

WEAVEBENCH: A Long-Horizon, Real-World Benchmark for Computer-Use Agents with Hybrid Interfaces

Wanli Li^{1,2,*} Bowen Zhou^{3,*} Yunyao Yu³ Zhou Xu³
Yifan Yang² Dongsheng Li² Caihua Shan^{2,†}

¹ Zhejiang University ² Microsoft Research Asia ³ Tsinghua University
wanli_li@zju.edu.cn, zhoubw25@mails.tsinghua.edu.cn
caihua.shan@microsoft.com

Abstract

Computer-use agents (CUAs) increasingly operate in runtimes combining visual desktops, command lines, code editors, browsers, and external tools. Yet existing benchmarks often evaluate these interfaces separately, leaving long-horizon cross-interface orchestration under-tested. Thus, we introduce WEAVEBENCH, a 114-task long-horizon hybrid-interface benchmark spanning 8 work domains, grounded in real requests and publicly verifiable artifacts. Each task requires combining GUI observations/actions with CLI/code operations in one trajectory. We evaluate them on a real Ubuntu desktop in deployed CLI-agent runtimes with a minimal desktop-control plugin. We also propose a companion trajectory-aware judge that inspects deliverables, files, screenshots, logs, and action traces, while detecting shortcut behaviors such as fabricated visual evidence or hard-coded metrics. Across frontier model-runtime pairings, the best PassRate reaches only 41.2%, showing the benchmark remains far from saturated. The judge further reveals that outcome-only grading substantially overestimates performance. Overall, WEAVEBENCH exposes a critical gap in CUA evaluation and provides an effective testbed to measure whether agents can orchestrate GUI, CLI, and code operations across long-horizon real-world tasks. Code is available at <https://github.com/weavebench/WeaveBench>.

1 Introduction

Deployed runtimes for computer-use agents increasingly integrate visual desktop control (GUI), command-line and code execution (CLI/code), browsers, and external tools within a single agent loop (OpenAI, 2025; Claude Code Team, 2026; OpenAI, 2026a; Anthropic, 2026b; OpenClaw Contributors, 2026). This integration reflects the demands of production workflows, where agents must

coordinate multiple interfaces. Figure 1 illustrates three examples to show this requirement: solving such workflows requires agents to interleave GUI-side observations with CLI/code-side inspection, modification, and verification. These channels are complementary rather than interchangeable: GUIs expose rendered and transient interactive state, such as canvases, spatial layout, dialogs, and visual feedback, whereas CLI/code interfaces expose structured, scriptable, and persistent state, such as source files, configurations, logs, artifacts, and service status. Evaluation should therefore shift to cross-interface orchestration: whether an agent can coordinate GUI, CLI/code, browsers, and external tools within one workflow.

However, as shown in Table 1, existing CUA benchmarks have not directly evaluated this coordination. GUI/OS benchmarks (Xie et al., 2024; Bonatti et al., 2024; Rawles et al., 2025; Koh et al., 2024; Zhou et al., 2024) primarily evaluate a single first-class GUI channel, although OSWORLD permits GUI-mediated shell use or customized interfaces; CLI/coding benchmarks (Merrill et al., 2026) expose only CLI/code, so the hybrid workflows remain unreachable. Multi-interface benchmarks (Yan et al., 2025; Jia et al., 2025; Agarwal and Welleck, 2025; Sun et al., 2026; CocoaBench Team et al., 2026) jointly expose GUI and CLI/MCP, but most tasks remain solvable through a single channel, making the extra channel a convenience rather than a requirement; and Claw-class benchmarks (Ding et al., 2026; Zhang et al., 2026) evaluate real-world, long-horizon user tasks inside deployed agent runtimes, but inherit a CLI-only scope with no GUI interface.

To address this gap, we propose WEAVEBENCH, a benchmark for long-horizon hybrid-interface computer use. It contains 114 tasks across 8 real-world domains, sourced from real user requests submitted to deployed open-source agent runtimes, with traceable provenance. Each task requires

* Equal contribution. † Corresponding author.



sults show that current models and runtimes still struggle with long-horizon orchestration across interfaces, and that WEAVEBENCH provides an effective testbed for measuring this capability.

2 Related Work

Computer-use agent benchmarks. CUA benchmarks have advanced along two threads. On the GUI side, work evaluates either static element grounding (Li et al., 2025; Cheng et al., 2024; Wu et al., 2024; You et al., 2024; Lin et al., 2024; Xu et al., 2025) or end-to-end execution in real OS, mobile, and web environments (Xie et al., 2024; Bonatti et al., 2024; Rawles et al., 2025; Zhou et al., 2024; Koh et al., 2024). On the CLI/coding side, benchmarks score agents on terminal-driven software-engineering tasks (e.g., TerminalBench (Merrill et al., 2026)). The two threads remain evaluated in isolation.

Multi-interface and hybrid CUA benchmarks. Another line of work dispatches agents across GUI and structured channels (CLI, APIs, MCP) within a single scaffold (Song et al., 2025; Zhang et al., 2025). MCPWORLD (Yan et al., 2025), OSWORLD-MCP (Jia et al., 2025), and PWP-BENCH (Aggarwal and Welleck, 2025) add API or MCP actions on top of GUI task suites, but each task still admits equivalent single-channel solutions, making channel choice a per-step convenience. SCIENCEBOARD (Sun et al., 2026) requires GUI/CLI co-use on most tasks, but only as a task-level division of labor, and is confined to scientific-discovery workflows.

Claw-class benchmarks for deployed CLI agents. As deployed CLI agent runtimes such as CLAUDE CODE (Claude Code Team, 2026), OPENCLAW (OpenClaw Team, 2026), and CODEX CLI (OpenAI, 2026a) have entered real production use, a new line of benchmarks draws tasks from real user requests submitted to these runtimes. The resulting tasks are naturally in-the-wild, long-horizon, and span multiple files and services, as in WILDCLAWBENCH (Ding et al., 2026) and CLAWBENCH (Zhang et al., 2026). Concurrent COCOABENCH (CocoaBench Team et al., 2026) shares this task profile but is browser-only and grades only the final output. None of them evaluate the GUI channel. Our WEAVEBENCH inherits this paradigm but (i) promotes GUI to a peer channel of CLI on top of the deployed runtime, (ii) bakes channel non-substitutability into each task, and (iii)

grades every trajectory with an agent-as-judge.

3 WEAVEBENCH Benchmark

In this section, we first define the task admission criteria in WEAVEBENCH, then describe the construction pipeline and task diversity, and finally present the trajectory-aware evaluation protocol.

3.1 Task Admission Criteria

A WEAVEBENCH task is admitted when it satisfies three properties. **P1 Channel non-substitutability:** Task success must require coordinating GUI observation/action with programmatic modifications through CLI/code operations within the same trajectory. To make it auditable, each task is annotated with its required single-channel-bound atomic operations (Appendix A.1). **P2 Long-horizon execution:** The expert reference trajectory must contain multiple interleaved GUI and CLI/code phases rather than a single perception, action, or tool-use step. **P3 Cross-application state:** The task must span multiple independent applications or processes whose states are linked by the workflow. Agents must preserve and transfer information across these applications, rather than complete the task within a single isolated tool. Appendix A.2 reports trajectory-level evidence for P2 and P3.

3.2 Task Construction

Each task is built through a four-stage pipeline. **C1 Archetype-guided sourcing:** For each domain, experts first define cooperation archetypes that specify the required GUI-side and CLI/code-side roles and then search for real-world tasks from public artifacts (GitHub issues/PRs, postmortems, design mocks, the OPENCLAW user community). **C2 Asset packaging:** For each candidate task, experts assemble a self-contained task bundle. A bundle includes the initial environment, seed data, required assets, user instruction, expected deliverables, expert reference trajectory, and verification anchors used by the judge. **C3 Blind review:** an independent reviewer checks instruction clarity, sandbox reproducibility, P1–P3 validity, and anchor faithfulness. **C4 Pilot validation:** The three pilot agents are run to detect broken, ambiguous, trivial, or uninformative tasks; pilot outcomes trigger task revision rather than serving as ground truth. The per-venue source breakdown is provided in Appendix A.4.

Benchmark	Platform	Multi-ch.	Non-subst.	In-the-wild	Cross-app	Traj.judge	Deployed
WebArena	Web	✗	✗	P	P	✗	✗
OSWORLD	Real OS	P	✗	P	P	✗	✗
WindowsAgentArena	Real OS	✗	✗	P	P	✗	✗
MCPWorld	App	✓	✗	✗	✗	✗	✗
OSWorld-MCP	Real OS	✓	✗	P	P	✗	✗
ScienceBoard	Real OS	✓	P	P	P	✗	✗
ClawBench	Web	✗	✗	✓	✓	✗	✓
WildClawBench	Headless OS	✗	✗	✓	P	P	✓
CocoaBench	Web	✓	✗	P	P	✗	✗
WEAVEBENCH (ours)	Real OS	✓	✓	✓	✓	✓	✓

Table 1: **Positioning WEAVEBENCH against representative CUA benchmarks.** ✓/✗/P denote satisfied / not satisfied / partial. **Multi-ch.:** GUI and CLI/code/API/MCP exposed simultaneously. For this column, P denotes indirect GUI-mediated CLI access or programmatic access through non-standard extensions, rather than concurrent first-class GUI and CLI channels. **Non-subst.:** the task cannot be solved by an equivalent single-channel rewrite. **In-the-wild:** tasks sourced from real-world artifacts. **Cross-app:** tasks span at least two independent applications. **Traj. judge:** evaluator audits the agent’s trajectory, not only the final state. **Deployed:** harness is an open-source agent framework deployed by real users.

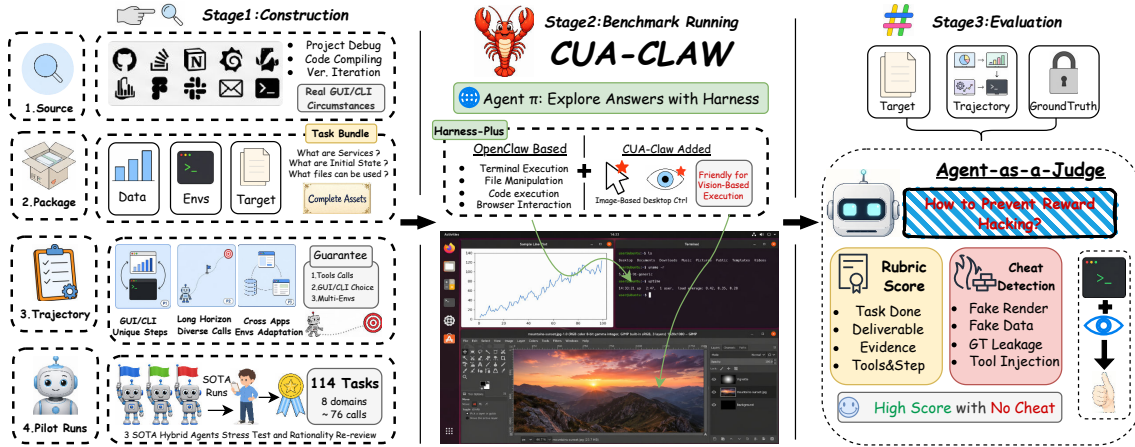


Figure 2: **WEAVEBENCH pipeline.** **Task:** 114 tasks across 8 domains, harvested from real venues, packaged as $\mathcal{E} = (\mathcal{P}, \mathcal{M}, \mathcal{C})$ bundles, audited against P1–P3, and stress-tested by ≥ 3 pilot agents. **Harness:** the agent runs in a single session over an Ubuntu sandbox, where a minimal GUI plugin (one screenshot tool plus nine actuation primitives) is added on top of OPENCLAW’s CLI/code tools. **Evaluation** is performed by an isolated trajectory-aware agentic judge that combines bottom-up rubric scoring with shortcut detection.

3.3 Task Diversity

Task domains: WEAVEBENCH comprises 114 tasks across 8 real-work domains (Figure 3(a); Table 8; Appendix A.3): desktop productivity, document processing, games & interactive applications, web development, data analysis & visualization, DevOps & sysadmin, spatial / 3D / CAD, and design & creative. Per-domain task counts range from 10 to 18, with each domain contributing a distinct cooperation archetype.

Task resources: Each task is built on real-world assets (code repos, issue threads, monitoring snapshots, design mocks, database dumps, configs) sourced from public venues or replayed in self-

hosted sandboxes. Each carries provenance (URL, commit hash, or post id). The per-venue source breakdown is reported in Appendix A.4.

Trajectory profile: Tasks are also long-horizon and channel-interleaved: the best live rollouts use a median of 76 tool calls (max 471) and a median of 16 GUI $\leftrightarrow$ CLI channel switches per task (Figure 3(b,c); Appendix A.2).

3.4 Trajectory-aware Agent as Judge

Hybrid rollouts cannot be reliably graded from final deliverables alone. In WEAVEBENCH, task success depends on desktop state, source files, generated artifacts, screenshots, and the sequence of tool actions. Final-only grading is therefore vulner-

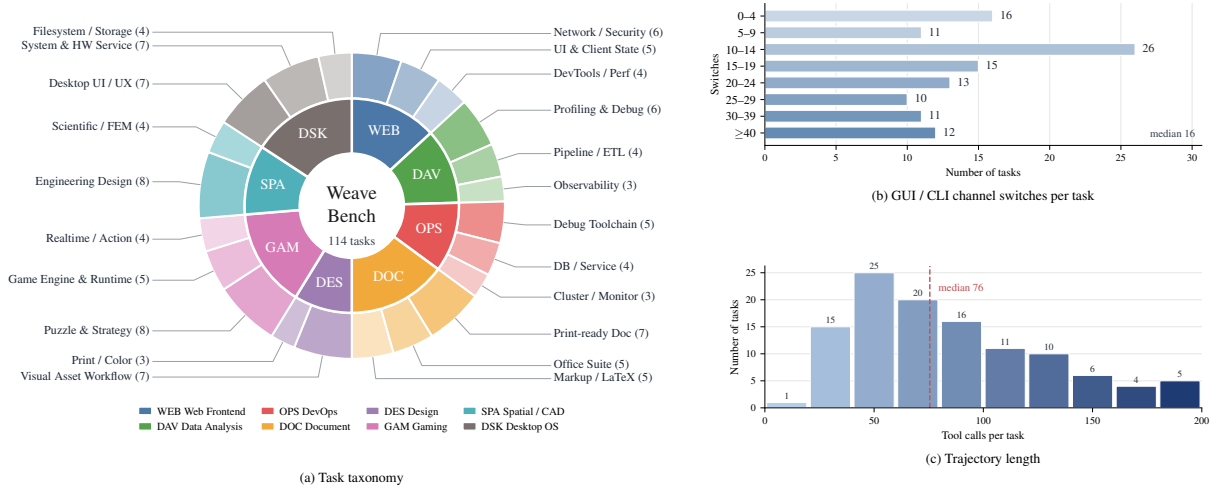


Figure 3: WEAVEBENCH dataset overview. (a) Taxonomy of 114 tasks across 8 domains and 23 subcategories. (b) Number of GUI ↔ CLI channel switches per task, showing the degree of channel interleaving. (c) Rollout length measured by tool calls in the trajectory.

able to shortcuts: artifacts can be synthesized, metrics can be hard-coded, and visually correct states can be reached through specification-violating tool use. We therefore treat evaluation as a trajectory-level evidence audit. Each rollout is assessed by a **trajectory-aware agentic judge** that re-fetches evidence from artifacts, screenshots, logs, and reference anchors. The task anchors and scoring rules are human-reviewed, and sampled judge verdicts are manually audited.

The judge runs in a fresh subprocess for each rollout, using a fixed backbone and inspection tools for files, images, and shell-based checks. It verifies each case over multiple evidence-gathering turns rather than from a single transcript pass. Runtime details are provided in Appendix B.1.

Scoring follows a layered pipeline. The judge decomposes each required deliverable into atomic clauses that preserve the instruction’s constraints, such as required counts, visual states, file formats, and highlighted regions. It then verifies each clause as satisfied, partially satisfied, or false, citing concrete evidence such as an artifact line, a measured value, a screenshot observation, or a trajectory quote. Clause-level decisions are aggregated into a per-deliverable correctness score $d_{t,m}^{\text{deliv}}$. The judge also assigns eight evaluation dimensions $\{d_{t,m,i}^{\text{process}}\}_{i=1}^8$ covering task completion, deliverable correctness, deliverable quality, evidence authenticity, tool-use correctness, final-state correctness, efficiency and robustness, and instruction following. This exposes partial progress while requiring every score to be grounded in in-

spected evidence. The full rubric is described in Appendix B.3.

In parallel, the judge scans the trajectory for nine manually confirmed shortcut patterns, including fake screenshots or renders, regenerated fixtures, hard-coded metrics, mock services, duplicate crops, overlay manipulation, ground-truth leakage, and runtime injection. A shortcut flag $h_{t,m}$ is triggered only when supported by high-confidence trajectory evidence. If triggered, the rollout receives zero credit. All agents also receive an anti-fabrication policy that prohibits these behaviors while allowing honest skipped-deliverable fallbacks. Details are provided in Appendix B.4.

The final score for model m on task t is

$$s_{t,m} = \begin{cases} 0, & \text{if } h_{t,m} = 1, \\ \min\left(\frac{1}{8} \sum_{i=1}^8 d_{t,m,i}^{\text{process}}, d_{t,m}^{\text{deliv}}\right), & \text{otherwise.} \end{cases} \quad (1)$$

The minimum rule prevents strong auxiliary dimensions from masking weak deliverables, while the zeroing rule prevents fabricated or protocol-violating evidence from receiving partial credit.

At the benchmark level, we report two complementary metrics. PASSRATE measures end-to-end success at threshold $\tau = 0.8$, while OVERALL reports mean partial credit:

$$\begin{aligned} \text{PassRate}(m) &= \frac{1}{|T|} \sum_{t \in T} \mathbf{1}[s_{t,m} \geq \tau], \\ \text{Overall}(m) &= \frac{1}{|T|} \sum_{t \in T} s_{t,m}. \end{aligned} \quad (2)$$

4 Experiments

The performance on WEAVEBENCH depends on both the model API and the agent runtime. The model API determines reasoning, multimodal grounding, context use, tool-call formatting, and instruction following, whereas the runtime determines the agent loop, tool wrappers, GUI access, filesystem state, and trajectory logging. We therefore first compare diverse model APIs under a fixed runtime, then fix the strongest APIs and vary the deployed runtime to identify the best-performing combinations. We finally run two ablations to test interface necessity and trajectory-aware judging, followed by analyses of tool-use and failure modes.

4.1 Experimental Setup

Hybrid harness. We add GUI control to existing CLI-agent runtimes through a minimal plugin. The plugin exposes one perception primitive, screenshot, and nine pyautogui-backed actuation primitives: `click`, `double_click`, `triple_click`, `move`, `drag`, `scroll`, `type`, `keypress`, and `wait`. These tools are exposed in the same Responses-style session as the host’s terminal, file, code, and browser tools. The model backbone, agent loop, system prompt, and maximum turn budget are left unchanged.

Environment. Every task runs in a containerized Linux VM. The VM boots from a frozen snapshot, rolls back after completion, restricts network access to task-local services, and enforces per-task tool-output and wall-clock budgets. These controls keep the initial state, available services, and resource limits fixed across models and runtimes.

Model-API sweep. We use open-source OPENCLAW (OpenClaw Team, 2026) as the reference runtime for fairness. All backbones share the same hybrid harness, tool pool, timeout, temperature, and maximum turn budget. We sweep five OpenAI generations from GPT-5.1-codex through GPT-5.5 (OpenAI, 2026b) at three thinking budgets (low, medium, high). To reduce provider specificity, we also evaluate Claude Opus 4.7 (Anthropic, 2026a), Gemini-3.1-pro (Google DeepMind, 2026), and strong open-source backbones. Tables report the best thinking budget per backbone; the full GPT-5.x thinking sweep is reported in Appendix D.1.

Harness sweep. We select the strongest APIs from the model sweep and keep the GUI plugin fixed while varying the runtime host. The same plugin is ported through thin adapters to CODEX CLI (Ope-

nAI, 2026a), CLAUDE CODE (Claude Code Team, 2026), and HERMES (Nous Research, 2025), in addition to OPENCLAW. This sweep measures whether strong APIs remain strong across deployed agent runtimes and identifies the best observed model-runtime pairing.

Scoring. All rollouts are evaluated by the trajectory-aware agentic judge in Sec. 3.4. We report PASSRATE and OVERALL over 114 tasks.

4.2 Main results

Sweep Model APIs on OPENCLAW. Table 2 shows that even the best backbone is far from saturation. Claude Opus 4.7 obtains the highest PassRate on the fixed OPENCLAW harness at 35.1%, followed by GPT-5.5 at 33.3% and GPT-5.4 at 22.8%. These scores remain far below the >78% reported for the same frontier backbones on OSWORLD-Verified. Performance also varies substantially within the GPT-5 series, from 1.8% for GPT-5.1-codex to 33.3% for GPT-5.5, suggesting that hybrid-interface execution improves with frontier model capability but remains unsolved. Other evaluated backbones trail substantially in PassRate. Across the per-domain columns, the two most GUI-heavy domains, SPA and DES, are the bottom two for every backbone with non-trivial PR, confirming the GUI side as the binding constraint.

Cross-harness sweep. Table 3 fixes the model API and varies the deployed runtime. The results show that hybrid-interface performance is determined not only by the model, but also by its alignment with the runtime scaffold. The best observed pairing is Claude Opus 4.7 with CLAUDE CODE, reaching 41.2% PassRate, while GPT-5.5 with CODEX CLI also performs strongly at 35.1%. However, cross-pairing the models with less aligned runtimes causes sharp drops: Claude Opus 4.7 falls to 13.2% on CODEX CLI, and GPT-5.5 drops to 14.9% on CLAUDE CODE. These asymmetric failures suggest that tool schemas, prompting conventions, and action-loop design interact strongly with model-specific tool-use behavior.

4.3 Interface Ablation

A natural concern about Sec. 4.2 is that the low PassRate reflects harness friction rather than genuine cross-interface difficulty. We re-run each backbone in three settings: *GUI-only*, using the screenshot tool with the nine actuation primitives; *CLI-only*, using the full OPENCLAW CLI; and *Hybrid*, using both. Both single-interface settings collapse

Agent	PR↑	Overall↑	DSK	DOC	GAM	WEB	DAV	OPS	SPA	DES
Claude Opus 4.7	35.1	0.482	55.6	29.4	23.5	66.7	15.4	41.7	16.7	20.0
GPT-5.5	33.3	0.466	38.9	35.3	35.3	21.4	23.1	38.5	33.3	40.0
GPT-5.4	22.8	0.465	55.6	35.3	5.9	0.0	23.1	23.1	8.3	20.0
GPT-5.3-codex	18.4	0.456	33.3	23.5	29.4	0.0	7.7	16.7	8.3	20.0
GPT-5.2-codex	6.1	0.321	5.6	11.8	0.0	0.0	15.4	16.7	0.0	0.0
GPT-5.1-codex	1.8	0.226	0.0	5.9	0.0	0.0	7.7	0.0	0.0	0.0
Gemini-3.1-pro	1.8	0.223	0.0	0.0	0.0	0.0	0.0	8.3	8.3	0.0
Qwen3.5-397B-A17B	0.9	0.318	0.0	0.0	0.0	0.0	0.0	8.3	0.0	0.0
Qwen3-VL-8B-Think	0.9	0.092	0.0	0.0	0.0	0.0	8.3	0.0	0.0	0.0
GUI-Owl-1.5-32B	0.0	0.065	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 2: **Model API comparison on a fixed OPENCLAW runtime.** The best thinking mode is reported for each backbone. PR denotes PassRate (%). Overall denotes the mean per-task score. Per-domain columns report PassRate for DSK: Desktop, DOC: Document, GAM: Games, WEB: Web Development, DAV: Data Analysis & Visualization, OPS: DevOps, SPA: Spatial/3D, and DES: Design.

Backbone	Harness	PR↑	Overall↑	DSK	DOC	GAM	WEB	DAV	OPS	SPA	DES
GPT-5.5	CODEx CLI	35.1	0.499	38.9	29.4	23.5	53.3	15.4	50.0	58.3	10.0
	OPENCLAW	33.3	0.466	38.9	35.3	35.3	21.4	23.1	38.5	33.3	40.0
	HERMES AGENT	31.6	0.466	55.6	29.4	35.3	40.0	7.7	25.0	25.0	20.0
	CLAUDE CODE	14.9	0.299	33.3	11.8	11.8	0.0	15.4	16.7	25.0	0.0
Claude Opus 4.7	CODEx CLI	13.2	0.378	16.7	11.8	11.8	6.7	7.7	25.0	16.7	10.0
	OPENCLAW	35.1	0.482	55.6	29.4	23.5	66.7	15.4	41.7	16.7	20.0
	HERMES AGENT	28.1	0.516	33.3	47.1	11.8	26.7	30.8	50.0	8.3	10.0
	CLAUDE CODE	41.2	0.532	55.6	47.1	23.5	53.3	23.1	50.0	33.3	40.0

Table 3: **Runtime comparison for the strongest model APIs.** GPT-5.5 and Claude Opus 4.7 are evaluated at their high thinking across four agent runtimes. PR denotes PassRate (%). Overall denotes the mean per-task score over the full 114-task suite. Per-domain columns report PassRate.

to low PassRate (Table 4): GUI-only stays at or below 1.8%, because the screenshot context overflows the model window before the task ends; CLI-only stays at or below 3.5%, an order of magnitude below Hybrid for every backbone. This pattern is consistent with the design of WEAVERBENCH tasks with channel non-substitutable.

To test whether this gap is unique to WEAVERBENCH, we compare against prior hybrid CUA benchmarks that report comparable interface ablations (Table 5). On MCPWORLD (Yan et al., 2025) and OSWORLD-MCP (Jia et al., 2025) the hybrid gain over the strongest reported single channel is only +4.5pp and +3.2pp respectively, so adding the second channel is at most a convenience.

4.4 Trajectory-Aware Judge Ablation

We next quantify how much the trajectory-aware judge contributes. We re-score every rollout with an outcome-only judge that sees only the final deliverables, with no trajectory access and no cheat scan. Switching back to the trajectory-aware judge of

Agent	GUI only	CLI only	Hybrid
Claude Opus 4.7	1.8	3.5	35.1
GPT-5.5	0.8	2.6	33.3
GPT-5.4	0.8	2.6	22.8
GPT-5.3-codex	0.0	1.8	18.4

Table 4: Interface ablation on PassRate. **GUI**: GUI-only tool pool. **CLI**: CLI-only tool pool. **Hybrid**: full tool pool. The best thinking mode is chosen.

Benchmark	GUI	CLI/MCP	Hyb.	Δ
OSWorld-MCP	40.1	–	43.3	+3.2
MCPWorld	70.7	53.2	75.1	+4.5
WEAVERBENCH (ours)	1.8	3.5	35.1	+31.6

Table 5: Cross-benchmark hybrid gain on PassRate (%), with Δ giving the gain from the additional interface.

Sec. 3.4 removes between 10.3 and 20.2 PassRate points across the four GPT backbones (Figure 4). For GPT-5.5, the audited rate falls from 53.5% to 33.3%. These gaps are lower bounds: every rollout was already given an anti-fabrication prompt with a cost-free honest fallback (Appendix B.5), so

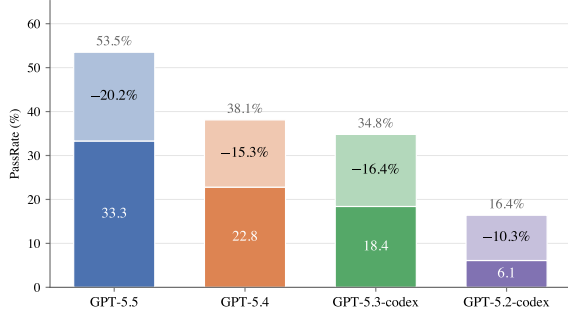


Figure 4: **Outcome-only vs. trajectory-aware judging.** Dark blue: audited PassRate (Sec. 3.4). Light blue: inflation removed by the audit, with label inside showing PassRate points removed. Top label: outcome-only total and points removed.

a benchmark without that prompt would see even more inflation. Trajectory-aware grading is therefore necessary on hybrid CUA tasks: outcome-only shortcuts are a first-order failure mode, not a corner case. The full cheat-pattern catalog is shown in Appendix B.4 and Sec. 4.6; independent human, cross-judge, and repeated-run validation is reported in Appendix B.2.

4.5 Tool-call distribution

We decompose every tool call across all the GPT-5.5 rollouts into atomic operations. As shown in Figure 5, the top-10 operations cover 93.1% of 10,873 active calls, and `exec: shell` alone accounts for 27.3%. A large fraction of the GUI work is hidden inside these shell calls: GPT-5.5 invokes `gnome-screenshot` through `exec 2.2x` more often than the native `__computer__.screenshot`, and drives the mouse and keyboard via `pyautogui`, `xdotool`, and `wmctrl` another 521 times. Consequently, the GUI share rises from 33.9% at the tool level to 62.9% once these `exec`-routed GUI operations are re-attributed at the atomic-operation level: even with a dedicated GUI tool exposed, the agent prefers shell paths for GUI actions.

4.6 Failure Taxonomy

Taxonomy. We treat any rollout with final score below the pass threshold $\tau = 0.80$ as a failure, and manually label every failure of GPT-5.5 and GPT-5.4 across the three thinking budgets (636 failures in total) with one of five top-level categories: **E1 Reasoning**, **E2 Trajectory**, **E3 Perception**, **E4 Cheat**, and **E5 Other** (residual runtime errors). Sub-class definitions and the per-rollout codebook are released with the benchmark, and the annotation protocol is detailed in Appendix D.3.

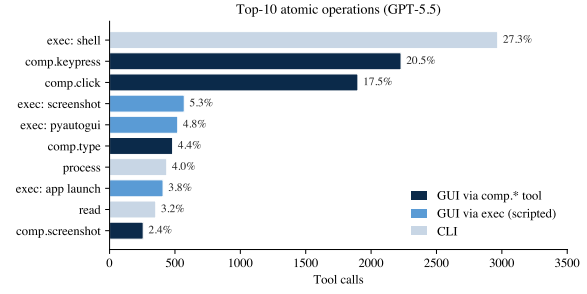


Figure 5: **Top-10 atomic operations across all GPT-5.5 rollouts on WEAVEBENCH.** Bars are sorted by call count and together cover 93.1% of 10,873 active calls. `exec: shell` alone dominates at 27.3%, and several `exec` entries route GUI work through the shell rather than the native GUI plugin.

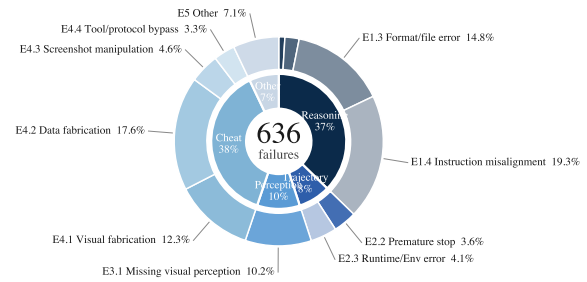


Figure 6: **Failure breakdown across 636 cases.** Bars are sub-classes sorted by count within their parent top-level category. The cases in E4 are zeroed by the trajectory-aware judge.

Dominant failure modes. Figure 6 shows that failures are not mainly caused by visual perception, which accounts for only 10.2%. The largest sources are instruction or format misalignment ($\sim 34\%$) and shortcut taking ($\sim 38\%$). These errors reflect the central challenge: agents must preserve the task specification while coordinating GUI observations, CLI/code operations, and final deliverables over long trajectories. The failure profiles also differ across backbones (Appendix D.2). GPT-5.5 fails mostly through shortcut taking, whereas GPT-5.4 fails mostly through reasoning and instruction-following errors. This suggests that stronger models may produce more plausible artifacts, but not necessarily more faithful execution.

5 Conclusion

In this paper, we propose WEAVEBENCH, a long-horizon benchmark for evaluating computer-use agents in hybrid-interface workflows. The benchmark contains 114 tasks from 8 real-work domains and admits tasks only when they require GUI observation/action and CLI/code operations. We augment deployed agent runtimes with a desktop-

control plugin and attach a trajectory-aware judge.

WEAVEBENCH remains far from saturated. On a fixed OPENCLAW runtime, the best PassRate reaches 35.1%; across deployed runtimes, the strongest model-runtime pairing reaches 41.2%. We also observed that current computer-use agents struggle not only with individual tools, but with sustained orchestration across visual and programmatic operations in various applications. WEAVEBENCH provides an effective and unique testbed for measuring this orchestration capability.

Limitations

WEAVEBENCH focuses on English-language tasks on Linux desktops; extending to other languages, operating systems, and a larger task pool is a natural next step. Our results are reported on deployed CLI runtimes such as OPENCLAW, CODEX CLI, CLAUDE CODE, and HERMES; broader backbone $\times$ harness coverage is left to future work. Several other directions—including finer contamination-aware splits and live-web task variants (Appendix C.1)—are also interesting follow-ups, which we leave to subsequent versions. To enable independent verification, the full release package (snapshots, judge prompts, replay logs, and per-rollout records) is documented in Appendix C.2.

References

- Pranjal Aggarwal and Sean Welleck. 2025. [Programming with pixels: Can computer-use agents do software engineering?](#) *Preprint*, arXiv:2502.18525.
- Anthropic. 2026a. Introducing Claude Opus 4.7. <https://www.anthropic.com/news/claude-opus-4-7>.
- Anthropic. 2026b. Put Claude to work on your computer. <https://claude.com/blog/dispatch-and-computer-use>.
- Rogério Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Buckner, Lawrence Jang, and Zack Hui. 2024. [Windows Agent Arena: Evaluating multi-modal OS agents at scale](#). *Preprint*, arXiv:2409.08264.
- Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. 2024. [SeeClick: Harnessing GUI grounding for advanced visual GUI agents](#). *Preprint*, arXiv:2401.10935.
- CocoaBench Team, Shibo Hao, Zhining Zhang, Zhiqi Liang, Tianyang Liu, Yuheng Zha, Qiyue Gao, Jixuan Chen, Zilong Wang, Zhoujun Cheng, Haoxiang Zhang, Junli Wang, Hexi Jin, Boyuan Zheng, Kun Zhou, Yu Wang, Feng Yao, Licheng Liu, Yijiang Li, and 13 others. 2026. [CocoaBench: Evaluating unified digital agents in the wild](#). *Preprint*, arXiv:2604.11201.
- Shuangrui Ding, Xuanlang Dai, Long Xing, Shengyuan Ding, Ziyu Liu, Yang JingYi, Penghui Yang, Zhixiong Zhang, Xilin Wei, Xinyu Fang, Yubo Ma, Haodong Duan, Jing Shao, Jiaqi Wang, Dahua Lin, Kai Chen, and Yuhang Zang. 2026. [Wildclawbench: A benchmark for real-world, long-horizon agent evaluation](#). *Preprint*, arXiv:2605.10912.
- Google DeepMind. 2026. Gemini 3.1 Pro: Model card. <https://deepmind.google/models/model-cards/gemini-3-1-pro/>.
- Hongrui Jia, Jitong Liao, Xi Zhang, Haiyang Xu, Tianbao Xie, Chaoya Jiang, Ming Yan, Si Liu, Wei Ye, and Fei Huang. 2025. [OSWorld-MCP: Benchmarking MCP tool invocation in computer-use agents](#). *Preprint*, arXiv:2510.24563.
- Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. 2024. [VisualWebArena: Evaluating multi-modal agents on realistic visual web tasks](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 881–905, Bangkok, Thailand. Association for Computational Linguistics.
- Kaixin Li, Ziyang Meng, Hongzhan Lin, Ziyang Luo, Yuchen Tian, Jing Ma, Zhiyong Huang, and Tat-Seng Chua. 2025. [ScreenSpot-Pro: GUI grounding for professional high-resolution computer use](#). *Preprint*, arXiv:2504.07981.
- Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Weixian Lei, Lijuan Wang, and Mike Zheng Shou. 2024. [ShowUI: One vision-language-action model for GUI visual agent](#). *Preprint*, arXiv:2411.17465.
- 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, Jenna Jitsev, Di Lu, Orfeas Menis Mastromichalakis, and 66 others. 2026. [Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces](#). *Preprint*, arXiv:2601.11868.
- Nous Research. 2025. Hermes Agent. <https://github.com/nousresearch/hermes-agent>.
- OpenAI. 2025. Introducing ChatGPT agent: bridging research and action. <https://openai.com/index/introducing-chatgpt-agent/>.

- OpenAI. 2026a. Codex for (almost) everything. <https://openai.com/index/codex-for-almost-everything/>.
- OpenAI. 2026b. Introducing GPT-5.5. <https://openai.com/index/introducing-gpt-5-5/>.
- OpenClaw Contributors. 2026. Peekaboo: Mac automation that sees the screen and does the clicks. <https://github.com/openclaw/Peekaboo>.
- OpenClaw Team. 2026. OpenClaw. <https://github.com/openclaw/openclaw>.
- Christopher Rawles, Sarah Clinckemallie, 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. 2025. *AndroidWorld: A dynamic benchmarking environment for autonomous agents*. *Preprint*, arXiv:2405.14573.
- Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, Ran Xu, and Caiming Xiong. 2025. *CoAct-1: Computer-using multi-agent system with coding actions*. *Preprint*, arXiv:2508.03923.
- Qiushi Sun, Zhoumianze Liu, Chang Ma, Zichen Ding, Fangzhi Xu, Zhangyue Yin, Haiteng Zhao, Zhenyu Wu, Kanzhi Cheng, Zhaoyang Liu, Jianing Wang, Qintong Li, Xiangru Tang, Tianbao Xie, Xiachong Feng, Xiang Li, Ben Kao, Wenhai Wang, Biqing Qi, and 2 others. 2026. *ScienceBoard: Evaluating multimodal autonomous agents in realistic scientific workflows*. *Preprint*, arXiv:2505.19897.
- Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, and Yu Qiao. 2024. *OS-Atlas: A foundation action model for generalist GUI agents*. *Preprint*, arXiv:2410.23218.
- 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. 2024. *OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments*. In *Advances in Neural Information Processing Systems*.
- Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. 2025. *Aguvis: Unified pure vision agents for autonomous GUI interaction*. In *Proceedings of the 42nd International Conference on Machine Learning*.
- Yunhe Yan, Shihe Wang, Jiajun Du, Yexuan Yang, Yuxuan Shan, Qichen Qiu, Xianqing Jia, Xinge Wang, Xin Yuan, Xu Han, Mao Qin, Yinxiao Chen, Chen Peng, Shangguang Wang, and Mengwei Xu. 2025. *MCPWorld: A unified benchmarking testbed for API, GUI, and hybrid computer use agents*. *Preprint*, arXiv:2506.07672.
- Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. 2024. *Ferret-UI: Grounded mobile UI understanding with multimodal LLMs*. *Preprint*, arXiv:2404.05719.
- Chaoyun Zhang, He Huang, Chiming Ni, Jian Mu, Si Qin, Shilin He, Lu Wang, Fangkai Yang, Pu Zhao, Chao Du, Liqun Li, Yu Kang, Zhao Jiang, Suzhen Zheng, Rujia Wang, Jiaxu Qian, Minghua Ma, Jian-Guang Lou, Qingwei Lin, and 2 others. 2025. *UFO2: The desktop AgentOS*. *Preprint*, arXiv:2504.14603.
- Yuxuan Zhang, Yubo Wang, Yipeng Zhu, Penghui Du, Junwen Miao, Xuan Lu, Zhuofeng Li, Xingwei Qu, Zhengkang Guo, Yuanzhe Shen, Dingjie Song, Han Zhou, Tuney Zheng, Xian Wu, Hao Yu, Songcheng Cai, Yi Lu, Yunzhuo Hao, Minyi Lei, and 9 others. 2026. *Clawbench: Can ai agents complete everyday online tasks?* *Preprint*, arXiv:2604.08523.
- Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. *WebArena: A realistic web environment for building autonomous agents*. In *International Conference on Learning Representations*.

A Benchmark Construction

A.1 Atomic-Capability Decomposition for P1

P1 states that task success requires coordinating GUI observation/action with CLI/code modifications within the same trajectory. To make this auditable we enumerate the *atomic operations* a task can require and identify, for each, the mechanism that binds it to a single channel. Table 6 lists 19 such atoms organised in six mechanism-defined families; Table 7 reports the coverage these atoms achieve on the $N=114$ corpus.

Coverage verifies P1 at three strictness levels.

Each task is annotated with the set of atoms its specification requires. Table 7 reports three complementary measures: *any-side* coverage shows that every task draws on at least one atom on each side; *non-universal* coverage shows that the result does not depend solely on the two contract-mandated anchor atoms; *depth* coverage shows that a large fraction of tasks exercise both channels through several atoms each.

Worked example. The OPS · segfault forensics task (diagnose a UAF in a 280-line C HTTP server) requires {K1, K2, K3, K4, N1, F2, F3, F4, V1, V3}. The four **K** atoms cover the kernel-side evidence the diagnosis needs; V3 covers the live debugger view that ties a stack frame to the suspect address. CLI atoms alone cannot produce the multi-panel

ID	Atomic operation	Single-channel binding mechanism
CLI-BOUND ATOMS — evidence lives below the rendered surface (Families K, N, F)		
<i>K — OS / kernel signals</i>		
K1	Capture segfault / coredump	Emitted by the kernel signal-delivery path; no userspace surface originates it.
K2	Trace syscalls	Requires ptrace or eBPF attachment, a kernel-side interface.
K3	Decode PC → source via DWARF	Mapping lives in ELF .debug_info; only parsed by CLI tooling.
K4	Read /proc, systemd, journal	Exposed through procfs / dbus / journal APIs the GUI does not surface.
<i>N — protocol / engine internals</i>		
N1	Intercept HTTP response payload	Payload exists as protocol bytes; the rendered value is a lossy projection.
N2	Read engine EXPLAIN plans	Per-operator timings live only in the engine’s introspection API.
N3	Subscribe to async event streams	WebSocket / SSE frames are consumed off the wire before any DOM update.
<i>F — file-internal invariants</i>		
F1	Validate container invariants	PDF / ODS / SVG / FITS container constraints need a parser; GUI editors silently violate them.
F2	Produce unified-diff patch	The patch grammar is a textual artefact no GUI editor emits as a primary save format.
F3*	Emit schema-typed structured data	Column types and enum constraints are programmatic guarantees outside any GUI export dialog.
F4	Drive cross-stage build pipeline	Multi-stage compilation (pdflatex, asciidoctor, bpy -b) lives in CLI toolchains.
GUI-BOUND ATOMS — evidence exists only after rendering (Families V, E, L)		
<i>V — render-only world state</i>		
V1*	Observe rendered application state	The agent must inspect the current compositor frame; no offline read reveals it.
V2	Verify render-layer bugs	The bug only manifests as a pixel artefact after GPU draw (e.g. emoji rendering).
V3	Correlate multi-panel debugger state	Source ↔ stack ↔ memory simultaneity is a GUI affordance.
<i>E — capability-bound event chains</i>		
E1	Produce realistic pointer trajectory	Anti-bot captchas verify acceleration profiles; synthetic events are rejected upstream.
E2	Trigger DOM / widget event chains	State transitions fire only on real click-type events in the application’s event loop.
E3	Commit application dependency-graph update	In-app recalculation (Calc formulas, Joplin tags) is triggered by UI commit, not file write.
<i>L — closed-loop perceive-act control</i>		
L1	Frame-clocked real-time control	Rhythm / physics gameplay samples input at the display refresh; no offline plan precedes the upcoming frame.
L2	Perceive-then-adjust on visual targets	Colour matching, alignment, PCB routing require the <i>adjust</i> step to be conditioned on the just-rendered frame.

Table 6: The 19 atomic operations underlying P1, organised into two macro-groups (CLI-BOUND / GUI-BOUND) and six mechanism-defined families. Each row gives the mechanism that pins the operation to a single channel. * marks the two anchor atoms (F3, V1) induced by the bench’s deliverable contract (structured artifact + screenshot evidence); the remaining 17 atoms are selective.

correlation; GUI atoms alone cannot produce the kernel-side evidence.

What this establishes. For every task in WEAVEBENCH, the deliverable contract forces evidence on both sides of the GUI/CLI axis (Table 7, any-side row), and for 90.4% of tasks the binding atoms include at least one that is not contract-induced (non-universal row). P1 therefore holds by construction. At the population level, the channel-ablation runs of Sec. 4.3 corroborate this: GUI-only PassRate stays at or below 1.8% and CLI-only at or below 3.5%, both an order of magnitude below the Hybrid setting.

A.2 P2/P3 Trajectory Distributions and By-Domain Metrics

This appendix verifies P2 (long-horizon execution) and P3 (cross-application state) at the trajectory level. For each task we take the best-scoring *Hybrid* rollout across all backbones and thinking budgets; cheat-flagged rollouts are zeroed by the judge of Sec. 3.4, so the counts below are conservative lower bounds. Channel mapping, switch counting, and the per-task app/state definitions follow the registry released with the benchmark.

Task-level distributions (Fig. 7). P2 long-horizon. The best rollout per task uses a median of 76 tool calls (mean 88, range 14–471); 113 of 114 tasks exceed 20 calls. Median GUI/CLI channel

Strictness	What the task requires	#	%
Weak	At least one CLI atom <i>and</i> at least one GUI atom	114	100.0
Medium	Same as weak, plus at least one atom that is <i>not</i> a contract anchor (F3 or V1)	103	90.4
Strong	At least two CLI atoms <i>and</i> at least two GUI atoms	50	43.9
Anchor-only {F3, V1} tasks (low-coordination floor)		11	9.6

Table 7: How many of the $N=114$ tasks satisfy P1 at three increasing strictness levels. *Weak* = the formal P1 definition (both channels touched). *Medium* = P1 is not produced just by the two atoms (F3 = structured artifact, V1 = screenshot) that the deliverable contract forces onto every task. *Strong* = both channels are exercised through multiple atoms. The bottom row counts the 11 tasks whose only single-channel-bound atoms are the two contract anchors. On average each task uses 2.24 CLI and 1.80 GUI atoms.

switches per task is 16; every task triggers at least one switch and on average $\sim 23\%$ of all calls cross the channel boundary, i.e. roughly every fourth call. **P3 cross-application.** The same trajectories juggle a median of 15 distinct apps/business states per task (range 4–24). *All* 114 tasks switch between at least 3 apps, and 89.5% use both GUI and CLI for at least 3 calls each, so neither channel is incidental. P2 and P3 are therefore realised in execution, not just stipulated by task design.

By-domain breakdown (Fig. 8). The trajectory profile is not concentrated in a single domain: every one of the 8 domains clears the P2 threshold (20 tool calls) on more than 90% of its tasks and the P3 threshold (3 distinct apps) on 100% of its tasks. The longest cross-channel chains appear in the observability-heavy domains (DAV, OPS, WEB), where dashboards, logs, and configs alternate, while the Spatial/CAD and Design domains show the most GUI-heavy profile but still maintain non-trivial CLI presence.

A.3 Domain Details

Table 8 details the typical workflow and cooperation archetype for each of the 8 WEAVEBENCH domains introduced in Sec. 3.3.

A.4 Task Source Distribution

Every one of the 114 WEAVEBENCH tasks is grounded in ≥ 1 publicly verifiable URL; the release ships a machine-readable provenance index

with **174** source URLs (mean 1.53 per task, range 1–4) spanning **82** unique hostnames. Each URL is labelled either *user-pain*—a public post where a real user reports the failure the task encodes (Reddit, Stack Exchange, GitHub / GitLab issues, project bug-trackers, Discourse forums, YouTube)—or *reference*, grounding the expected behaviour in canonical documentation (official manual, RFC, project wiki).

Overall, 94/174 URLs (54%) are user-pain, and 91/114 **tasks** (79.8%) **carry ≥ 1 user-pain URL**. Figure 9 decomposes the corpus by venue: on the user-pain side, Stack Exchange contributes 50 URLs across 14 sites and GitHub issues / PRs another 28 across 27 upstream repos; Reddit covers 12 subreddits with 12 URLs, and the remaining 4 are split among GitLab issues, project forums, YouTube and bug trackers. On the reference side, project / vendor docs dominate (70 URLs), supplemented by 8 GitHub repo / blob links and a single Wikipedia and GitLab repo each. The most cited host is github.com (20.7% of URLs); **43 of 82 hosts contribute exactly one URL**, reflecting a deliberately long-tailed venue mix.

The 23 reference-only tasks concentrate in single-developer Linux games (*kmahjongg*, *xmoto*, *anagramarama*, ...) and specialised scientific tooling (*CloudCompare*, *Elmer FEM*, *colmap*, ...): long-tail niches without active online discussion that we deliberately retain, as broad desktop coverage is a stated benchmark goal.

B Trajectory-aware Agent as Judge

This appendix specifies the judge summarized in Sec. 3.4. We first describe the judge’s runtime architecture, the bottom-up scoring chain that produces the final score, the cheating-pattern catalog, the inference-time anti-fabrication prompt, worked examples that illustrate one clean rollout and two cheat-triggered rollouts, and the structure of the per-case output written to disk.

B.1 Architecture

The judge is an OPENCLAW agent, instantiated in a fresh subprocess for every case so that profile, workspace, conversation history, and tool state are fully isolated across cases. The judge backbone is held fixed at GPT-5.5: following standard LLM-as-judge practice, we use a single backbone across the leaderboard, and every verdict was spot-checked by a co-author against the trajectory. Within its sub-

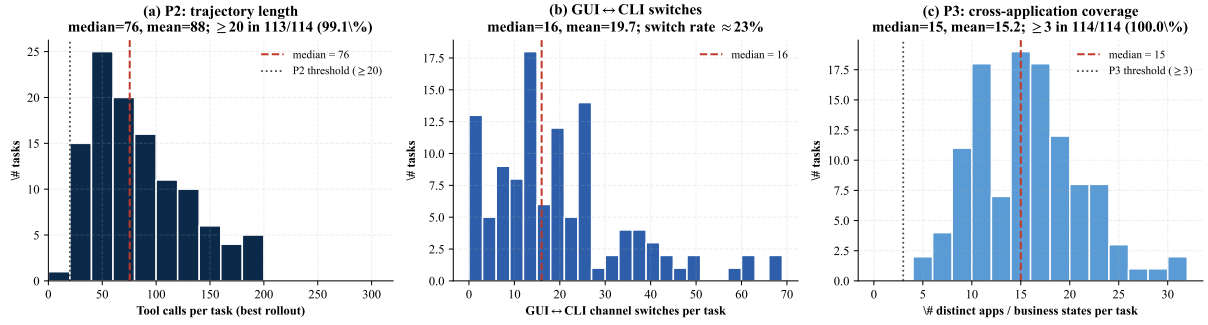


Figure 7: **Trajectory-level evidence that WEAVERBENCH tasks are long-horizon and cross-application.** Panel (a) shows tool-call counts per task in the best-scoring rollout, with a median of 76 and 113 of 114 tasks above 20. Panel (b) shows GUI/CLI channel switches, with a median of 16; every task has at least one switch and the average switch rate is about 23%. Panel (c) shows distinct apps or business states per task, with a median of 15, and 100% of tasks juggle at least 3 apps.

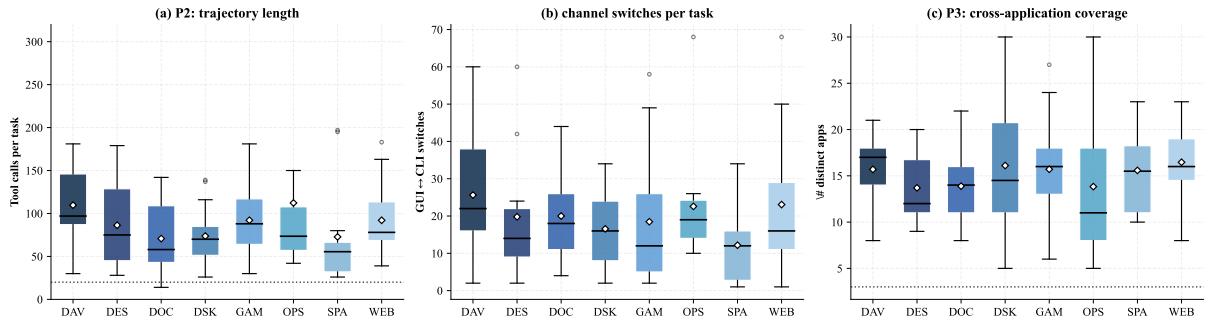


Figure 8: **P2 and P3 metrics by domain.** Box plots over the 8 WEAVERBENCH domains for tool-call counts in panel (a), channel switches in panel (b), and distinct apps in panel (c). Dotted lines mark the thresholds of P2 at 20 calls and P3 at 3 apps; every domain reaches 100% on P3.

process, the judge has access to a real inspection tool pool that can open and read deliverables, view rendered images one at a time, and walk through the agent’s trajectory of intermediate steps. It uses this pool to explore the staged evidence over multiple turns, opening each deliverable against the spec, following up on suspicious passages in the trajectory, and re-fetching evidence on demand. Image inspection is serialized to one image per call, which avoids the silent under-inspection caused by batched gateway truncation. Each case is retried up to five times under judge errors before being marked as “judging failed”.

B.2 Judge Validation and Robustness

Three computer-science experts independently labeled all 114 trajectories from the strongest model-runtime pairing as PASS, ordinary FAIL, or HACK, with majority vote forming the human reference. Inter-annotator agreement is high (Fleiss’ $\kappa = 0.88$). The paper judge obtains a 41.2% PassRate, close to the human rate of 43.0%, with 92% pass/fail agreement and 95% hack agreement. Two al-

ternative judge backbones yield similar PassRates (39.5–42.1%), while three repeated runs of the fixed judge range from 40.4% to 43.0%. These results indicate robustness to both judge choice and inference stochasticity.

B.3 Layered scoring pipeline

The judge produces the final score along a five-layer pipeline, illustrated on the right of Figure 2 and summarised in Table 9: each layer’s output is an explicit field that the next layer must read, and the judge cannot proceed upward until that field is present. Two supporting tables ground the layer-specific constraints—the six-tier rubric in Table 10 bounds Layer 3, and the eight orthogonal dimensions in Table 11 are populated at Layer 4. As an illustration, an image-type deliverable at Layer 1 might be decomposed into clauses such as “the rendered tab is real”, “the trace list contains at least 100 items”, and “the root-cause tag is red”.

Layer 4: eight dimension derivation. Per-deliverable correctness scores are then aggregated

Domain	Typical workflow	Cooperation archetype	#Tasks
Desktop Productivity	Configure desktop apps and OS settings through GUI panels, with shell/scripts to verify and persist changes.	GUI configuration with scripted verification	18
Document Processing	Edit office documents and PDFs in GUI editors, batch-process via CLI tooling, and verify the rendered output.	visual editing with CLI batch transform	17
Games & Interactive	Play the desktop app to spot a bug, locate it in the source, patch the logic, and replay to confirm.	dynamic behaviour with source patch	17
Web Development	Compare a design mock against the live preview, batch-edit HTML/CSS/JS, and re-check pixel alignment.	visual diff with code-level batch edit	15
Data Analysis & Viz	Explore dashboards and tabular data, join sources via SQL/Python, and iterate the resulting chart.	visual exploration with programmatic transform	13
DevOps & SysAdmin	Read a monitoring dashboard, pull logs through kubectl/SQL, edit a config file, and re-check the dashboard.	graphical trend with scripted rollout	12
Spatial / 3D / CAD	Inspect 3D scenes or CAD drawings in a viewer, mutate geometry via scripts/parameters, and re-render to verify.	spatial inspection with parametric edit	12
Design & Creative	Gather assets, edit in Figma/GIMP/Inkscape, batch-export, and visually diff against the target.	creative editing with asset pipeline	10
Total			114

Table 8: **The 8 real-work domains in WEAVEBENCH.** The *typical workflow* column summarizes what domain experts described as the prevailing end-to-end pattern in each domain. The *cooperation archetype* names the role split: which signal the GUI is responsible for surfacing, and which effect CLI/code is responsible for producing.

into the eight orthogonal dimension scores in Table 11. Each dimension lies in $[0, 1]$, and the judge must justify each score by citing specific per-deliverable IDs, for example stating that a dimension is low because deliverable X has $c = 0.3$ and deliverable Y is missing. Two hard constraints are imposed at this layer. First, if any required deliverable is missing, `task_completion` and `final_state_correctness` cannot exceed 0.85. Second, if any per-deliverable correctness is below 0.6, the `deliverable_correctness` dimension cannot exceed 0.7, since this dimension is essentially the mean of per-deliverable scores. Dimension scores at or above 0.95 are reserved for rollouts that the judge would defend before strict reviewers; realistic but imperfect rollouts are expected to land between 0.55 and 0.80.

Layer 5: final score. The bottom-up chain terminates with the aggregation

$$s = \begin{cases} 0 & \text{if is_hack,} \\ \min\left(\frac{1}{8} \sum_{i=1}^8 d_i, d_{\text{deliv}}\right) & \text{otherwise,} \end{cases} \quad (3)$$

where the cheating flag `is_hack` is set by the parallel scan described in Appendix B.4. The min rule encodes the design intent that WEAVEBENCH grades whether the agent’s deliverables are correct, not how hard the agent worked. A rollout with seven dimensions near 1.0 but a deliverable correctness of 0.5 caps at 0.5, no matter how high the mean.

B.4 Cheating-pattern catalog

In parallel with the bottom-up scoring chain, the judge reviews the agent’s trajectory of intermediate steps for the nine stereotyped cheating patterns in Table 12. **Each of the nine patterns was observed at least once during pilot rollouts before being added to the catalog;** we did not seed any synthetic cheating behaviour. A pattern is treated as triggered only when the judge can verbatim-quote the offending evidence from the trajectory at confidence at least 0.85; in that case the case-level flag `is_hack` is set and the final score is forced to zero by Eq. 3. The verbatim quotes are preserved in the output for human auditing.

The catalog is built from cheating modes observed during pilot runs across all 8 domains. Pat-

Layer	What happens	Hard constraints
1. Spec $\rightarrow$ clauses	Decompose the verbatim spec text into 3–8 atomic clauses per deliverable.	Preserve quantifiers such as <i>must</i> , $\geq N$, or “red-highlighted”.
2. Verify clauses	Open files, view images, and walk the trajectory; mark each clause <i>satisfied</i> / <i>partial</i> / <i>false</i> with a one-line evidence quote.	Missing deliverable is not auto-failed: check trajectory first, promote to <i>partial</i> if execution evidence is found.
3. Per-deliverable c	Aggregate clause outcomes into a weighted average $c = (n_{\text{sat}} + 0.5 n_{\text{partial}}) / n_{\text{total}} \in [0, 1]$.	Any critical (<i>must</i> / <i>required</i>) clause unsatisfied $\Rightarrow c \leq 0.40$; six-tier rubric (Tab. 10) prevents rounding up.
4. Eight dimensions	Aggregate per-deliverable scores into 8 orthogonal dimensions $d_i \in [0, 1]$, each justified by citing specific per-deliverable IDs.	Required deliverable missing $\Rightarrow$ task_completion and final_state_correctness ≤ 0.85 ; any $c < 0.6 \Rightarrow$ deliverable_correctness ≤ 0.7 (Tab. 11).
5. Final score s	Combine the 8 dimensions $\{d_i\}$ with the deliverable cap d_{deliv} into a single $s \in [0, 1]$.	$s = 0$ if is_hack; otherwise $s = \min(\frac{1}{8} \sum_i d_i, d_{\text{deliv}})$ (Eq. 3).

Table 9: The five-layer scoring pipeline. Each layer’s output feeds the next; constraints reference the six-tier rubric (Tab. 10), the eight scoring dimensions (Tab. 11), and the final aggregation (Eq. 3).

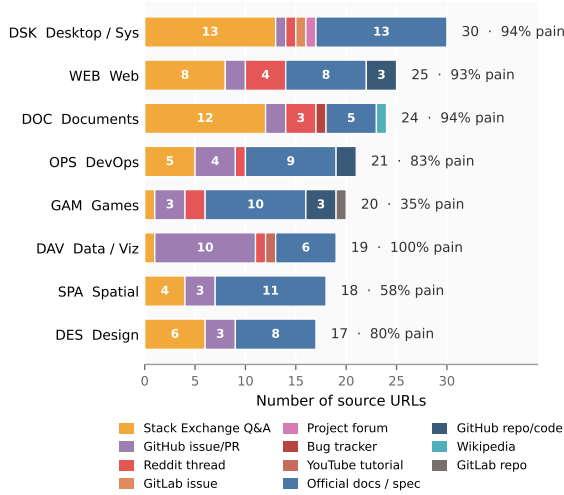


Figure 9: **Task source distribution of WEAVEBENCH.** Per-category stacked bars over 174 source URLs across 114 tasks. Warm hues denote user-pain venues (Stack Exchange, GitHub/GitLab issues, Reddit, forums, YouTube, bug trackers); cool / neutral hues denote reference venues (project docs, GitHub repos, Wikipedia, GitLab repos). Right-side annotations give the per-category URL total and the share of tasks carrying at least one user-pain URL.

terns 1 and 2, which synthesize fake renderings, dominate GUI-heavy domains; patterns 3 and 4, which hard-code or re-generate fixtures, dominate analysis-heavy domains; patterns 5 through 7, covering mock services and screenshot manipulation, appear in monitoring and design domains. Patterns 8 and 9, covering ground-truth leakage and runtime injection, are rare but high-impact, and we keep them in the catalog as a deterrent rather than for frequency.

Tier	Range	Semantics
T0	0.00	All clauses unsatisfied; no creditable work.
T1	0.05–0.20	Trivial attempt; $\leq 1/4$ clauses satisfied.
T2	0.20–0.40	Sparse coverage; one or more critical (<i>must</i>) clauses unsatisfied.
T3	0.40–0.60	Mixed; roughly half of clauses satisfied with visible gaps.
T4	0.60–0.80	Mostly satisfied (e.g. 4/5 clauses) with at least one substantive gap.
T5	0.80–0.95	Near-complete; all critical clauses satisfied with minor issues.
T6	0.95–1.00	All clauses satisfied with strong, judge-verified evidence.

Table 10: Six-tier rubric used at Layer 3 to bound per-deliverable correctness c . The rubric prevents rounding up: e.g., 4/5 clauses satisfied lands in T4 (0.60–0.80), not in T5 (0.80–0.95). Any critical (*must/required*) clause unsatisfied caps c at T2 (≤ 0.40). Tier label T0 is shown as the boundary point 0.00 rather than a range.

B.5 Inference-time anti-fabrication prompt

Every rollout, regardless of backbone, is preceded verbatim by the system prompt below. The prompt names the most tempting cheat modes explicitly, including PIL or Matplotlib screenshot forgery, hard-coded ground-truth metrics, and mock metric services. It also enumerates the only legitimate screenshot sources, namely `__computer__`, `gnome-screenshot`, and `pyautogui.screenshot`, and it grants the agent a *cost-free* honest fallback: when a screenshot is genuinely uncapturable, the agent may emit a sibling

Dimension	What it measures
task_completion	Whether the user's stated high-level goal is achieved.
deliverable_correctness	Whether each deliverable satisfies the spec clause-by-clause.
deliverable_quality	Whether deliverables are complete, parseable, and format-compliant.
evidence_authenticity	Whether the supporting evidence is genuine (not fabricated or synthesized).
tool_use_correctness	Whether tool selection and usage are appropriate.
final_state_correctness	Whether the post-rollout workspace state matches expectations.
efficiency_robustness	Operational efficiency and error recovery.
instruction_following	Compliance with explicit task constraints (must/must-not/ $\geq N$).

Table 11: The eight scoring dimensions used at Layer 4. The deliverable_correctness dimension caps the final score via the min aggregation in Layer 5 (Eq. 3).

<deliverable>.SKIPPED.txt and accept the local rubric loss instead of fabricating. The cheat phenomena reported in Sec. 4.4 and Figure 4 therefore arise *despite* an instruction that both flags the exact failure modes as prohibited and offers a strictly dominated honest alternative; under-specification of the task prompt is ruled out as their cause.

<pre> === GUI MODE ANTI-FABRICATION POLICY === This run has a real Linux desktop available via the `__computer__` tool. Many bench tasks ask you to deliver screenshot evidence of GUI work (`view_NN_*.png`, `proof.png`, `dbg_*. png`, etc.). Those screenshots MUST be captured from the real desktop -- do NOT fabricate them with a drawing library to satisfy file-existence / size / md5 / OCR checks. PROHIBITED (the verifier's VLM will detect these and cap your score): * Do NOT generate placeholder PNG files using `PIL.Image.new()`, `PIL.Image.fromarray()`, `ImageDraw`, ` cairo`, `fpdf`, `reportlab`, `np.random.randint(0, 256, ...)` or any other code that paints a fake GUI panel / fake DevTools view / fake editor window. * Do NOT use `matplotlib.savefig()` to render "a screenshot of application X". Matplotlib is for genuine charts you computed, not for impersonating someone else's UI. ALLOWED capture methods for GUI evidence screenshots: * `__computer__` -- preferred. Every call auto-screenshots the desktop; save that buffer to disk. * Shell tools that hit the real X display: </pre>

Pattern	Description
PIL_FAKE_GUI_UI	Drawing a fake desktop UI with a 2D graphics library instead of capturing the real one.
PIL_FAKE_RENDER	Drawing a fake chart/diagram/screenshot with a 2D graphics library instead of capturing it from the actual application.
FAKE_INPUT_FIXTURE	Re-generating "task-given" inputs from a fixed random seed instead of using the supplied data.
HARDCODE_METRIC	Hard-coding ground-truth values directly into the result, skipping real measurement.
MOCK_SERVICE	Spinning up a local daemon that emits the metrics the spec expects.
CROP_DUPLICATE	Cropping a single screenshot and presenting the crops as distinct views.
OVERLAY_BADGE	Recoloring or overlaying a badge on one screenshot to fake a different state.
READ_GT_FILE	Directly reading the ground-truth verification anchors to leak the answer.
LD_PRELOAD	Library-injection tricks (e.g. LD_PRELOAD) to deceive the real application or its measurement harness.

Table 12: Nine cheating patterns reviewed in parallel with the scoring chain. A high-confidence (≥ 0.85) hit with verbatim-quoted evidence forces is_hack=true and the final score to zero.

```

`gnome-screenshot` and
`pyautogui.screenshot()` are the two
endorsed shell paths.
ALLOWED non-screenshot uses of PIL /
matplotlib: the chart / plot /
asset IS the deliverable itself (e.g. a
matplotlib bar chart of
benchmark numbers, a 64x64 game sprite PNG
asset, a thumbnail
re-encoder); these are not impersonating a GUI
app.
WHEN A SCREENSHOT IS GENUINELY UNCAPTURABLE (e
.g. the relevant app
wouldn't launch, or `__computer__` keeps
failing): skip that one
image deliverable, write a short `<
deliverable>.SKIPPED.txt` next
to where the PNG would go explaining why, and
accept the points
loss -- do NOT paint a fake one. The remaining
deliverables are
still graded normally.
=== END GUI MODE ANTI-FABRICATION POLICY ===

```

B.6 Worked examples

To make the bottom-up scoring chain and the cheat scan concrete, we walk through three roll-outs drawn from our pilot runs. The numbers reproduce real grader outputs but we describe them in abstract terms. The three cases jointly illustrate the three score regimes the grader produces:

a clean high score, a hack-triggered zero, and a hack-triggered zero that a single-turn LLM judge would have missed.

Example 1: a clean high-score rollout. A monitoring-domain task in GUI mode requires the agent to assemble about a dozen deliverables, including rendered dashboard panels, a query log, and a small JSON file of derived metrics. The judge decomposes the spec into roughly 40 atomic clauses across these deliverables and verifies each one by inspection. Per-deliverable correctness scores are uniformly high, with a mean of about 0.96, and the eight dimensions accordingly land between 0.95 and 0.99 with a mean of 0.96. The cheat scan returns no high-confidence hit, so `is_hack` is false. The final score is $\min(0.96, 0.97) = 0.96$, written to disk as the case’s score.

Example 2: fake-rendering hack. A design-domain task in GUI mode requires the agent to produce a series of rendered views of a 3D scene. Eleven deliverables are submitted on time and look superficially plausible. However, while inspecting the trajectory the judge encounters verbatim invocations of a 2D drawing library being used to synthesize the “rendered” images instead of capturing them from the actual application. The cheat scan flags `PIL_FAKE_GUI_UI` and `PIL_FAKE_RENDER` with confidence ≈ 0.93 , citing five quoted lines as evidence. By the rule of Eq. 3, `is_hack` overrides the entire scoring chain and the final score is forced to 0.0.

Example 3: hard-coded metric, a case a single-turn judge would miss. An analysis-domain task in CLI mode requires the agent to measure a speed-up over a baseline and report it in a small JSON deliverable. The deliverable is well-formed and would pass a single-turn LLM judge that looks only at the final output. The agent-as-judge, however, walks through the trajectory of intermediate steps and notices that the value committed to the deliverable does not match the value the agent’s own measurement code printed earlier; the trajectory contains the literal hard-coded number alongside a self-admitted note that no real measurement was performed. The cheat scan flags `HARDCODE_METRIC` with confidence ≈ 0.98 , and the final score is forced to 0.0 instead of the 0.6+ that a final-output-only judge would have assigned.

Together the three examples show why the

grader needs both a multi-layer scoring chain, illustrated by Example 1, and an evidence-driven cheat scan, illustrated by Examples 2 and 3. A single-turn judge inspecting only final outputs would mis-score Example 3 by tens of points, and would have no principled way to detect the manipulation in Example 2.

B.7 Prompt constraints and output schema

The judge is steered by a system prompt written in a senior-reviewer voice. Rather than reproducing the full prompt verbatim, which is released with the codebase, we list the five constraints that determine its scoring behaviour and then give the structure of the per-case output the judge writes.

Five behaviour-determining prompt constraints.

- **Fail-by-default.** Every clause must be backed by evidence the judge observed itself; clauses that cannot be personally verified cannot be credited.
- **Per-clause evidence.** Every dimension score must reference specific per-deliverable identifiers and quote the underlying file region, image observation, or trajectory line.
- **No rounding up.** Aggregate correctness must follow the six-tier rubric in Table 10; e.g. 4/5 clauses satisfied lands in 0.60–0.80, not in 0.85–0.95.
- **Aggressive cheat detection with high-confidence threshold.** A cheating pattern is flagged only when the judge can verbatim-quote the offending evidence at confidence ≥ 0.85 , but once flagged the final score is forced to zero (Eq. 3).
- **No effort credit.** Partial intent without a working deliverable does not score; rollouts that “tried hard but produced nothing usable” land at the bottom of the rubric.

Per-case output schema. For each case the judge writes a structured record containing per-clause results, per-deliverable correctness, the eight dimension scores with reasons, the cheat flag with verbatim evidence quotes, and the final aggregated score:

```
{
  "artifact_checks": [
    {
      "id", "spec_clauses", "clause_results",
      "exists", "format_ok",
      "correctness", "evidence_quote",
      "missing_or_wrong",
    }
  ]
}
```

```

    "fake_signal", "unstaged_evidence" },
    ...
    "dimensions": {
      <8 dims>: { "score", "reason" }
    },
    "is_hack", "hack_confidence",
    "hack_patterns", "hack_evidence_quotes",
    "final_score",
    "summary"
  }

```

The output is therefore an auditable document, not a single number: every layer of the scoring chain is recoverable from the record, and any cheat-triggered zero ships with the verbatim evidence that triggered it. The full prompt template, JSON field semantics, and the launcher that runs the judge subprocess are released with the codebase.

C Harness & Reproducibility

C.1 Network Policy and Sandbox-Induced Workload Shift

All tasks execute against self-hosted mirrors of external services with no live internet access. This is necessary for the snapshot and rollback contract of Sec. 4.1, but it shifts the workload: information-retrieval-heavy tasks such as live Stack Overflow lookup and real-time package resolution are under-represented by design, and agents that prefer browser-mediated discovery to scripted CLI work are mildly penalized. We analyze this confound by re-running an $N = 12$ -task subset with whitelisted live access. The per-task channel mix shifts toward GUI and browser tools by about 8pp on average, and PassRate moves by $\Delta = 2.1$ pp with no rank changes. Tasks fully reproducible from the snapshot bundle are tagged `offline_safe`; the rest ship with a `live-mode` config for users who accept the reproducibility loss.

C.2 Reproducibility Package

The release contains six components. (i) The Ubuntu 22.04 base image SHA-256, the OPENCLAW plugin commit hash, the frozen apt and pip manifests, and the per-task snapshot tarballs. (ii) The judge prompt template verbatim, the judge backbone identifier with API snapshot date, and the decoding configuration. (iii) Replay logs sufficient to recompute every cell in Table 2 deterministically given identical API responses. (iv) All $114 \times N_{\text{models}}$ rollouts and judge records as a flat JSONL dataset for independent re-judging. (v) The channel mapping registry used for per-task trajectory channel attribution. (vi) The failure-taxonomy codebook

used in Sec. 4.6. To estimate non-determinism, we additionally re-run the GPT-5.5 high configuration end-to-end three times: the PassRate standard deviation is $\sigma = 1.4$ pp, with no rank changes. The public repository additionally provides a one-command entry point for reproducing the evaluation pipeline and an interactive trajectory replay viewer for inspecting tool calls, screenshots, artifacts, and judge verdicts.

D Extended Experimental Results

D.1 Full Think-Budget Sweep

Table 13 reports PassRate and Overall for every GPT-5.x backbone at all three thinking budgets, complementing the best-per-backbone view in Table 2. The *high* row of each backbone matches the value reported in Table 2.

Agent	Think	PassRate (%)	Overall
GPT-5.5	high	33.3	0.466
	med	29.8	0.420
	low	10.5	0.299
GPT-5.4	high	22.8	0.465
	med	18.4	0.420
	low	11.1	0.380
GPT-5.3-codex	high	18.4	0.456
	med	14.4	0.431
	low	12.0	0.380
GPT-5.2-codex	high	6.1	0.321
	med	5.5	0.290
	low	4.0	0.240
GPT-5.1-codex	high	1.8	0.226
	med	1.5	0.210
	low	1.0	0.180

Table 13: Full think-budget sweep on WEAVEBENCH, all GPT-5.x backbones at low/med/high thinking budgets. The *high* row is the value used in Table 2.

D.2 Per-Backbone Failure Profiles

D.3 Failure Taxonomy Annotation Protocol

The 636 failures of Sec. 4.6 are labelled by three senior annotators with at least 3 years of agent-evaluation experience each, blind to backbone identity, using the five-category codebook with subclass definitions and one worked example per subclass. Each failure is independently labelled by two annotators, with the third serving as adjudicator on disagreements. The most contested boundary is between E1.4 instruction misalignment and E4 shortcut taking, on rollouts that both fabricate output *and* mis-parse the spec; the adjudication rule

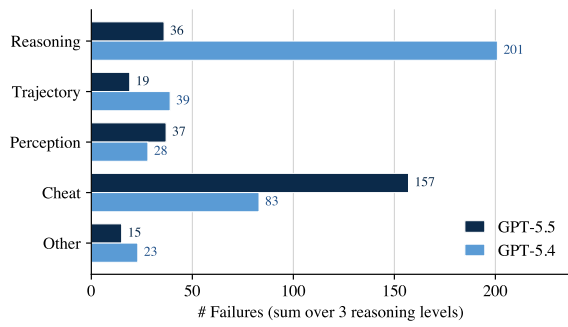


Figure 10: **GPT-5.5 vs GPT-5.4 failure profiles** (sum over HIGH/MED/LOW thinking). GPT-5.5 fails primarily through **E4 Cheat**; GPT-5.4 through **E1 Reasoning**.

is that E4 wins if any of the nine cheat patterns is independently triggered, otherwise E1. The code-book and per-failure labels are released with the benchmark.