instantiated with that domain's application environments, reference repairs, and native verifiers. For each task, a fixed reference policy executes the gold repair in a fresh environment: it starts the browser and receives a screenshot, applies the gold patch through a shell action, runs the reference validation command, waits 750 ms through the computer tool to obtain an updated screenshot, and finishes the episode. The task's native verifier then checks the resulting repair. A complete replay supplies training data only after passing verification. These scripted replays provide supervision for repair execution and its surrounding tool interactions, using screenshots returned by the running application.

Action prefixes and sampling weights. Each accepted replay yields five cumulative conversation prefixes, one for each action. A prefix contains the task instruction and the messages and observations available before its target action, followed by that action as the supervised response. This preserves the information available at each decision point. We oversample these prefixes with the fixed weights in Table 3, emphasizing the patch-application action. Thus, N accepted replays produce $5N$ distinct task/action prefix positions and $44N$ weighted training rows. Repeated rows change the sampling weight of a prefix; the underlying task and demonstration remain the same.

Table 3: **SFT prefix construction for each domain.** Each row supervises the indicated action, conditioned on the preceding conversation. Screenshot counts refer to images available before that action.

Supervised action	Copies per task	Screenshots in context
Start browser	2	0
Apply gold patch	32	1
Validate repair	4	1
Wait and observe	2	1
Finish episode	4	2
Total	44	—

Multimodal representation and target masking. Each example stores the conversation, references to its screenshot inputs, and metadata linking it to the task version, reference replay, and prefix position. The loader applies the supervised loss only to the final assistant action. Earlier assistant actions, task instructions, tool outputs, and screenshots provide conditioning context and receive no prediction loss. The browser-start prefix is text-only because its target action precedes the first screenshot; the finish prefix conditions on both observations. Before training, token-level checks verify the target mask, context length, and action-token limit for every weighted example.

E.2 SUPERVISED FINE-TUNING

Training configuration. The SFT experiment in Section 4.4 uses Qwen3.8-27B, the weighted action-prefix examples described in Appendix E.1, and 1,801 optimizer updates. Training starts from the base checkpoint with shuffled examples, per-token supervised loss, and global and rollout batch sizes of four. The vision encoder and linear-attention parameters are frozen. The learning rate decreases from 10^{-6} to 2×10^{-7} with zero warm-up, and the token budget is 16,384 per GPU. The update budget corresponds to 7,204 nominal sample slots, approximately 4.96 passes over the weighted dataset. Figure 5 includes every recorded batch loss. Its smooth curve is the trailing arithmetic mean over at most 50 updates, rather than an epoch-level average.

Evaluation. Both checkpoints are evaluated on the same set of tasks, with one greedy attempt per task and matched interaction settings. The outcome is the native task-verifier result P_i: 11.1% for the base model and 33.3% for the SFT checkpoint. The task-success metric used in the benchmark comparison also includes trajectory compliance, $S_i = P_i U_i$; these SFT results report P_i.

E.3 VERIFIABLE REWARDS AND THE TRAJECTORY-LEVEL GRPO OBJECTIVE

Verifiable rewards. For trajectory i ending with codebase c_{T_i}, the terminal repair reward is $R_i = P_i = V_u(c_{T_i})$. It is determined by the verifier, which checks the requested behavior, designated existing functionality, and restrictions on permitted changes. Scoring starts from a clean baseline, transfers permitted edits, rebuilds the application, and runs the task's tests. Changes to protected evaluator or build files are rejected; candidate dependencies and generated build outputs are not used for scoring. A compilation failure or failed requirement earns zero. A successful repair supplies supervision for the entire sequence of coding and computer use that produced it. Test progress is recorded for analysis; its optional use in reward shaping is disabled in the default configuration described below.

Group-relative learning. The RLVR extension adapts GRPO (Shao et al., 2024) to complete development episodes. For each task, the behavior policy samples G trajectories from fresh workspaces. For a group with valid scoring outcomes, define

$$\bar{R} = \frac{1}{G}\sum_{i=1}^{G} R_i, \qquad s_R = \sqrt{\frac{1}{G-1}\sum_{i=1}^{G}(R_i - \bar{R})^2}, \qquad A_i = \frac{R_i - \bar{R}}{s_R + \delta}. \tag{1}$$

Here $\delta > 0$ is a numerical stability constant. Every generated token in trajectory i receives the same advantage A_i. Groups in which all repairs succeed or all fail are retained with zero advantage.

An interleaved trajectory contains both model outputs and environment observations. Let $m_{i,k} = 1$ at generated-token positions and zero elsewhere, $L_i = \sum_k m_{i,k}$, and $H_{i,k}$ denote the multimodal prefix preceding token $z_{i,k}$. With $\rho_{i,k}(\theta) = \pi_\theta(z_{i,k} \mid H_{i,k})/\pi_{\text{old}}(z_{i,k} \mid H_{i,k})$, the policy objective is

$$J(\theta) = \mathbb{E}_{u,\,\tau_{1:G}|F=1}\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{L_i}\sum_{k} m_{i,k}\ell_{i,k}(\theta)\right], \tag{2}$$
$$\ell_{i,k}(\theta) = \min(\rho_{i,k}A_i, \text{clip}(\rho_{i,k}, 1-\epsilon_{\text{low}}, 1+\epsilon_{\text{high}})A_i)\,.$$

The hyperparameters ϵ_{low} and ϵ_{high} control the lower and upper clipping bounds on the probability ratio. Here $F = 1$ means that all trajectories in the group have usable scoring outcomes. The old-policy probabilities are computed by the training backend. Both KL regularization and entropy bonuses are disabled. The mask excludes observations from the prediction loss while retaining them as conditioning context. Per-trajectory normalization by L_i prevents longer responses from receiving more weight solely because they contain more generated tokens.

This adaptation changes the unit of sampling from a standalone answer to a development episode, replaces answer correctness with executable repair verification, and restricts the loss to generated tokens across all turns. We implement the objective using SLIME (SLIME Contributors, 2026), coupling its policy updates to the coding and computer-use environment. Worker failures require separate treatment: a failed attempt is retried from a fresh workspace, and if retries are exhausted, all trajectories sampled for that task prompt are excluded and replaced before normalization. Such failures have no model reward. Evaluation reports a failure status after retry exhaustion.

Training procedure. Each iteration samples a batch of task prompts, executes G development episodes per prompt, computes verifier rewards and relative advantages, and updates the policy. Updated weights are synchronized to the rollout engine for the next iteration. Appendix E.4 describes the rollout and update interface. This procedure defines the RLVR extension; the completed post-training experiment in Section 4.4 uses SFT.

E.4 ROLLOUT AND UPDATE INTERFACE

Parallel development episodes. The implementation connects a persistent browser and editable workspace to SLIME's multimodal rollout interface. Each attempt receives fresh files and application ports, allowing independent development episodes to run in parallel. The policy's generated tokens, their sampling log probabilities, image inputs, and observation masks are preserved in one interleaved sample. The policy conditions on the sequence of observations and actions up to its context limit. SGLang supplies inference and the Megatron backend supplies policy updates, with weights synchronized through SLIME's training loop. Figure 9 illustrates this interface.

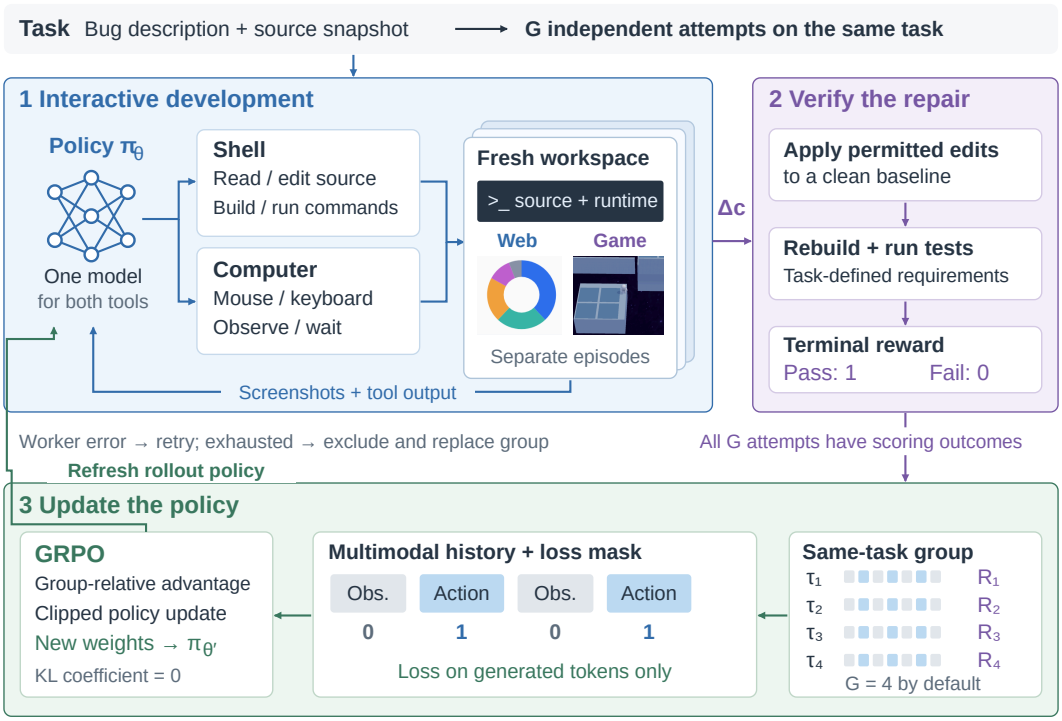


Figure 9: **RLVR extension using verifiable software repairs.** For the same task, the policy samples several development trajectories in separate workspaces. Shell commands and computer actions return text and screenshots. At termination, permitted source changes are transferred to a clean scoring copy and checked by deterministic tests. Complete, successfully scored groups provide relative advantages for a policy update; observations condition the policy but receive no direct prediction loss. Application thumbnails show example Web and Game applications.

Verifiable feedback. The scoring copy rejects protected-file and symlink edits and retains baseline dependencies and build configuration. Baseline fingerprints detect mutation outside the candidate workspace. Reset, browser, inference, and verifier-runtime failures are distinguished from unsuccessful repairs; unresolved infrastructure failures are excluded before computing group advantages. The default reward is binary repair correctness, with no intermediate rewards. Generated tokens share the episode-level advantage, while environment observations condition the policy without receiving prediction loss.

Scaling RLVR. Independent rollouts and executable scoring permit additional workers to collect verifiable experience without manually labeling each attempted repair. Scaling this process to long hybrid-CUA trajectories requires balancing rollout throughput, optimization stability, and credit assignment across coding and interaction. Scalable RLVR is immediate future work; the completed experiment reported here is supervised fine-tuning.

F REPEATED-ATTEMPT EVALUATION

Table 4 measures the coverage gained from two additional attempts relative to the first-attempt results in Table 8. Each attempt starts from a fresh workspace under the domain's fixed task, access, and verification settings.

Additional attempts expand task coverage. GPT-6-Astra reaches pass@3 of 75.0% on Web, 62.1% on Game, 90.0% on DevOps, and 60.0% on Mobile. On Game, its cumulative pass@1, pass@2, and pass@3 are 37.9%, 58.6%, and 62.1%, respectively: 11 tasks are solved in the first attempt, 17 in at least one of the first two attempts, and 18 in at least one of all three attempts. Appendix A.5 describes the repeated evaluation's cohort composition.

Stable competence and additional-attempt coverage. Repeated evaluation distinguishes tasks an agent completes consistently from those reached through another development trajectory. In the three-attempt Game study, GPT-6-Astra solves 34.5% of tasks in all three attempts, compared with 3.4% for GPT-5.6-Sol. Ten of its 18 solved tasks succeed in every attempt; the remaining eight succeed in one or two attempts. On Mobile, an additional attempt lets GPT-5.6-Sol complete the sprite task and GPT-6-Astra complete one schedule variant; the remaining schedule variants continue to distinguish this family from the broadly solved application tasks. Thus additional attempts expand coverage while the task-level success patterns identify capabilities requiring more reliable execution. Appendix F reports all rates and uncertainty estimates within the repeated evaluation.

Task coverage and sampling uncertainty. We assess task-sampling uncertainty on Web and Game with 20,000 paired bootstrap resamples, preserving each task's outcomes across models and attempts. On Game, GPT-6-Astra's pass@3 advantage over GPT-5.6-Sol is 31.0 points, with a 95% interval of [13.8, 48.3]. Resampling whole game families gives an interval of [11.5, 50.0], supporting the same ordering under variation in the composition of source games. Appendix F gives both model comparisons, the pass@k profiles, and the resampling methods.

Table 4: **Three-attempt hybrid CUA success (%).** Each cell reports pass@1 / pass@3. Pass@1 uses the first attempt; pass@3 uses all three.

Domain	GPT-6-Astra	GPT-5.6-Sol	Claude-Fable-5
Web	**66.7 / 75.0**	58.3 / 58.3	47.2 / 61.1
Game	**37.9 / 62.1**	10.3 / 31.0	17.2 / 41.4
DevOps	**80.0 / 90.0**	55.0 / 60.0	55.0 / 70.0
Mobile	**55.0 / 60.0**	45.0 / 50.0	40.0 / 40.0

F.1 METRICS AND AGGREGATION

We evaluate GPT-6-Astra, GPT-5.6-Sol, and Claude-Fable-5 with three attempts per task across Web, Game, DevOps, and Mobile. For task i, let $s_{ij} \in \{0, 1\}$ denote success in attempt j, in the recorded execution order. For a domain with N tasks, cumulative success is $\text{pass@}k = 100N^{-1}\sum_i \mathbb{1}[\sum_{j=1}^{k} s_{ij} > 0]$, for $k \in \{1, 2, 3\}$. A task remains covered once any of its first k attempts succeeds. Pass@1 therefore matches the single-attempt comparison, and pass@3 counts tasks solved at least once across all three attempts. Table 5 reports this cumulative profile for Web and Game, where ordered task-level outcomes are available. Appendix A.5 describes the execution protocol and attempt composition. Four-domain means average the unrounded domain rates equally. Web and Game statistics use the recorded ordered task-level outcomes. DevOps and Mobile pass@1 and pass@3 use the recorded aggregate outcomes. We compute task-level bootstrap intervals for Web and Game.

The task-level success profiles and success-frequency distributions below describe Web and Game. The frequency distributions give the percentage of tasks solved in zero, one, two, or all three attempts, distinguishing consistently solved tasks from those reached through additional attempts.

Table 5: **Task success across attempt budgets (%).** Mean of three is the average single-attempt success within this repeated cohort. Pass@2 averages all two-attempt subsets; pass@3 uses all three. The difference is pass@3 minus mean single-attempt success, in percentage points. Results cover Web and Game.

Domain	Model	Mean of three	Pass@2	Pass@3	Diff. (pp)
Web	GPT-6-Astra	67.6	72.2	75.0	+7.4
	GPT-5.6-Sol	52.8	57.4	58.3	+5.6
	Claude-Fable-5	52.8	59.3	61.1	+8.3
Game	GPT-6-Astra	50.6	59.8	62.1	+11.5
	GPT-5.6-Sol	20.7	29.9	31.0	+10.3
	Claude-Fable-5	27.6	36.8	41.4	+13.8

Table 6: **Success frequency across three attempts.** Entries give the percentage of tasks with zero, one, two, or three successful attempts. Each row sums to 100% before rounding. Results cover Web and Game.

Domain	Model	0 successes	1 success	2 successes	3 successes
Web	GPT-6-Astra	25.0	8.3	5.6	61.1
	GPT-5.6-Sol	41.7	2.8	11.1	44.4
	Claude-Fable-5	38.9	5.6	13.9	41.7
Game	GPT-6-Astra	37.9	6.9	20.7	34.5
	GPT-5.6-Sol	69.0	3.4	24.1	3.4
	Claude-Fable-5	58.6	13.8	13.8	13.8

F.2 TASK-SAMPLING UNCERTAINTY

For the paired model comparisons in Table 7, we also report the mean success rate across all three attempts, $\overline{S} = 100(3N)^{-1} \sum_i \sum_{j=1}^{3} s_{ij}$. For statistical intervals on Web and Game, we resample tasks within each domain, drawing the domain's original number of tasks with replacement. Each sampled task carries the outcomes of all three models and all three attempts, preserving the pairing in model comparisons. We recompute the three-attempt mean $\overline{S}$, cumulative pass@3, and the paired model differences in each of 20,000 bootstrap replicates, and use the 2.5th and 97.5th percentiles as the 95% interval. These intervals quantify variation under resampling of the evaluated tasks.

We also perform a family-level sensitivity analysis, using nine taxonomy-defined task families on Web and twelve source games on Game. Each replicate resamples the domain's original number of groups and includes every task from each sampled group. Rates retain task weighting, with the total number of sampled tasks providing the replicate's normalization. All model and attempt outcomes remain paired. This analysis preserves dependence among related tasks while measuring sensitivity to the benchmark's composition of task families and source games.

Table 7: **Paired task-success differences for GPT-6-Astra (pp).** Intervals are 95% percentile intervals from 20,000 paired resamples. Task resampling operates within each domain; family resampling selects whole task families and retains task weighting. All model outcomes and three attempts for each task remain together. Results cover Web and Game.

Domain	Compared with	Measure	Difference	Task 95% CI	Family 95% CI
Web	GPT-5.6-Sol	Mean of three	+14.8	[3.7, 25.9]	[-1.3, 33.3]
	GPT-5.6-Sol	Pass@3	+16.7	[2.8, 30.6]	[-6.7, 42.9]
	Claude-Fable-5	Mean of three	+14.8	[3.7, 25.9]	[0.0, 32.5]
	Claude-Fable-5	Pass@3	+13.9	[0.0, 27.8]	[-4.3, 33.3]
Game	GPT-5.6-Sol	Mean of three	+29.9	[17.2, 43.7]	[15.3, 47.2]
	GPT-5.6-Sol	Pass@3	+31.0	[13.8, 48.3]	[11.5, 50.0]
	Claude-Fable-5	Mean of three	+23.0	[8.0, 39.1]	[10.0, 42.0]
	Claude-Fable-5	Pass@3	+20.7	[0.0, 41.4]	[0.0, 41.4]

G DOMAIN-LEVEL CUA PERFORMANCE

We summarize CUA performance across Web, Game, DevOps, and Mobile, giving each domain equal weight. GPT-6-Astra achieves a mean task success rate of 59.9%, followed by GPT-5.6-Sol at 42.2%, Grok-4.6 and Codex with GPT-5.6-Sol at 41.9%, and Claude-Fable-5 at 39.9%. Table 8 provides the complete domain-level results.

Domain and family strengths. GPT-6-Astra achieves the highest observed CUA task success rate in each domain: 66.7% on Web, 37.9% on Game, 80.0% on DevOps, and 55.0% on Mobile (Appendix G). Its Web advantage over GPT-5.6-Sol is concentrated in Canvas interactions after geometry changes, while its Mobile results distinguish precise gear relationships and sprite pose assignments (Figure 13). Across Game, the union of successful CUA repairs covers 58.6% of tasks.

Complementary strengths among frontier agents. Individual tasks reveal capabilities that the aggregate ranking compresses. Claude-Opus-4.8 uniquely solves a DevOps alert-routing task by preserving the labels consumed by receiver selectors. Grok-4.6 and Codex with GPT-5.6-Sol solve a log-normalization task by combining transport format with the original producer's severity convention. GPT-6-Astra misses both tasks. These contrasts expose distinct demands on identifying which application facts determine a correct repair.

Complete agent systems. Codex with GPT-5.6-Sol reaches 55.6% on Web, 17.2% on Game, 55.0% on DevOps, and 40.0% on Mobile. Claude Code with Claude-Opus-5 reaches 55.6%, 20.7%, and 65.0% on Web, Game, and DevOps. Relative to the corresponding frontier models, Codex's Game rate is 6.9 percentage points higher and Claude Code's DevOps rate is 5.0 points higher. These comparisons characterize the combination of model, tools, and execution strategy.

Table 8: **Single-attempt task success (%) across four domains.** Code denotes code-only access; hybrid CUA adds screenshots and graphical actions. Teal marks column maxima; an em dash indicates no reported result.

	Web		Game		DevOps		Mobile	
Model / system	Code	Hybrid CUA	Code	Hybrid CUA	Code	Hybrid CUA	Code	Hybrid CUA
GPT-6-Astra	27.8	**66.7**	**17.2**	**37.9**	0.0	**80.0**	0.0	**55.0**
GPT-5.6-Sol	19.4	58.3	10.3	10.3	0.0	55.0	0.0	45.0
GPT-5.6-Luna	11.1	5.6	0.0	6.9	0.0	15.0	0.0	35.0
GPT-5.6-Terra	11.1	13.9	3.4	6.9	0.0	25.0	0.0	35.0
Claude-Opus-5	22.2	55.6	13.8	20.7	0.0	60.0	—	—
Claude-Opus-4.8	16.7	27.8	3.4	20.7	0.0	55.0	0.0	40.0
Claude-Sonnet-5	8.3	0.0	6.9	3.4	0.0	40.0	0.0	35.0
Claude-Fable-5	19.4	47.2	13.8	17.2	**5.0**	55.0	**5.0**	40.0
Grok-4.6	**30.6**	55.6	10.3	17.2	0.0	65.0	0.0	30.0
Codex + GPT-5.6-Sol	—	55.6	—	17.2	—	55.0	—	40.0
Claude Code + Claude-Opus-5	—	55.6	—	20.7	—	65.0	—	—

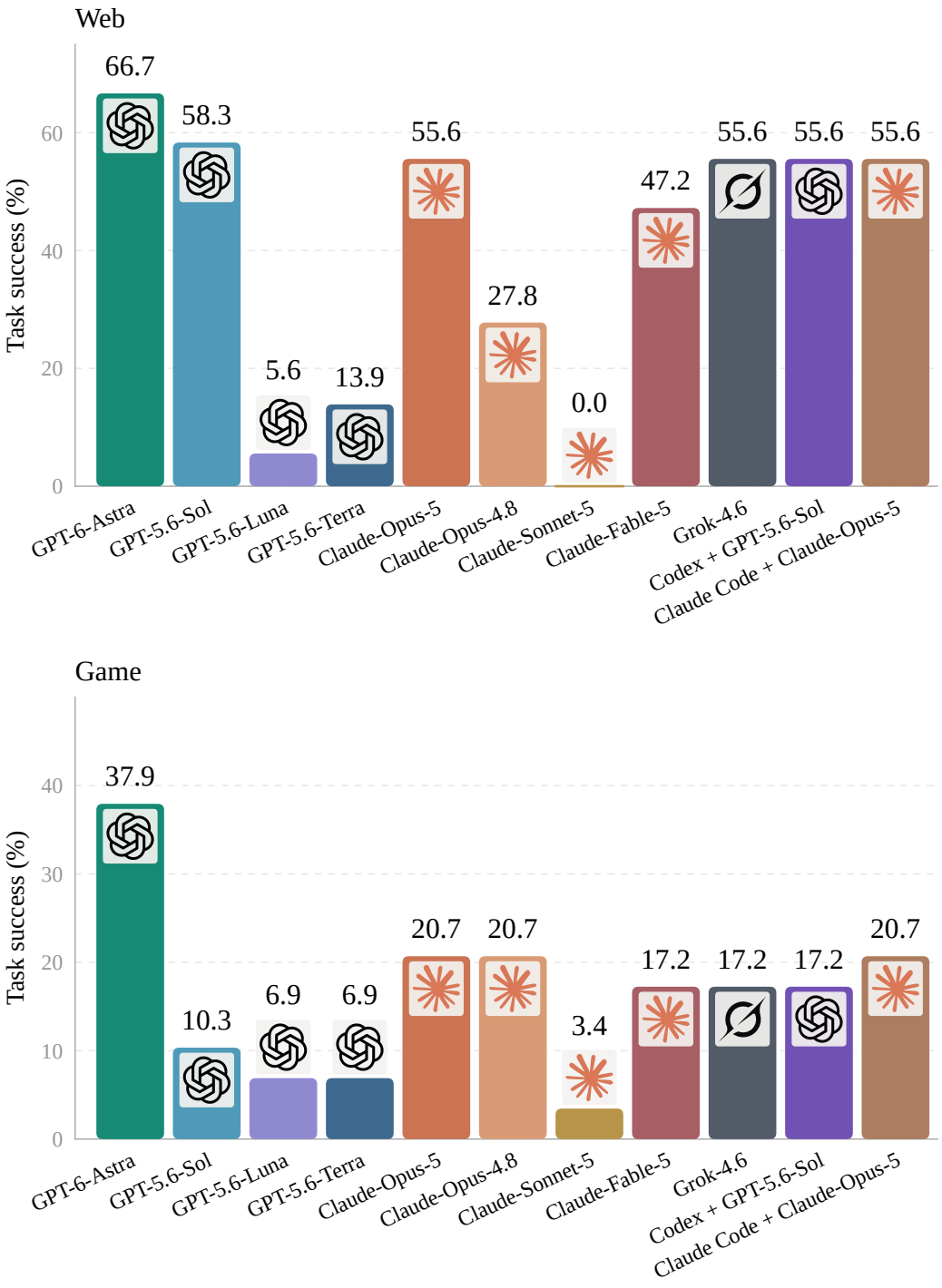


Figure 10: **CUA task success on Web (top) and Game (bottom).** Bars report success rates for nine frontier models and two agent systems over the full task inventories: 36 tasks on Web and 29 on Game.

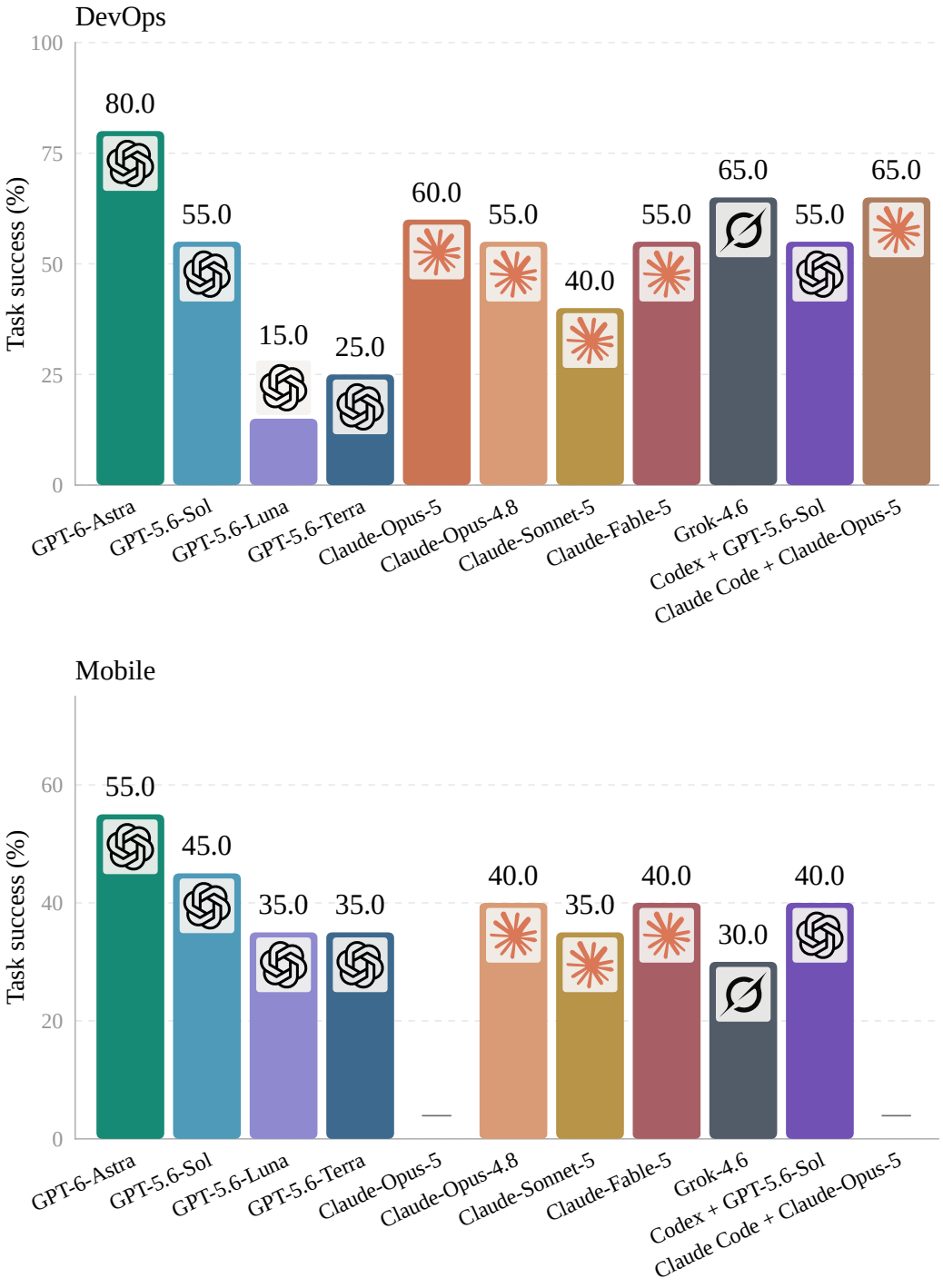


Figure 11: **CUA task success on DevOps (top) and Mobile (bottom).** Bars report CUA task success rates for the evaluated models and agent systems. Colors follow Figure 10.

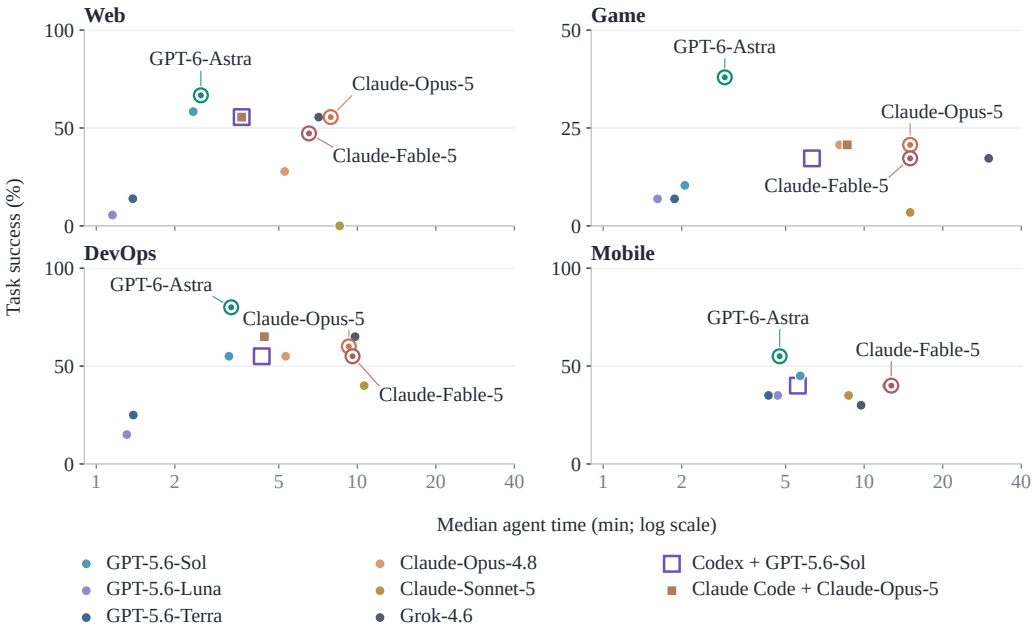


Figure 12: **Task success and median development time within each domain.** Points pair task success (vertical axis) with median agent time (horizontal axis, logarithmic). Resource coverage and aggregation are specified in Appendix A.6. Colored rings and direct labels highlight GPT-6-Astra, Claude-Opus-5, and Claude-Fable-5; the shared legend names the other entries. Circles denote frontier models; open and filled squares denote Codex and Claude Code, respectively, with nested squares for the nearly coincident Web estimates. Success axes span 0–50% for Game and 0–100% elsewhere.

H TASK-FAMILY PERFORMANCE

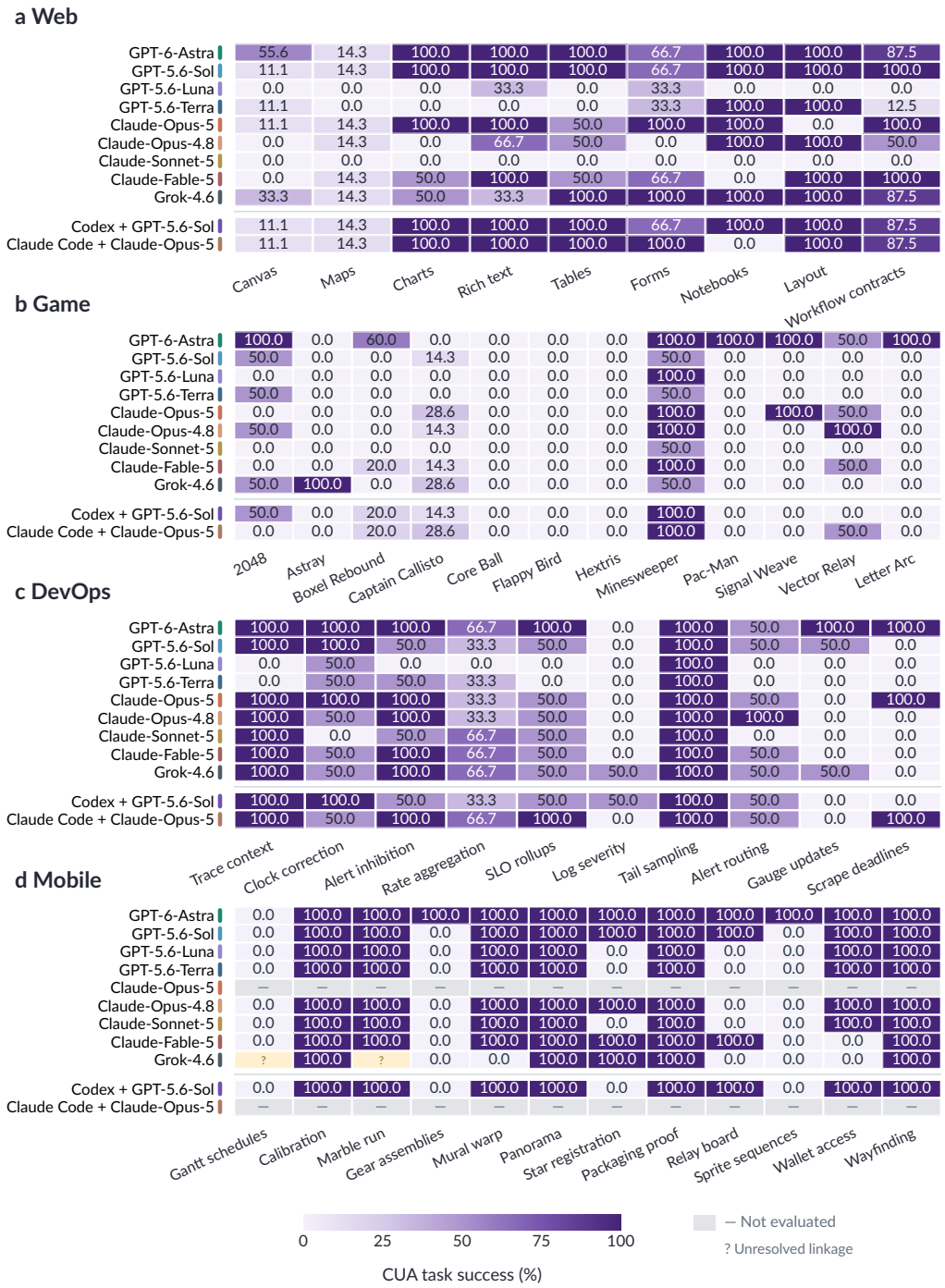


Figure 13: **CUA task success across task families.** Cells report task success rates within each family, with purple intensity encoding the percentage. Web families group applications and tools; Game families retain source-game identity; DevOps and Mobile follow task-contract categories. Rows below the separator show agent systems. Rates use each family's full task inventory. Em dashes denote unevaluated models. Question marks denote Grok-4.6's two Mobile family estimates with unresolved task outcomes: 0–11.1% for Gantt schedules and 0–100% for Marble run (Appendix A.5).

I TASK-LEVEL OUTCOMES AND PAIRED DIFFERENCES

Figure 1 compares development with source-level execution against development that additionally uses application screenshots and GUI actions. Both conditions provide feedback from nonvisual commands; hybrid CUA also exposes the assembled application's visual state and materials and supports interactive checks of the effects of code changes.

Connecting visual evidence to the repair. The additional successes reveal two ways application access supports software development. It exposes material that determines the required implementation, such as received drawings and displayed service behavior. It also lets agents exercise a changed application and inspect the outcome: GPT-6-Astra's additional Game successes include delayed terminal state, checkpoint synchronization, and camera behavior after restart. In Web, its nested-selection trajectory uses a drag after resizing to discover and repair a second interaction defect. These cases connect the access comparison to concrete changes in the development process.

Web and Game. On Web, GPT-5.6-Sol achieves 19.4% task success with code-only access and 58.3% with hybrid access; Claude-Opus-5 achieves 22.2% and 55.6%, respectively. On Game, the corresponding rates are 3.4% and 20.7% for Claude-Opus-4.8, and 17.2% and 37.9% for GPT-6-Astra.

DevOps and Mobile. Agents inspect service interfaces and visual materials, use graphical controls to explore their behavior, and translate these observations into code changes. On DevOps, the nine frontier models average 50.0% with hybrid access and 0.6% with code alone. On Mobile, GPT-5.6-Sol achieves 0.0% with code-only access and 45.0% with hybrid access; GPT-6-Astra achieves 0.0% and 55.0%, respectively. Figure 18 and Appendix I provide task-level comparisons.

Performance by task information requirements. Application-material-dependent tasks account for 14 of 36 Web tasks, 9 of 29 Game tasks, and all 20 tasks in each of DevOps and Mobile. Across the eight frontier models evaluated in both conditions in all four domains, the mean hybrid advantage on these tasks is 42.0, 23.6, 48.1, and 38.8 percentage points, respectively. On source-specified tasks, the corresponding mean differences are 0.0 points on Web and -0.6 on Game. This distribution connects the aggregate comparison to the tasks' information requirements: the largest gains occur when agents must turn application materials into working software. Appendix I reports all model-level comparisons on matched task subsets. For source-specified tasks, code-only agents can check behavior through program execution, as the 2048 trajectory in Appendix C illustrates. GUI access adds application-level evidence; completing the repair requires the agent to interpret that evidence and act on it.

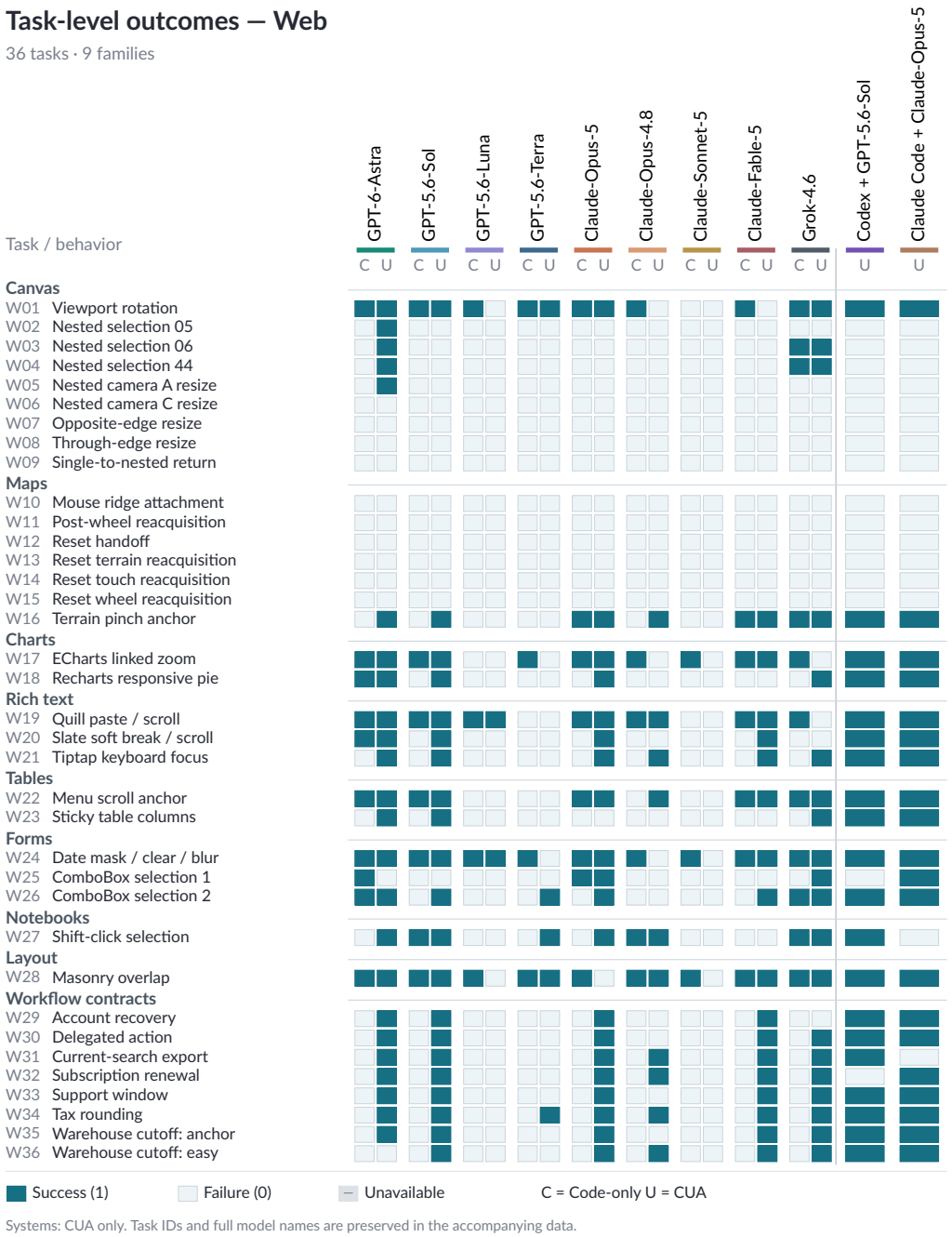


Figure 14: **Task-level Web outcomes.** Rows cover all 36 Web tasks, grouped by task family. Each frontier model has adjacent code-only (C) and CUA (U) cells; each agent system has one CUA cell. Colors encode the recorded outcome according to the key, and gray em dashes indicate unavailable outcomes.

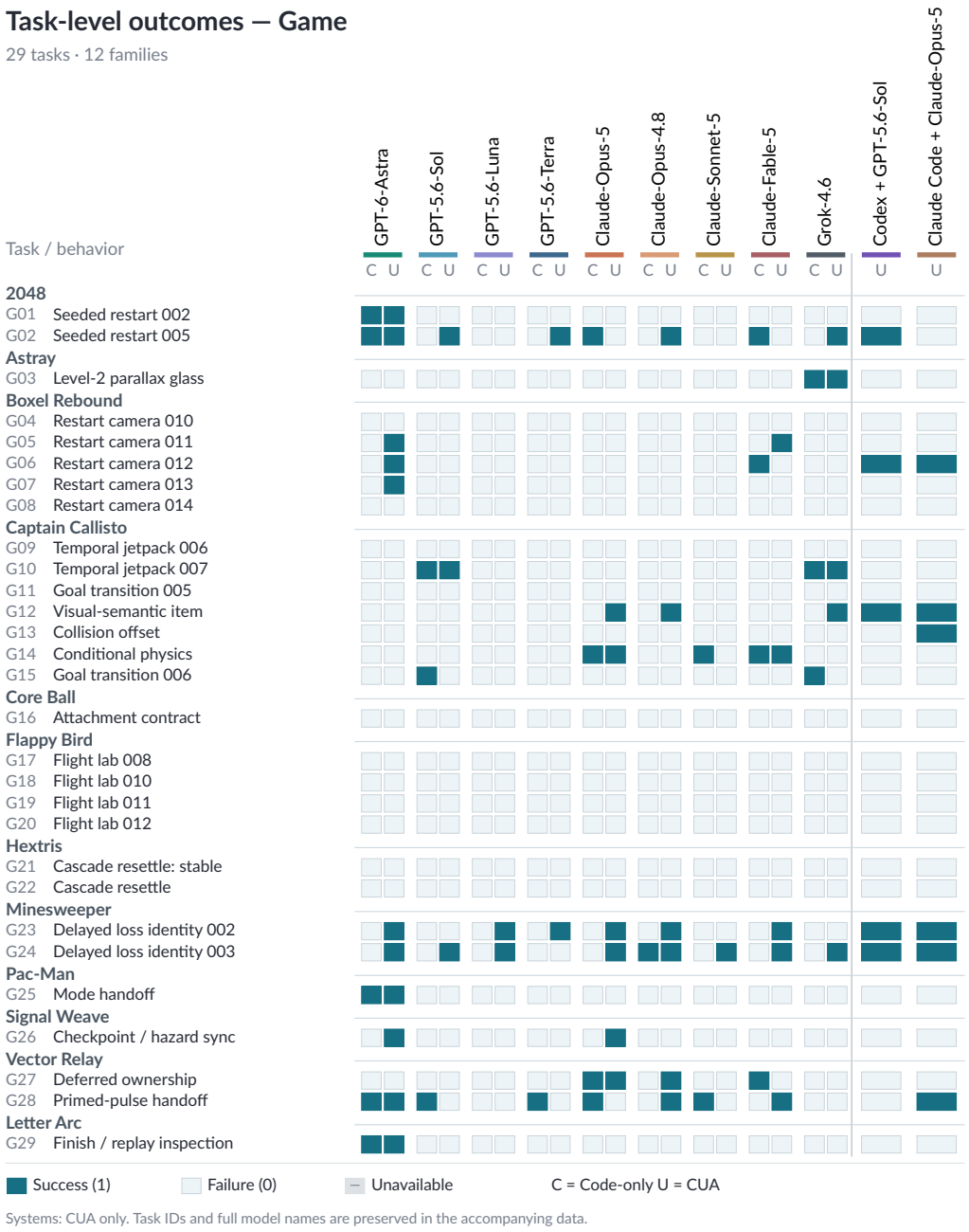


Figure 15: **Task-level Game outcomes.** Rows cover all 29 Game tasks, grouped by source game. Adjacent C and U cells show code-only and CUA outcomes for frontier models; the two system columns show CUA outcomes. The encoding follows Figure 14.

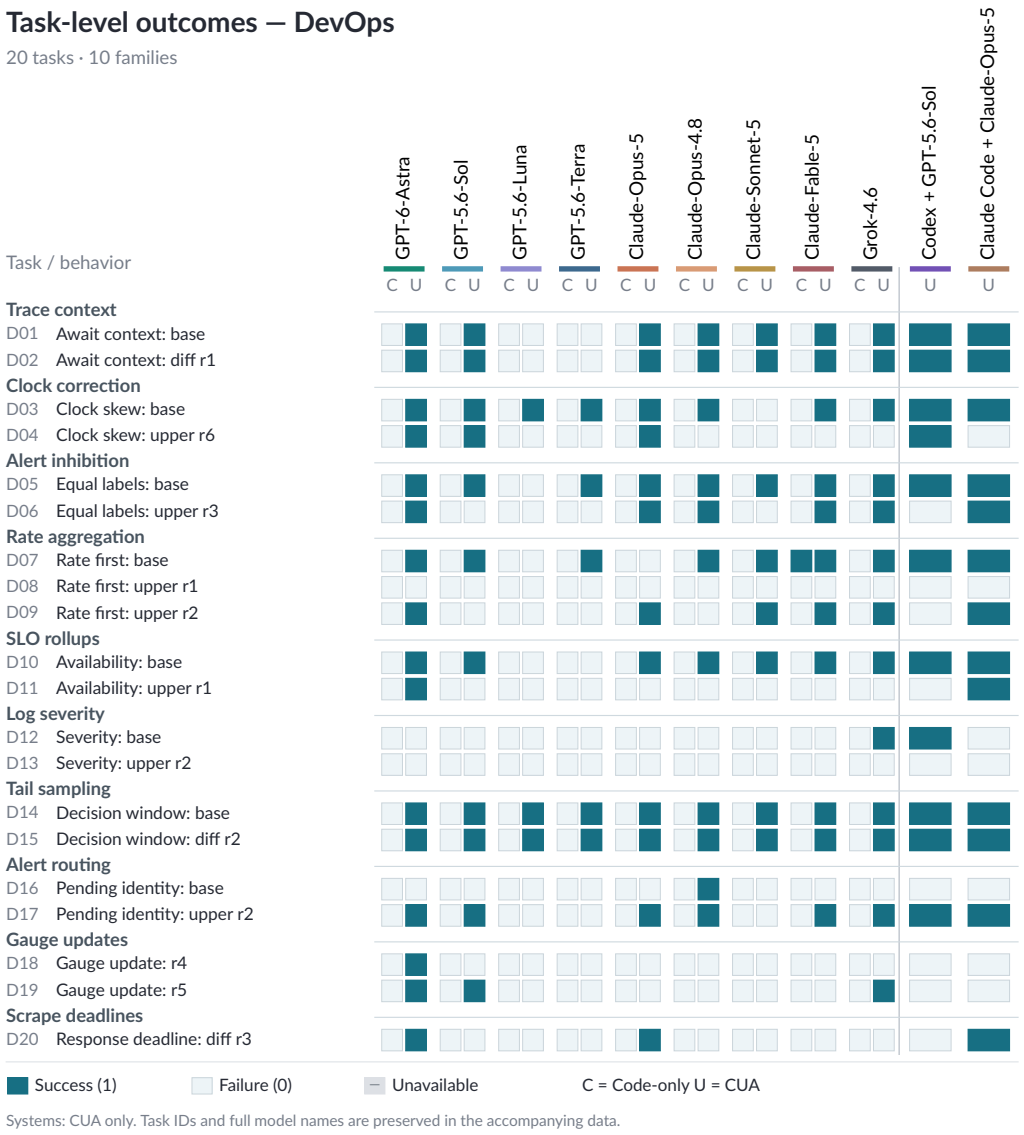


Figure 16: **Task-level DevOps outcomes.** Rows cover all 20 DevOps tasks, grouped by operational contract. Adjacent C and U cells show code-only and CUA outcomes for frontier models; the two system columns show CUA outcomes. The encoding follows Figure 14.

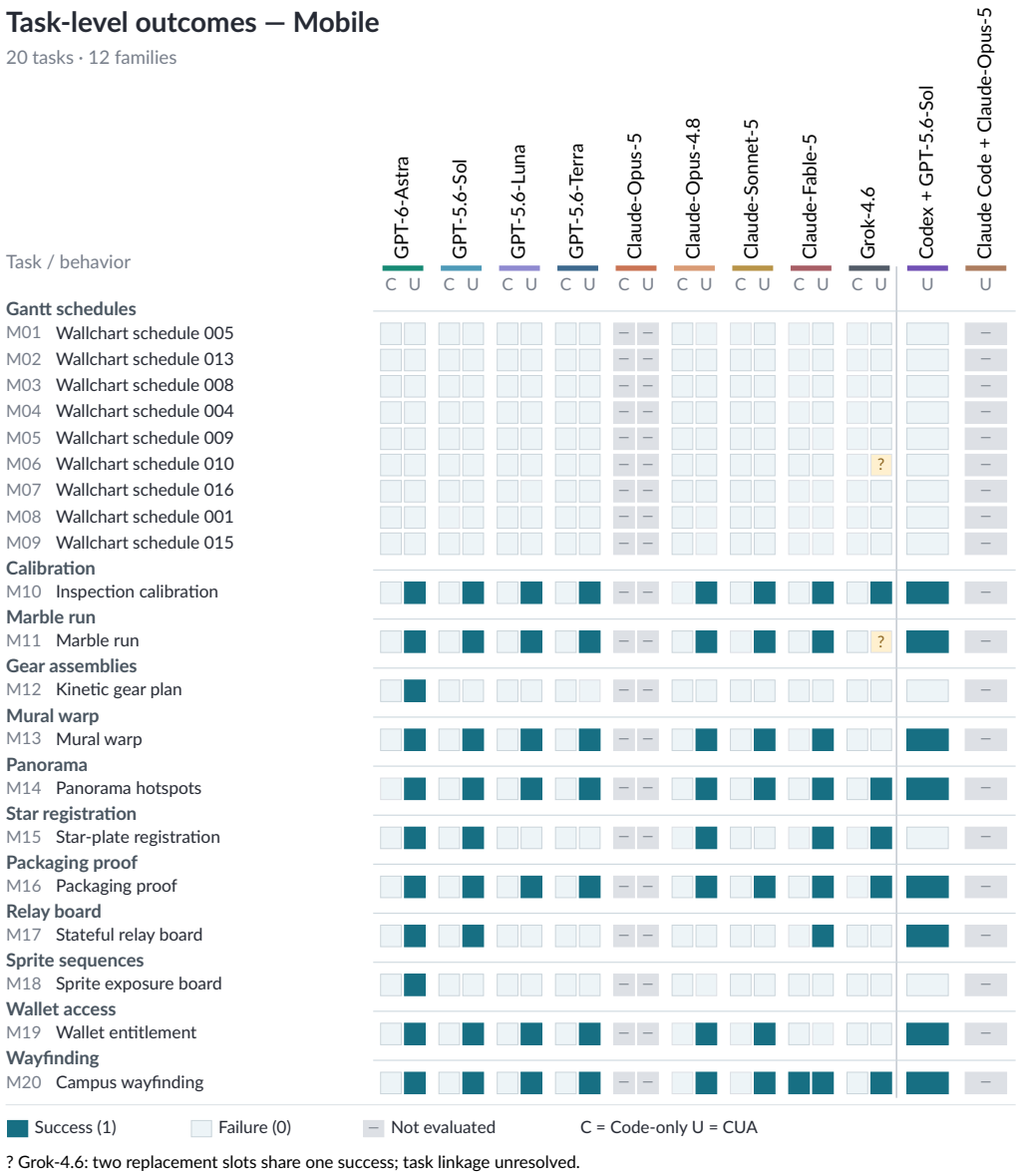


Figure 17: **Task-level Mobile outcomes.** Rows cover all 20 Mobile tasks, grouped by application contract. Adjacent C and U cells show code-only and CUA outcomes for frontier models; the two system columns show CUA outcomes. Gray em dashes denote unevaluated models. Grok-4.6's two question marks identify tasks with one success between them and unresolved individual outcomes (Appendix A.5). Other colors follow Figure 14.

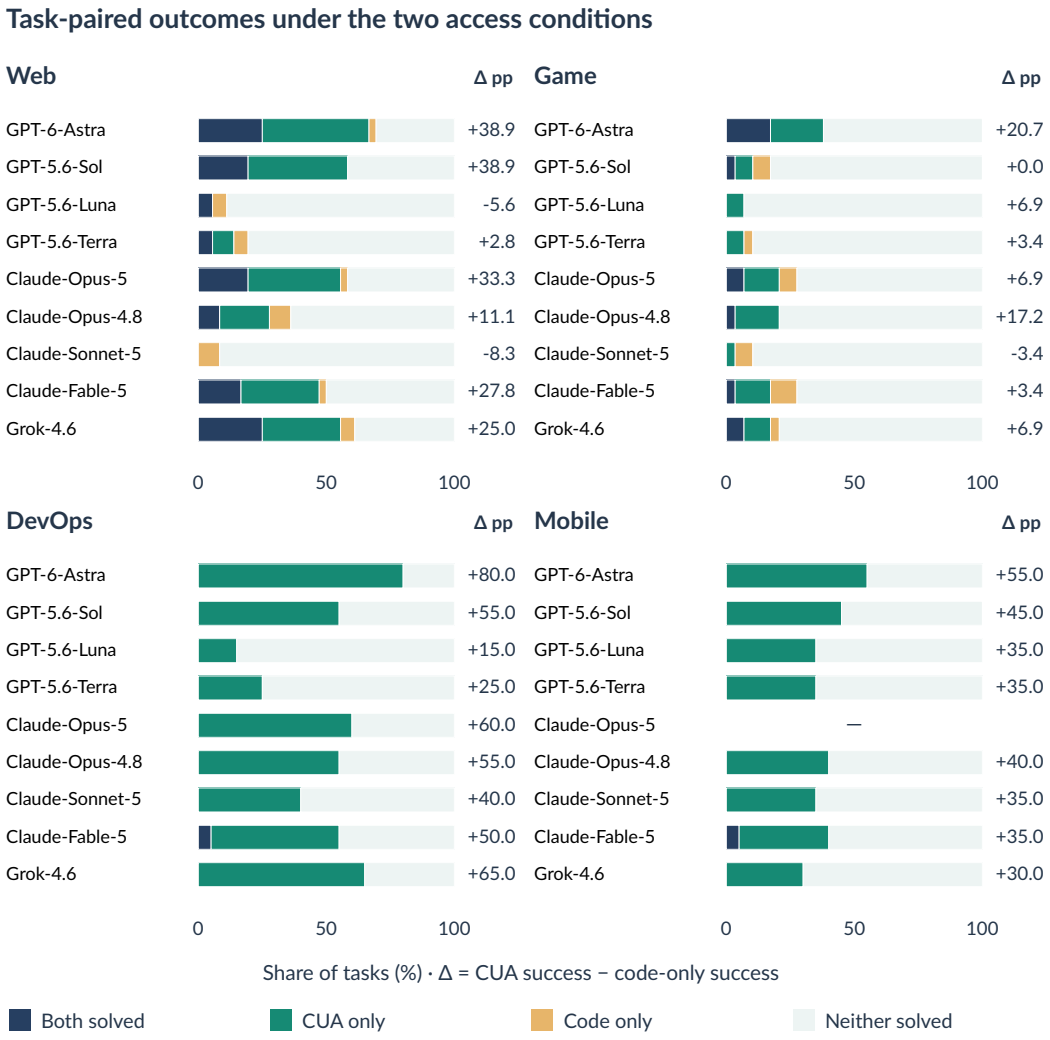


Figure 18: **Task-paired outcomes under the two access conditions.** Each bar partitions a model's matched tasks by the joint code-only and CUA outcome. The four segments sum to 100%. Values at right give the CUA minus code-only success difference in percentage points, computed as the share solved only with CUA minus the share solved only with code. Task membership is identical across conditions within each model and domain. Both conditions include source-level execution and command feedback; hybrid CUA additionally provides application screenshots and graphical actions. Information-requirement strata are reported in Table 10.

Paired outcomes by task information requirements. Tables 9 and 10 stratify the original single-attempt comparison by specification provenance. The aggregate summary uses the same eight frontier models in every domain; the detailed table includes every frontier model evaluated in both conditions in the relevant domain. All rates within a row use exactly the same tasks. The M group includes five Game tasks with bundled graphical reference cards; its other 58 tasks use runtime-supplied materials. These comparisons describe performance across task requirements under the two access conditions.

Table 9: Success by task information requirement, averaged over the eight frontier models evaluated under both access conditions in all four domains. Each domain–type group uses the same tasks for both conditions. M: application-material dependent; S: source-specified. The labels identify specification provenance; both groups require implementation and verification. Full model results appear in Appendix I.

Domain	Type	Tasks	Code-only	Hybrid CUA	Δ (pp)
Web	S	22	23.9	23.9	+0.0
Web	M	14	8.9	50.9	+42.0
Game	S	20	10.6	10.0	-0.6
Game	M	9	2.8	26.4	+23.6
DevOps	M	20	0.6	48.8	+48.1
Mobile	M	20	0.6	39.4	+38.8

Table 10: Task success by information requirement (%). Each row compares the same tasks and original single attempts under code-only and hybrid CUA access. M and S follow Appendix A.3; Δ is hybrid minus code-only in percentage points. DevOps and Mobile contain no S tasks; their S results are not applicable. Only models evaluated under both access conditions are included.

Type	Model	N	Code-only	Hybrid CUA	Δ
Web					
S	GPT-6-Astra	22	27.3	54.5	+27.3
S	GPT-5.6-Sol	22	27.3	36.4	+9.1
S	GPT-5.6-Luna	22	13.6	4.5	-9.1
S	GPT-5.6-Terra	22	18.2	13.6	-4.5
S	Claude-Opus-5	22	27.3	31.8	+4.5
S	Claude-Opus-4.8	22	22.7	18.2	-4.5
S	Claude-Sonnet-5	22	13.6	0.0	-13.6
S	Claude-Fable-5	22	27.3	27.3	+0.0
S	Grok-4.6	22	40.9	36.4	-4.5
M	GPT-6-Astra	14	28.6	85.7	+57.1
M	GPT-5.6-Sol	14	7.1	92.9	+85.7
M	GPT-5.6-Luna	14	7.1	7.1	+0.0
M	GPT-5.6-Terra	14	0.0	14.3	+14.3
M	Claude-Opus-5	14	14.3	92.9	+78.6
M	Claude-Opus-4.8	14	7.1	42.9	+35.7
M	Claude-Sonnet-5	14	0.0	0.0	+0.0
M	Claude-Fable-5	14	7.1	78.6	+71.4
M	Grok-4.6	14	14.3	85.7	+71.4
Game					
S	GPT-6-Astra	20	25.0	25.0	+0.0
S	GPT-5.6-Sol	20	15.0	10.0	-5.0
S	GPT-5.6-Luna	20	0.0	0.0	+0.0
S	GPT-5.6-Terra	20	5.0	5.0	+0.0
S	Claude-Opus-5	20	20.0	10.0	-10.0
S	Claude-Opus-4.8	20	0.0	15.0	+15.0
S	Claude-Sonnet-5	20	10.0	0.0	-10.0
S	Claude-Fable-5	20	15.0	10.0	-5.0
S	Grok-4.6	20	15.0	15.0	+0.0
M	GPT-6-Astra	9	0.0	66.7	+66.7
M	GPT-5.6-Sol	9	0.0	11.1	+11.1
M	GPT-5.6-Luna	9	0.0	22.2	+22.2

Type	Model	N	Code-only	Hybrid CUA	Δ
M	GPT-5.6-Terra	9	0.0	11.1	+11.1
M	Claude-Opus-5	9	0.0	44.4	+44.4
M	Claude-Opus-4.8	9	11.1	33.3	+22.2
M	Claude-Sonnet-5	9	0.0	11.1	+11.1
M	Claude-Fable-5	9	11.1	33.3	+22.2
M	Grok-4.6	9	0.0	22.2	+22.2
DevOps					
M	GPT-6-Astra	20	0.0	80.0	+80.0
M	GPT-5.6-Sol	20	0.0	55.0	+55.0
M	GPT-5.6-Luna	20	0.0	15.0	+15.0
M	GPT-5.6-Terra	20	0.0	25.0	+25.0
M	Claude-Opus-5	20	0.0	60.0	+60.0
M	Claude-Opus-4.8	20	0.0	55.0	+55.0
M	Claude-Sonnet-5	20	0.0	40.0	+40.0
M	Claude-Fable-5	20	5.0	55.0	+50.0
M	Grok-4.6	20	0.0	65.0	+65.0
Mobile					
M	GPT-6-Astra	20	0.0	55.0	+55.0
M	GPT-5.6-Sol	20	0.0	45.0	+45.0
M	GPT-5.6-Luna	20	0.0	35.0	+35.0
M	GPT-5.6-Terra	20	0.0	35.0	+35.0
M	Claude-Opus-4.8	20	0.0	40.0	+40.0
M	Claude-Sonnet-5	20	0.0	35.0	+35.0
M	Claude-Fable-5	20	5.0	40.0	+35.0
M	Grok-4.6	20	0.0	30.0	+30.0

J INTERACTION WORKLOADS

Time and workload measures. Time is measured overthe agent's development episode. Agent time per success divides mean episode time by task-success probability, using equal domain weights for cross-domain summaries. Appendix A describes the evaluation protocol, condition comparisons, and workload accounting.

GPT-6-Astra achieves the highest four-domain task success rate, 59.9%, and the lowest agent time per success, 7.2 minutes, among the eight frontier models (Figure 19). Its Hybrid episodes average 46.3 model responses with 145.8 output tokens per response. GPT-5.6-Sol uses 37.7 responses and 203.7 tokens per response, with 8.7 agent minutes per success. These profiles characterize how each model distributes its work across interaction rounds and response length.

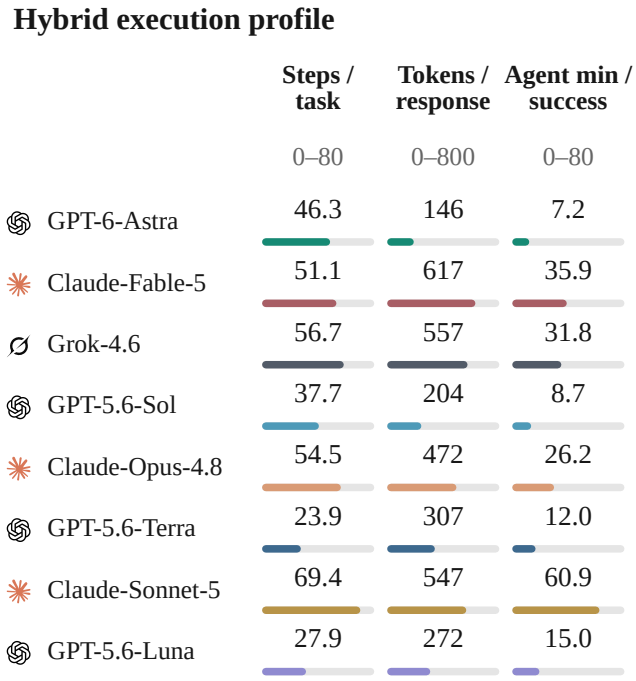


Figure 19: **Hybrid execution efficiency.** Model responses per task, output tokens per response, and agent minutes per success for eight frontier models. Episode workloads are averaged within each domain and then weighted equally across the four domains. Agent minutes per success divides that mean cost by the full-inventory task-success probability. Appendix A.6 specifies workload coverage.

Success and episode time. Figure 4(a) summarizes task success and mean development time across four equally weighted domains. GPT-6-Astra achieves 59.9% task success with a mean of 4.3 minutes per attempt; GPT-5.6-Sol reaches 42.2% at 3.7 minutes. The comparison covers nine models and agent systems evaluated in all four domains. Appendix G provides the domain-level profiles.

Effort on shared successful tasks. On Web tasks successfully repaired by both models, GPT-6-Astra uses fewer responses on 83.3% of tasks shared with Grok-4.6 and 72.2% shared with Claude-Opus-5 (Figure 4(b)). Median response counts are 28.5 versus 43.5 and 21.5 versus 27.5, respectively. Each comparison uses its own set of 18 shared successful tasks, linking interaction effort to the same completed repairs. Appendix K gives the task-level pairs and resource comparisons across domains. System-level results also show different development profiles: Claude Code with Claude-Opus-5 achieves 65.0% on DevOps at a median of 4.4 minutes, while the frontier model achieves

60.0% at 9.3 minutes. Appendix L gives the response, token, and system-specific workload distributions.

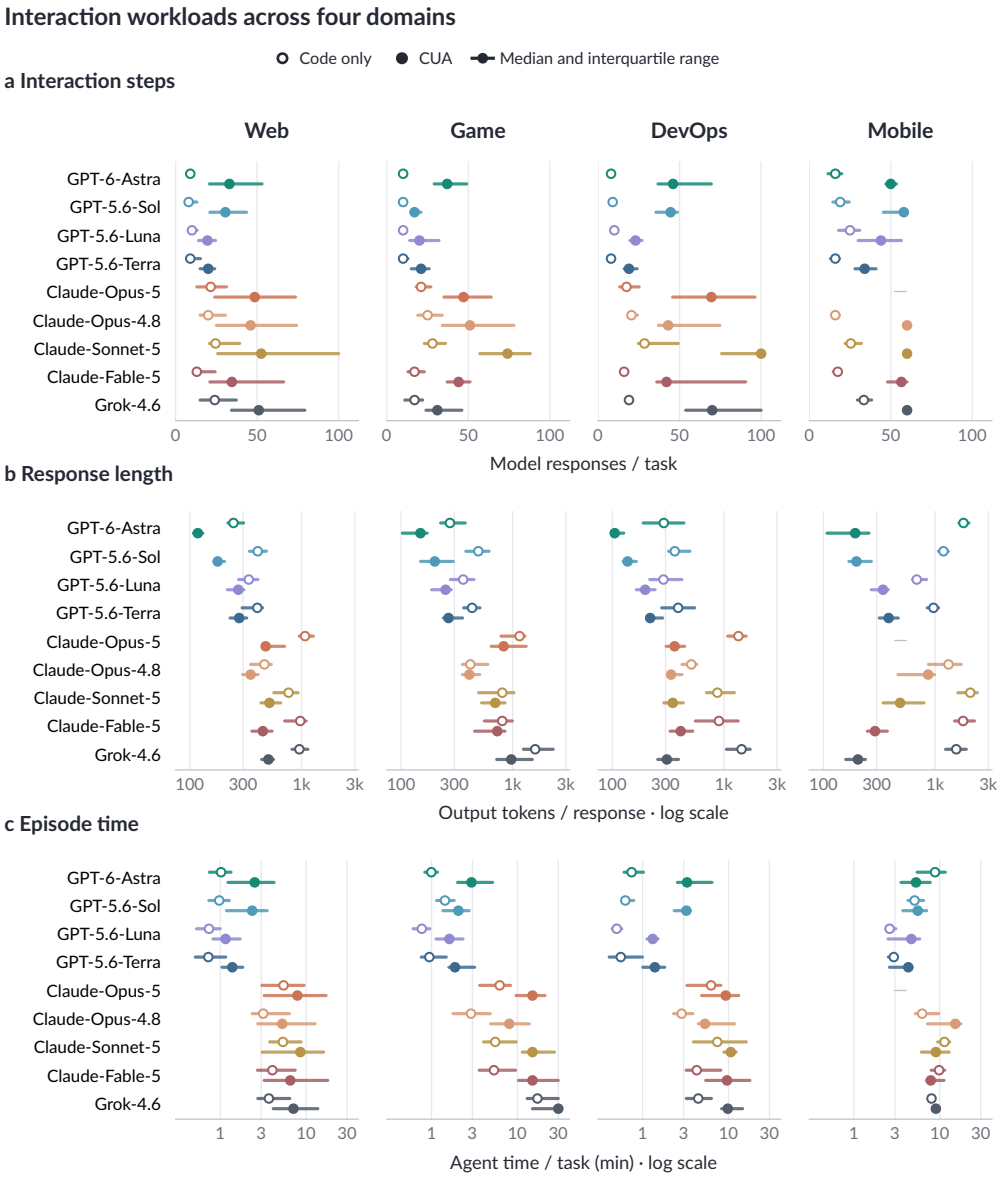


Figure 20: **Interaction steps, response length, and development time.** Hollow circles show code-only and filled circles show CUA. Markers give task medians; intervals span the 25th–75th percentiles. Steps count completed model responses. Response length is recorded output tokens divided by completed responses for each episode, summarized across tasks. Time covers the agent episode. Each metric uses the same eligible tasks across conditions within a model and domain, retaining the full range of task outcomes. Resource coverage is specified in Appendix A.6.

K EFFORT ON SHARED SUCCESSFUL TASKS



Figure 21: **Effort ratios on shared successful tasks.** Each comparison uses the tasks successfully completed by both GPT-6-Astra and the corresponding frontier model with CUA. For each task, effort is divided by the peer's effort; markers show median ratios and intervals span the 25th–75th percentiles. The line at one indicates equal effort. Axes are logarithmic. Steps count model responses; response length uses recorded output tokens per response; time covers the agent episode. Each peer has its own shared-success cohort, whose size appears in the top row. Pairs include executions with resource measurements for both models (Appendix A.6).

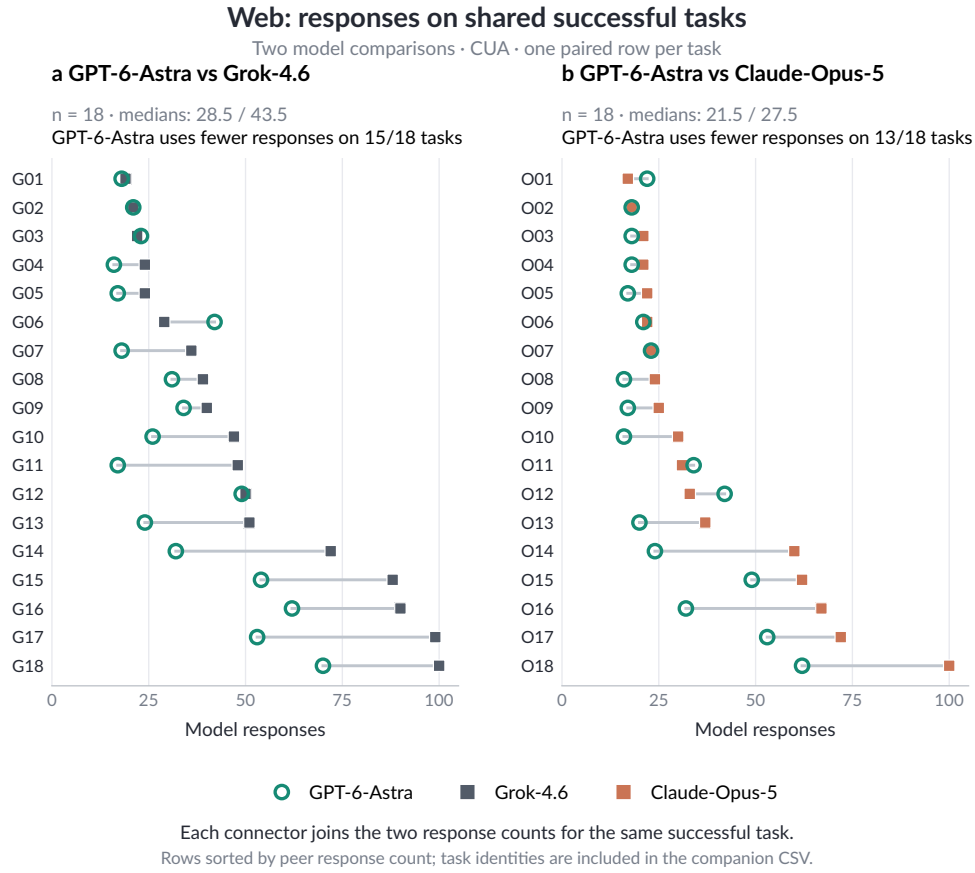


Figure 22: **Response counts on shared successful Web tasks.** Each connected pair gives GPT-6-Astra's and its peer's response counts on one task. The GPT-6-Astra–Grok-4.6 comparison covers 18 shared successes, with medians of 28.5 and 43.5 responses. The GPT-6-Astra–Claude-Opus-5 comparison also covers 18 shared successes, with medians of 21.5 and 27.5 responses. Each panel uses its corresponding pair's shared-success task set.

L RESOURCE PROFILES

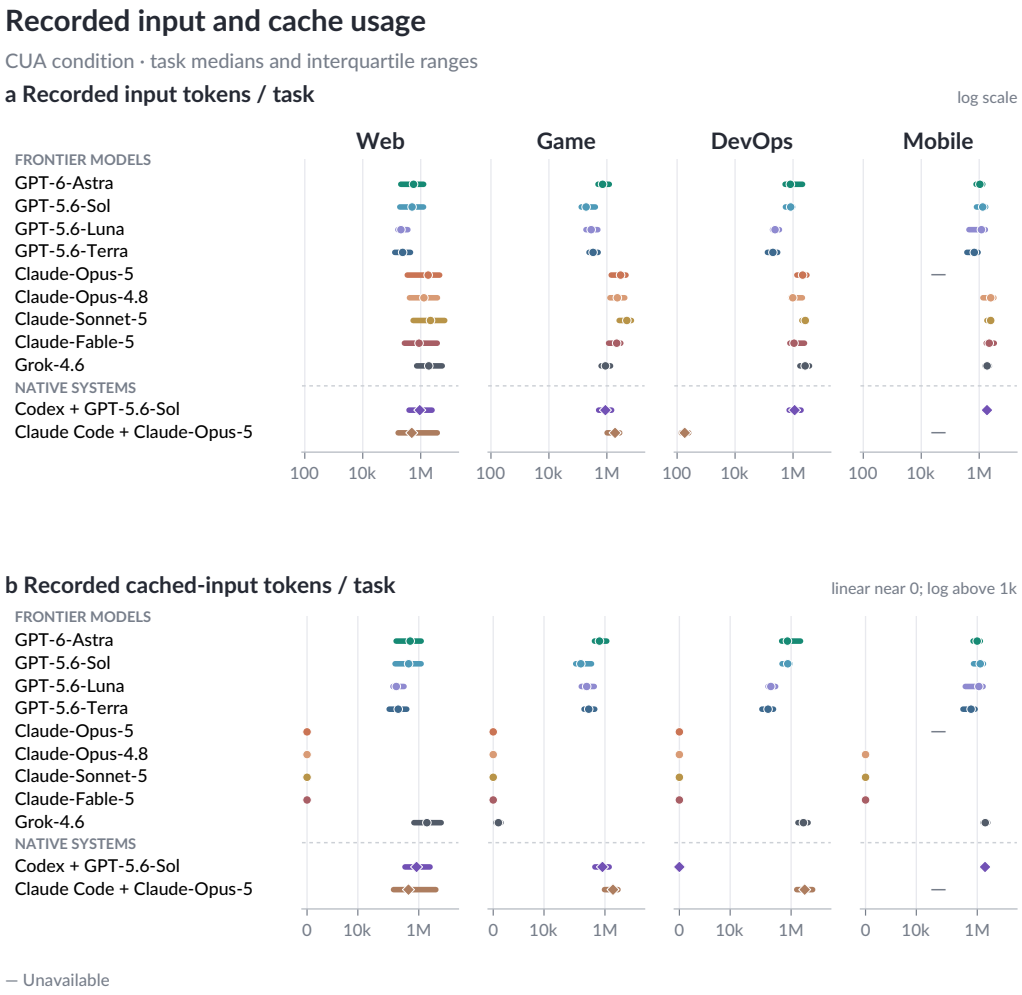


Figure 23: **Recorded input and cached-input tokens per CUA task.** Markers show medians and intervals span the 25th–75th percentiles over each entry's measured CUA cohort. The two counters retain their provider-specific definitions and are summarized separately. Input tokens use a logarithmic axis. Cached-input tokens use a symmetric logarithmic axis with a linear region from zero to 1,000 tokens; tick labels give token counts. Token summaries use episodes with complete recorded usage.



Figure 24: **CUA interaction and output profiles.** Markers show medians and intervals span the 25th–75th percentiles over each entry's measured task cohort. Circles denote frontier models and diamonds denote agent systems. GUI actions and recorded screenshots use linear axes; output tokens use a logarithmic axis. Screenshot counts retain each source's capture definition, and token summaries use complete recorded usage.

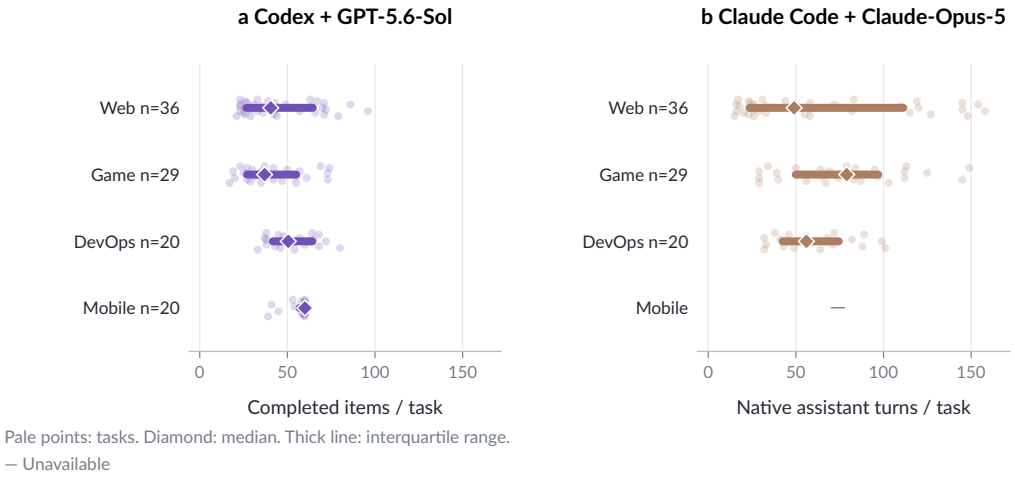


Figure 25: **Native agent-system steps with CUA.** Codex with GPT-5.6-Sol counts completed items; Claude Code with Claude-Opus-5 counts native assistant turns. The two step units appear in separately labeled panels. Pale points show individual tasks, diamonds give medians, and thick intervals span the 25th–75th percentiles. Each panel uses all available native step measurements for the system and domain.

M SUCCESSFUL REPAIR TRAJECTORIES

These cases illustrate four repair mechanisms through first-eligible trajectories from the single-attempt evaluation. Each case follows the agent's observations, source edits, and verification results.

Interaction guides successive edits. In the Web nested-selection task, GPT-6-Astra repairs resizing and then tests a drag beyond the containing frame. The resulting selection behavior motivates a second edit: the group's hit testing is extended to consider selectable descendants outside its bounds. The agent repeats the interaction and submits a passing patch, completing both geometry and selection behavior (Figure 26).



Figure 26: **Successive interaction-guided edits on Web.** GPT-6-Astra repairs resizing, tests selection beyond the containing frame, and extends hit testing to selectable descendants. The scene panels are conceptual schematics of the reviewed trajectory; the edit description summarizes the source change. The resulting patch passes the task's evaluator.

State updates jointly restore the scene. In the Game checkpoint-restart task, GPT-6-Astra reconstructs lift phase from a faded reference frame, restores camera progress in course coordinates, and reattaches the player at its saved support-relative height. The patch also preserves the checkpoint against updates from the beacon's static map cell. Independent build and state-replay checks pass, establishing the combined repair (Figure 27).

S5b GAME

Restore checkpoint state from geometry

Boxel Rebound · Level 50 restart-camera .013 · released first-valid attempt

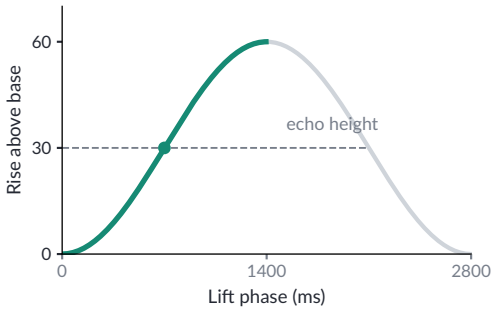
1 Observation evidence · retained transcript

The checkpoint stores geometry; motion phase and camera position are reconstructed from that state.

Paraphrase · GPT-6-Astra, turn 18 · retained transcript

2 Source change · reconstruct, then reattach

Lift reconstruction schematic



Earlier half-cycle is selected.
Echo height here is illustrative.

Exact patch excerpts

```
this.cameraProgress = progress;
this.x = -this.cameraProgress;
```

```
this.supportId = support.id;
this.y = support.top+
this.supportOffset;
```

Camera uses checkpoint course space; player height uses the restored support. The lift phase is recovered by inverting the cosine curve in the source patch.

3 Evaluated behavior · native verifier

PASS **Deterministic state replay passes**

The submitted restart repair passes the task-specific state check and build.

Evidence: September 21 attempt-1 transcript, source.patch and verifier_report.json.
The schematic visualizes restoreCheckpointFrame; the pass comes from the verifier.

Figure 27: **Coordinated restart restoration in Game.** GPT-6-Astra combines lift-phase recovery, course-space camera restoration, and support-relative player attachment. The phase curve is a source-based schematic with an illustrative echo height, and the turn-18 text paraphrases the released first-valid trajectory. Deterministic state replay verifies the resulting repair. This episode belongs to the single-attempt evaluation in Table 8.

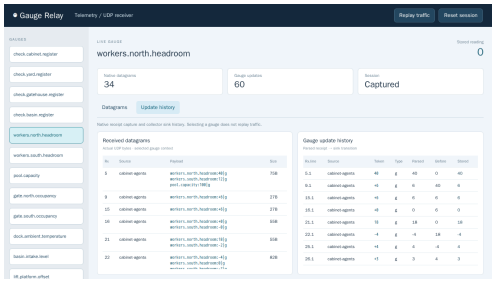
Live behavior identifies the operative contract. In the DevOps gauge task, GPT-6-Astra uses a pulse–pulse–settle sequence to infer each endpoint's update mode. The repair resets that mode at each epoch and preserves receipt order across rollout transitions (Figure 28). This connects visual observations of the service's responses to the semantics implemented in code.

S5c DEVOPS

Infer each producer's gauge-update semantics

StatsD gauge update modes · task 9 · released GPT-6-Astra CUA attempt

1 Observed behavior · original capture 0009



Before the final repair

Signed pulses overwrite the gauge.
For workers.north.headroom:

Tokens 40 → +6 → +6 → +0
Stored 40 → 6 → 6 → 0

Same capture · enlarged update-history detail (unchanged crop)

Rx.line	Source	Token	Type	Parsed	Before	Stored
5.1	cabinet-agents	40	g	40	0	40
9.1	cabinet-agents	+6	g	6	40	6
15.1	cabinet-agents	+6	g	6	6	6
16.1	cabinet-agents	+0	g	0	6	0

2 Source change · learn a mode before applying updates

Provisioning pulses identify delta vs. snapshot semantics for each endpoint.

```
const relative = mode === 'delta' && /^[+-]/.test(measurement.token);
const value = relative ? before + measurement.value : measurement.value;
```

Lexical signs govern addition; unsigned readings replace. Modes are relearned each epoch.

3 Evaluated behavior · native browser verifier

PASS **Gauge transitions and replay checks pass**

The final repair passes native state, receipt-order and epoch-specific mode checks.

Evidence: original screenshot 0009; src/ingest.mjs, src/receiver.mjs and src/update.mjs patch; verifier-report.json. The observation precedes the final repair; evaluated behavior is reported separately.

Figure 28: **Recovering gauge-update semantics in DevOps.** The recorded observation and its update-history crop show behavior during development. Opening provisioning pulses reveal producer update modes; the source excerpts preserve signed update tokens and apply the inferred modes. The final browser check verifies GPT-6-Astra's repair.

Visual structure becomes executable behavior. On Mobile, GPT-6-Astra reconstructs all 23 gear relations and implements their phase, angle, and editing behavior (Figure 29). In the sprite task, it recovers all 28 approved pose assignments together with timing and transport behavior. Both repairs carry precise visual relationships through the application's interactive operations.

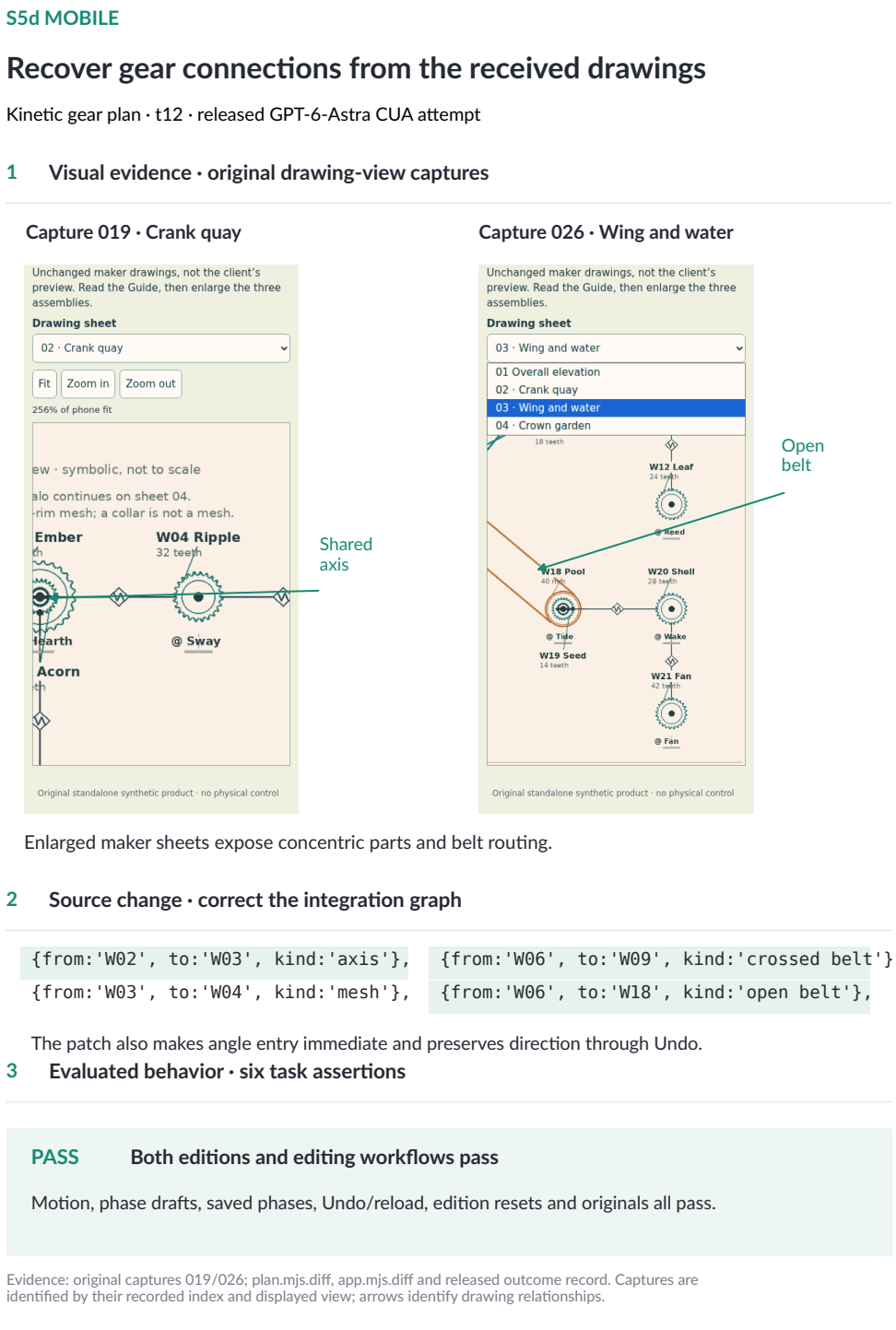


Figure 29: **Translating a Mobile drawing into gear relationships.** The two recorded drawing views guide GPT-6-Astra's graph repair. The source excerpt shows four selected added edges from the patch. The completed repair passes all six task assertions.

M.1 PAIRED REPAIRS AND REMAINING CAPABILITY DEMANDS

Selection after a geometry change. The nested-selection comparison uses the same task under hybrid access. GPT-6-Astra repairs signed resizing and then extends descendant hit testing after its own overflow drag loses selection. Its final patch passes both the resize and selection workflows. GPT-5.6-Sol removes the resize clamps but reverts its experimental hit-testing changes before submission; Claude-Opus-5 also removes the clamps without repairing descendant discovery. Both patches pass the reached resize prefix but lose selection when the visible object is pressed outside the containing frame. This contrast links their different outcomes to the second repair, which preserves the interaction after the geometry change.

Exact visual relations and animation assignments. In the kinetic-gear task, GPT-6-Astra recovers the 23 driving relations. GPT-5.6-Sol's reconstruction selects the wrong immediate driver for two compound assemblies: it substitutes an outer rim for the connected inner rim, changing downstream motion. In the sprite task, GPT-6-Astra recovers all 28 approved pose assignments, while GPT-5.6-Terra swaps two crossing poses and selects an incorrect late catching pose. GPT-5.6-Sol completes the sprite task on its second independent attempt. These comparisons distinguish the accuracy of the recovered relations from the ability to implement the surrounding interaction and animation machinery.

Service semantics and complementary model strengths. In the gauge-calibration task, GPT-6-Astra is the only successful entry. It infers update modes from the opening pulse–pulse–settle sequence, resets them at each deployment epoch, and preserves receipt order. Other service tasks favor different agents. Claude-Opus-4.8 uniquely repairs the receiver-selector task by propagating route metadata into rule compilation and retaining the required selector labels. GPT-6-Astra instead separates literal and templated enrichments without recovering the route's actual identity keys. In log normalization, Grok-4.6 and Codex combine delivered format with producer provenance; GPT-6-Astra changes the format selection but leaves relayed syslog severities interpreted under the wrong convention. The final verifier outcomes match these differences in the submitted implementations.

Coupled game state and dense schedule dependencies. No entry completes Core Ball in the original single-attempt evaluation. GPT-6-Astra's patch corrects impact-angle ownership but leaves failures in waiting-shot promotion, post-commit ordering, pace-change persistence, completion, and rejected-collision occupancy. In Mobile, the original hybrid attempts leave all nine schedule variants unsolved. The inspected GPT-6-Astra schedule patch recovers starts and durations but misassigns two dependency endpoints and one lag; its self-tests encode the same incorrect transcription. The repeated evaluation adds one GPT-6-Astra schedule success on the third attempt. These observations identify the joint recovery of event ordering and relational constraints as a demand that remains beyond the successful local repairs in these cases.

N PATCH CORRECTNESS AND TASK SUCCESS

We report three complementary measures for each model and access condition. Patch correctness averages P_i, task success averages $S_i = P_i U_i$, and the third measure averages $P_i(1 - U_i)$ over the same N task attempts. All three are expressed as percentages. The first measure summarizes verified repairs; the latter two partition them by trajectory validation. Table 11 gives the results across the four domains.

Table 11: **Patch correctness and task success (%).** Patch reports $P = 1$; Task reports $PU = 1$; the third column reports $P = 1, U = 0$. Each evaluated entry uses the same N tasks and single attempts as Table 8. An em dash denotes an unevaluated condition.

	Code-only			Hybrid CUA		
Model / system	Patch	Task	P=1, U=0	Patch	Task	P=1, U=0
Web ($N = 36$)						
GPT-6-Astra	27.8	27.8	0.0	66.7	66.7	0.0
GPT-5.6-Sol	19.4	19.4	0.0	58.3	58.3	0.0
GPT-5.6-Luna	11.1	11.1	0.0	13.9	5.6	8.3
GPT-5.6-Terra	11.1	11.1	0.0	13.9	13.9	0.0
Claude-Opus-5	22.2	22.2	0.0	55.6	55.6	0.0
Claude-Opus-4.8	16.7	16.7	0.0	55.6	27.8	27.8
Claude-Sonnet-5	8.3	8.3	0.0	50.0	0.0	50.0
Claude-Fable-5	22.2	19.4	2.8	55.6	47.2	8.3
Grok-4.6	30.6	30.6	0.0	58.3	55.6	2.8
Codex + GPT-5.6-Sol	—	—	—	55.6	55.6	0.0
Claude Code + Claude-Opus-5	—	—	—	55.6	55.6	0.0
Game ($N = 29$)						
GPT-6-Astra	17.2	17.2	0.0	37.9	37.9	0.0
GPT-5.6-Sol	10.3	10.3	0.0	10.3	10.3	0.0
GPT-5.6-Luna	0.0	0.0	0.0	6.9	6.9	0.0
GPT-5.6-Terra	3.4	3.4	0.0	10.3	6.9	3.4
Claude-Opus-5	13.8	13.8	0.0	20.7	20.7	0.0
Claude-Opus-4.8	3.4	3.4	0.0	20.7	20.7	0.0
Claude-Sonnet-5	6.9	6.9	0.0	10.3	3.4	6.9
Claude-Fable-5	13.8	13.8	0.0	20.7	17.2	3.4
Grok-4.6	10.3	10.3	0.0	20.7	17.2	3.4
Codex + GPT-5.6-Sol	—	—	—	17.2	17.2	0.0
Claude Code + Claude-Opus-5	—	—	—	20.7	20.7	0.0
DevOps ($N = 20$)						
GPT-6-Astra	0.0	0.0	0.0	80.0	80.0	0.0
GPT-5.6-Sol	0.0	0.0	0.0	55.0	55.0	0.0
GPT-5.6-Luna	0.0	0.0	0.0	15.0	15.0	0.0
GPT-5.6-Terra	0.0	0.0	0.0	25.0	25.0	0.0
Claude-Opus-5	0.0	0.0	0.0	60.0	60.0	0.0
Claude-Opus-4.8	0.0	0.0	0.0	55.0	55.0	0.0
Claude-Sonnet-5	0.0	0.0	0.0	40.0	40.0	0.0
Claude-Fable-5	5.0	5.0	0.0	55.0	55.0	0.0
Grok-4.6	0.0	0.0	0.0	65.0	65.0	0.0
Codex + GPT-5.6-Sol	—	—	—	55.0	55.0	0.0
Claude Code + Claude-Opus-5	—	—	—	65.0	65.0	0.0
Mobile ($N = 20$)						
GPT-6-Astra	0.0	0.0	0.0	55.0	55.0	0.0
GPT-5.6-Sol	0.0	0.0	0.0	45.0	45.0	0.0
GPT-5.6-Luna	0.0	0.0	0.0	35.0	35.0	0.0
GPT-5.6-Terra	0.0	0.0	0.0	35.0	35.0	0.0
Claude-Opus-5	—	—	—	—	—	—
Claude-Opus-4.8	0.0	0.0	0.0	40.0	40.0	0.0
Claude-Sonnet-5	0.0	0.0	0.0	35.0	35.0	0.0
Claude-Fable-5	5.0	5.0	0.0	40.0	40.0	0.0
Grok-4.6	0.0	0.0	0.0	30.0	30.0	0.0
Codex + GPT-5.6-Sol	—	—	—	40.0	40.0	0.0
Claude Code + Claude-Opus-5	—	—	—	—	—	—

O EXTENDED RELATED WORK

O.1 CODING AND SOFTWARE ENGINEERING BENCHMARKS

HumanEval measures whether programs synthesized from docstrings satisfy functional tests (Chen et al., 2021). LiveCodeBench uses newly collected programming problems to evaluate generation, self-repair, execution, and test-output prediction (Jain et al., 2024). BigCodeBench broadens function-level evaluation to complex instructions and compositions of library calls (Zhuo et al., 2025). These benchmarks make functional correctness measurable while varying the program understanding and tool knowledge required to produce a solution.

Repository-level benchmarks introduce dependencies between changes across an existing project. SWE-bench evaluates issue resolution in real repositories (Jimenez et al., 2024); SWE-bench Pro emphasizes longer engineering tasks involving substantial, often multi-file changes (Deng et al., 2025). SWE-Lancer includes both independent engineering work, from bug fixes to feature implementations, and decisions among implementation proposals. Its independent engineering tasks use end-to-end tests (Miserendino et al., 2025). Terminal-Bench evaluates multi-step work in command-line environments, pairing tasks with dedicated environments, reference solutions, and verification tests (Merrill et al., 2026). These settings establish that SWE evaluation extends beyond isolated code generation. CUA-SWE adds direct interaction with the running software to this development process, so that agents can connect implementation choices with observed application behavior.

O.2 COMPUTER-USE BENCHMARKS AND VISUAL GROUNDING

Mind2Web provides demonstrations of tasks on real websites and studies generalization across websites and domains (Deng et al., 2023). WebArena supplies reproducible, functional websites and evaluates task completion (Zhou et al., 2023). VisualWebArena adds tasks for which visual information is needed to identify the appropriate objects and actions (Koh et al., 2024). WebVoyager studies end-to-end interaction with live websites and uses a multimodal model to judge task completion (He et al., 2024). These evaluation settings differ in whether they use recorded demonstrations or online execution and in how they assess the final outcome.

OSWorld supports tasks across desktop applications with execution-based evaluation (Xie et al., 2024). Windows Agent Arena adapts this approach to a real Windows environment (Bonatti et al., 2024), while AndroidWorld defines mobile tasks with programmatic initialization, success checks, and teardown (Rawles et al., 2024). ScreenSpot-Pro isolates a constituent capability: locating interface targets in high-resolution screenshots of professional applications (Li et al., 2025). Grounding accuracy, successful navigation, and correct software changes measure different capabilities. CUA-SWE requires agents to coordinate interface interaction with source or configuration changes, and checks whether the resulting software satisfies the task requirements.

O.3 VISUAL AND INTERACTIVE SOFTWARE DEVELOPMENT

SWE-bench Multimodal introduces images into issues and tests involving user-facing JavaScript software (Yang et al., 2024b). Design2Code studies the reconstruction of webpages from reference screenshots, with visual metrics and human evaluation (Si et al., 2025). WebGen-Bench evaluates websites generated from natural-language requirements by using a navigation agent to exercise and check their functionality (Lu et al., 2025). GameDevBench studies development in a game engine and investigates image- and video-based feedback during implementation (Chi et al., 2026). Together, these works show how visual information enters development as a specification, an observation, or an evaluation signal. CUA-SWE emphasizes observations acquired while using the software: the agent chooses what to inspect and how to interact, then relates the observed behavior to its changes. Its task-specific tests check explicit requirements across web development, game development, mobile app development, and DevOps.

O.4 AGENT INTERFACES AND POST-TRAINING

SWE-agent studies interfaces for repository navigation, editing, and execution (Yang et al., 2024a). OpenHands provides a platform for software agents with code, command-line, and browser

tools (Wang et al., 2024). On the computer-use side, Agent S2 combines generalist planning with specialist components for interaction (Agashe et al., 2025). UI-TARS learns screenshot-grounded interaction policies (Qin et al., 2025), and OpenCUA provides computer-use demonstrations, data-processing infrastructure, and trained agent models (Wang et al., 2025). AndroidControl studies how training data scale and instruction granularity affect mobile-agent performance (Li et al., 2024). These approaches inform the interfaces and supervision through which an agent learns to act. CUA-SWE provides a task environment in which coding actions and computer use belong to the same development episode, with executable feedback on the software produced by that episode.

O.5 Hybrid coding and computer-use environments

Programming with Pixels provides a visual IDE for diverse SWE tasks and evaluates both visual interaction and access to file-editing, shell, and IDE APIs. Its programmatic verifiers can check compilation and test outcomes independently of the agent's interaction channel (Aggarwal & Welleck, 2025). This directly connects software engineering with computer use and executable evaluation. WeaveBench studies long-horizon work across graphical interfaces, command-line tools, and code in multiple domains. Its trajectory-aware agentic judge examines artifacts and interaction evidence against task requirements (Li et al., 2026).

CUA-SWE organizes this combination around software engineering tasks with deterministic tests of the resulting implementation and behavior. The benchmark, evaluation pipeline, and post-training environment share these executable requirements. This supports the study of how coding agents choose when to use software, acquire visual feedback, revise code, and verify the consequences of their changes.

P QUANTITATIVE ANALYSIS OF DEVELOPMENT TRAJECTORIES

We analyze interaction counts from 2,040 evaluation attempts and detailed annotations of 214 trajectories, complementing the examples in Appendix M with measurements of observation, implementation, and verification behavior.

P.1 EVALUATION SAMPLE AND ANNOTATION PROCEDURE

Coverage and sampling. The records comprise 720 Web, 580 Game, 400 DevOps, and 340 Mobile attempts, with the task-success outcomes of Table 8. Interaction counts use the full collection. Detailed review selects up to two failures and one success per model–domain–condition combination, using a fixed hash ranking of task identities within each outcome stratum. The sample contains 59 Web, 59 Game, 52 DevOps, and 44 Mobile trajectories: 154 failures and 60 successes across 77 evaluated combinations. Review examines instructions, interactions, authored patches, verifier findings, and relevant screenshots. Annotation counts describe this stratified sample; outcome associations weight each attempt by N_h/n_h, where N_h and n_h are its stratum's population and sample sizes.

Annotations and outcomes. Annotations distinguish relevant runtime information, relevant GUI behavior, substantive edits, and GUI checks of an edited revision. Runtime information includes execution of product code in tests or simulations; static source inspection alone does not qualify. Fields with insufficient evidence remain unknown. We report task success $S_i = P_i U_i$ and patch correctness P_i separately. Reviewers annotated the sample; a second reviewer annotated 32 overlapping trajectories before seeing the primary labels, followed by adjudication. Section P.6 reports original agreement and scope.

P.2 FAILURE STAGES ACROSS MODELS AND DOMAINS

A taxonomy of development behavior. We annotate five overlapping stages: *observation-acquisition gaps* (O), where relevant evidence has not been acquired; *incorrect interpretation* (Perc) of observed evidence; *mistaken diagnosis* (D) of the underlying mechanism; *implementation errors or incomplete repairs* (I); and *verification-coverage gaps* (V), including unexamined interactions or checks of stale revisions. Labels describe evidenced behavior, including subsequently corrected errors, rather than mutually exclusive terminal causes.

Domain profiles. Among 84 sampled hybrid failures, implementation errors or incomplete repairs occur in 61 trajectories (21 of 22 in Game), observation gaps in 59, and verification gaps in 82. Verification gaps also occur in 36 of 41 sampled hybrid successes, reflecting incomplete coverage beyond a repaired interaction. Code-only O/V labels also reflect the condition's exclusion of GUI access. Tables 12 and 13 retain separate conditions and unknown counts.

Table 12: **Failure-stage annotations by domain and access condition.** Each cell gives supported/not observed/unknown counts among n sampled task failures. O: observation; Perc: interpretation; D: diagnosis; I: implementation; V: verification coverage. Stages may overlap.

Domain	Condition	n	O	Perc	D	I	V
Web	Code-only	18	18/0/0	0/18/0	2/14/2	5/0/13	18/0/0
Web	Hybrid	22	16/6/0	6/13/3	6/8/8	12/3/7	21/1/0
Game	Code-only	18	18/0/0	0/16/2	3/13/2	16/0/2	18/0/0
Game	Hybrid	22	15/7/0	6/12/4	7/9/6	21/0/1	22/0/0
DevOps	Code-only	18	18/0/0	1/16/1	14/0/4	16/0/2	18/0/0
DevOps	Hybrid	22	13/8/1	6/12/4	1/17/4	15/0/7	22/0/0
Mobile	Code-only	16	16/0/0	1/15/0	9/7/0	14/1/1	16/0/0
Mobile	Hybrid	18	15/1/2	2/13/3	0/16/2	13/3/2	17/1/0

Table 13: **Failure-stage counts for each model and domain.** Cells report supported/unknown counts; the remaining annotations are not observed. Counts describe the outcome-stratified review sample, with at most two failed attempts per model, domain, and condition. Stage definitions are shared with Table 12.

Model / system	Condition	n	O	Perc	D	I	V
Web							
GPT-6-Astra	Code-only	2	2/0	0/0	0/0	1/1	2/0
GPT-6-Astra	Hybrid	2	2/0	0/0	0/1	1/1	2/0
GPT-5.6-Sol	Code-only	2	2/0	0/0	0/0	0/2	2/0
GPT-5.6-Sol	Hybrid	2	2/0	0/0	1/0	0/2	2/0
GPT-5.6-Luna	Code-only	2	2/0	0/0	0/1	0/2	2/0
GPT-5.6-Luna	Hybrid	2	2/0	0/0	1/0	2/0	2/0
GPT-5.6-Terra	Code-only	2	2/0	0/0	1/0	1/1	2/0
GPT-5.6-Terra	Hybrid	2	2/0	1/0	1/1	2/0	2/0
Claude-Opus-5	Code-only	2	2/0	0/0	0/0	1/1	2/0
Claude-Opus-5	Hybrid	2	0/0	0/2	0/2	2/0	2/0
Claude-Opus-4.8	Code-only	2	2/0	0/0	0/1	1/1	2/0
Claude-Opus-4.8	Hybrid	2	1/0	1/0	1/0	0/0	2/0
Claude-Sonnet-5	Code-only	2	2/0	0/0	1/0	1/1	2/0
Claude-Sonnet-5	Hybrid	2	0/0	0/1	0/0	1/0	1/0
Claude-Fable-5	Code-only	2	2/0	0/0	0/0	0/2	2/0
Claude-Fable-5	Hybrid	2	2/0	0/0	1/0	1/1	2/0
Grok-4.6	Code-only	2	2/0	0/0	0/0	0/2	2/0
Grok-4.6	Hybrid	2	2/0	0/0	0/1	1/1	2/0
Codex + GPT-5.6-Sol	Hybrid	2	2/0	2/0	0/2	0/2	2/0
Claude Code + Claude-Opus-5	Hybrid	2	1/0	2/0	1/1	2/0	2/0
Game							
GPT-6-Astra	Code-only	2	2/0	0/0	0/0	2/0	2/0
GPT-6-Astra	Hybrid	2	2/0	0/0	1/0	2/0	2/0
GPT-5.6-Sol	Code-only	2	2/0	0/0	1/0	1/1	2/0
GPT-5.6-Sol	Hybrid	2	2/0	0/0	0/0	2/0	2/0
GPT-5.6-Luna	Code-only	2	2/0	0/1	0/1	2/0	2/0
GPT-5.6-Luna	Hybrid	2	0/0	0/0	1/0	2/0	2/0
GPT-5.6-Terra	Code-only	2	2/0	0/0	0/0	2/0	2/0
GPT-5.6-Terra	Hybrid	2	2/0	0/0	1/0	2/0	2/0
Claude-Opus-5	Code-only	2	2/0	0/0	1/0	1/1	2/0
Claude-Opus-5	Hybrid	2	1/0	1/0	1/1	2/0	2/0
Claude-Opus-4.8	Code-only	2	2/0	0/0	1/0	2/0	2/0
Claude-Opus-4.8	Hybrid	2	2/0	2/0	0/2	2/0	2/0
Claude-Sonnet-5	Code-only	2	2/0	0/0	0/0	2/0	2/0
Claude-Sonnet-5	Hybrid	2	2/0	2/0	2/0	2/0	2/0
Claude-Fable-5	Code-only	2	2/0	0/1	0/1	2/0	2/0
Claude-Fable-5	Hybrid	2	0/0	0/0	0/0	2/0	2/0
Grok-4.6	Code-only	2	2/0	0/0	0/0	2/0	2/0
Grok-4.6	Hybrid	2	2/0	0/2	0/2	1/1	2/0
Codex + GPT-5.6-Sol	Hybrid	2	0/0	1/0	0/0	2/0	2/0
Claude Code + Claude-Opus-5	Hybrid	2	2/0	0/2	1/1	2/0	2/0
DevOps							
GPT-6-Astra	Code-only	2	2/0	0/0	2/0	2/0	2/0
GPT-6-Astra	Hybrid	2	0/1	0/0	0/0	1/1	2/0
GPT-5.6-Sol	Code-only	2	2/0	0/0	2/0	2/0	2/0
GPT-5.6-Sol	Hybrid	2	2/0	0/1	0/1	1/1	2/0
GPT-5.6-Luna	Code-only	2	2/0	0/0	2/0	2/0	2/0
GPT-5.6-Luna	Hybrid	2	2/0	1/0	1/1	2/0	2/0
GPT-5.6-Terra	Code-only	2	2/0	0/0	2/0	2/0	2/0
GPT-5.6-Terra	Hybrid	2	2/0	0/1	0/1	0/2	2/0

Table 13 (continued)

Model / system	Condition	n	O	Perc	D	I	V
Claude-Opus-5	Code-only	2	2/0	0/0	1/1	2/0	2/0
Claude-Opus-5	Hybrid	2	1/0	2/0	0/0	2/0	2/0
Claude-Opus-4.8	Code-only	2	2/0	0/0	1/1	1/1	2/0
Claude-Opus-4.8	Hybrid	2	2/0	1/0	0/0	1/1	2/0
Claude-Sonnet-5	Code-only	2	2/0	1/1	1/1	2/0	2/0
Claude-Sonnet-5	Hybrid	2	2/0	0/1	0/1	2/0	2/0
Claude-Fable-5	Code-only	2	2/0	0/0	2/0	2/0	2/0
Claude-Fable-5	Hybrid	2	1/0	0/1	0/0	1/1	2/0
Grok-4.6	Code-only	2	2/0	0/0	1/1	1/1	2/0
Grok-4.6	Hybrid	2	1/0	2/0	0/0	2/0	2/0
Codex + GPT-5.6-Sol	Hybrid	2	0/0	0/0	0/0	2/0	2/0
Claude Code + Claude-Opus-5	Hybrid	2	0/0	0/0	0/0	1/1	2/0
Mobile							
GPT-6-Astra	Code-only	2	2/0	0/0	0/0	1/1	2/0
GPT-6-Astra	Hybrid	2	0/1	0/1	0/0	1/1	2/0
GPT-5.6-Sol	Code-only	2	2/0	0/0	2/0	2/0	2/0
GPT-5.6-Sol	Hybrid	2	2/0	0/0	0/0	1/0	2/0
GPT-5.6-Luna	Code-only	2	2/0	0/0	2/0	2/0	2/0
GPT-5.6-Luna	Hybrid	2	2/0	0/0	0/0	2/0	2/0
GPT-5.6-Terra	Code-only	2	2/0	0/0	1/0	2/0	2/0
GPT-5.6-Terra	Hybrid	2	2/0	0/0	0/0	1/1	2/0
Claude-Opus-4.8	Code-only	2	2/0	0/0	2/0	2/0	2/0
Claude-Opus-4.8	Hybrid	2	2/0	0/0	0/0	2/0	2/0
Claude-Sonnet-5	Code-only	2	2/0	1/0	1/0	2/0	2/0
Claude-Sonnet-5	Hybrid	2	2/0	0/0	0/0	1/0	1/0
Claude-Fable-5	Code-only	2	2/0	0/0	0/0	2/0	2/0
Claude-Fable-5	Hybrid	2	2/0	1/0	0/0	2/0	2/0
Grok-4.6	Code-only	2	2/0	0/0	1/0	1/0	2/0
Grok-4.6	Hybrid	2	2/0	1/0	0/0	1/0	2/0
Codex + GPT-5.6-Sol	Hybrid	2	1/1	0/2	0/2	2/0	2/0

P.3 Observation, editing, and GUI re-verification

When agents observe relevant behavior. Of the 214 reviewed trajectories, 201 contain substantive edits and 13 contain none. Table 14 separates observation before the first edit from observation at any point in the attempt. Among the 116 edited hybrid trajectories, 83 establish relevant observation before editing, 32 do not, and one has unknown ordering. All 116 acquire relevant runtime information at some point, while six never observe the relevant GUI behavior. Among the 85 edited code-only trajectories, 47 obtain relevant runtime information through non-GUI channels. These counts distinguish an early edit made before observation from an attempt that never acquires relevant evidence.

Table 14: **Observation and editing in the reviewed trajectories.** N/E gives all/edited attempts. Before-edit observation reports yes/no/unknown among the E edited attempts; no-edit attempts are excluded. The last two columns count edited attempts with no relevant runtime information or no relevant GUI observation at any point.

Domain	Condition	N/E	Before edit Y/N/U	No runtime information	No GUI observation
Web	Code-only	27/27	0/27/0	21	27
Web	Hybrid	32/32	19/13/0	0	4
Game	Code-only	26/24	0/24/0	13	24
Game	Hybrid	33/31	14/16/1	0	2
DevOps	Code-only	19/18	1/17/0	3	18
DevOps	Hybrid	33/32	32/0/0	0	0
Mobile	Code-only	17/16	1/15/0	1	16
Mobile	Hybrid	27/21	18/3/0	0	0

Checking the changed application. Post-edit GUI re-verification requires that an edited revision is active, that the relevant behavior is exercised, and that the resulting GUI observation is delivered before submission. Final-edit re-verification additionally requires this sequence after the last substantive edit. The check may reveal success or a remaining defect; its label records whether the agent checked the behavior. Source-level tests and the evaluator's subsequent checks are recorded separately from this agent-side GUI activity.

Table 15 reports weighted outcome rates among the 116 edited hybrid trajectories. Observed final-edit re-verification is associated with higher task success in each domain: 45.8% versus 5.7% on Web, 16.6% versus 12.8% on Game, 68.1% versus 5.9% on DevOps, and 83.7% versus 10.6% on Mobile. Patch-correctness rates show the same ordering; for Web, they are 61.3% versus 5.7%. The table retains unknown check labels, including 13 of the 31 edited Game trajectories for the final-edit measure. These are descriptive associations within the sampled attempts; task difficulty, model, and the trajectory-validation component of task success can also contribute. The small outcome strata support descriptive estimates rather than significance tests or design-based confidence intervals.

Table 15: **GUI re-verification and repair outcomes among edited hybrid attempts.** Y/N/U are raw counts with an observed/absent/unknown GUI check. Task success PU and patch correctness P are inverse-inclusion-weighted percentages, shown separately for Y and N. Unknown check labels are retained in U and excluded from the Y–N comparison. All protocol statuses are included.

			Task success (%)		Patch correctness (%)	
Domain	GUI check	Y/N/U	Y	N	Y	N
Web	Post-edit	27/5/0	47.6	6.9	61.2	6.9
Web	Final-edit	24/7/1	45.8	5.7	61.3	5.7
Game	Post-edit	10/9/12	16.6	9.3	16.6	9.3
Game	Final-edit	7/11/13	16.6	12.8	16.6	12.8
DevOps	Post-edit	23/9/0	68.1	5.9	68.1	5.9
DevOps	Final-edit	23/9/0	68.1	5.9	68.1	5.9
Mobile	Post-edit	19/2/0	54.8	0.0	54.8	0.0
Mobile	Final-edit	10/11/0	83.7	10.6	83.7	10.6

P.4 Interaction counts for successful and unsuccessful repairs

Image deliveries and GUI actions. Table 16 reports all 1,115 hybrid attempts in the execution collection, separating task successes and failures within each model and domain. We report the median and first and third quartiles, retaining observed zeros. Image counts measure delivered screenshot content: successful image-view results for Web, Game, and DevOps frontier models; pending-image deliveries for Mobile frontier models; and returned image blocks for CLI agents. Repeated transmission of a Mobile image is a separate transport operation and is not added to its delivery count.

Table 16: **Observed image deliveries and GUI actions by repair outcome.** Every entry is a median [first quartile, third quartile] over hybrid task successes (S) or failures (F); n gives S/F attempts. An em dash marks an empty success group. Counts are retained-log observations with interface-specific units, defined in the text.

		Image deliveries		GUI actions	
Model / system	n (S/F)	S	F	S	F
Web					
GPT-6-Astra	24/12	7.5 [6, 16.8]	19 [14, 26.5]	6 [4, 15.5]	17.5 [12.2, 25.2]
GPT-5.6-Sol	21/15	7 [5, 10]	13 [9.5, 18]	6 [4, 8]	12 [8.5, 17]
GPT-5.6-Luna	2/34	4 [3.5, 4.5]	2.5 [1, 3]	3.5 [3.2, 3.8]	3.5 [1, 7]
GPT-5.6-Terra	5/31	4 [2, 4]	2 [1, 4.5]	3 [2, 4]	3 [1, 5]
Claude-Opus-5	20/16	6.5 [5, 11]	20 [15.8, 24]	9.5 [5, 13.2]	36.5 [31.8, 50]
Claude-Opus-4.8	10/26	7.5 [5.2, 8.8]	10.5 [4, 15.8]	7 [5, 18.2]	16 [6, 22.8]
Claude-Sonnet-5	0/36	–	9.5 [3, 19]	–	11 [3, 23.5]
Claude-Fable-5	17/19	6 [5, 6]	16 [12, 21]	5 [5, 11]	22 [17.5, 37]
Grok-4.6	20/16	12.5 [6.8, 15.2]	18 [12, 21]	11.5 [5.8, 14.2]	17 [11.8, 22.2]
Codex + GPT-5.6-Sol	20/16	9 [6, 14.2]	29 [19, 32.5]	8 [5, 13.2]	27.5 [18, 30.8]
Claude Code + Claude-Opus-5	20/16	7 [5, 9.2]	32 [25.2, 46]	4.5 [3.8, 8]	30 [21.2, 42]
Game					
GPT-6-Astra	11/18	8 [6, 14]	14.5 [9.5, 27.8]	6 [5, 12]	12 [8.5, 26.8]
GPT-5.6-Sol	3/26	1 [1, 2]	2.5 [1, 4.8]	4 [3, 6]	4 [2, 5]
GPT-5.6-Luna	2/27	3.5 [2.8, 4.2]	2 [1, 3.5]	13 [10.5, 15.5]	3 [0.5, 7]
GPT-5.6-Terra	2/27	4.5 [3.8, 5.2]	3 [1, 4]	5 [3.5, 6.5]	5 [1.5, 8.5]
Claude-Opus-5	6/23	19 [12.8, 25.2]	10 [7, 12.5]	17 [14, 22.2]	10 [7, 23.5]
Claude-Opus-4.8	6/23	5.5 [4.2, 18.8]	11 [6, 18.5]	5 [4.2, 28.2]	13 [5, 25]
Claude-Sonnet-5	1/28	12 [12, 12]	14.5 [9.8, 19.2]	8 [8, 8]	19 [10.8, 44]
Claude-Fable-5	5/24	12 [11, 14]	9 [6.8, 14.5]	15 [10, 27]	14 [6.8, 36]
Grok-4.6	5/24	12 [10, 25]	8 [2.8, 11.2]	12 [10, 25]	7 [1, 10]
Codex + GPT-5.6-Sol	5/24	11 [9, 13]	11 [6, 25.8]	10 [8, 11]	10 [5, 23.8]
Claude Code + Claude-Opus-5	6/23	24 [18.8, 33.8]	24 [11.5, 34]	21 [14, 28.8]	18 [10, 28]
DevOps					
GPT-6-Astra	16/4	18 [13.8, 27]	31 [18, 44.2]	13.5 [9.8, 21]	24.5 [13.5, 35.8]
GPT-5.6-Sol	11/9	18 [16, 18.5]	15 [11, 19]	17 [15, 17.5]	14 [10, 18]
GPT-5.6-Luna	3/17	3 [3, 4]	2 [1, 3]	9 [8, 11]	7 [3, 11]
GPT-5.6-Terra	5/15	6 [4, 7]	5 [2.5, 5.5]	6 [5, 7]	4 [3, 8.5]
Claude-Opus-5	12/8	20.5 [13.8, 26]	30 [26, 36.2]	25 [13.8, 35.5]	33.5 [26.5, 64]
Claude-Opus-4.8	11/9	10 [9.5, 14.5]	12 [9, 27]	12 [10.5, 16]	13 [9, 27]
Claude-Sonnet-5	8/12	19.5 [15, 24.8]	28 [24.5, 33]	18.5 [14, 23.5]	28 [20, 34.8]
Claude-Fable-5	11/9	13 [10.5, 18.5]	30 [11, 35]	16 [11.5, 29]	50 [12, 60]
Grok-4.6	13/7	24 [18, 32]	41 [32, 44.5]	26 [18, 31]	34 [31, 40]
Codex + GPT-5.6-Sol	11/9	33 [20, 38.5]	24 [17, 29]	32 [19, 37.5]	23 [16, 28]
Claude Code + Claude-Opus-5	13/7	21 [14, 38]	35 [23, 43.5]	17 [9, 26]	32 [20, 33]
Mobile					
GPT-6-Astra	11/9	41 [35.5, 44.5]	48 [45, 50]	36 [33.5, 38]	46 [44, 47]
GPT-5.6-Sol	9/11	37 [33, 39]	54 [52, 55]	36 [32, 38]	54 [53, 55]
GPT-5.6-Luna	7/13	21 [19, 21.5]	35 [25, 42]	19 [17.5, 20]	33 [23, 39]
GPT-5.6-Terra	7/13	19 [18.5, 20]	27 [23, 32]	17 [16.5, 18]	25 [21, 30]
Claude-Opus-4.8	8/12	38.5 [31.5, 45.5]	58 [48.8, 65.5]	37.5 [29.8, 44.8]	58 [48, 65.2]
Claude-Sonnet-5	7/13	46 [39.5, 49.5]	54 [52, 55]	45 [39, 49]	52 [52, 55]
Claude-Fable-5	8/12	47 [42.5, 53.5]	43.5 [40.2, 52.5]	46 [41.5, 53.5]	41 [28.8, 52.5]
Grok-4.6	6/14	49 [48.2, 52.8]	39 [37, 51.5]	48 [46.2, 52]	39 [36, 51.2]
Codex + GPT-5.6-Sol	8/12	53.5 [47.5, 66.5]	54.5 [53.8, 64.8]	51.5 [46.5, 61.2]	53.5 [52.8, 63.8]

Action units. GUI actions use the domain's recorded action unit: completed browser operations on Web, scaffold actions on Game, non-observation protocol actions on DevOps, and completed non-screenshot GUI actions on Mobile. These units are kept separate by model and interface.

Interaction quantity and repair outcome. More interaction does not uniformly accompany a successful repair. For GPT-6-Astra on Web, successful attempts have a median of 7.5 image deliveries and 6 GUI actions, compared with 19 deliveries and 17.5 actions for failures. The complete distributions in Table 16 situate such patterns across all models, alongside the behavior-specific reverification measure above. Counts describe the recorded execution: 80 streams in the full collection lack a recognized terminal marker (4 Web, 71 Game, 4 DevOps, and 1 Mobile), so their observed image counts are lower bounds. Quartiles use linear interpolation at position $(n-1)p$ in the sorted observations.

P.5 Patch correctness and paired task outcomes

Separating repair quality from trajectory validation. We compare the same tasks under code-only and hybrid access in 925 frontier model pairs across 35 model–domain groups. Three groups have lower hybrid task success: GPT-5.6-Luna on Web and Claude-Sonnet-5 on Web and Game. Table 17 separates their passing patches from task success. Across these 101 paired tasks, passing patches increase from 9 to 26, while task successes change from 9 to 3; 23 hybrid passing patches do not satisfy trajectory validation. The seven task-level losses comprise three incorrect patches and four passing patches with invalid tool use; one hybrid win offsets part of these losses. The passing-patch losses involve prohibited HTTP or structured observations. The three incorrect-patch losses involve two masonry-layout repairs and one conditional-physics repair, and also have invalid tool use. This decomposition locates the lower task-success rates in specific repair and tool-use outcomes rather than treating every loss as an implementation failure. It uses the complete paired outcome records and retained patch evidence for the loss cases.

Table 17: **Matched tasks for model–domain pairs with lower hybrid task success.** C/H denotes code-only/hybrid. Wins and losses compare task success on the same task. The final column counts all hybrid passing patches with $U = 0$, including tasks where code-only also fails.

Domain	Model	N	Task C/H	Patch C/H	Hybrid wins/losses	$P{=}1, U{=}0$
Web	GPT-5.6-Luna	36	4 / 2	4 / 5	0 / 2	3
Web	Claude-Sonnet-5	36	3 / 0	3 / 18	0 / 3	18
Game	Claude-Sonnet-5	29	2 / 1	2 / 3	1 / 2	2

P.6 ANNOTATION CONSISTENCY

Independent overlap and adjudication. Table 18 reports agreement on the 32 doubly annotated trajectories before adjudication. Agreement is 100% for acquiring relevant runtime information and observing relevant GUI behavior, 90.6% for observation before the first edit, and 87.5% for both GUI re-verification measures. Stage annotations have lower agreement, especially interpretation and diagnosis; many disagreements concern the distinction between not observed and unknown. We therefore report stage labels as exploratory descriptive annotations, alongside unknown counts and the original agreement statistics. Adjudication examined 110 differing fields and changed 17 primary annotations. The second reviewer had prior exposure to earlier population summaries and selected calibration examples, but not to the new primary annotations. The overlap measures reviewer consistency. Substantive-edit occurrence was not separately annotated by the second reviewer, so agreement for that field is unavailable.

Table 18: **Reviewer agreement before adjudication.** The same 32 trajectories receive both annotations. Unknown is an explicit category in nominal Cohen's κ.

Annotation	Agreement	(%)	κ
Editing before relevant observation	31/32	96.9	0.939
Relevant GUI behavior observed at any point	32/32	100.0	1.000
Final-edit GUI re-verification	28/32	87.5	0.771
Post-edit GUI re-verification	28/32	87.5	0.775
Relevant observation before first edit	29/32	90.6	0.819
Relevant runtime information observed	32/32	100.0	1.000
O: observation-acquisition gap	16/32	50.0	0.238
Perc: incorrect interpretation	1/32	3.1	0.028
D: mistaken diagnosis	2/32	6.2	0.000
I: implementation error or incomplete repair	15/32	46.9	0.240
V: verification-coverage gap	28/32	87.5	0.301

Q BENCHMARK DATASET AND ENVIRONMENT AVAILABILITY

The CUA-SWE benchmark dataset and environment are available in our anonymous GitHub repository at `https://anonymous.4open.science/r/cua-swe-iclr-D151/readme.md`.